Flatpak Documentation

Flatpak Team

24.10.2022
## Inhaltsverzeichnis

1 Inhalt 3

1.1 Einführung in Flatpak ......................................................... 3
1.2 Erste Schritte ............................................................... 5
1.3 Erstellen von Flatpaks ........................................................ 12
1.4 Debugging ................................................................. 34
1.5 Publishing ................................................................. 35
1.6 Desktop Integration ......................................................... 39
1.7 Tips and Tricks ............................................................ 43
1.8 Reference Documentation ................................................... 46
Diese Dokumentation deckt alles ab, was Sie wissen müssen, um Anwendungen, unter Verwendung von Flatpak, zu erstellen und zu verteilen. Sie beginnt mit einer grundlegenden Einführung zu Flatpak, Hintergrundinformationen zu grundsätzlichen Konzepten und einer Anleitung zu der Kommandozeilenschnittstelle von Flatpak. Spätere Abschnitte enthalten detaillierte Informationen, wie man Anwendungen erzeugt und verteilt.

Die Dokumente richten sich vorrangig an Anwendungsentwickler und Distributoren. Der Inhalt ist aber auch ebenso wissenswert für diejenigen, die ein allgemeines Interesse an Flatpak haben.

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

Flatpak is a framework for distributing desktop applications across various Linux distributions. 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 Terminology

- Flatpak: a system for building, distributing, and running sandboxed desktop applications on Linux.
- Flatpak application: these are the applications the user installs via the `flatpak` command or via a different UI like GNOME Software or KDE Discover.
- Runtime: also called platforms, these are integrated platforms to provide basic utilities needed for a Flatpak application to work.
- BaseApp: these are integrated platforms for frameworks like Electron.
- Flatpak bundle: a specific single-file export format which contains a Flatpak app or runtime.

1.1.2 Zielgruppe

Flatpak kann von allen Arten von Desktop-Anwendungen verwendet werden und verfolgt dabei das Ziel so flexibel wie möglich zu sein, wie Anwendungen erstellt werden. Es existieren keine Vorgaben, welche Programmiersprachen, Build-Tools, Toolkits oder Frameworks verwendet werden können.

Während Flatpak lediglich unter Linux lauffähig ist, kann es genauso von Anwendungen verwendet werden, die andere Betriebssysteme adressieren, als auch Linux-spezifisch sind. Anwendungen können Open Source oder proprietär sein (auch wenn einige Distributions-Dienste, wie Flathub, Restriktionen mitbringen im Hinblick auf diesen Aspekt).

Die einzige technische Vorgabe, die seitens Flatpak gegeben ist, lautet, dass Anwendungen einer kleinen Zahl von Freedesktop-Standards folgen, um eine Desktop-Integration zu ermöglichen (siehe Anforderungen & Konventionen).
1.1.3 Issues of current model of packaging

It is important to understand the problems of the current model of packaging applications to understand the existence of Flatpak:

- **Duplicated work packaging apps:** many Linux distributions come with their own package manager, package format and repository. This requires a lot of maintainers to package the same application in various distributions, or the application developer to learn the language of each format and then package the application in those distributions, or ignore most distributions and package and support a couple of distributions. This makes the Linux desktop a difficult platform for software vendors to target.

- **Limited to apps that are packaged:** not all applications are natively available in every Linux distribution. If an application is not available in a specific distribution, the user will have to rely on manually downloading the archive of the application, extracting it and hoping the application will launch.

- **Limited to distributions that have the apps:** the user is limited to the number of distributions that have the needed applications for them to properly setup their workflow. This reduces the amount of distributions that can be suitable for a user.

- **Hard to innovate in OS space:** the maintainers of the distributions have to spend a lot of time packaging applications to make the distribution suitable for the end user, instead of focusing on their end goals. This delays the progress of each distribution.

- **Old and outdated packages:** LTS distributions often have very old versions of applications packaged natively. Bug reproducibility is hindered by the different environments that applications are run in, and application developers often have little control over how their application is packaged by distributions.

Flatpak strives to fix the issues listed above, by conveniently enabling developers to distribute applications from one source and to target the entire Linux desktop.

1.1.4 Gründe Flatpak zu verwenden

Flatpak has some major advantages over most system package managers:

- **Universality:** Flatpak allows applications to be installed and run on virtually any Linux distribution. This includes non-GNU distributions, systemd-free distributions, distributions with a read-only operating system (OS), and various architectures without the developer needing the relevant hardware on hand.

- **Space for innovations:** Flatpak facilitates distribution maintainers to focus on their goals to innovate their distribution.

- **Stability:** breakage in a Flatpak application will not risk the system from breaking. This is because Flatpak applications and runtimes are contained to not interfere with the system altogether.

- **Rootless install:** elevated privileges are not required when installing a Flatpak application or a runtime.

- **Sandboxed applications:** 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 limited access to the host environment.

Flatpak has some major advantages over other universal approaches to distributing applications on Linux:

- **Decentralized by design:** 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.

- **Desktop integration:** 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.

- **Space efficiency:** Flatpak deduplicates libraries and other files used by multiple applications to save megabytes or even gigabytes worth of storage depending on the amount of applications installed.
• **Delta updates**: only changed files are downloaded for updates.

Andere Vorteile für Entwickler beinhalten:

• **Forward-compatibility**: the same Flatpak application 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.

• **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.

• **Branches**: this allows developers to ship applications from different branches, e.g. `stable`, `beta`, etc. while retaining the same name.

• **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.


### 1.2 Erste Schritte

Dieser Abschnitt beinhaltet eine Einführung zu grundlegenden Konzepten von Flatpak, einer Anleitung, wie das Kommandozeileninterface verwendet wird und einem Tutorial, um eine einfache Anwendung zu erzeugen.

Um diesen Abschnitt durchzugehen, sollte Flatpak installiert und das Flathub-Repository aktiviert sein. Die Flatpak-Website stellt Instruktionen, wie dies mit einer Reihe von Distributionen durchgeführt wird, zur Verfügung.

#### 1.2.1 Grundlegende Konzepte

Flatpak wird verständlich durch eine kleine Zahl an Schlüsselkonzepten. Es ist sinnvoll, mit diesen vertraut zu sein, bevor man lernt Flatpak auf der Kommandozeile zu verwenden oder um Applikationen zu generieren.

**Laufzeitumgebungen**

Laufzeitumgebungen beinhalten die grundlegenden Abhängigkeiten, die von den Anwendungen verwendet werden. Jede Anwendung muss gegen eine Laufzeitumgebung erzeugt werden und diese Laufzeitumgebung muss auf einem Gastsystem installiert sein, als Voraussetzung damit die Anwendung lauffähig ist (Flatpak kann die Laufzeitumgebung, die von einer Applikation benötigt wird, automatisch installieren). Es können mehrere unterschiedliche Laufzeitumgebungen zur gleichen Zeit installiert werden, dies beinhaltet auch verschiedene Versionen der gleichen Laufzeitumgebung.


**Gebündelte Bibliotheken**

Wenn eine Anwendung Abhängigkeiten benötigt, die nicht in seiner Laufzeitumgebung vorhanden sind, können diese gebündelt werden als Teil der Applikation. Dies gibt Anwendungsentwicklern Flexibilität im Hinblick auf die Abhängigkeiten, die sie verwenden. Dies beinhaltet die Verwendung von:
Flatpak Documentation

- libraries that aren’t available in a runtime
- different versions of libraries from the ones that are in a runtime
- gepatchte Versionen von Bibliotheken

**Sandboxen**


Einige Ressourcen, die innerhalb der Sandbox liegen, müssen bei Bedarf nach außen freigegeben werden, um vom Gastsystem verwendet zu werden. Diese sind bekannt als »Exports«, da dies Dateien sind, die aus der Sandbox heraus exportiert werden und Dinge enthalten wie die `.desktop`-Datei und Icon einer Anwendung.

**Portale**

Portale sind ein Mechanismus durch den Applikationen, aus einer Sandbox heraus, mit der Gastumgebung interagieren können. Sie geben die Möglichkeit mit Daten, Dateien und Diensten zu interagieren, ohne die Notwendigkeit Sandbox-Berechtigungen hinzuzufügen.


Weitere Informationen zu Portalen können unter *Sandbox Permissions* gefunden werden.

**Repositorys**

Anwendungen und Laufzeitumgebungen von Flatpaks werden normalerweise gespeichert und veröffentlicht unter Verwendung von Repositorys, die sich sehr ähnlich zu Git-Repositorys verhalten. Ein Flatpak-Repository kann ein einziges Objekt oder mehrere enthalten, jedes Objekt ist dabei versioniert, was upgraden und sogar downgraden ermöglicht.


Wenn eine Aktualisierung durchgeführt wird, werden neue Versionen der installierten Anwendungen und Laufzeitumgebungen von dem relevanten Remote heruntergeladen. Wie mit Git, werden lediglich Unterschiede zwischen Versionen heruntergeladen, was den Prozess sehr effizient macht.

**1.2.2 Verwenden von Flatpak**

Diese Seite enthält eine Einführung zu der Flatpak-Kommandozeilenschnittstelle und erläutert essenzielle technische Konventionen, sowie die häufigst verwendeten Befehle.

Endbenutzer sollten auf diese Seite oder die Flatpak-Kommandozeilenschnittstelle für gewöhnlich nicht angewiesen sein, da Flatpak mit grafischen Softwaremanagement-Tools verwendet werden kann. Wenn es bevorzugt wird, steht es natürlich frei, die Kommandozeile zu verwenden.
Das Flatpak-Kommando

flatpak ist das primäre Flatpak-Kommando, an dieses werden spezifische Befehle angefügt. Zum Beispiel lautet das Komando etwas zu installieren flatpak install und etwas zu deinstallieren flatpak uninstall.

Identifikatoren


Identifikator-Tripel

Normalerweise ist es ausreichend auf Objekte unter Verwendung ihrer ID zu referenzieren. Jedoch ist es in manchen Situationen notwendig, auf eine bestimmte Version eines Objektes oder eine spezifische Architektur zu referenzieren. Beispielsweise können einige Anwendungen als stabile Versionen verfügbar sein oder als Testversion, in diesem Fall ist es notwendig anzugeben, welche installiert werden soll.


Das Flatpak-CLI gibt Feedback, ob Architektur oder Branch eines Objektes spezifiziert werden muss.

System versus Benutzer


Das gleiche Prinzip gilt für Repositories – Repositories, die systemweit hinzugefügt wurden, sind verfügbar für alle Benutzer, während benutzerbasierte Repositories lediglich von einem bestimmten Benutzer verwendet werden können.


Befehle verhalten sich exakt in der gleichen Weise, wenn sie benutzerbasiert laufen, als wenn sie systemweit ange- wendet werden.
Grundlegende Kommandos

Dieser Abschnitt deckt grundlegende Befehle ab, die benötigt werden, um Flatpak-Anwendungen zu installieren, zu starten und zu verwalten. Für die vollständige Liste der Flatpak-Kommandos kann man `flatpak --help` aufrufen oder in die Flatpak Command Reference schauen.

Auflisten von Remotes

Um die Remotes aufzulisten, die auf dem System konfiguriert sind, verwendet man diesen Befehl:

```bash
$ flatpak remotes
```

Ausgegeben wird eine Liste der existierenden Remotes, die hinzugefügt wurden. Die Auflistung zeigt an, ob die jeweiligen Remotes benutzerbasiert oder systemweit installiert wurden.

Hinzufügen eines Remotes

Der zweckmäßigste Weg einen Remote hinzuzufügen, ist eine flatpakrepo-Datei zu verwenden, welche sowohl die Details des Remote und seines GPG-Schlüssels enthält:

```bash
```


Entfernen eines Remotes

Um ein Remote zu entfernen, ruft man auf:

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

In diesem Fall ist flathub der lokal vergebene Name des Remotes.

Suchen

Anwendungen können in allen systemweiten und benutzerbasierten Remotes gefunden werden unter Hinzunahme des search-Kommandos. Beispielsweise:

```bash
$ flatpak search gimp
```


Installieren von Anwendungen

Um eine Anwendung zu installieren, verwendet man:
$ flatpak install flathub org.gimp.GIMP

flathub ist hierbei der Name des Remotes, von dem die Applikation installiert werden soll und org.gimp.GIMP ist die ID der Applikation.

Gelegentlich benötigt eine Anwendung eine bestimmte Laufzeitumgebung, diese wird dann vor der Anwendung installiert.

Die Details für die Installation der Anwendung können auch in einer .flatpakref-Datei zur Verfügung gestellt werden, diese kann entweder remote oder lokal vorliegen. Um ein .flatpakref zur Verfügung zu stellen, anstatt händisch das Remote und die Anwendungs-ID anzugeben, verwendet man:

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

Wenn die .flatpakref-Datei spezifiziert, dass die Anwendung von einem Remote installiert wird, das noch nicht hinzugefügt wurde, wird gefragt, ob dieses hinzugefügt werden soll, bevor die Applikation installiert wird.

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.

**Starten von Anwendungen**

Sobald eine Anwendung installiert wurde, kann diese aufgerufen werden unter Verwendung des run-Kommandos und der zugehörigen Anwendungs-ID:

$ flatpak run org.gimp.GIMP

**Aktualisieren**

Um alle installierten Anwendungen und Laufzeitumgebungen auf die aktuellsten Versionen zu aktualisieren, verwendet man:

$ flatpak update

**Auflisten von installierten Anwendungen**

Um alle installierten Anwendungen und Laufzeitumgebungen aufzulisten, verwendet man:

$ flatpak list

Um lediglich die installierten Anwendungen aufzulisten, verwendet man:

$ flatpak list --app

**Entfernen einer Anwendung**

Um eine Anwendung zu entfernen, verwendet man:
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:

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

To fix inconsistencies with your local installation, use:

```bash
$ 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`:

```bash
$ 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):

```bash
$ flatpak history
```

1.2.3 Erstellen des ersten Flatpaks

Dieses Tutorial liefert eine schnelle Einführung in die Erstellung von Flatpaks. In diesem lernt man, wie man grund- sätzlich eine einfache Flatpak-Anwendung erstellt, die dann installiert und gestartet werden kann.

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. Installieren einer Laufzeitumgebung und des zugehörigen SDK

Flatpak fordert von jeder Anwendung, eine Laufzeitumgebung zu spezifizieren, die für es für die Auflösung der grund- sätzlichen Abhängigkeiten benötigt. Jede Laufzeitumgebung hat ein zugehöriges SDK (Software Development Kit), das alle Bestandteile der Laufzeitumgebung enthält, sowie Header-Dateien und Entwicklung-Tools. Das SDK wird benötigt, um Anwendungen für die Laufzeitumgebung zu erstellen.

In this tutorial we will use the Freedesktop 21.08 runtime and SDK. To install these, run:

```bash
```

2. Erstellen der App

Die Anwendung, die für dieses Tutorial erstellt wird, ist ein einfaches Skript. Um sie zu erstellen, kopiert man Folgen- des:

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

Dies wird in eine leere Datei eingefügt und als `hello.sh` gespeichert.
3. Hinzufügen eines Manifests

Jedes Flatpak wird unter Verwendung einer Manifest-Datei gebaut, dieses enthält grundlegende Informationen über die Anwendung und Instruktionen darüber, wie es gebaut werden soll. Um ein Manifest zu der hello world-anwendung hinzuzufügen, fügt man das Folgende in eine leere Datei ein:

```
app-id: org.flatpak.Hello
runtime: org.freedesktop.Platform
runtime-version: '21.08'
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.yml.

In einer komplexeren Anwendung würde das Manifest verschiedene Module auflisten. Das letzte Modul würde standardmäßig die Anwendung selbst sein und die davor aufgelisteten Module würden Abhängigkeiten sein, die mit der Anwendung gebündelt werden, da sie kein Bestandteil der Laufzeitumgebung sind.

4. Erzeugen der Anwendung

Nachdem die Anwendung ein Manifest besitzt, kann flatpak-builder verwendet werden, um sie zu erzeugen. Durchgeführt wird dies, indem ein Zielverzeichnis und die Manifest-Datei angegeben wird:

```
$ flatpak-builder build-dir org.flatpak.Hello.yml
```

Dieses Kommando erzeugt jedes Modul, das im Manifest gelistet ist und installiert es, innerhalb des build-dir-Verzeichnisses, in das /app-Unterverzeichnis.

5. Testen des Builds

Um zu testen, ob der Build erfolgreich war, verwendet man dieses Kommando:

```
$ flatpak-builder --user --install --force-clean build-dir org.flatpak.Hello.yml
$ flatpak run org.flatpak.Hello
```

Im zweiten Durchlauf haben wir --force-clean durchgereicht, was bedeutet, dass das zuvor erzeugte build-dir-Verzeichnis gelöscht wurde, bevor der neue Build angestoßen wird.

Die erste Anwendung wurde damit erfolgreich erstellt.

6. Die Anwendung in ein Repository legen

If you want to share the application you can put it 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.yml
```

1.2. Erste Schritte
Dies stößt den Build-Prozess erneut an und exportiert das Resultat am Ende in ein lokales Verzeichnis namens repo. Zu beachten ist, dass flatpak-builder einen Cache behält von vorherigen Builds im flatpak-builder-Unterverzeichnis, somit kann sehr schnell ein zweiter Build, wie dieser, erzeugt werden.

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. Installieren der Anwendung

Nun ist alles so weit vorbereitet, um das Repository, das eben erstellt wurde, hinzuzufügen und die Anwendung zu installieren. Dies wird mit zwei Kommandos durchgeführt:

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

Das erste Kommando fügt das Repository hinzu, das im vorherigen Schritt erzeugt wurde. Das zweite Kommando installiert die Anwendung vom Repository.

Beide Kommandos verwenden das --user-Argument, damit wird das Repository für den jeweiligen Benutzer hinzugefügt – und nicht systemweit. Dies ist sinnvoll für Testläufe.

Hierbei ist zu beachten, dass das Repository mit --no-gpg-verify hinzugefügt wurde, da kein GPG-Schlüssel spezifiziert wurde, als die Anwendung erzeugt wurde. Dies ist gut zum Testen, aber für offizielle Repositorys sollten diese mit einem privaten GPG-Schlüssel signiert sein.

8. Starten der Anwendung

Zum Abschluss kann die Anwendung getestet werden. Um dies durchzuführen wird folgendes Kommando eingegeben:

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

Das startet die Anwendung und gibt den Text 'Hello world, from a sandbox' aus.

1.3 Erstellen von Flatpaks


Wenn noch nicht geschehen, wäre es ein guter Zeitpunkt den Abschnitt Erstellen des ersten Flatpaks durchzugehen, bevor dieser Abschnitt gelesen wird.

1.3.1 Einführung zum Erstellen von Flatpaks

Der Abschnitt Erstellen des ersten Flatpaks beinhaltete bereits eine kurze Demonstration, wie man Anwendungen mit Flatpak erzeugt. Diese Seite bietet zusätzlich eine allgemeine Übersicht, was hierbei involviert ist.
flatpak-builder


flatpak-builder erhält als Eingabe eine Manifest-Datei. Diese spezifiziert die Parameter für die Anwendung, die erstellt werden sollen, so wie den Namen der Anwendung und von welcher Laufzeitumgebung es abhängig ist. Das Manifest listet zudem alle Module, die erzeugt werden als Teil des Build-Prozesses. Eine Quelle für jedes Modul kann benannt werden, dies beinhaltet auch Dateiarchive oder Repositorys mit Versionskontrolle. Eines der Module (gewöhnlich das Letzte) ist der Programmcode der Anwendung selbst.

Das zugrundeliegende Format für den Aufruf von flatpak-builder sieht so aus:

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

<build-dir> bezeichnet den Pfad zum Verzeichnis, in dem die Anwendung erzeugt wird und <manifest> entspricht dem Pfad zur Manifest-Datei. Der Inhalt von <build-dir> kann zum Testen und Debuggen nützlich sein, wird aber generell als Artefakt des Build-Prozesses behandelt.

Das Ausführen von flatpak-builder setzt folgende Prozesse in Gang:

- Das Build-Verzeichnis wird erzeugt, sofern es noch nicht existiert.
- Der Quellcode wird für jedes Modul heruntergeladen und verifiziert.
- Der Quellcode wird für jedes Modul kompiliert und installiert.
- The build is finished by setting sandbox permissions
- Das Resultat des Erstellungsprozesses wird zu einem Repository exportiert (welches erzeugt wird, wenn es nicht bereits existiert).

Die Anwendung kann vom Repository aus installiert und anschließend gestartet werden.

Software Development Kits (SDKs)

Für den Erstellungsprozess wird nicht das Gastsystem verwendet, anstelle dessen werden Flatpak-Anwendungen innerhalb einer separaten Umgebung erzeugt, dem SDK.

SDKs sind wie die regulären Laufzeitumgebungen in denen Anwendungen laufen. Der Unterschied ist, dass SDKs zusätzlich alle Entwicklungs-Ressourcen beinhalten und Tools, die benötigt werden, um eine Anwendung zu erzeugen, sowie Tools für die Erstellung und Paketierung, Header-Dateien, Compiler und Debugger.

Jede Laufzeitumgebung verfügt über ein korrespondierendes SDK. Beispielsweise existiert eine GNOME 3.26 Laufzeitumgebung und ein GNOME 3.26 SDK. Anwendungen, die diese Laufzeitumgebung verwenden, werden mit dem zugehörigen SDK erstellt.

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 Anforderungen & Konventionen

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 *Verwenden von 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 generally a domain controlled by the project and the trailing section represents the specific project. There are some exceptions to this, such as extensions using the base application-id of the project they extend rather than their own.

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 for bus names 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 it is 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 `.metainfo.xml` file extension, and should be placed in `/app/share/metainfo/`. For example:

```
/app/share/metainfo/org.gnome.Dictionary.metainfo.xml
```
A legacy convention of having the `.appdata.xml` installed in `/app/share/appdata` is also accepted as well, and `flatpak-builder` will check either directory with either extension. 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 128x128px version of GNOME Dictionary’s icon is:

```
```

**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
```

1.3. Erstellen von Flatpaks 15
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:

<table>
<thead>
<tr>
<th>Base directory</th>
<th>Usage</th>
<th>Default location</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDG_CONFIG_HOME</td>
<td>User-specific configuration files</td>
<td>~/.var/app/&lt;app-id&gt;/config</td>
</tr>
<tr>
<td>XDG_DATA_HOME</td>
<td>User-specific data</td>
<td>~/.var/app/&lt;app-id&gt;/data</td>
</tr>
<tr>
<td>XDG_CACHE_HOME</td>
<td>Non-essential user-specific data</td>
<td>~/.var/app/&lt;app-id&gt;/cache</td>
</tr>
<tr>
<td>XDG_STATE_HOME</td>
<td>State data such as undo history</td>
<td>~/.var/app/&lt;app-id&gt;/local/state</td>
</tr>
</tbody>
</table>

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).

Note that $XDG_STATE_HOME is only supported by Flatpak 1.13 and later. If your app needs to work on earlier versions of Flatpak, you can use the --persist=.local/state and --unset-env=XDG_STATE_HOME finish args so the app will use the correct directory, even after Flatpak is later upgraded to >1.13.
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 Grundlegende Konzepte, 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!

Tipp: 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 in which 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 dependencies). 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 Erstellen des ersten Flatpaks and Einführung zum Erstellen von Flatpaks. 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.

You can put more than one application in the same repository by using the same --repo path for multiple invocations of flatpak-builder.

Bemerkung: 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 [Erstellen des ersten Flatpaks](#) 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.yml`. This page uses this manifest file 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:

```yaml
app-id: org.gnome.Dictionary
runtime: org.gnome.Platform
runtime-version: '3.36'
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**

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 be 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.

---

1.3. Erstellen von Flatpaks
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:

- \texttt{rename-icon} - rename the application icon
- \texttt{rename-desktop-file} - rename the .desktop filename
- \texttt{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 \texttt{finish-args} property, which can be seen in the Dictionary manifest file:

```plaintext
definitions:
  finish-args:
    # X11 + XShm access
    - --share=ipc
    - --socket=x11
    # Wayland access
    - --socket=wayland
    # Needs to talk to the network:
    - --share=network
    # Needs to save files locally
    - --filesystem=xdg-documents
    - --metadata=X-DConf=migrate-path=/org/gnome/dictionary/
```

Guidance on which permissions to use can be found in the *Sandbox Permissions*, and a full list of \texttt{finish-args} options can be found in *Sandbox Permissions Reference*.

If you’re wondering about the last finish arg, see this blog post.

**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:

```plaintext
cleanup:
- '/include'
- '/bin/foo-*'
- '*.*'
```

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

```plaintext
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
    buildsystem: meson
    config-opts:
      - --build-man=false
    sources:
      - type: archive
        url: https://download.gnome.org/sources/gnome-dictionary/3.26/gnome-
        dictionary-3.26.1.tar.xz
        sha256: 16b8bc248dcf68987826d5e39234b1bb7fd24a2607fcd6f4258fde88f012f300
      - type: patch
        path: appdata_oars.patch
```

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
- bzf - 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
- qmake
- the „Build API“

A „simple“ build method is also available, which allows a series of commands to be specified.
**Shared Modules**

Shared Modules (or shared-modules) is a repository containing various manifests to build common libraries. It is intended to be used as a git submodule.

To add it to your repository, run this command:

```
git submodule add https://github.com/flathub/shared-modules.git
```

Then, add whichever module you want. In this example, we will use `gtk2`:

```
modules:
  - shared-modules/gtk2/gtk2.json
```

To update the submodule, run this command:

```
git submodule update --remote --merge
```

To remove the submodule, run these commands:

```
git submodule deinit -f -- shared-modules
rm -rf .git/modules/shared-modules
git rm -f shared-modules
rm .gitmodules
```

**Flatpak Builder Tools**

Flatpak Builder Tools (or flatpak-builder-tools) is a collection of scripts to aid using `flatpak-builder`. In this repository, each directory contains instructions to generate a manifest for the respective platform.

**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, `~/.var/app/$FLATPAK_ID`, and `$XDG_RUNTIME_DIR/app/$FLATPAK_ID`. Only the latter two being 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 *Grundlegende Konzepte*. 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, overrides x11 socket permission. Note that you must still use `--socket=wayland` for wayland permission
- `--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 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 of available filesystem options can be found in the Sandbox Permissions Reference. 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 a wrapper script that sets it to $XDG_RUNTIME_DIR/app/$FLATPAK_ID.
• 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 and newer.

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:

```plaintext
--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.

gvfs access

As of gvfs 1.48, the gvfs daemons and applications use an on-disk socket to communicate, rather than an abstract socket so that the gvfs infrastructure still works when network support is disabled in the application’s sandbox.

A number of different options need to be passed depending on the application’s use of gvfs.

`--talk-name=org.gtk.vfs.*` is necessary to talk to the gvfs daemons over D-Bus and list mounts using the GIO APIs.

`--filesystem=xdg-run/gvfsd` is necessary to use the GIO APIs to list and access non-native files using the GIO APIs, using URLs rather than FUSE paths.

`--filesystem=xdg-run/gvfs` is necessary to give access to the FUSE mounts non-GIO and legacy applications can use. This is what will make native files appear under `/run/user/`id -u`/gvfs/`.

Typical GNOME and GTK applications should use:

```plaintext
--talk-name=org.gtk.vfs.*
--filesystem=xdg-run/gvfsd
```

Typical non-GNOME and non-GTK applications should use:

```plaintext
--filesystem=xdg-run/gvfs
```

No application should be using `--talk-name=org.gtk.vfs` in its manifest, as there are no D-Bus services named org.gtk.vfs.
**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 YAML 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: 9c443e7324ba5b85070c4a818ade28bfbadef16ea10206da1132edaa6dda237e
```

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 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

Or if you have a requirements.txt file you can generate modules for every listed package:

$ python3 flatpak-pip-generator --requirements-file=requirements.txt

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. Even if your manifest uses YAML, you can still include JSON like this:

```yaml
modules:
  - python3-requests.json
  # (other modules go here)
```

**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 and the Node.js SDK extension:

```
$ flatpak install flathub org.freedesktop.Sdk.Extension.node14//21.08
```

Then you can run the build:

```
$ flatpak-builder build org.flathub.electron-sample-app.yml --install --force-clean --user
```

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:

```yaml
app-id: org.flathub.electron-sample-app
runtime: org.freedesktop.Platform
runtime-version: '21.08'
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:

```yaml
base-version: '21.08'
```

Base apps are described in Dependencies. Using the Electron base app is much faster and more convenient than manually building Electron 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.

**The Node.js SDK extension**

In order to build Electron-based apps, you need Node.js available at build time. Flathub provides Node.js LTS versions as extensions for the SDK, so you can install one of them and add it in your apps' manifest:

```yaml
sdk-extensions:
  - org.freedesktop.Sdk.Extension.node14
```

Enable the extension by adding it to `PATH`:

```yaml
build-options:
  append-path: /usr/lib/sdk/node14/bin
```

Note that the extension name (last portion of reverse-dns notation, `node14` in this example) must be the same in `sdk-extensions` and `append-path`.

**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.

```yaml
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:

```yaml
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.

```json
build-options:
  cflags: -O2 -g
  cxxflags: -O2 -g
  env:
    NPM_CONFIG_LOGLEVEL: info
```

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.

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.

```manifest
name: electron-sample-app
buildsystem: simple
build-options:
  env:
    XDG_CACHE_HOME: /run/build/electron-sample-app/flatpak-node/cache
    npm_config_cache: /run/build/electron-sample-app/flatpak-node/npm-cache
    npm_config_nodedir: /usr/lib/sdk/node14
    npm_config_offline: 'true'
  subdir: main
sources:
  - type: archive
    url: https://github.com/flathub/electron-sample-app/archive/1.0.1.tar.gz
    sha256: a2feb3f1cf002a2e4e8900f718cc5c54db4ad174e48bf3fe3d588c7b716d5b
    dest: main
```

Bundling NPM packages

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

```manifest
- 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 or Yarn and include them in an application’s sources.

The Python script requires a `package-lock.json` (or `yarn.lock`) file. This file 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 script is then run as follows:
This generates the manifest JSON needed to build the NPM/Yarn 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-sources.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. See below a sample wrapper for launching app:

```python
- type: script
dest-filename: run.sh
commands:
- zypak-wrapper.sh /app/main/electron-sample-app "$@
```

### 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 npm_config_offline=true environment variable, installing dependencies from 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:

```bash
build-commands:
  # Install npm dependencies
  - npm install --offline
  # Build the app; in this example the `dist` script
  # in package.json runs electron-builder
  - |
    .../flatpak-node/electron-builder-arch-args.sh
    npm run dist -- $ELECTRON_BUILDER_ARCH_ARGS --linux --dir
  # Bundle app and dependencies
  - cp -a dist/linux*unpacked /app/main
  # Install app wrapper
  - install -Dm755 -t /app/bin/ ../run.sh
```

Note that if the application you are trying to package contains a build block in package.json with instructions for Linux, this can cause electron-builder to try to fetch additional binaries at build-time (Even if --dir option is used). The following example shows a configuration that will try to download AppImage binaries:

```json
"build": {
  "linux": {
    "target": "AppImage",
  }
}
```

The preferred way of fixing this, is not a patch, but a build-time edit using jq. The following command will replace "target": "AppImage" with "target": "dir":

```bash
jq '.build.linux.target="dir"' <<<$(<package.json) > package.json
```
Qt

For Qt-based applications we have the org.kde.Platform runtime (and corresponding org.kde.Sdk sdk runtime) that will offer us most Qt modules and KDE Frameworks for our applications to use.

For example, the following YAML makes building a random Qt application really straight-forward.

```yaml
app-id: org.flatpak.qtdemo
runtime: org.kde.Platform
runtime-version: '5.11'
sdk: org.kde.Sdk
command: flatpak-demo
finish-args:
  - --share=ipc
  - --socket=x11
  - --socket=wayland
  - --filesystem=host
  - --device=dri
modules:
  - name: flatpak-demo
    buildsystem: cmake-ninja
    config-opts:
      - DCMAKE_BUILD_TYPE=RelWithDebInfo
    sources:
      - type: archive
        url: https://github.com/flatpak/qt-flatpak-demo/archive/v1.1.2.tar.gz
        sha256: 1a1cc5d0f06ad949d6854c772ec9624b8856a0a4f880355f51058bc0dd52ba7a
```

Contents

The org.kde.Platform runtime includes all of Qt, including some KDE Frameworks. If you discover any issues we encourage you to report them. If you want more control, it’s also possible to use the org.freedesktop.Platform as a base and bundle the parts of Qt you need.

Multiarch support

Flatpak has multiarch/multilib support, but it’s not enabled by default and require some additional steps to enable it. This section covers enabling multiarch/multilib in your application bundle.

Running 32-bit programs

In order to set up the run time environment for 32-bit executables, first you’ll need to allow it in `finish-args`:

```yaml
finish-args:
  - --allow=multiarch
```

This is enough for static binaries, but most real-world GNU/Linux programs are linked dynamically. Those need some shared libraries to work.

Freedesktop.org and GNOME SDKs both provide a special flatpak extension with a set of libraries for corresponding architecture. This extension can be attached to an app of different architecture. In order to enable the extension for your app, define an extension point for it in the app manifest:
Flatpak Documentation

add-extensions:
  org.freedesktop.Platform.Compat.i386:
    directory: lib/i386-linux-gnu
    version: '21.08'

  # This is not strictly required, but needed for debugging 32-bit programs
  org.freedesktop.Platform.Compat.i386.Debug:
    directory: lib/debug/lib/i386-linux-gnu
    version: '21.08'
    no-autodownload: true


Note that this extension version must match the runtime-version of the application.

If the 32-bit programs make use of GPU acceleration, or have some graphical UI in general, you’ll also need 32-bit GL drivers. Add an extension point for it:

add-extensions:
  org.freedesktop.Platform.GL32:
    directory: lib/i386-linux-gnu/GL
    version: '1.4'
    versions: 21.08;1.4
    subdirectories: true
    no-autodownload: true
    autodelete: false
    add-ld-path: lib
    merge-dirs: vulkan/icd.d;glvnd/egl_vendor.d;OpenCL/vendors;lib/dri;lib/d3d;vulkan/
                → explicit_layer.d;vulkan/implicit_layer.d
    download-if: active-gl-driver
    enable-if: active-gl-driver

Note that the versions property here must contain both 1.4 and the same value as in runtime-version.

Make sure to create directories where the extensions will be mounted (the mount points are specified in directory properties and are relative to the app bundle mount point, i.e. to /app/). This can be done at stage of the build.

Finally, you need to make the dynamic library loader know the paths to 32-bit libraries. In order to do this, you can install a /app/etc/ld.so.conf file with contents like this:

```
/app/lib32
/app/lib/i386-linux-gnu
```

Here /app/lib32 is the directory where you install additional 32-bit libraries, if any.

You can combine the above two steps in a special module, e.g.

modules:
  - name: bundle-setup
    buildsystem: simple
    build-commands:
      - mkdir -p /app/lib/i386-linux-gnu
      - mkdir -p /app/debug/lib/i386-linux-gnu
      - mkdir -p /app/lib/i386-linux-gnu/GL
      - install -Dm644 ld.so.conf /app/etc/ld.so.conf
    sources:
      - type: file
        dest-filename: ld.so.conf
        url: data:/app/lib32%0A/app/lib/i386-linux-gnu%0A
Building 32-bit modules

The section above describes how to run 32-bit programs that are already built. This section will describe the process of building 32-bit components yourself to ship them with the application. It assumes that you are already familiar with building (single-arch) flatpaks. If not, please refer to Flatpak Builder guide first.

First of all, you'll need to enable some SDK extensions at build time:

```yaml
sdk-extensions:
- org.freedesktop.Sdk.Compat.i386
- org.freedesktop.Sdk.Extension.toolchain-i386
```

The first one is the 32-bit portion of the SDK, containing 32-bit libraries and development files.

The second one is a cross-compiler. Usually `gcc -m32` is used for multilib builds, but the flatpak SDK comes with `gcc` without multilib support. Thus, you will need a cross-compiler for building x86 on x86_64 just as you would need it for any foreign architecture like aarch64.

In order to build a 32-bit module, some global build options needs to be overridden. Examples here assume that 32-bit libraries are installed in `/app/lib32` directory:

```yaml
modules:
- name: some-lib-32bit
  build-options: &compat-i386-build-options
    # Make sure 32-bit dependencies are first on pkg-config search path
    prepend-pkg-config-path: /app/lib32/pkgconfig:/usr/lib/i386-linux-gnu/pkgconfig
    # Add /app/lib32 to linker search path for modules without pkg-config
    ldflags: -L/app/lib32
    # Add the cross-compiler to PATH
    prepend-path: /usr/lib/sdk/toolchain-i386/bin
    # Tell the build systems to use the cross-compiler for compilation
    env:
      CC: i686-unknown-linux-gnu-gcc
      CXX: i686-unknown-linux-gnu-g++
    # Tell the build systems to install libraries to /app/lib32
    libdir: /app/lib32
```

These `build-options` need to be set for each 32-bit module. If your app manifest is in YAML format, the YAML anchors can come in handy and save you from copy-pasting the same snippet. You can define the 32-bit `build-options` object somewhere in the manifest, add an anchor to it, and then point each 32-bit modules’ `build-options` to that anchor:

```yaml
x-compat-i386-build-options: &compat-i386-build-options
  prepend-pkg-config-path: /app/lib32/pkgconfig:/usr/lib/i386-linux-gnu/pkgconfig
  ldflags: -L/app/lib32
  prepend-path: /usr/lib/sdk/toolchain-i386/bin
  env:
    CC: i686-unknown-linux-gnu-gcc
    CXX: i686-unknown-linux-gnu-g++
  libdir: /app/lib32

modules:
- name: some-lib-32bit
  build-options: *compat-i386-build-options
- name: some-other-lib-32bit
  build-options: *compat-i386-build-options
```
Of course, in order to actually use 32-bit modules you’ve build, you’ll also need all the same setup from the previous section.

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:

```bash
$ 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:

```bash
$ 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.

Bemerkung: 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:

```bash
$ 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:

```bash
$ 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:

```bash
$ 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:

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

To reset all portal permissions of an application, use:

```bash
$ 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):

```bash
$ flatpak ps
```

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

```bash
$ 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 Verwenden von 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=mQINBF1D2sABEADsiUZUO...
```

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.

_Bemerkung:_ .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

Bemerkung: Flathub uses flat-manager to host its Flatpak repository. See https://github.com/flatpak/flat-manager

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=mQINBF1D2sABEADsiUZUO...
```

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:

```
→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.

Bemerkung: .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.

**Warnung:** Since single-file bundles don’t include dependencies or AppStream data, the preferred way of distributing applications offline is with the create-usb command; see *USB Drives*.

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:

```
```

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
```

1.5.4 USB Drives

One can distribute flatpaks along with their dependencies on USB drives (or network shares, etc.) which is especially helpful in situations where Internet access is limited or non-existent.

For offline distribution to work there are a few prerequisites:

- the remote repositories providing the app or any of its dependencies must utilize GPG signatures
- the remote repos must all have a collection ID set on the server side
- the locally configured remotes must have a collection ID set (on the client side)
- the relevant remotes must be configured on the receiving computer (the one installing from the drive)

Apps can then be copied to USB drives using the `flatpak create-usb` command. You can refer to this blog post for an introduction.

For example, if you want to put Gedit on a USB drive:

1. First identify the Application ID using `flatpak list --app`. In the case of Gedit it is `org.gnome.gedit`. Use `flatpak info -o org.gnome.gedit` to determine the origin remote. For example that may be `flathub`. 
2. Ensure the origin remote has a collection ID set by using `flatpak remotes -d` and checking the ”Collection ID” column. If not, configure one with e.g. `flatpak remote-modify --collection-id=org.flathub.Stable flathub`. If any dependencies come from other remotes, those will also need a collection ID configured.

3. Next, use the `df` command to identify the mount point for the USB drive. It may be something like `/media/user/FLATPAKS`.

4. Now copy the flatpak and its dependencies to the drive:

   ```bash
   $ flatpak create-usb /media/user/FLATPAKS org.gnome.gedit
   ```

5. Wait for the copying process to complete, at which point you should get a command prompt (`$`). This process can take tens of minutes especially if the USB drive and USB port aren’t USB 3.0+. Then unmount the drive before removing it:

   ```bash
   $ umount /media/user/FLATPAKS
   ```

   The process for installing from such a USB drive (for example on an offline machine) differs between Flatpak versions before 1.8.0 and those after. With earlier versions you can simply use the `flatpak install` command as you normally would online:

   ```bash
   $ flatpak install flathub org.gnome.gedit
   ```

   For versions after 1.8.0, if your Linux distribution has packaged the relevant `systemd` units, using `flatpak install` with no extra arguments still works. Otherwise, you can use the `--sideload-repo` option in your command invocation:

   ```bash
   $ flatpak install --sideload-repo=/media/user/FLATPAKS/.ostree/repo flathub org.gnome.gedit
   ```

   The `flatpak update` command also accepts a `--sideload-repo` option.

   Alternatively, it’s possible to specify sideload sources using symbolic links placed in system-wide or user-specific directories and such sources will then be used for all Flatpak operations without the need for a `--sideload-repo` option. See the `flatpak man` page.

### 1.6 Desktop Integration

**Anforderungen & Konventionen** 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:

**Portal support in GTK**

GTK will transparently use portals for some functionality when it detects that it is being used inside a Flatpak sandbox. Here are some hints for what GTK applications should do to benefit from this.

- Use `g_get_user_config_dir()`, `g_get_user_cache_dir()` and `g_get_user_data_dir()` to find the right place to store configuration and data
- Use `GtkFileChooserNative` (or `GtkFileChooserButton`) to open files. As of `xdg-desktop-portal-gtk` 1.7.1 it can also open directories.
- Use `GtkPrintOperation` for printing
- Use `gtk_show_uri_on_window()` or `g_app_info_launch_default_for_uri()` to open URIs
- Use `gtk_application_inhibit()` if you want to inhibit idle or logout
- Use `g_application_send_notification()` to show notifications
- Use the `GtkApplication::screensaver-active` property to monitorscreensaver status

**Portal support in Qt and KDE**

Qt and KDE libraries will transparently use portals for some functionality when they detect that they are being used inside a Flatpak sandbox. Here are some hints for what Qt or KDE applications should do to benefit from this.

- Use `QDesktopServices::openUrl(const QUrl &url)` or `KIO::OpenUrlJob` to open URIs or send an email when using `mailto URL`
- Use `QFileDialog` class to open files and, as of Qt 5.18.90, directories. Avoid using `QFileDialog::DontUseNativeDialog`. Note that portals cannot currently give access to directories on the host filesystem
- Use `KNotification::notify()` to show notifications

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).

On X11 applications can rely on system-provided titlebars if they don’t want to draw their own window controls. For a consistent Wayland experience applications must always provide their own. Typically toolkits handle this but raw wayland clients can use libdecor for a general solution.

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 Gtk.Application:menubar or QMenuBar it will work as expected from within a sandboxed application.
1.6.9 Theming

Flatpak applications cannot directly use the system theme. This happens because flatpaks do not have the ability to use data files or libraries in /usr (where system themes are typically located). The solution to this was to package themes as Flatpaks, as relying upon the host to have the correct version for every flatpak defeats the portability benefits it provides. These themes are provided as extensions, to the Freedesktop runtime when the extension point is Gtk, and to the KDE runtime when the extension point is Qt.

The theming system requires Flatpak 0.8.4+ and applications using up to date org.gnome.Platform 3.24+, or org.freedesktop.Platform 1.6+, or org.kde.Platform 5.9+.

Installing themes

Instructions for Gtk

The current Gtk themes are packaged in the flathub repository which you can add (if it’s not already added) by running:

```
$ flatpak remote-add flathub https://flathub.org/repo/flathub.flatpakrepo
```

To see a list of currently packaged themes you can use the command flatpak search gtk3theme (available since Flatpak version 0.10.1). In case you use an older version of Flatpak than that, you can use the command flatpak remote-ls flathub | grep org.gtk.Gtk3theme. The difference in output between these two commands is that the first prints the application ID, the remote from which the theme comes and the theme’s description, while the second prints only the full name of the theme’s flatpak package.

You can install themes with the command flatpak install flathub org.gtk.Gtk3theme.Foo, replacing Foo with the name of the desired theme.

Instructions for Qt

For the Qt theming to work, the flatpak packages kstyle and platformtheme must be installed. These are packed in the kdeapps repository which you can add by running:

```
$ flatpak remote-add kdeapps https://distribute.kde.org/kdeapps.flatpakrepo
```

Afterwards the two packages can be installed with the following commands:

```
$ flatpak install kdeapps org.kde.KStyle.Adwaita//5.9
$ flatpak install kdeapps org.kde.PlatformTheme.QGnomePlatform//5.9
```

Applying themes

There is no ideal way to specify the theme Flatpak applications use. The applications will try to match the system theme currently being used, if it corresponds to any of the Flatpak themes installed, and will fall back to Adwaita (if they use Gtk2 or Gtk3) or the default Qt theme (if they use Qt) if a corresponding theme isn’t detected.

As of Flatpak 0.10.1, the Flatpak system can detect whether the system themes available correspond to any Flatpak themes installed, and, if so, will automatically install found themes at update time based upon the gtk-theme Dconf key. This key however can contain only one value, the one of the currently being used theme, which means that the Flatpak versions of matching themes that aren’t currently being used aren’t installed until those themes are enabled. If none of the corresponding system themes are currently being used, the applications will fall back to Adwaita or the default Qt theme.
On X11, Gtk3 picks up the themes via XSettings. Specifically, the GNOME XSettings daemon `gsd-xsettings` reads the DConf values and converts them into the XSettings values. For this to work, you need an xsettings daemon that is correctly configured. Gtk3 on Wayland picks up themes directly via Dconf. For this to work, you can either use KDE (with `kde-gtk-config` > 5.11.95), GNOME, which works out of the box, or manually configure the dconf keys under `/org/gnome/desktop/interface/`. For the DConf option to work on Wayland the application must also be configured to have DConf access.

**Other notes on theming**

In regards to icon themes, since Flatpak 0.8.8 the host icons are exposed to the guest, so that there is usually no need for the presence of Flatpak icon themes.

If you use the *Global Dark Theme* option (removed in GNOME-Tweaks 3.28) in `gnome-tweak-tool` it will not work as that simply writes to `settings.ini` which isn’t available in the sandbox. Use dark versions of themes instead if they exist.

### 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:

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

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

```bash
$ 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`.

**Warnung:** 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:

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

Then you deploy the commit:

```bash
$ sudo flatpak update \
--commit=ec07ad6c54e803d1428e5580426a41315e50a14376af033458e7a65bfb2b64f0 \
org.gnome.Recipes
```
**Bemerkung:** The example here uses `sudo` for system installations because, unlike normal updates, downgrades are considered a privileged action. If the application is installed per-user you would run it as that user.

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/main/scripts/flatpak-bisect](https://github.com/flatpak/flatpak/blob/main/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
   → 5cd2b0648618c9038fbc6830733817309ade29541cdd8383830b6f6accf0d
```

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.

### 1.7.4 Adding a custom installation

By default Flatpak installs apps system-wide, and can also be made to install per-user with the `--user` option accepted by most commands. A third option is to set up a custom installation, which could be stored on an external hard drive.

First ensure that the config directory exists:
$ sudo mkdir -p /etc/flatpak/installations.d

Then open a file in that directory as root:

$ sudo edit /etc/flatpak/installations.d/extra.conf

And write something like this:

```ini
[Installation "extra"]
Path=/run/media/mwleeds/ext4_4tb/flatpak/
DisplayName=Extra Installation
StorageType=harddisk
```

See flatpak-installation(5) for the full format specification. Replace the path with the actual path you want to use. You can use df to see mounted file systems and mkdir to create a flatpak directory so the path specified by Path= exists.

Then you can add a remote using a command like:

```
   →flatpakrepo
```

And install to it with:

```
$ flatpak --installation=extra install flathub org.inkscape.Inkscape
```

**Bemerkung:** If your custom installation is the only one with the remote you're installing from, --installation can be omitted.

And run apps from it with:

```
$ flatpak --installation=extra run org.inkscape.Inkscape
```

**Bemerkung:** If your custom installation is the only one with the app you're running, --installation can be omitted.

### 1.7.5 Configuring resource limits for apps

When systemd is available, Flatpak tries to put app processes in a scope such as `app-com.brave.Browser-*.scope` (in the case of Brave), with * replaced by an arbitrary suffix. This means you can create a file like `~/.config/systemd/user/app-flatpak-com.brave.Browser-.scope.d/memory.conf` with contents like:

```ini
[Scope]
MemoryHigh=1G
```

Then after a systemctl --user daemon-reload, those systemd.resource-control(5) parameters will apply to all instances of that app.
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. Each runtime comes with the corresponding SDK for building, and extensions for specific uses.

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//21.08`) 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.

The Freedesktop runtime is maintained here and has a website here.

Available Freedesktop runtimes:

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.freedesktop.Platform</td>
<td>Runtime</td>
</tr>
<tr>
<td>org.freedesktop.Sdk</td>
<td>SDK</td>
</tr>
</tbody>
</table>

The following runtime extensions are available:

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.freedesktop.Platform.Locale</td>
<td>Runtime translations (extension)</td>
</tr>
<tr>
<td>org.freedesktop.Platform.VAAPI.Intel{,.i386}</td>
<td>Intel vaapi drivers (extension)</td>
</tr>
<tr>
<td>org.freedesktop.Platform.ffmpeg-full</td>
<td>All ffmpeg codecs (extension)</td>
</tr>
<tr>
<td>org.freedesktop.Platform.Compat.{architecture}</td>
<td>32 bits compatible extension</td>
</tr>
<tr>
<td>org.freedesktop.Platform.Compat.{architecture}.debug</td>
<td>32 bits compatible extension (debug)</td>
</tr>
<tr>
<td>org.freedesktop.Platform.GL{,.32}.default</td>
<td>Mesa drivers (extension)</td>
</tr>
<tr>
<td>org.freedesktop.Platform.GL{,.32}.mesa-git</td>
<td>Mesa drivers, latest (extension)</td>
</tr>
<tr>
<td>org.freedesktop.Sdk.Debug</td>
<td>SDK debug information (extension)</td>
</tr>
<tr>
<td>org.freedesktop.Sdk.Locale</td>
<td>SDK translations (extension)</td>
</tr>
<tr>
<td>org.freedesktop.Sdk.Docs</td>
<td>SDK documentation (extension)</td>
</tr>
<tr>
<td>org.freedesktop.Sdk.Extension.toolchain-{architecture}</td>
<td>SDK cross compilers (extension)</td>
</tr>
</tbody>
</table>
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

The GNOME runtime is maintained here.

Available GNOME runtimes:

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.gnome.Platform</td>
<td>Runtime</td>
</tr>
<tr>
<td>org.gnome.Sdk</td>
<td>SDK</td>
</tr>
</tbody>
</table>

The following runtime extensions are available:

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.gnome.Platform.Locale</td>
<td>Runtime translations (extension)</td>
</tr>
<tr>
<td>org.gnome.Sdk.Debug</td>
<td>SDK debug information (extension)</td>
</tr>
<tr>
<td>org.gnome.Sdk.Locale</td>
<td>SDK translations (extension)</td>
</tr>
<tr>
<td>org.gnome.Sdk.Docs</td>
<td>SDK documentation (extension)</td>
</tr>
</tbody>
</table>

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.

The KDE runtime is maintained here.

Available KDE runtimes:

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.kde.Platform</td>
<td>Runtime</td>
</tr>
<tr>
<td>org.kde.Sdk</td>
<td>SDK</td>
</tr>
</tbody>
</table>

The following runtime extensions are available:
elementary

The elementary runtime is appropriate for any application that would like to publish in elementary AppCenter. It is based on the GNOME runtime and adds the elementary platform, including:

- elementary Icons
- elementary Stylesheet
- elementary Sound Theme
- Granite

The elementary runtime is maintained here.

Available elementary runtimes:

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>io.elementary.Platform</td>
<td>Runtime</td>
</tr>
<tr>
<td>io.elementary.Sdk</td>
<td>SDK</td>
</tr>
</tbody>
</table>

The following runtime extensions are available:

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>io.elementary.Platform.Locale</td>
<td>Runtime translations (extension)</td>
</tr>
<tr>
<td>io.elementary.Sdk.Debug</td>
<td>SDK debug information (extension)</td>
</tr>
<tr>
<td>io.elementary.Sdk.Locale</td>
<td>SDK translations (extension)</td>
</tr>
<tr>
<td>io.elementary.Sdk.Docs</td>
<td>SDK documentation (extension)</td>
</tr>
</tbody>
</table>

1.8.4 Sandbox Permissions Reference

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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--socket=x11</td>
<td>Show windows using X11</td>
</tr>
<tr>
<td>--share=ipc</td>
<td>Share IPC namespace with the host’</td>
</tr>
<tr>
<td>--allow=bluetooth</td>
<td>Allow access to Bluetooth</td>
</tr>
<tr>
<td>--device=dri</td>
<td>OpenGL rendering</td>
</tr>
<tr>
<td>--socket=wayland</td>
<td>Show windows using Wayland</td>
</tr>
<tr>
<td>--socket=pulseaudio</td>
<td>Play sounds using PulseAudio</td>
</tr>
<tr>
<td>--share=network</td>
<td>Access the network’</td>
</tr>
<tr>
<td>--talk-name=org.freedesktop.secrets</td>
<td>Talk to a named service on the session bus</td>
</tr>
<tr>
<td>--system-talk-name=org.freedesktop.GeoClue2</td>
<td>Talk to a named service on the system bus</td>
</tr>
<tr>
<td>--socket=system-bus</td>
<td>Unlimited access to all of D-Bus</td>
</tr>
</tbody>
</table>
**Filesystem permissions**

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

<table>
<thead>
<tr>
<th>Permission</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>Access all files</td>
</tr>
<tr>
<td>home</td>
<td>Access the home directory</td>
</tr>
<tr>
<td>/some/dir</td>
<td>Access an arbitrary path</td>
</tr>
<tr>
<td>~/some/dir</td>
<td>Access an arbitrary path relative to the home directory</td>
</tr>
<tr>
<td>xdg-desktop</td>
<td>Access the XDG desktop directory</td>
</tr>
<tr>
<td>xdg-documents</td>
<td>Access the XDG documents directory</td>
</tr>
<tr>
<td>xdg-download</td>
<td>Access the XDG download directory</td>
</tr>
<tr>
<td>xdg-music</td>
<td>Access the XDG music directory</td>
</tr>
<tr>
<td>xdg-pictures</td>
<td>Access the XDG pictures directory</td>
</tr>
<tr>
<td>xdg-public-share</td>
<td>Access the XDG public directory</td>
</tr>
<tr>
<td>xdg-videos</td>
<td>Access the XDG videos directory</td>
</tr>
<tr>
<td>xdg-templates</td>
<td>Access the XDG templates directory</td>
</tr>
<tr>
<td>xdg-config</td>
<td>Access the XDG config directory</td>
</tr>
<tr>
<td>xdg-cache</td>
<td>Access the XDG cache directory</td>
</tr>
<tr>
<td>xdg-data</td>
<td>Access the XDG data directory</td>
</tr>
<tr>
<td>xdg-run/path</td>
<td>Access subdirectories of the XDG runtime directory (where path is any subdirectory)</td>
</tr>
</tbody>
</table>

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

**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:

```
```

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=gnome-dictionary --database=dictionary.db
Type=Application
Icon=org.gnome.Dictionary
```

Your desktop file should be prefixed with your application’s appid and placed in `/app/share/applications/` within your app.

Example:

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

It’s recommended to use `desktop-file-validate` to check your file for errors before including it.

A special note about the `Exec` line: When installing an app, Flatpak will automatically rewrite the included .desktop file so that the app will be started through Flatpak. The rewritten desktop file is then exported to a path such as `exports/share/applications/org.gnome.Dictionary.desktop` under your Flatpak installation directory (usually `~/var/lib/flatpak/` or `~/.local/share/flatpak/`). In the case of `org.gnome.Dictionary.desktop`, the rewritten `Exec` line looks like this:

```
Exec=/usr/bin/flatpak run --branch=stable --arch=x86_64 --command=gnome-dictionary --database=dictionary.db
```

The command from the original desktop file will be part of the `--command` argument to Flatpak and arguments will be passed through. This means that in most cases, it should match the value of the `command` line in your app’s manifest.

If you want the `--command` argument to be omitted from the `flatpak run` command in the generated desktop file, you can leave the `Exec` value in the source desktop file empty:

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

This way, the generated `Exec` line looks like this:

```
Exec=/usr/bin/flatpak run --branch=stable --arch=x86_64 org.gnome.Dictionary
```

**Bemerkung:** With Flatpak 1.12.7, a warning may be shown when exporting a build with an empty `Exec=` line to a repository:

```
(flatpak build-export:189863): GLib-CRITICAL **: 22:15:27.398: g_path_is_absolute: assertion 'file_name != NULL' failed
```

This warning can be ignored.

You can find more general information about desktop files [here](#). If interested, you can read also the full spec [here](#).
**Appdata files**

Appdata files are used by application stores (e.g. KDE Discover, GNOME Software) in order to display metadata about your application, such as a description, screenshots, changelogs when updates are available, and other miscellaneous things.

Your Appdata 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:
  - 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 OSTree system for versioning and distributing filesystem trees
  • The OCI format from the Open Container Initiative, as an alternative to OSTree used by the Fedora infrastructure
  • Flatpak can use either OSTree or OCI for single-file bundles.
  • 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