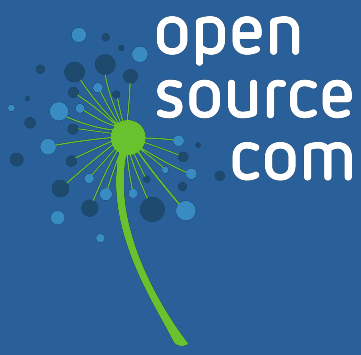


Installing applications on Linux

by Seth Kenlon, Chris Hermansen, Patrick H. Mullins

We are Opensource.com

Opensource.com is a community website publishing stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Do you have an open source story to tell? Submit a story idea at opensource.com/story

Email us at open@opensource.com

Supported by
Red Hat

Table of Contents

We are Opensource.com....1
5 reasons to use Linux package managers....4
How to install software applications on Linux....8
How to install software from the Linux command line....20
Linux package management with dnf....23
Linux package management with apt....29
Run your favorite Windows applications on Linux....34
Anyone can compile open source code in these three simple steps....38
Why programmers love Linux packaging....46
Install apps on Linux with Flatpak....49
How to build a Flatpak....53

Seth Kenlon

Seth Kenlon is a UNIX geek, free culture advocate, independent multimedia artist, and D&D nerd. He has worked in the film and computing industry, often at the same time. He is one of the maintainers of the Slackware-based multimedia production project Slackermedia.

5 reasons to use Linux package managers

Before I used Linux, I took the applications I had installed on my computer for granted. I would install applications as needed, and if I didn't end up using them, I'd forget about them, letting them languish as they took up space on my hard drive. Eventually, space on my drive would become scarce, and I'd end up frantically removing applications to make room for more important data. Inevitably, though, the applications would only free up so much space, and so I'd turn my attention to all of the other bits and pieces that got installed along with those apps, whether it was media assets or configuration files and documentation. It wasn't a great way to manage my computer. I knew that, but it didn't occur to me to imagine an alternative, because as they say, you don't know what you don't know.

When I switched to Linux, I found that installing applications worked a little differently. On Linux, you were encouraged not to go out to websites for an application installer. Instead, you ran a command, and the application was installed on the system, with every individual file, library, configuration file, documentation, and asset recorded.

What is a software repository?

The default method of installing applications on Linux is from a distribution software repository. That might sound like an app store, and that's because modern app stores have borrowed much from the concept of software repositories. Linux has app stores, too, but software repositories are unique. You get an application from a software repository through a *package manager*, which enables your Linux system to record and track every component of what you've installed.

Here are five reasons that knowing exactly what's on your system can be surprisingly useful.

1. Removing old applications

When your computer knows every file that was installed with any given application, it's really easy to uninstall files you no longer need. On Linux, there's no problem with installing 31 different text editors only to later uninstall the 30 you don't love. When you uninstall on Linux, you really uninstall.

2. Reinstall like you mean it

Not only is an uninstall thorough, a *reinstall* is meaningful. On many platforms, should something go wrong with an application, you're sometimes advised to reinstall it. Usually, nobody can say why you should reinstall an application. Still, there's often the vague suspicion that some file somewhere has become corrupt (in other words, data got written incorrectly), and so the hope is that a reinstall might overwrite the bad files and make things work again. It's not bad advice, but it's frustrating for any technician not to know what's gone wrong. Worse still, there's no guarantee, without careful tracking, that all files will be refreshed during a reinstall because there's often no way of knowing that all the files installed with an application were removed in the first place. With a package manager, you can force a complete removal of old files to ensure a fresh installation of new files. Just as significantly, you can account for every file and probably find out which one is causing problems, but that's a feature of open source and Linux rather than package management.

3. Keep your applications updated

Don't let anybody tell you that Linux is "more secure" than other operating systems. Computers are made of code, and we humans find ways to exploit that code in new and interesting ways every day. Because the vast majority of applications on Linux are open source, many exploits are filed publically as Common Vulnerability and Exposures (CVE). A flood of incoming security bug reports may seem like a bad thing, but this is definitely a case when *knowing* is far better than *not knowing*. After all, just because nobody's told you that there's a problem doesn't mean that there's not a problem. Bug reports are good. They benefit everyone. And when developers fix security bugs, it's important for you to be able to get those fixes promptly, and preferably without having to remember to do it yourself.

A package manager is designed to do exactly that. When applications receive updates, whether it's to patch a potential security problem or introduce an exciting new feature, your package manager application alerts you of the available update.

4. Keep it light

Say you have application A and application B, both of which require library C. On some operating systems, by getting A and B, you get two copies of C. That's obviously redundant, so imagine it happening several times per application. Redundant libraries add up quickly, and by having no single source of "truth" for a given library, it's nearly impossible to ensure you're using the most up-to-date or even just a consistent version of it.

I admit I don't tend to sit around pondering software libraries all day, but I do remember the days when I did, even though I didn't know that's what was troubling me. Before I had switched to Linux, it wasn't uncommon for me to encounter errors when dealing with media files for work, or glitches when playing different video games, or quirks when reading a PDF, and so on. I spent a lot of time investigating these errors back then. I still remember learning that two major applications on my system each had bundled the same (but different) graphic backend technologies. The mismatch was causing errors when the output of one was imported into the other. It was meant to work, but because of a bug in an older version of the same collection of library files, a hotfix for one application didn't benefit the other.

A package manager knows what backends (referred to as a *dependency*) are needed for each application and refrains from reinstalling software that's already on your system.

5. Keep it simple

As a Linux user, I appreciate a good package manager because it helps make my life simple. I don't have to think about the software I install, what I need to update, or whether something's really been uninstalled when I'm finished with it. I audition software without hesitation. And when I'm setting up a new computer, I run a simple Ansible script to automate the installation of the latest versions of all the software I rely upon. It's simple, smart, and uniquely liberating.

Better package management

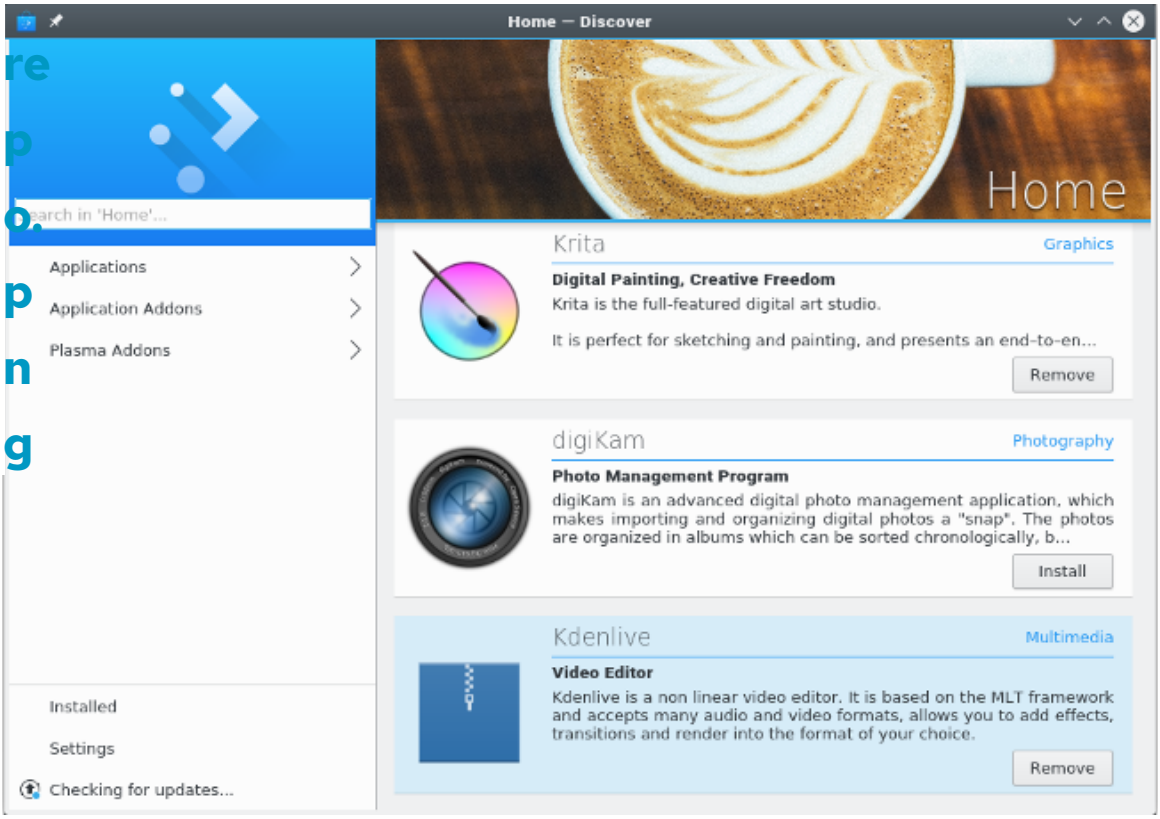
Linux takes a holistic view of applications and the operating system. After all, open source is built upon the work of other open source, so distribution maintainers understand the concept of a dependency *stack*. Package management on Linux has an awareness of your whole system, the libraries and support files on it, and the applications you install. These disparate parts work together to provide you with an efficient, optimized, and robust set of applications.

How to install software applications on Linux

How do you install an application on Linux? As with many operating systems, there isn't just one answer to that question. Applications can come from so many sources—it's nearly impossible to count—and each development team may deliver their software whatever way they feel is best. Knowing how to install what you're given is part of being a true power user of your operating system.

Repositories

For well over a decade, Linux has used software repositories to distribute software. A "repository" in this context is a public server hosting installable software packages. A Linux distribution provides a command, and usually a graphical interface to that command, that pulls the software from the server and installs it onto your computer. It's such a simple concept that it has served as the model for all major cellphone operating systems and, more recently, the "app stores" of the two major closed source computer operating systems.



Not an app store

Installing from a software repository is the primary method of installing apps on Linux. It should be the first place you look for any application you intend to install. To install from a software repository, there's usually a command:

```
$ sudo dnf install inkscape
```

The actual command you use depends on what distribution of Linux you use. Fedora uses **`dnf`**, OpenSUSE uses **`zypper`**, Debian and Ubuntu use **`apt`**, Slackware uses **`sbopkg`**, NetBSD uses **`pkgin`**, and Illumos-based OpenIndiana uses **`pkg`**. Whatever you use, the incantation usually involves searching for the proper name of what you want to install, because sometimes what you call software is not its official or solitary designation:

```
$ sudo dnf search pyqt
PyQt.x86_64 : Python bindings for Qt3
PyQt4.x86_64 : Python bindings for Qt4
python-qt5.x86_64 : PyQt5 is Python bindings for Qt5
```

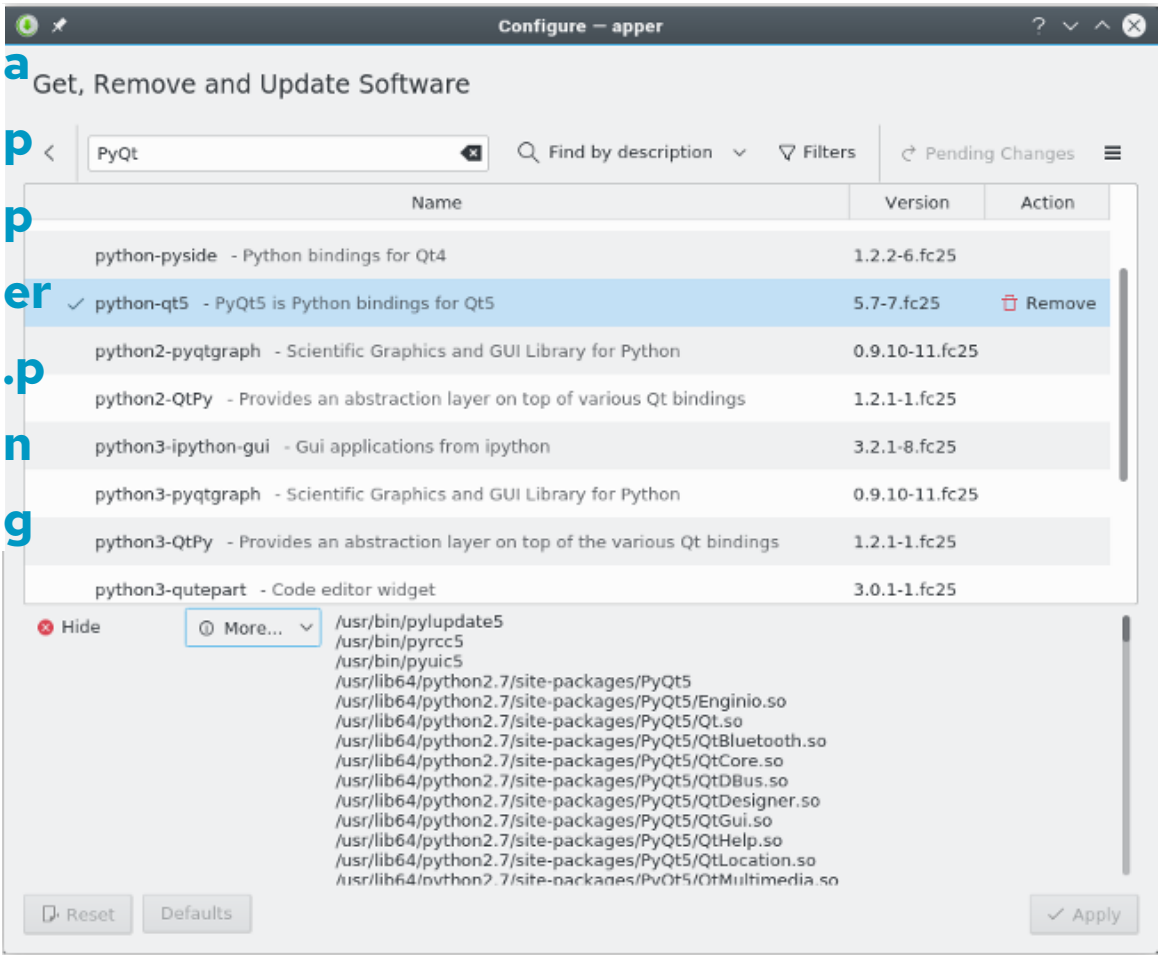
Once you have located the name of the package you want to install, use the **`install`** subcommand to perform the actual download and automated install:

```
$ sudo dnf install python-qt5
```

For specifics on installing from a software repository, see your distribution's documentation.

Installing apps with an app

The same generally holds true with the graphical tools. Search for what you think you want, and then install it.



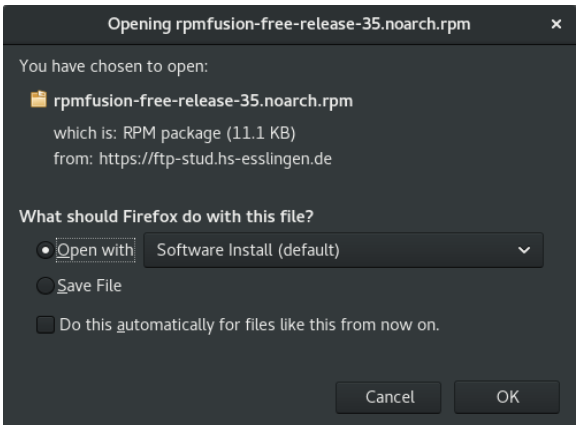
Apper

Like the underlying command, the name of the graphical installer depends on what distribution you are running. The relevant application is usually tagged with the software or package keywords, so search your launcher or menu for those terms, and you'll find what you need. Since open source is all about user choice, if you don't like the graphical user interface (GUI) that your distribution provides, there may be an alternative that you can install. And now you know how to do that.

Extra repositories

Your distribution has a standard repository for software that it packages for you, and there are usually extra repositories common to your distribution. For example, EPEL serves Red Hat Enterprise Linux and CentOS, RPMFusion serves Fedora, Ubuntu has various levels of support as well as a Personal Package Archive (PPA) network, Packman provides extra software for OpenSUSE, and SlackBuilds.org provides community build scripts for Slackware.

By default, your Linux OS is set to look at just its official repositories, so if you want to use additional software collections, you must add extra repositories yourself. You can usually install a repository as though it were a software package. In fact, when you install certain software, such as GNU Ring video chat application, the Google Chrome browser, and many others, what you are actually installing is access to their private repositories, from which the latest version of their application is installed to your machine.



You can also add the repository manually by editing a text file and adding it to your package manager's configuration directory, or by running a command to install the repository. As usual, the exact command you use depends on the distribution you are running; for example, here is a **dnf** command that adds a repository to the system:

```
$ sudo dnf config-manager --add-repo=http://example.com/pub/centos/7
```

Installing apps without repositories

The repository model is so popular because it provides a link between the user (you) and the developer. When important updates are released, your system kindly prompts you to accept the updates, and you can accept them all from one centralized location.

Sometimes, though, there are times when a package is made available with no repository attached. These installable packages come in several forms.

Linux packages

Sometimes, a developer distributes software in a common Linux packaging format, such as RPM, DEB, or the newer but very popular Flatpak or Snap formats. You may not get access to a repository with this download; you might just get the package.

When this happens, you download a `.deb` file (for **apt** users) and an `.rpm` file (for **dnf** users). When you want to update, you return to the website and download the latest appropriate file.

These one-off packages can be installed with all the same tools used when installing from a repository. If you double-click the package you download, a graphical installer launches and steps you through the install process.

Alternately, you can install from a terminal. The difference here is that a lone package file you've downloaded from the internet isn't coming from a repository. It's a "local" install, meaning your package management software doesn't need to download it to install it. Most package managers handle this transparently:

```
$ sudo dnf install ~/Downloads/example-14.0.0-amd64.rpm
```

In some cases, you need to take additional steps to get the application to run, so carefully read the documentation about the software you're installing.

Generic install scripts

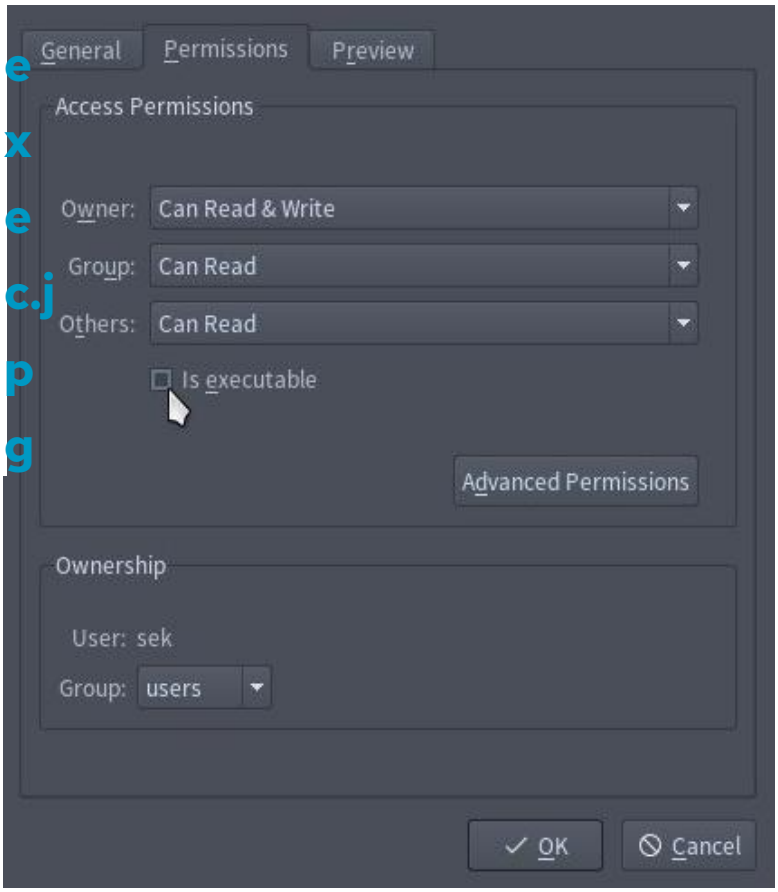
Some developers release their packages in one of several generic formats. Common extensions include `.run` and `.sh`. NVIDIA graphic card drivers, Foundry visual FX packages like Nuke and Mari, and many DRM-free games from GOG use this style of installer.

This model of installation relies on the developer to deliver an installation "wizard." Some of the installers are graphical, while others just run in a terminal. There are two ways to run these types of installers.

1. You can run the installer directly from a terminal:

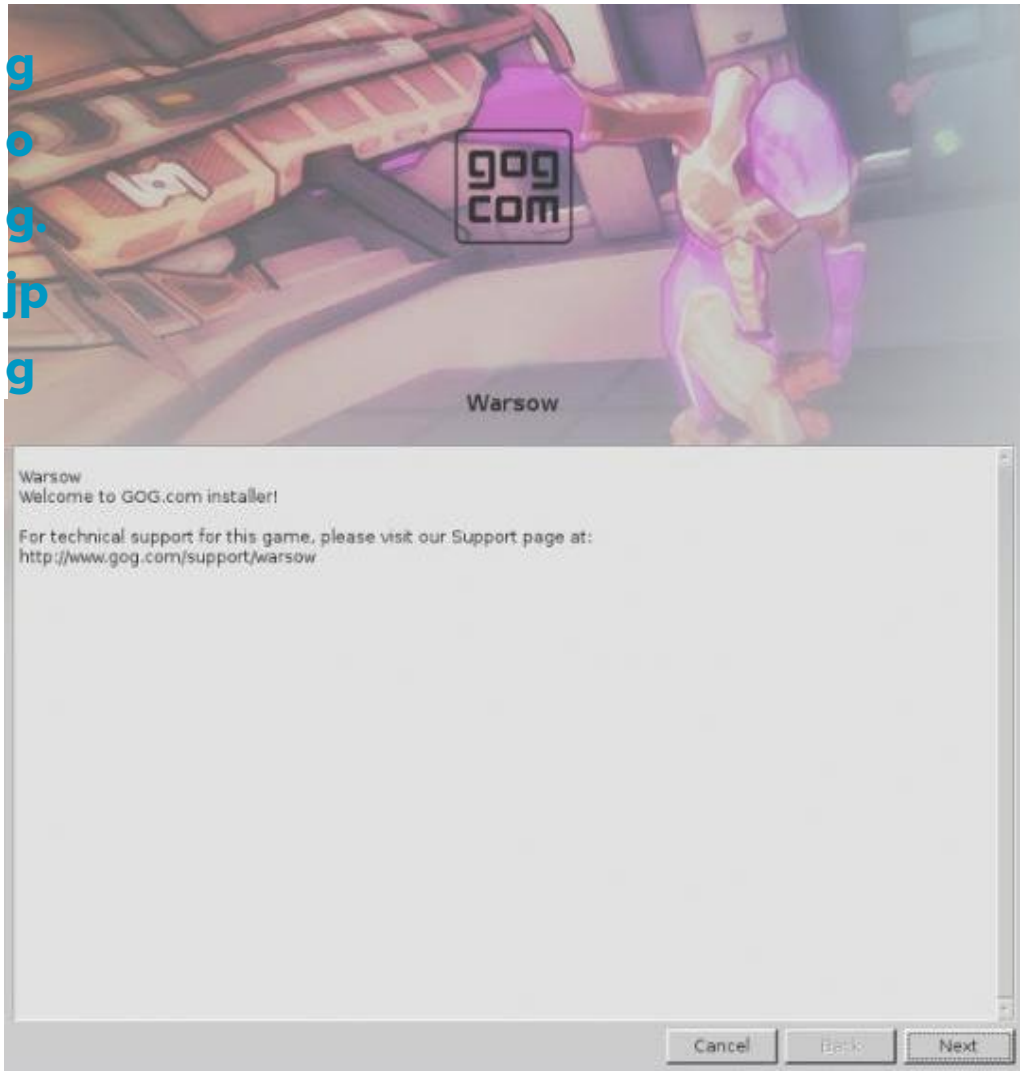
```
$ sh ./game/gog_warsow_x.y.z.sh
```

2. Alternately, you can run it from your desktop by marking it as executable. To mark an installer executable, right-click on its icon and select **Properties**.



Giving an installer executable permission

Once you've given permission for it to run, double-click the icon to start the install.



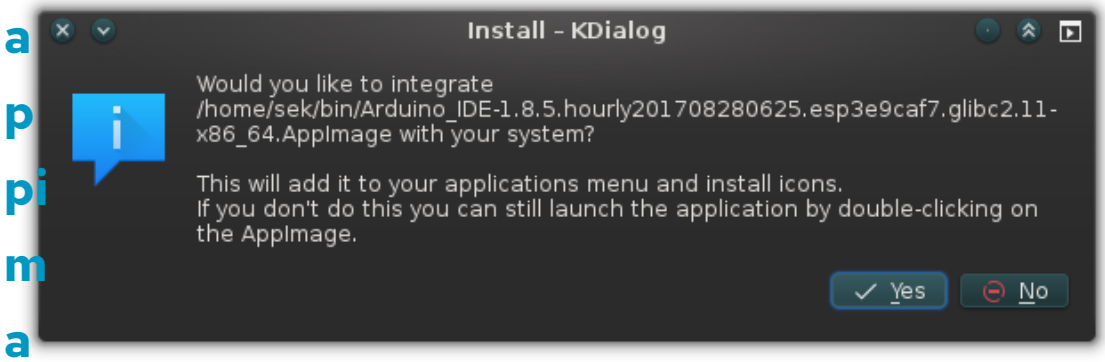
A common game installer

For the rest of the install, just follow the instructions on the screen.

AppImage portable apps

The AppImage format is relatively new to Linux, although its concept is based on both NeXT and Rox. The idea is simple: everything required to run an application is placed into one directory, and then that directory is treated as an "app." To run the application, you just double-click the icon, and it runs. There's no need or expectation that the application is installed in the traditional sense; it just runs from wherever you have it lying around on your hard drive.

Despite its ability to run as a self-contained app, an AppImage usually offers to do some soft system integration.

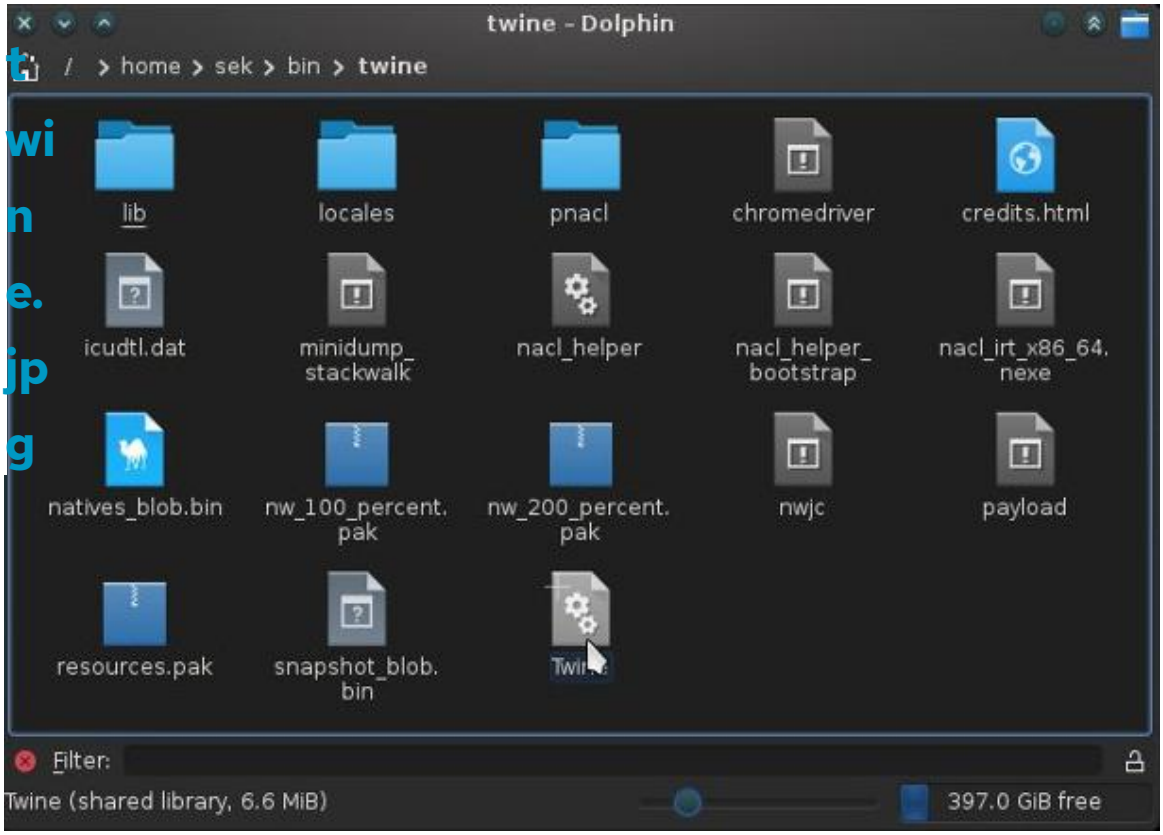


AppImage system integration

If you accept this offer, a local **`.desktop`** file is installed to your home directory. A **`.desktop`** file is a small configuration file used by the Applications menu and mimetype system of a Linux desktop. Essentially, placing the desktop config file in your home directory's application list "installs" the application without actually installing it. You get all the benefits of having installed something, and the benefits of being able to run something locally, as a "portable app."

Application directory

Sometimes, a developer just compiles an application and posts the result as a download, with no install script and no packaging. Usually, this means that you download a TAR file, extract it, and then double-click the executable file (it's usually the one with the name of the software you downloaded).



Twine downloaded for Linux

When presented with this style of software delivery, you can either leave it where you downloaded it and launch it manually when you need it, or you can do a quick and dirty install yourself. This involves two simple steps:

1. Save the directory to a standard location and launch it manually when you need it.
2. Save the directory to a standard location and create a **`.desktop`** file to integrate it into your system.

If you're just installing applications for yourself, it's traditional to keep a **`bin`** directory (short for "binary") in your home directory as a storage location for locally installed applications and scripts. If you have other users on your system who need access to the applications, it's traditional to place the binaries in **`/opt`**. Ultimately, it's up to you where you store the application.

Downloads often come in directories with versioned names, such as **`twine_2.13`** or **`pcgen-v6.07.04`**. Since it's reasonable to assume you'll update the application at some point, it's a

good idea to either remove the version number or to create a symlink to the directory. This way, the launcher that you create for the application can remain the same, even though you update the application itself.

To create a **`.desktop`** launcher file, open a text editor and create a file called **`twine.desktop`**. The Desktop Entry Specification is defined by FreeDesktop.org. Here is a simple launcher for a game development IDE called Twine, installed to the system-wide **`/opt`** directory:

```
[Desktop Entry]
Encoding=UTF-8
Name=Twine
GenericName=Twine
Comment=Twine
Exec=/opt/twine/Twine
Icon=/usr/share/icons/oxygen/64x64/categories/applications-games.png
Terminal=false
Type=Application
Categories=Development;IDE;
```

The tricky line is the **`Exec`** line. It must contain a valid command to start the application. Usually, it's just the full path to the thing you downloaded, but in some cases, it's something more complex. For example, a Java application might need to be launched as an argument to Java itself:

```
Exec=java -jar /path/to/foo.jar
```

Sometimes, a project includes a wrapper script that you can run so you don't have to figure out the right command:

```
Exec=/opt/foo/foo-launcher.sh
```

In the Twine example, there's no icon bundled with the download, so the example **`.desktop`** file assigns a generic gaming icon that shipped with the KDE desktop. You can use workarounds like that, but if you're more artistic, you can just create your own icon, or you can search the Internet for a good icon. As long as the **`Icon`** line points to a valid PNG or SVG file, your application will inherit the icon.

The example script also sets the application category primarily to Development, so in KDE, GNOME, and most other Application menus, Twine appears under the Development category.

To get this example to appear in an Application menu, place the **`twine.desktop`** file into one of two places:

- Place it in **`~/.local/share/applications`** if you're storing the application in your own home directory.
- Place it in **`/usr/share/applications`** if you're storing the application in **`/opt`** or another system-wide location and want it to appear in all your users' Application menus.

And now the application is installed as it needs to be and integrated with the rest of your system.

Compiling from source

Finally, there's the truly universal install format: source code. Compiling an application from source code is a great way to learn how applications are structured, how they interact with your system, and how they can be customized. It's by no means a push-button process, though. It requires a build environment, it usually involves installing dependency libraries and header files, and sometimes a little bit of debugging.

To learn more about compiling from source code, read my article on the topic.

Now you know

Some people think installing software is a magical process that only developers understand, or they think it "activates" an application, as if the binary executable file isn't valid until it has been "installed." Hopefully, learning about the many different methods of installing has shown you that install is really just shorthand for "copying files from one place to the appropriate places on your system." There's nothing mysterious about it. As long as you approach each install without expectations of how it's supposed to happen, and instead look for what the developer has set up as the install process, it's generally easy, even if it is different from what you're used to.

The important thing is that an installer is honest with you. If you come across an installer that attempts to install additional software without your consent (or maybe it asks for consent, but in a confusing or misleading way), or that attempts to run checks on your system for no apparent reason, then don't continue an install.

Good software is flexible, honest, and open. And now you know how to get good software onto your computer.

How to install software from the Linux command line

Contributed by Patrick H. Mullins

If you use Linux for any amount of time, you'll soon learn there are many different ways to do the same thing. This includes installing applications on a Linux machine via the command line. I have been a Linux user for roughly 25 years, and time and time again I find myself going back to the command line to install my apps.

The most common method of installing apps from the command line is through software repositories (a place where software is stored) using what's called a package manager. All Linux apps are distributed as packages, which are nothing more than files associated with a package management system. Every Linux distribution comes with a package management system, but they are not all the same.

What is a package management system?

A package management system is comprised of sets of tools and file formats that are used together to install, update, and uninstall Linux apps. The two most common package management systems are from Red Hat and Debian. Red Hat, CentOS, and Fedora all use the **`rpm`** system (.rpm files), while Debian, Ubuntu, Mint, and Ubuntu use **`dpkg`** (.deb files). Gentoo Linux uses a system called Portage, and Arch Linux uses nothing but tarballs (.tar files). The primary difference between these systems is how they install and maintain apps.

You might be wondering what's inside an **`.rpm`**, **`.deb`**, or **`.tar`** file. You might be surprised to learn that all are nothing more than plain old archive files (like **`.zip`**) that contain an application's code, instructions on how to install it, dependencies (what other apps it may depend on), and where its configuration files should be placed. The software that reads and executes all of those instructions is called a package manager.

Debian, Ubuntu, Mint, and others

Debian, Ubuntu, Mint, and other Debian-based distributions all use **`.deb`** files and the **`dpkg`** package management system. There are two ways to install apps via this system. You can use the **`apt`** application to install from a repository, or you can use the **`dpkg`** app to install apps from **`.deb`** files. Let's take a look at how to do both.

Installing apps using **`apt`** is as easy as:

```
$ sudo apt install app_name
```

Uninstalling an app via **`apt`** is also super easy:

```
$ sudo apt remove app_name
```

To upgrade your installed apps, you'll first need to update the app repository:

```
$ sudo apt update
```

Once finished, you can update any apps that need updating with the following:

```
$ sudo apt upgrade
```

What if you want to update only a single app? No problem.

```
$ sudo apt update app_name
```

Finally, let's say the app you want to install is not available in the Debian repository, but it is available as a **`.deb`** download. You can install it manually using **`dpkg`**, the system that **`apt`** helps manage:

```
$ sudo dpkg -i app_name.deb
```

RHEL, CentOS, Fedora, Mageia, and OpenMandriva

Red Hat, its upstream project Fedora, and its "midstream" project CentOS, use the **dnf** package manager. It has its own syntax, and is a front-end to the RPM system. Although the syntax is different, **dnf** is similar to apt in the sense that the mechanisms and goals are the same. The Mageia and OpenMandriva distributions, once focused exclusively on **urpmi** for package management, now also includes **dnf** in their distributions.

The **dnf** package manager is the successor to the previous **yum** command. The **yum** had a long time to engrain itself in the minds and servers of users, so to avoid breaking custom scripts that have been around on users' systems for over a decade, **yum** and **dnf** are now interchangeable (in fact, **yum** is now based on **dnf**).

To install an app:

```
$ sudo dnf install app_name
```

Removing unwanted applications is just as easy.

```
$ sudo dnf remove app_name
```

Updating apps:

```
$ sudo dnf upgrade --refresh
```

The **dnf** (or **yum**) command is a front-end for the RPM packaging system. If you can't find an app in your software repository but you can find it for download directly from its vendor site, you can use **dnf** to manually install an **.rpm** file.

```
$ sudo dnf install ./app_name.rpm
```

As you can see, installing, uninstalling, and updating Linux apps from the command line isn't hard at all. In fact, once you get used to it, you'll find it's faster than using desktop GUI-based management tools!

For more information on installing apps from the command line, please visit the Debian Apt wiki, the Yum cheat sheet, and the DNF wiki.

Linux package management with dnf

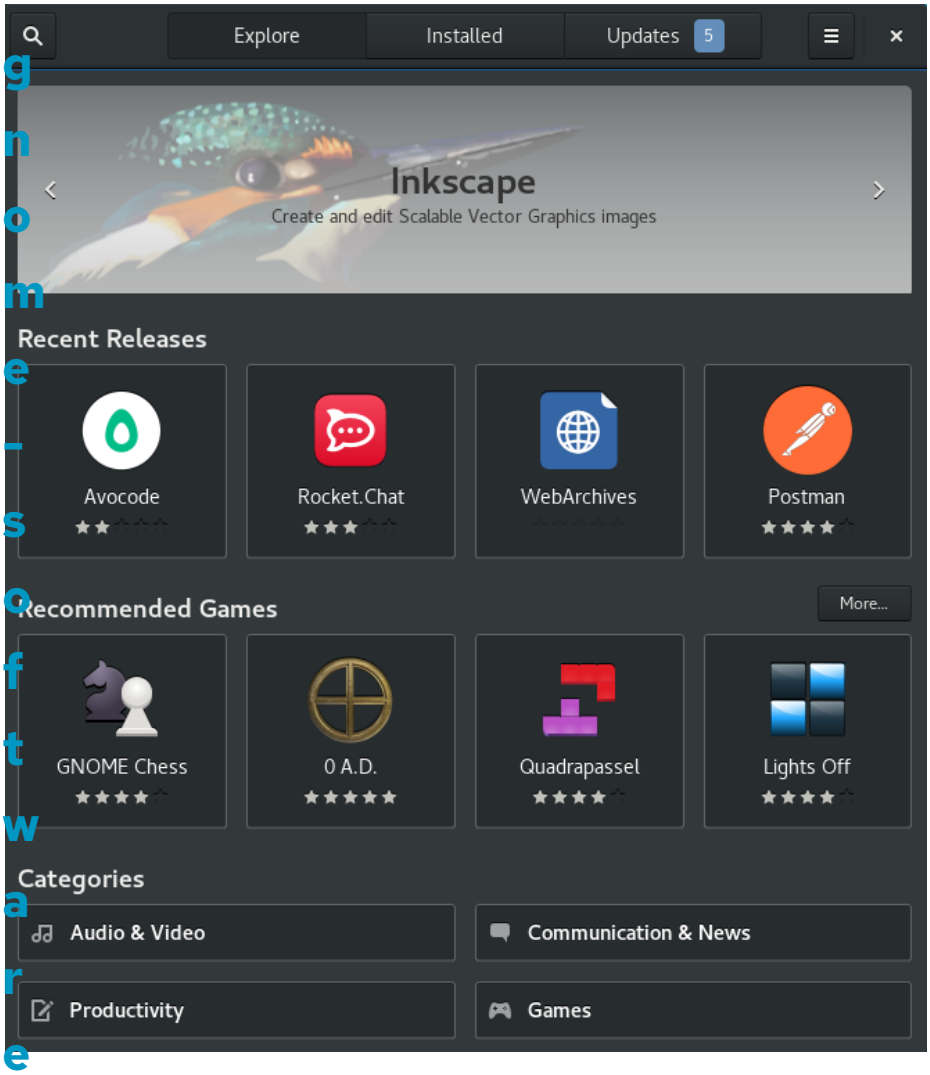
Installing an application on a computer system is pretty simple. You copy files from an archive (like a .zip file) onto the target computer in a place the operating system expects there to be applications. Because many of us are accustomed to having fancy installer "wizards" to help us get software on our computers, the process seems like it should be technically more complex than it is.

What *is* complex, though, is the issue of what makes up an application. What users think of as a single application actually contains code borrowing from software libraries (i.e., `.so` files on Linux, `.dll` files on Windows, and `.dylib` on macOS) scattered throughout an operating system.

So that users don't have to worry about that veritable matrix of interdependent code, Linux uses a **package management system** to track what application needs what library, and which library or application has security or feature updates, and what extra data files were installed with each software title. A package manager is, essentially, an installer wizard. They're easy to use, they provide both graphical interfaces and terminal-based interfaces, and they make your life easier. The better you know your distribution's package manager, the easier your life gets.

Installing applications on Linux

If you're a casual desktop user who wants to install an application on Linux, then you may be looking for GNOME Software, a desktop application browser.



gnome-software

It works as you'd expect: You click through its interface until you find an application that seems like it would be useful, and then you click the **Install** button.

Alternately, you can open `.rpm` or `.flatpakref` packages downloaded from the web in GNOME Software for it to install them for you.

If you're inclined toward controlling your computer with typed commands, read on!

dnf

Finding software with dnf

Before you can install an application, you may need to confirm that it exists on your distribution's servers. Usually, searching for the common name of an application with **dnf** suffices. For instance, say you recently read an article about Cockpit and decide you want to try it. You could search for **cockpit** to verify that your distribution includes it:

```
$ dnf search cockpit
 Last metadata expiration check: 0:01:46 ago on Tue 18 May 2021 19:18:15 NZST.
 ==== Name Exactly Matched: cockpit ====
 cockpit.x86_64 : Web Console for Linux servers
==== Name & Summary Matched: cockpit ==
 cockpit-bridge.x86_64 : Cockpit bridge server-side component
 cockpit-composer.noarch : Composer GUI for use with Cockpit
 [...]
```

There's an exact match. The package listed as a match is called **cockpit.x86_64**, but the **.x86_64** part of the name only denotes the CPU architecture it's compatible with. By default, your system installs packages with matching CPU architectures, so you can ignore that extension. Therefore, you've confirmed that the package you're looking for is indeed called simply **cockpit**.

Now you can confidently install it with **dnf install.** This step requires administrative privileges:

```
$ sudo dnf install cockpit
```

More often than not, that's the typical **dnf** workflow: search and install.

Sometimes, however, the results of **dnf search** aren't clear to you, or you want more information about a package than just its common name. There are a few relevant **dnf** subcommands, depending on what information you're after.

Package metadata

If you feel like your search got you close to the package you want, but you're just not sure yet, it's often helpful to take a look at the package's metadata, such as the project's URL and description. To get this info, use the pleasantly intuitive **`dnf info`** command:

```
$ dnf info terminator
Available Packages
Name         : terminator
Version      : 1.92
Release      : 2.el8
Architecture : noarch
Size         : 526 k
Source       : terminator-1.92-2.el8.src.rpm
Repository   : epel
Summary      : Store and run multiple GNOME terminals in one window
URL          : https://github.com/gnome-terminator
License      : GPLv2
Description  : Multiple GNOME terminals in one window.  This is a project to \
               produce
             : an efficient way of filling a large area of screen space with
             : terminals. This is done by splitting the window into a resizeable
             : grid of terminals. As such, you can  produce a very flexible
             : arrangements of terminals for different tasks.
```

This info dump tells you the version of the available package, which repository registered with your system provides it, the project's website, and a long description of what it does.

What package provides a file?

Package names don't always match what you're looking for. For instance, suppose you're reading documentation telling you that you must install something called **`qmake-qt5`**:

```
$ dnf search qmake-qt5
No matches found.
```

The **dnf** database is extensive, so you don't have to restrict yourself to searches for exact matches. You can use the **dnf provides** command to learn whether anything provides what you're looking for as part of some larger package:

```
$ dnf provides qmake-qt5
qt5-qtbase-devel-5.12.5-8.el8.i686 : Development files for qt5-qtbase
Repo        : appstream
Matched from:
Filename    : /usr/bin/qmake-qt5
qt5-qtbase-devel-5.15.2-3.el8.x86_64 : Development files for qt5-qtbase
Repo        : appstream
Matched from:
Filename    : /usr/bin/qmake-qt5
```

This confirms that the application **qmake-qt5** is a part of a package named **qt5-qtbase-devel**. It also tells you that the application gets installed to **/usr/bin**, so you know exactly where to find it once it's installed.

What files are included in a package?

There are times when I find myself approaching **dnf** from a different angle entirely. Sometimes, I've already confirmed that an application is installed on my system; I just can't figure out how I got it. Other times, I know I have a specific package installed, but I'm not clear on exactly what that package put on my system.

If you ever need to "reverse engineer" a package's payload, you can use the **dnf repoquery** command along with the **--list** option. This looks at the repository's metadata about a package and returns a list of all files provided by that package:

```
$ dnf repoquery --list qt5-qtbase-devel
/usr/bin/fixqt4headers.pl
/usr/bin/moc-qt5
/usr/bin/qdbuscpp2xml-qt5
/usr/bin/qdbusxml2cpp-qt5
/usr/bin/qlalr
/usr/bin/qmake-qt5
/usr/bin/qvkgen
/usr/bin/rcc-qt5
[...]
```

These lists can get long, so it helps to pipe the command through **less** or your favorite pager.

Removing an application

Should you decide you no longer need an application installed on your system, you can use **dnf remove** to uninstall it, all of the files that were installed as part of its package, and any dependencies that are no longer necessary:

```
$ dnf remove bigapp
```

Sometimes, dependencies get installed with one app and are later found useful by some other application you install. In the event that two packages require the same dependency, **dnf remove** does not remove the dependency. It's not unheard of to end up with a stray package here and there after installing and uninstalling lots of applications. About once a year, I perform a **dnf autoremove** to clear out any unused packages:

```
$ dnf autoremove
```

This isn't necessary, but it's a housecleaning step that makes me feel better about my computer.

Getting to know dnf

The more you know about how your package manager works, the easier it is for you to install and query applications when necessary. Even if you're not a regular **dnf** user, it can be useful to know it when you find yourself interfacing with an RPM-based distro.

Having graduated from **yum**, one of my favorite package managers is the **dnf** command. While I don't love all its subcommands, I find it to be one of the more robust package management systems out there. **Download our dnf cheat sheet** to get used to the command, and don't be afraid to try some new tricks with it. Once you get familiar with it, you might find it hard to use anything else.

Linux package management with apt

Contributed by Chris Hermansen

On Linux, package managers help you handle updates, uninstalls, troubleshooting, and more for the software on your computer. Seth Kenlon wrote about **dnf**, the command-line package management tool for installing software in RHEL, CentOS, Fedora, Mageia, OpenMandriva, and other Linux distros.

Debian and Debian-based distros such as MX Linux, Deepin, Ubuntu—and distros based on Ubuntu, such as Linux Mint and Pop!_OS—have **apt**, a "similar but different" tool. In this article, I'll follow Seth's examples—but with **apt**—to show you how to use it.

Before I start, I want to mention four **apt**-related tools for installing software:

- Synaptic is a GTK+ based graphical user interface (GUI) front end for **apt**.
- Aptitude is an Ncurses-based full-screen command-line front end for **apt**.
- There are **apt-get**, **apt-cache**, and other predecessors of **apt**.
- Dpkg is the "behind the scenes" package manager **apt** uses to do the heavy lifting.

There are other packaging systems, such as Flatpak and Snap, that you might run into on Debian and Debian-based systems, but I'm not going to discuss them here. There are also application "stores," such as GNOME Software, that overlap with apt and other packaging technologies; again, I'm not going to discuss them here. Finally, there are other Linux distros such as Arch and Gentoo that use neither dnf nor apt, and I'm not going to discuss those here either!

With all the things I'm not going to discuss here, you may be wondering what tiny subset of software **apt** handles. Well, on my Ubuntu 20.04, **apt** gives me access to 69,371 packages, from the **0ad** real-time strategy game of ancient warfare to the **zzuf** transparent application fuzzer. Not bad at all.

Finding software with apt

The first step in using a package manager such as **apt** is finding a software package of interest. Seth's **dnf** article used the Cockpit server management application as an example, so I will, too:

```
$ apt search cockpit
Sorting... Done
Full Text Search... Done
389-ds/hirsute,hirsute 1.4.4.11-1 all
  389 Directory Server suite - metapackage
cockpit/hirsute,hirsute 238-1 all
  Web Console for Linux servers
...
$
```

The second package above is the one you're after (it's the line beginning with **cockpit/hirsute**). If you decide you want to install it, enter:

```
$ sudo apt install cockpit
```

apt will take care of installing Cockpit and all the bits and pieces, or dependencies, needed to make it work. Sometimes that's all that's needed; sometimes it's not. It's possible that having a bit more information could be useful in deciding whether you really want to install this application.

Package metadata

To find out more about a package, use the **`apt show`** command:

```
$ apt show cockpit
Package: cockpit
Version: 238-1
Priority: optional
Section: universe/admin
Origin: Ubuntu
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Original-Maintainer: Utopia Maintenance Team <pkg-utopia-
maintainers@lists.alioth.debian.org>
Bugs: https://bugs.launchpad.net/ubuntu/+filebug
Installed-Size: 88.1 kB
Depends: cockpit-bridge (>= 238-1), cockpit-ws (>= 238-1), cockpit-system (>=
238-1)
Recommends: cockpit-storaged (>= 238-1), cockpit-networkmanager (>= 238-1),
cockpit-packagekit (>= 238-1)
Suggests: cockpit-doc (>= 238-1), cockpit-pcp (>= 238-1), cockpit-machines (>=
238-1), xdg-utils
Homepage: https://cockpit-project.org/
Download-Size: 21.3 kB
APT-Sources: http://ca.archive.ubuntu.com/ubuntu hirsute/universe amd64 Packages
Description: Web Console for Linux servers
 The Cockpit Web Console enables users to administer GNU/Linux servers using a
 web browser.
 .
 It offers network configuration, log inspection, diagnostic reports, SELinux
 troubleshooting, interactive command-line sessions, and more.
$
```

In particular, notice the **`Description`** field, which tells you more about the application. The **`Depends`** field says what else must be installed, and **`Recommends`** shows what other—if any—cooperating components are suggested alongside it. The **`Homepage`** field offers a URL in case you need more info.

What package provides a file?

Sometimes you don't know the package name, but you know a file that must be in a package. Seth offers as an example the **`qmake-qt5`** utility. Using **`apt search`** doesn't find it:

```
$ apt search qmake-qt5
Sorting... Done
Full Text Search... Done
$
```

However, a related command, **`apt-file`** will explore inside packages:

```
$ apt-file search qmake-qt5
qt5-qmake-bin: /usr/share/man/man1/qmake-qt5.1.gz
$
```

This turns up a man page for **`qmake-qt5`** that is part of a package called **`qt5-qmake-bin`**. Note that this package name reverses the **`qmake`** and **`qt5`** parts.

What files are included in a package?

That handy **`apt-file`** command also tells which files are included in a given package. For example:

```
$ apt-file list cockpit
cockpit: /usr/share/doc/cockpit/TODO.Debian
cockpit: /usr/share/doc/cockpit/changelog.Debian.gz
cockpit: /usr/share/doc/cockpit/copyright
cockpit: /usr/share/man/man1/cockpit.1.gz
cockpit: /usr/share/metainfo/cockpit.appdata.xml
cockpit: /usr/share/pixmaps/cockpit.png
$
```

Note that this is distinct from the info provided by the **`apt show`** command, which lists the package's dependencies (other packages that must be installed).

Removing an application

You can also remove packages with **apt**. For example, to remove the **apt-file** application:

```
$ sudo apt purge apt-file
```

Note that a superuser must run **apt** to install or remove applications.

Removing a package doesn't automatically remove all the dependencies that **apt** installs along the way. However, it's easy to carry out that little bit of tidying:

```
$ sudo apt autoremove
```

Getting to know apt

As Seth wrote, "the more you know about how your package manager works, the easier it is for you to install and query applications when necessary."

Even if you're not a regular **apt** user, knowing it can be useful when you need to work at the command line while installing or removing packages (for example, on a remote server or when following a how-to published by some helpful soul). You may also need to know a bit about Dkpg (mentioned above); for example, some software creators provide a bare **.pkg** file.

I find the Synaptic package manager to be a really useful tool on my desktop, but I also use **apt** on a handful of servers that I maintain for various purposes.

Download our apt cheat sheet to get used to the command and try some new tricks with it. Once you do, you might find it hard to use anything else.

Run your favorite Windows applications on Linux

Do you have an application that only runs on Windows? Is that one application the one and only thing holding you back from switching to Linux? If so, you'll be happy to know about WINE, an open source project that has all but reinvented key Windows libraries so that applications compiled for Windows can run on Linux.

WINE stands for "Wine Is Not an Emulator," which references the code driving this technology. Open source developers have worked since 1993 to translate any incoming Windows API calls an application makes to POSIX calls.

This is an astonishing feat of programming, especially given that the project operated independently, with no help from Microsoft (to say the least), but there are limits. The farther an application strays from the "core" of the Windows API, the less likely it is that WINE could have anticipated its requests. There are vendors that may make up for this, notably Codeweavers and Valve Software. There's no coordination between the producers of the applications requiring translation and the people and companies doing the translation, so there can be some lag time between, for instance, an updated software title and when it earns a "gold" status from WINE headquarters.

However, if you're looking to run a well-known Windows application on Linux, the chances are good that WINE is ready for it.

Installing WINE

You can install WINE from your Linux distribution's software repository. On Fedora, CentOS Stream, or RHEL:

```
$ sudo dnf install wine
```

On Debian, Linux Mint, Elementary, and similar:

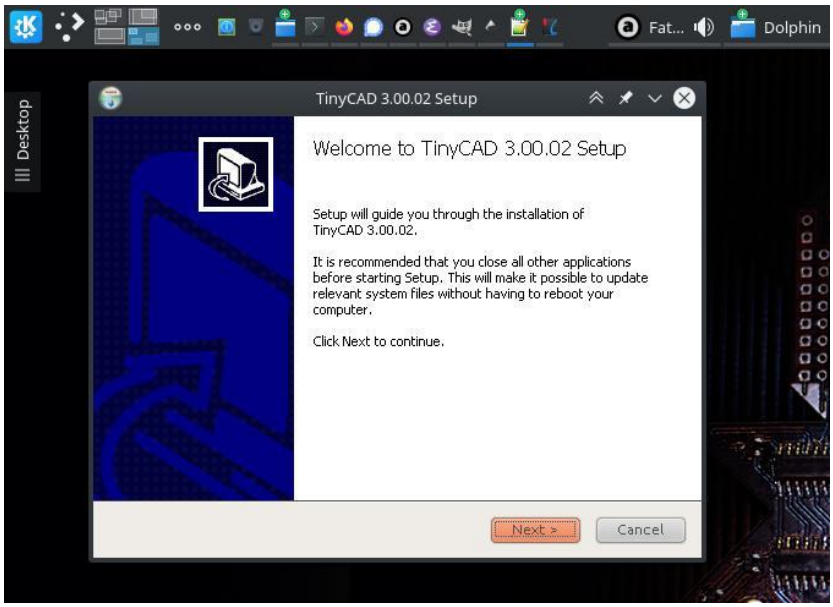
```
$ sudo apt install wine
```

WINE isn't an application that you launch on its own. It's a backend that gets invoked when a Windows application is launched. Your first interaction with WINE will most likely occur when you launch the installer of a Windows application.

Installing an application

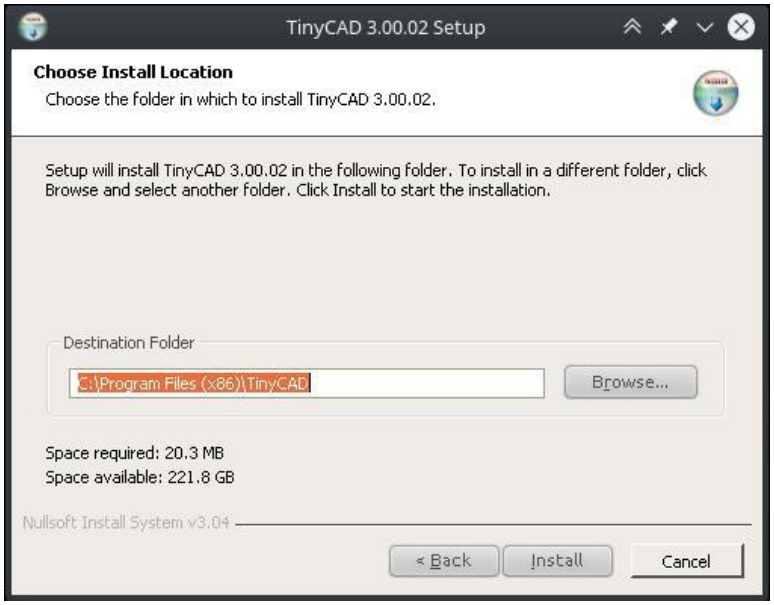
TinyCAD is a nice open source application for designing circuits, but it's only available for Windows. While it is a small application, it does incorporate some .NET components, so that ought to stress test WINE a little.

First, download the installer for TinyCAD. As is often the case for Windows installers, it's a **.exe** file. Once downloaded, double-click the file to launch it.



WINE installation wizard for TinyCAD

Step through the installer as you would on Windows. It's usually best to accept the defaults, especially where WINE is concerned. The WINE environment is largely self-contained, hidden away on your hard drive in a **drive_c** directory that gets used by a Windows application as the fake root directory of the file system.

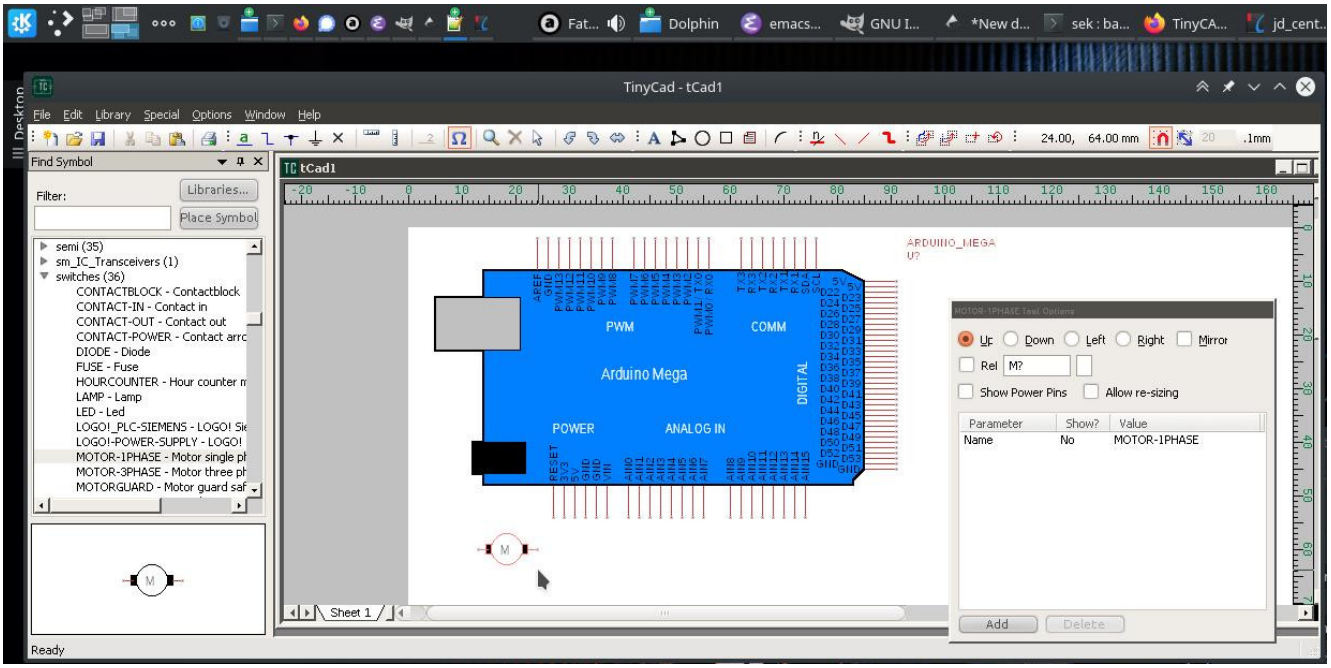


WINE TinyCAD destination drive

Once it's installed, the application usually offers to launch for you. If you're ready to test it out, launch the application.

Launching a Windows application

Aside from the first launch immediately after installation, you normally launch a WINE application the same way as you launch a native Linux application. Whether you use an applications menu or an Activities screen or just type the application's name into a runner, desktop Windows applications running in WINE are treated essentially as native applications on Linux.



TinyCAD running with WINE support

When WINE fails

Most applications I run in WINE, TinyCAD included, run as expected. There are exceptions, however. In those cases, you can either wait a few months to see whether WINE developers (or, if it's a game, Valve Software) manage to catch up, or you can contact a vendor like Codeweavers to find out whether they sell support for the application you require.

WINE is cheating, but in a good way

Some Linux users feel that if you use WINE, you're "cheating" on Linux. It might feel that way, but WINE is an open source project that's enabling users to switch to Linux and still run required applications for their work or hobbies. If WINE solves your problem and lets you use Linux, then use it, and embrace the flexibility of Linux.

Anyone can compile open source code in these three simple steps

There are many ways to install software, but you get an option not available elsewhere with open source: You can compile the code yourself. The classic three-step process to compile source code:

```
$ ./configure
$ make
$ sudo make install
```

Thanks to these commands, you might be surprised to find that you don't need to know how to write or even read code to compile it.

Install commands to build software

As this is your first time compiling, there's a one-time preparatory step to install the commands for building software. Specifically, you need a compiler. A compiler, such as GCC or LLVM, turns source code that looks like this—

```
#include <iostream>
using namespace std;
int main() {
  cout << "hello world";
}
```

—into *machine language*, the instructions that a CPU uses to process information. You can look at machine code, but it wouldn't make any sense to you (unless you're a CPU).

You can get the GNU C compiler (GCC) and the LLVM compiler, along with other essential commands for compiling on Fedora, CentOS, Mageia, and similar distributions, using your package manager:

```
$ sudo dnf install @development clang
```

On Debian, Elementary, Mint, and similar distributions:

```
$ sudo apt install build-essential clang
```

With your system set up, there are a few tasks that you'll repeat each time you want to compile your software:

1. Download the source code
2. Unarchive the source code
3. Compile

You have all the commands you need, so now you need some software to compile.

1. Download source code

Obtaining source code for an application is much like getting any downloadable software. You go to a website or a code management site like GitLab, SourceForge, or GitHub. Typically, open source software is available in both a work-in-progress ("current" or "nightly") form as well as a packaged "stable" release version. Use the stable version when possible unless you have reason to believe otherwise or are good enough with code to fix things when they break. The term **stable** suggests the code got tested and that the programmers of the application feel confident enough in the code to package it into a `.zip` or `.tar` archive, give it an official number and sometimes a release name, and offer it for download to the general non-programmer public.

For this exercise, I'm using Angband, an open source (GPLv2) ASCII dungeon crawler. It's a simple application with just enough complications to demonstrate what you need to consider when compiling software for yourself.

Download the source code from the website.

2. Unarchive the source code

Source code is often delivered as an archive because source code usually consists of multiple files. You have to extract it before interacting with it, whether it's a tarball, a zip file, a 7z file, or something else entirely.

```
$ tar --extract --file Angband-x.y.z.tar.gz
```

Once you've unarchived it, change the directory into the extracted directory and have a look around. There's usually a **`README`** file at the top level of the directory. This file, ideally, contains guidance on what you need to do to compile the code. The **`README`** often contains information on these important aspects of the code:

- **Language:** What language the code is in (for instance, C, C++, Rust, Python).
- **Dependencies:** What other software you need to have installed on your system for this application to build and run.
- **Instructions:** The literal steps you need to take to build the software. Occasionally, they include this information within a dedicated file intuitively entitled **`INSTALL`**.

If the **`README`** file doesn't contain that information, consider filing a bug report with the developer. You're not the only one who needs an introduction to source code. Regardless of how experienced they are, everyone is new to source code they've never seen before, and documentation is important!

Angband's maintainers link to online instructions to describe how to compile the code. This document also describes what other software you need to have installed, although it doesn't exactly spell it out. The site says, "There are several different front ends that you can optionally build (GCU, SDL, SDL2, and X11) using arguments to configure such as **`--enable-sdl`**, **`--disable-x11,`** etc." This may mean something to you or look like a foreign language, but this is the kind of stuff you get used to after compiling code frequently. Whether or not you understand what X11 or SDL2 is, they're both requirements that you see pretty often after regularly compiling code over a few months. You get comfortable with the idea that most software needs other software libraries because they build upon other

technologies. In this case, though, Angband is very flexible and compiles with or without these optional dependencies, so for now, you can pretend that there are no additional dependencies.

3. Compile the code

The canonical steps to build code are:

```
$ ./configure
$ make
$ sudo make install
```

Those are the steps for projects built with Autotools, which is a framework created to standardize how source code is delivered. Other frameworks (such as Cmake) exist, however, and they require different steps. When projects stray from Autotools or Cmake, they tend to warn you in the **README** file.

Configure

Angband uses Autotools, so it's time to compile code!

In the Angband directory, first, run the configuration script included with the source:

```
$ ./configure
```

This step scans your system to find the dependencies that Angband requires to build correctly. Some dependencies are so basic that your computer wouldn't be running without them, while others are specialized. At the end of the process, the script gives you a report on what it has found:

```
[...]
configure: creating ./config.status
config.status: creating mk/buildsys.mk
config.status: creating mk/extra.mk
config.status: creating src/autoconf.h
Configuration:
  Install path:    /usr/local
  binary path:     /usr/local/games
  config path:     /usr/local/etc/angband/
  lib path:        /usr/local/share/angband/
  doc path:        /usr/local/share/doc/angband/
  var path:        (not used)
  (save and score files in ~/.angband/Angband/)
-- Frontends --
- Curses            Yes
- X11               Yes
- SDL2              Disabled
- SDL               Disabled
- Windows           Disabled
- Test              No
- Stats             No
- Spoilers          Yes
- SDL2 sound        Disabled
- SDL sound         Disabled
```

Some of that output may make sense to you, some of it may not. Either way, you probably notice that SDL2 and SDL are marked as **`Disabled`**, and both Test and Stats are marked with **`No`**. Although negative, this isn't necessarily a bad thing. This, essentially, is the difference between a **Warning** and an **Error**. Had the configure script encountered something that would prevent it from building the code, it would have alerted you with an error.

If you want to optimize your build a little, you can choose to resolve these negative messages. By searching through the Angband documentation, you might decide that Test and Stats aren't actually of interest to you (they're developer options, specific to Angband). However, with a little online research, you might discover that SDL2 would be a nice feature to have.

To resolve a dependency when compiling code, you need to install the missing component and the *development libraries* for that missing component. In other words, Angband needs SDL2 to play sound, but it needs **`SDL2-devel`** (called **`libsdl2-dev`**, on Debian systems) to build. Install both with your package manager:

```
$ sudo dnf install sdl2 sdl2-devel
```

Try the configuration script again:

```
$ ./configure --enable-sdl2
[...]
Configuration:
[...]
- Curses                                    Yes
- X11                                       Yes
- SDL2                                      Yes
- SDL                                       Disabled
- Windows                                   Disabled
- Test                                      No
- Stats                                     No
- Spoilers                                  Yes
- SDL sound                                 Disabled
- SDL2 sound                                Yes
```

Make

Once everything's configured, run the **`make`** command:

```
$ make
```

This usually takes a while, but it provides lots of visual feedback, so you'll know code is getting compiled.

Install

The final step is to install the code you've just compiled. There's nothing magical about installing code. All that happens is that lots of files get copied to very specific directories. That's true whether you're compiling from source code or running a fancy graphical install

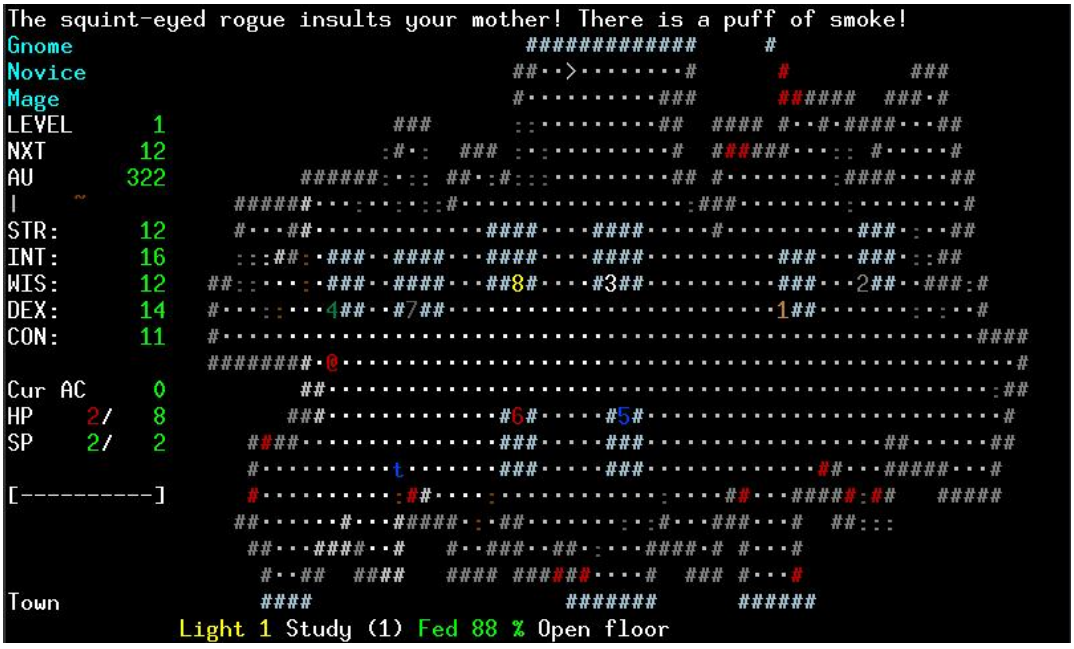
wizard. Because the code is getting copied to system-level directories, you must have root (administrative) privileges, which get granted by the **sudo** command.

```
$ sudo make install
```

Run the application

Once the application gets installed, you can run it. According to the Angband documentation, the command to start the game is **angband**, so try it out:

```
$ angband
```



(Seth Kenlon, CC BY-SA 4.0)

Compiling code

I compile most of my own applications, whether on my Slackware desktop computer or my CentOS laptop using NetBSD's pkgsrc system. I find that by compiling software myself, I can be as particular as I want to be about the features included in the application, how it's configured, which library version it uses, and so on. It's rewarding, and it helps me keep up to

date with new releases and, because I sometimes find bugs in the process, it helps me get involved with lots of different open source projects.

It's rare that you have no other option but to compile software. Most open source projects provide both the source code (that's why it's called "open source") and installable packages. Compiling from source code is a choice you get to make for yourself, maybe because you want new features not yet available in the latest release or just because you prefer to compile code yourself.

Homework

Angband can use either Autotools or Cmake, so if you want to experience another way of building code, try this:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo make install
```

You can also try compiling with the LLVM compiler instead of the GNU C compiler. For now, I'll leave that as an exercise for you to investigate on your own (hint: try setting the CC environment variable).

Once you finish exploring the source code of Angband and at least a few of its dungeons (you've earned some downtime), have a look at some other codebases. Many will use Autotools or Cmake, while others may use something different. See what you can build!

Why programmers love Linux packaging

Programmers love to program. That probably seems like an obvious statement, but it's important to understand that developing software involves a lot more than just writing code. It includes compiling, documentation, source code management, install scripts, configuration defaults, support files, delivery format, and more. Getting from a blank screen to a deliverable software installer requires much more than just programming, but most programmers would rather program than package.

What is packaging?

When food is sent to stores to be purchased, it is packaged. When buying directly from a farmer or from an eco-friendly bulk or bin store, the packaging is whatever container you've brought with you. When buying from a grocery store, packaging may be a cardboard box, plastic bag, a tin can, and so on.

When software is made available to computer users at large, it also must be packaged. Like food, there are several ways software can be packaged. Open source software can be left unpackaged because users, having access to the raw code, can compile and package it themselves. However, there are advantages to packages, so applications are commonly delivered in some format specific to the user's platform. And that's where the problems begin, because there's not just one format for software packages.

For the user, packages make it easy to install software because all the work is done by the system's installer. The software is extracted from its package and distributed to the appropriate places within the operating system. There's little opportunity for anything to go wrong.

For the software developer, however, packaging means that you have to learn how to create a package—and not just one package, but a unique package for every operating system you want your software to be installable on. To complicate matters, there are multiple packaging formats and options for each operating system, and sometimes even for the programming language being used.

Packaging on Linux

Packaging options for Linux have traditionally seemed pretty overwhelming. Linux distributions derived from Fedora, such as Red Hat and CentOS, default to `.rpm` packages. Debian and Ubuntu (and similar) default to `.deb` packages. Other distributions may use one or the other, or neither, opting for a custom format. When asked, many Linux users say that ideally, a programmer won't package their software for Linux at all but instead rely on the package maintainers of each distribution to create the package. All software installed onto any Linux system ought to come from that distribution's official repository. However, it remains unclear how to get your software reliably packaged and included by one distribution, let alone all distributions.

Flatpak for Linux

The Flatpak packaging system was introduced to unify and decentralize Linux as a delivery target for developers. With Flatpak, either a developer or anyone (a member of a Linux community, a different developer, a Flatpak team member, or anyone else) is free to package software. They can then submit the package to Flathub or choose to self-host the package and offer it to basically any Linux distribution. The Flatpak system is available to all Linux distributions, so targeting one is the same as targeting them all.

How Flatpak technology works

The secret to Flatpak's universal appeal is a standard base. The Flatpak system allows developers to reference a common set of Software Developer Kit (SDK) modules. These are packaged and managed by the maintainers of the Flatpak system. The SDKs get pulled in as needed whenever you install a Flatpak, ensuring compatibility with your system. Any given SDK is only required once because the libraries it contains can be shared across any Flatpak calling for it.

If a developer requires a library not already included in an existing SDK, the developer can add that library in the Flatpak.

The results speak for themselves. Users may install hundreds of packages on any Linux distribution from one central repository, called Flathub.

How developers use Flatpaks

Flatpaks are designed to be reproducible, so the build process is easily integrated into a CI/CD workflow. A Flatpak is defined in a YAML or JSON manifest file. You can create your first Flatpak by following my introductory article, and you can read the full documentation at docs.flatpak.org.

Linux makes it easy

Creating software on Linux is easy, and packaging it up for Linux is simple and automatable. If you're a programmer, Linux makes it easy for you to forget about packaging by targeting one system and integrating that into your build process.

Install apps on Linux with Flatpak

Computer applications consist of many small files that are linked together to perform a set of tasks. Because they get presented as "apps," colorful icons in the menu or on a desktop, most of us think of applications as a single, almost tangible thing. And in a way, it's comforting to think of them that way because they feel manageable that way. If an application is actually the amalgamation of hundreds of little library and asset files scattered throughout your computer, where's the application? And existential crisis aside, what happens when one application needs one version of a library while another application demands a different version?

In the world of cloud computing, containers are becoming more and more popular because they offer isolation and consolidation for applications. You can install all the files an application needs in a "container." That way, its libraries stay out of the way of other applications, and the memory it occupies doesn't leak data into the memory space of another. Everything ends up feeling very much like a single, almost tangible thing. On the Linux desktop, Flatpak, a cross-distribution, daemon-less, decentralized application delivery system, provides a similar technology.

Install Flatpak on Linux

Your Linux system may already have Flatpak installed. If not, you can install it from your package manager:

On Fedora, Mageia, and similar distributions:

```
$ sudo dnf install flatpak
```

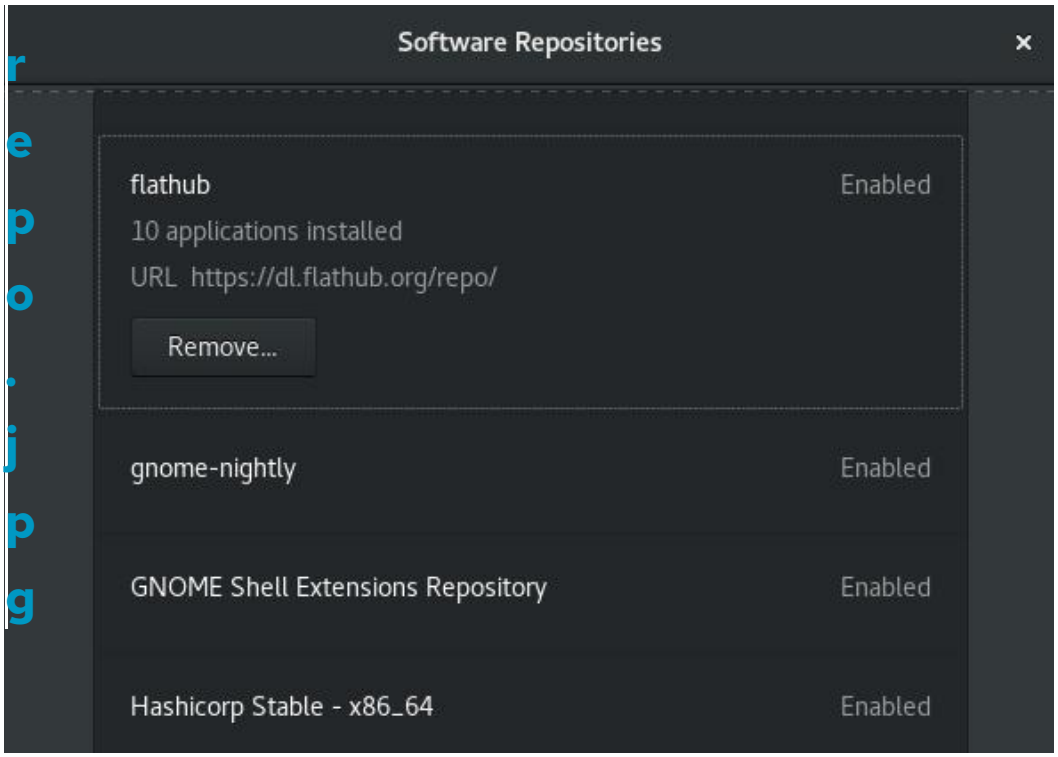
On Elementary, Mint, and other Debian-based distributions:

```
$ sudo apt install flatpak
```

On Slackware, Flatpak is available from SlackBuilds.org.

Select a Flatpak repo

You can install an application as a Flatpak by adding a Flatpak repository to your distribution's software center (such as **Software** on GNOME). Flatpak is a decentralized system, meaning anyone developing software can host their own repository. Still, in practice, Flathub is the biggest and most popular aggregation of applications in the Flatpak format. To add Flathub to **GNOME Software** or **KDE Discover**, navigate to flatpak.org/setup and find the instructions for your distribution and start with step #2, or just download the Flatpakrepo file. Depending on your network, it may take a few minutes for your software center to synchronize with Flathub or another Flatpak repository. Flathub has a lot of software, but there's no limit to how many Flatpak repositories you have on your system, so don't be afraid to add a new repository if you find one that has the software you want to try.



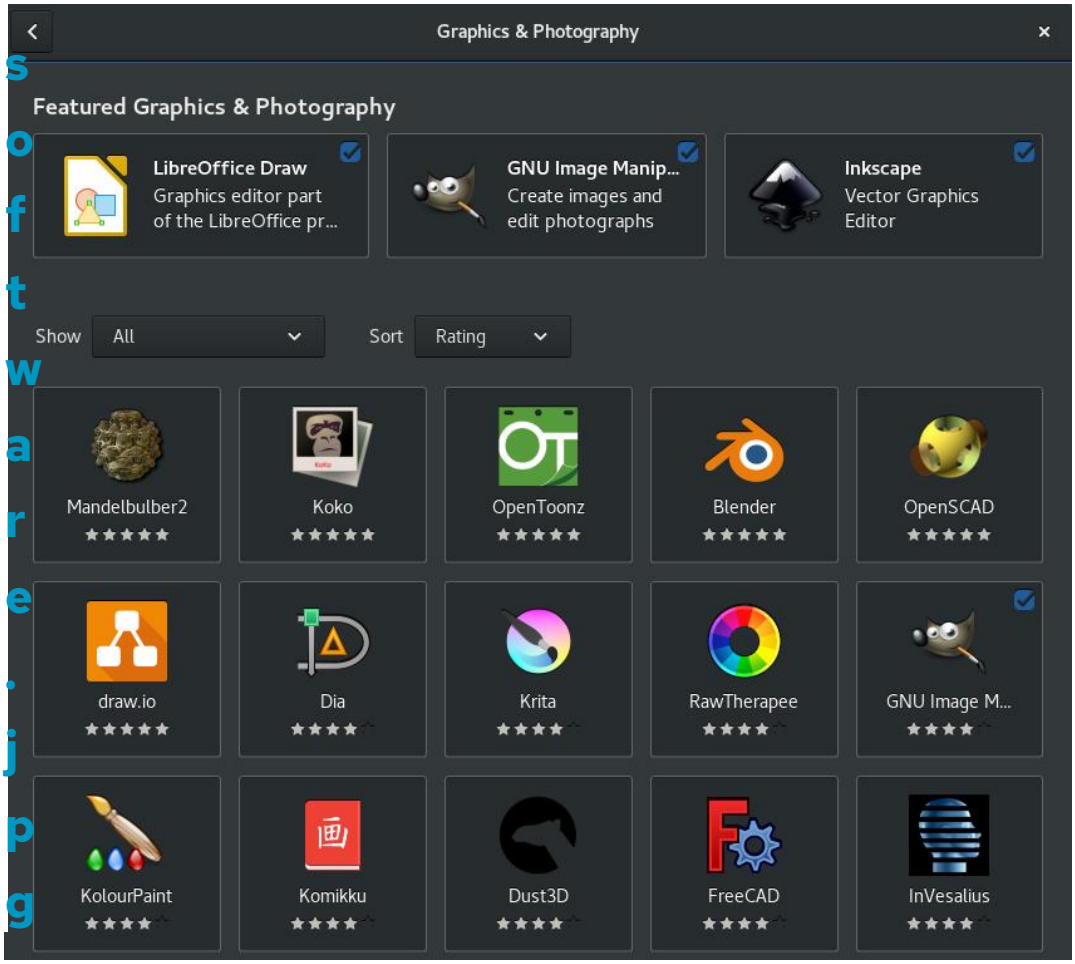
(Seth Kenlon, CC BY-SA 4.0)

If you prefer to work in the terminal, you can add repositories directly with the **`flatpak`** command:

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

Install an application

As long as you've added a Flatpak repository to your software center, you can browse through applications as usual.



(Seth Kenlon, CC BY-SA 4.0)

Click on an application that looks appealing, read up on it, and click the **Install** button when you're ready.

Installing flatpaks in the terminal

If you prefer to work in the terminal, you can treat Flatpak as a dedicated package manager. You can search for an application using the **`flatpak search`** command:

```
$ flatpak search paint
Name        Description                         Application ID
CorePaint   A simple painting tool              org.cubocore.CorePaint
Pinta       Edit images and paint digitally     com.github.PintaProject.Pinta
Glimpse     Create images and edit photographs  org.glimpse_editor.Glimpse
Tux Paint   A drawing program for children      org.tuxpaint.Tuxpaint
Krita       Digital Painting, Creative Freedom  org.kde.krita
```

Install with **`flatpak install`**:

```
$ flatpak install krita
```

Once installed, applications appear in your application menu or Activities screen along with all the other applications on your system.

Apps made easy

Flatpak makes installing applications easy for the user by eliminating version conflicts. They make distributing software easy for developers by targeting just one package format on either a self-hosted platform or a communal one like Flathub. I use Flatpaks on Fedora Silverblue, CentOS, and Slackware, and I can't quite imagine life without it now. Try Flatpak for your next app install!

How to build a Flatpak

This is an advanced chapter for people interested in programming, packaging, and distributing software. If you're just a Linux user, you don't need to know how a Flatpak is built, you can just use them as described in the previous chapter. However, if you're curious, or you're someone with an application to distribute, this chapter demonstrates how that's done.

A long time ago, a Linux distribution shipped an operating system along with all the software available for it. There was no concept of "third party" software because everything was a part of the distribution. Applications weren't so much installed as they were enabled from a great big software repository that you got on one of the many floppy disks or, later, CDs you purchased or downloaded.

This evolved into something even more convenient as the internet became ubiquitous, and the concept of what is now the "app store" was born. Of course, Linux distributions tend to call this a software repository or just repo for short, with some variations for "branding", such as Ubuntu Software Center or, with typical GNOME minimalism, simply Software.

This model worked well back when open source software was still a novelty and the number of open source applications was a number rather than a theoretical number. In today's world of GitLab and GitHub and Bitbucket (and many many more), it's hardly possible to count the number of open source projects, much less package them up in a repository. No Linux distribution today, even Debian and its formidable group of package maintainers, can claim or hope to have a package for every installable open source project.

Of course, a Linux package doesn't have to be in a repository to be installable. Any programmer can package up their software and distribute it from their own website. However, because repositories are seen as an integral part of a distribution, there isn't a universal packaging format, meaning that a programmer must decide whether to release a **`.deb`** or **`.rpm`**, or an AUR build script, or a Nix or Guix package, or a Homebrew script, or just a mostly-generic **`.tgz`** archive for **`/opt`**. It's overwhelming for a developer who lives and breathes Linux

every day, much less for a developer just trying to make a best-effort attempt at supporting a free and open source target.

Why Flatpak?

The Flatpak project provides a universal packaging format along with a decentralized means of distribution, plus portability, and sandboxing.

- **Universal:** Install the Flatpak system, and you can run Flatpaks, regardless of your distribution. No daemon or systemd required. The same Flatpak runs on Fedora, Ubuntu, Mageia, Pop OS, Arch, Slackware, and more.
- **Decentralized:** Developers can create and sign their own Flatpak packages and repositories. There's no repository to petition in order to get a package included.
- **Portability:** If you have a Flatpak on your system and want to hand it to a friend so they can run the same application, you can export the Flatpak to a USB thumbdrive.
- **Sandboxed:** Flatpaks use a container-based model, allowing multiple versions of libraries and applications to exist on one system. Yes, you can easily install the latest version of an app to test out while maintaining the old version you rely on.

Building a Flatpak

To build a Flatpak, you must first install Flatpak (the subsystem that enables you to use Flatpak packages) and the Flatpak-builder application.

On Fedora, CentOS, RHEL, and similar:

```
$ sudo dnf install flatpak flatpak-builder
```

On Debian, Ubuntu, and similar:

```
$ sudo apt install flatpak flatpak-builder
```

You must also install the development tools required to build the application you are packaging. By nature of developing the application you're now packaging, you may already have a development environment installed, so you might not notice that these components are required, but should you start building Flatpaks with Jenkins or from inside containers, then you must ensure that your build tools are a part of your toolchain.

For the first example build, this article assumes that your application uses GNU Autotools, but Flatpak itself supports other build systems, such as **`cmake`**, **`cmake-ninja`**, **`meson`**, **`ant`**, as well as custom commands (a **`simple`** build system, in Flatpak terminology, but by no means does this imply that the build itself is actually simple).

Project directory

Unlike the strict RPM build infrastructure, Flatpak doesn't impose a project directory structure. I prefer to create project directories based on the **dist** packages of software, but there's no technical reason you can't instead integrate your Flatpak build process **`with`** your source directory. It is technically easier to build a Flatpak from your **dist** package, though, and it's an easier demo too, so that's the model this article uses. Set up a project directory for GNU Hello, serving as your first Flatpak:

```
$ mkdir hello_flatpak
$ mkdir src
```

Download your distributable source. For this example, the source code is located at `https://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz.`

```
$ cd hello_flatpak
$ wget https://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz
```

Manifest

A Flatpak is defined by a manifest, which describes how to build and install the application it is delivering. A manifest is atomic and reproducible. A Flatpak exists in a "sandbox" container, though, so the manifest is based on a mostly empty environment with a root directory call **`/app.`**

The first two attributes are the ID of the application you are packaging and the command provided by it. The application ID must be unique to the application you are packaging. The canonical way of formulating a unique ID is to use a triplet value consisting of the entity responsible for the code followed by the name of the application, such as **`org.gnu.Hello`**.

The command provided by the application is whatever you type into a terminal to run the application. This does not imply that the application is intended to be run from a terminal instead of a **`.desktop`** file in the Activities or Applications menu.

In a file called **`org.gnu.Hello.yaml`**, enter this text:

```
id: org.gnu.Hello
command: hello
```

A manifest can be written in YAML or in JSON. This article uses YAML.

Next, you must define each "module" delivered by this Flatpak package. You can think of a module as a dependency or a component. For GNU Hello, there is only one module: GNU Hello. More complex applications may require a specific library or another application entirely.

```
modules:
  - name: hello
    buildsystem: autotools
    no-autogen: true
    sources:
      - type: archive
        path: src/hello-2.10.tar.gz
```

The **`buildsystem`** value identifies how Flatpak must build the module. Each module can use its own build system, so one Flatpak can have several build systems defined.

The **`no-autogen`** value tells Flatpak not to run the setup commands for **`autotools`**, which aren't necessary because the GNU Hello source code is the product of **`make dist`**. If the code you're building isn't in a easily buildable form, then you may need to install **`autogen`** and **`autoconf`** to prepare the source for **`autotools`**. This option doesn't apply at all to projects that don't use **`autotools`**.

The **`type`** value tells Flatpak that the source code is in an archive, which triggers the requisite unarchival tasks before building. The **`path`** points to the source code. In this example, the source exists in the **`src`** directory on your local build machine, but you could instead define the source as a remote location:

```
modules:
  - name: hello
    buildsystem: autotools
    no-autogen: true
    sources:
      - type: archive
        url: https://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz
```

Finally, you must define the platform required for the application to run and build. The Flatpak maintainers supply runtimes and SDKs that include common libraries, including **freedesktop**, **gnome**, and **kde**. The basic requirement is the **freedesk** runtime and SDK, although this may be superseded by GNOME or KDE, depending on what your code needs to run. For this GNU Hello example, only the basics are required.

```
runtime: org.freedesktop.Platform
runtime-version: '18.08'
sdk: org.freedesktop.Sdk
```

The entire GNU Hello flatpak manifest:

```
id: org.gnu.Hello
runtime: org.freedesktop.Platform
runtime-version: '18.08'
sdk: org.freedesktop.Sdk
command: hello
modules:
  - name: hello
    buildsystem: autotools
    no-autogen: true
    sources:
      - type: archive
        path: src/hello-2.10.tar.gz
```

Building a Flatpak

Now that the package is defined, you can build it. The build process prompts Flatpak-builder to parse the manifest and to resolve each requirement: it ensures that the necessary Platform and SDK are available (if they aren't, then you'll have to install them with the **`flatpak`** command), it unarchives the source code, and executes the **`buildsystem`** specified.

The command to start:

```
$ flatpak-builder build-dir org.gnu.Hello.yaml
```

The directory **`build-dir`** is created if it does not already exist. The name **`build-dir`** is arbitrary; you could call it **`build`** or **`bld`** or **`penguin`**, and you can have more than one build

destination in the same project directory. However, the term **build-dir** is a frequent value used in documentation, so using it as the literal value can be helpful.

Testing your application

You can test your application before or after it has been built by running the build command along with the **--run** option, and ending the command with the command provided by the Flatpak:

```
$ flatpak-builder --run build-dir \
org.gnu.Hello.yaml hello
Hello, world!
```

Packaging GUI apps with Flatpak

Packaging up a simple self-contained hello world application is trivial, and fortunately packaging up a GUI application isn't much harder. The most difficult applications to package are those that don't rely on common libraries and frameworks (in the context of packaging, "common" means anything not already packaged by someone else). The Flatpak community provides SDKs and SDK Extensions for many components you might otherwise have had to package yourself. For instance, when packaging the pure Java implementation of **pdftk**, I use the OpenJDK SDK extension I found in the Flatpak Github repository:

```
runtime: org.freedesktop.Platform
runtime-version: '18.08'
sdk: org.freedesktop.Sdk
sdk-extensions:
 - org.freedesktop.Sdk.Extension.openjdk11
```

The Flatpak community does a lot of work on the foundations required for applications to run upon in order to make the packaging process easy for developers. For instance, the Kblocks game from the KDE community requires the KDE platform to run, and that's already available from Flatpak. The additional **libkdegames** library is not included, but it's as easy to add it to your list of **modules** as **kblocks** itself.

Here's a manifest for the Kblocks game:

```
id: org.kde.kblocks
command: kblocks
modules:
- buildsystem: cmake-ninja
  name: libkdegames
  sources:
    type: archive
    path: src/libkdegames-19.08.2.tar.xz
- buildsystem: cmake-ninja
  name: kblocks
  sources:
    type: archive
    path: src/kblocks-19.08.2.tar.xz
runtime: org.kde.Platform
runtime-version: '5.13'
sdk: org.kde.Sdk
```

As you can see, the manifest is still straightforward and relatively intuitive. The build system is different, and the runtime and SDK point to KDE instead of the Freedesktop, but the structure and requirements are basically the same.

Because it's a GUI application, however, there are some new options required. First, it needs an icon so that when it's listed in the Activities or Application menu, it looks nice and recognizable. Kblocks includes an icon in its sources, but the names of files exported by a Flatpak must be prefixed using the application ID (such as **`org.kde.Kblocks.desktop`**). The easiest way to do this is to rename the file directly in the application source, which Flatpak can do for you as long as you include this directive in your manifest:

```
rename-icon: kblocks
```

Another unique trait of GUI applications is that they often require integration with common desktop services, like the graphics server (X11 or Wayland) itself, a sound server such as Pulse Audio, and the Inter-Process Communication (IPC) subsystem.

In the case of Kblocks, the requirements are:

```
finish-args:
- --share=ipc
- --socket=x11
- --socket=wayland
- --socket=pulseaudio
- --device=dri
- --filesystem=xdg-config/kdeglobals:ro
```

Here's the final, complete manifest, using URLs for the sources so you can try this on your own system easily:

```
command: kblocks
finish-args:
- --share=ipc
- --socket=x11
- --socket=wayland
- --socket=pulseaudio
- --device=dri
- --filesystem=xdg-config/kdeglobals:ro
id: org.kde.kblocks
modules:
- buildsystem: cmake-ninja
  name: libkdegames
  sources:
  - sha256: 83456cec44502a1f79c0be00c983090e32fd8aea5fec1461fbfbd37b5f8866ac
    type: archive
    url: https://download.kde.org/stable/applications/19.08.2/src/libkdegames-
19.08.2.tar.xz
- buildsystem: cmake-ninja
  name: kblocks
  sources:
  - sha256: 8b52c949e2d446a4ccf81b09818fc90234f2f55d8722c385491ee67e1f2abf93
    type: archive
    url: https://download.kde.org/stable/applications/19.08.2/src/kblocks-
19.08.2.tar.xz
rename-icon: kblocks
runtime: org.kde.Platform
runtime-version: '5.13'
sdk: org.kde.Sdk
```

To build the application, you must have the KDE Platform and SDK Flatpaks (version 5.13 as of this writing) installed. Once the application has been built, you can run it using the **`--run`** method, but to see the application icon, you must install it.

Distributing and installing your Flatpak

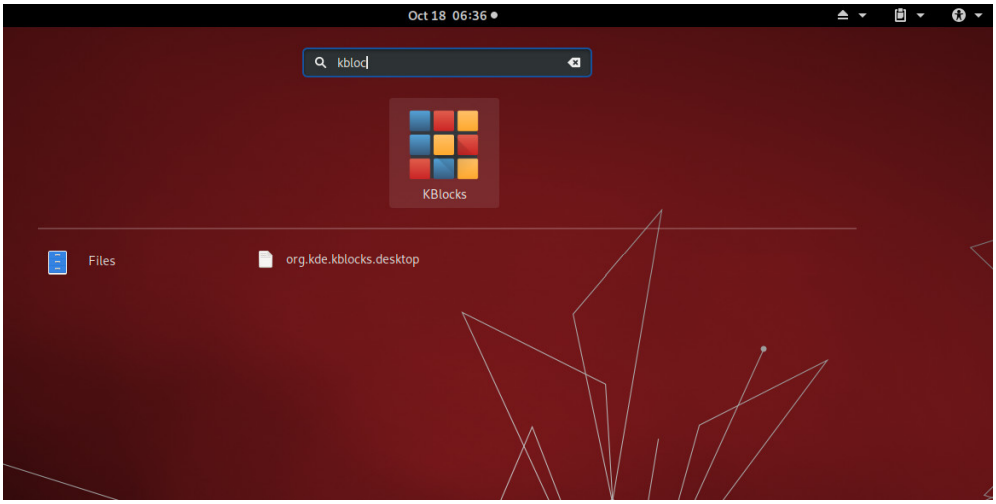
Distributing flatpaks happens through repositories.

You can list your apps on Flathub.org, a community website meant as a technically decentralised (but central in spirit) location for Flatpaks. To submit your Flatpak, place your manifest into a Git repository and submit a pull request on Github. Alternately, you can create your own repository using the **`flatpak build-export`** command.

You can also just install locally:

```
$ flatpak-builder --force-clean --install build-dir org.kde.Kblocks.yaml
```

Once installed, open your Activities or Applications menu and search for Kblocks.



Learning more

The Flatpak documentation site has a good walkthrough on building your first Flatpak. It's worth reading even if you've followed along with this article. Besides that, the docs provide details on what Platforms and SDKs are available.

For those who enjoy learning from examples, there are manifests for every application available on Flathub.

The resources to build and use Flatpaks are plentiful, and Flatpak, along with containers and sandboxed apps, are arguably the future, so get familiar with them, start integrating them with your Jenkins pipelines, and enjoy easy and universal Linux app packaging.