

Developer Documentation

Getting Started

What We Will (And Won't) Cover

We're going to assume that you have absolutely no experience in writing apps for elementary OS. But we will assume some basic programming knowledge, and hopefully a little experience in Vala or at least similarly syntaxed languages. If you're not familiar with Vala, we highly encourage you to brush up on it before coming here. There are great resources (text and video) available for learning Vala on [Valadoc.org](https://valadoc.org).

We're also not covering design too much in this guide; that's what the [Human Interface Guidelines \(HIG\)](#) are for, and you're highly encouraged to take a look there before beginning your app. We're going to assume you have a basic knowledge of (or at least a quick link to) the HIG and focus on coding. However, elementary OS is all about great design and stellar consistency. It's important you grasp these concepts before moving on.

Writing Apps

In this book, we're going to talk about building apps using GTK, Granite, and other tech available in elementary OS, setting up a build system, hosting your code for collaborative development, working with translations, a few other bits and pieces, and finally packaging and distributing your new app.

Some of you may feel confident enough to jump straight into coding. If that's the case, you might want to skip ahead and start writing your first app.

However, we strongly recommend to at least skim through the preparation step. Having the right setup is going to help you reach your goals faster, and having a solid foundation is going to help you throughout the rest of this book.

AppCenter Publishing Requirements

There are also a number of technical, metadata, legal, and other requirements for publishing your app to users via AppCenter.

→ **Publishing Requirements**

</appcenter/publishing-requirements>

Writing Apps

The Basic Setup

Before we even think about writing code, you'll need a certain basic setup. This chapter will walk you through the process of getting set up. We will cover the following topics:

- Creating an account on GitHub and importing an SSH key
- Setting up the Git revision control system
- Getting and using the elementary developer "SDK"

We're going to assume that you're working from a clean installation of elementary OS 5.1 Hera or later. This is important as the instructions you're given may reference apps that are not present (or even available) in other GNU/Linux based operating systems like Ubuntu. It is possible to apply the principles of this guide to Ubuntu development, but it may be more difficult to follow along.

GitHub

GitHub is an online platform for hosting code, reporting issues, tracking milestones, making releases, and more. If you're planning to publish your app through AppCenter, you'll need a GitHub account. If you already have an account, feel free to move on to the next section. Otherwise, [sign up for a GitHub account](#) and return when you're finished.

Git

To download and upload to GitHub, you'll need the Terminal program `git`. Git is a type of [revision control system](#) that allows multiple developers to collaboratively develop and maintain code while keeping track of each revision along the way.

If you're ready, let's get you set up to use Git:

1. Open the Terminal and install Git



```
sudo apt install git
```

2. We need to inform Git who we are so that when we upload code it is attributed correctly. Inform Git of your identity with the following commands

```
1 git config --global user.name "Your Name"  
2 git config --global user.email "You@email.com"
```

3. To authenticate and transfer code securely, you'll need to generate an [SSH](#) key pair (a kind of fingerprint for your computer) and import your public key to GitHub. Type the following in Terminal:

```
ssh-keygen -t rsa
```

4. When prompted, press Enter to accept the default file name for your key. You can choose to protect your key with a password or press Enter again to use no password when pushing code.
5. Now we're going to import your public key to GitHub. View your public SSH key with the following command, then copy the text that appears

```
cat ~/.ssh/id_rsa.pub
```

6. Visit [your SSH keys page](#) and click the green button in the upper right-hand corner that says "New SSH key". Paste your key in the "Key" box and give it a title.

We're all done! Now you can download source code hosted on GitHub and upload your own code. We'll revisit using `git` in a bit, but for now you're set up.



For a more in-depth intro to Git, we recommend [Codecademy's course on Git](#).

Developer "SDK"

At the time of this writing, elementary OS doesn't have a full SDK like Android or iOS. But luckily, we only need a couple apps to get started writing code.

Code



Code icon

The first piece of our "SDK" is Code. This comes by default with elementary OS. It comes with some helpful features like syntax highlighting, auto-save, and a Folder Manager. There are other extensions for Code as well, like the Outline, Terminal, Word Completion, or Devhelp extensions. Play around with what works best for you.

Terminal



Terminal icon

We're going to use Terminal in order to compile our code, push revisions to GitHub (using `git`), and other good stuff. Throughout this guide, we'll be issuing Terminal commands. You should assume that any command is executed from the directory "Projects" in your

home folder unless otherwise stated. Since elementary OS doesn't come with that folder by default, you'll need to create it.

Open Terminal and issue the following command:

```
mkdir Projects
```

Development Libraries



Generic application icon

In order to build apps you're going to need their development libraries. We can fetch a basic set of libraries and other development tools with the following terminal command:

```
sudo apt install elementary-sdk
```

Flatpak

On elementary OS 6 beta and newer, you should already have the required Flatpak remote and platform pre-installed. On earlier versions or other OSes, you can add the remote and install the Flatpak platform and SDK:

```
1 flatpak remote-add --if-not-exists --system appcenter https://flatpak.elementary.io
2 flatpak install -y appcenter io.elementary.Platform io.elementary.Sdk
```


And with that, we're ready to dive into development! Let's move on!

Hello World

The first app we'll create will be a basic and generic "Hello World". We'll walk through the steps of creating folders to store our source code, compiling our first app, and pushing the project to a Git branch. Let's begin.

Setting Up

Apps on elementary OS are organized into standardized directories contained in your project's "root" folder. Let's create a couple of these to get started:

1. Create your root folder called "gtk-hello"
2. Create a folder inside that one called "src". This folder will contain all of our source code.

Later on, We'll talk about adding other directories like "po" and "data". For now, this is all we need.

Gtk.Application

Now what you've been waiting for! We're going to create a window that contains a button. When pressed, the button will display the text "Hello World!" To do this, we're going to use a widget toolkit called GTK+ and the programming language Vala. Before we begin, we highly recommend that you do not copy and paste. Typing each section manually will help you to practice and remember. Let's get started:

Create a new file in Code and save it as "Application.vala" inside your "src" folder

In this file, we're going to create a special class called a `Gtk.Application` .

`Gtk.Application` is a class that handles many important aspects of a Gtk app like uniqueness and the ID you need to identify your app to the notifications server. If you want

some more details about `Gtk.Application` , [check out Valadoc](#). For now, type the following in "Application.vala".

```
1 public class MyApp : Gtk.Application {
2     public MyApp () {
3         Object (
4             application_id: "com.github.yourusername.yourrepositoryname",
5             flags: ApplicationFlags.FLAGS_NONE
6         );
7     }
8
9     protected override void activate () {
10         var main_window = new Gtk.ApplicationWindow (this) {
11             default_height = 300,
12             default_width = 300,
13             title = "Hello World"
14         };
15         main_window.show_all ();
16     }
17
18     public static int main (string[] args) {
19         return new MyApp ().run (args);
20     }
21 }
```

You'll notice that most of these property names are pretty straightforward. Inside `MyApp ()` we set a couple of properties for our `Gtk.Application` object, namely our app's ID and [flags](#). The naming scheme we used for our app's ID is called [Reverse Domain Name Notation](#) and will ensure that your app has a unique identifier. The first line inside the `activate` method creates a new `Gtk.ApplicationWindow` called `main_window`. The fourth line sets the window title that you see at the top of the window. We also must give our window a default size so that it does not appear too small for the user to interact with it. Then in our `main ()` method we create a new instance of our `Gtk.Application` and run it.

Ready to test it out? Fire up your terminal and make sure you're in "`~/Projects/gtk-hello/src`". Then execute the following commands to compile and run your first Gtk+ app:

```
1 valac --pkg gtk+-3.0 Application.vala
```

2 ./Application

Do you see a new, empty window called "Hello World"? If so, congratulations! If not, read over your source code again and look for errors. Also check the output of your terminal. Usually there is helpful output that will help you track down your mistake.

Now that we've defined a nice window, let's put a button inside of it. Add the following to your application at the beginning of the `activate ()` function:

```
1 var button_hello = new Gtk.Button.with_label ("Click me!") {  
2     margin = 12  
3 };  
4  
5 button_hello.clicked.connect (() => {  
6     button_hello.label = "Hello World!";  
7     button_hello.sensitive = false;  
8 });
```

Then add this line right before `main_window.show_all ()`:

```
main_window.add (button_hello);
```

Any ideas about what happened here?

- We've created a new `Gtk.Button` with the label "Click me!"
- Then we add a margin to the button so that it doesn't bump up against the sides of the window.
- We've said that if this button is clicked, we want to change the label to say "Hello World!" instead.
- We've also said that we want to make the button insensitive after it's clicked; We do this because clicking the button again has no visible effect
- Finally, we add the button to our `Gtk.ApplicationWindow` and declare that we want to show all of the window's contents.

Compile and run your application one more time and test it out. Nice job! You've just written your first Gtk+ app!

 If you're having trouble, you can view the full example code [here on GitHub](#)

Pushing to GitHub

After we do anything significant, we must remember to push our code. This is especially important in collaborative development where not pushing your code soon enough can lead to unintentional forks and pushing too much code at once can make it hard to track down any bugs introduced by your code.

First we need to create a new repository on GitHub. Visit [the new repository page](#) and create a new repository for your code.

Open Terminal and make sure you're in your project's root directory "`~/Projects/gtk-hello`", then issue the following commands

```
1 git init
2 git add src/Application.vala
3 git commit -m "Create initial structure. Create window with button."
4 git remote add origin git@github.com:yourusername/yourrepositoryname.git
5 git push -u origin master
```

With these commands:

- We've told `git` to track revisions in this folder
- That we'd like to track revisions on the file "`Application.vala`" specifically
- We've committed our first revision and explained what we did in the revision
- Then we've told `git` to push your code to GitHub.



Remember to replace "yourusername" with your GitHub username and "yourrepositoryname" with the name of the new repository you created

Victory!

Let's recap what we've learned to do in this first section:

- We created a new project containing a "src" folder
- We created our main vala file and inside it we created a new `Gtk.Window` and `Gtk.Button`
- We built and ran our app to make sure that everything worked properly
- Finally, we committed our first revision and pushed code to GitHub

Feel free to play around with this example. Make the window a different size, set different margins, make the button say other things. When you're comfortable with what you've learned, go on to the next section.

A Note About Libraries

Remember how when we compiled our code, we used the `valac` command and the argument `--pkg gtk+-3.0`? What we did there was make use of a "library". If you're not familiar with the idea of libraries, a library is a collection of methods that your program can use. So this argument tells `valac` to include the GTK+ library (version 3.0) when compiling our app.

In our code, we've used the `Gtk` "Namespace" to declare that we want to use methods from GTK (specifically, `Gtk.Window` and `Gtk.Button.with_label`). Notice that there is a hierarchy at play. If you want to explore that hierarchy in more detail, you can [check out Valadoc](#).

Our First App

In the previous chapter, we created a "Hello World!" app to show off our vala and Gtk skills. But what if we wanted to share our new app with a friend? They'd have to know which packages to include with the `valac` command we used to build our app, and after they'd built it they'd have to run it from the build directory like we did. Clearly, we need to do some more stuff to make our app fit for people to use, to make it a *real* app.

Hello (again) World!

To create our first real app, we're going to need all the old stuff that we used in the last example. But don't just copy and paste! Let's take this time to practice our skills and see if we can recreate the last example from memory. Additionally, now that you have the basics, we're going to get a little more complex and a little more organized:

1. Create a new folder inside "~/Projects" called "hello-again". Then, go into "hello-again" and create our directory structure including the "src" folder.
2. Create "Application.vala" in the "src" folder. This time we're going to prefix our file with a small legal header. Make sure this header matches the license of your code and assigns copyright to you. More info about this later.

```
1  /*
2   * SPDX-License-Identifier: GPL-3.0-or-later
3   * SPDX-FileCopyrightText: 2021 Your Name <you@email.com>
4   */
```

3. Now, let's create a `Gtk.Application`, a `Gtk.ApplicationWindow`, and set the window's default properties. Refer back to the last chapter if you need a refresher.
4. For the sake of time let's just put a `Gtk.Label` instead of a `Gtk.Button`. We don't need to try to make the label do anything when you click it.

```
var label = new Gtk.Label ("Hello Again World!");
```

Don't forget to add it to your window and show the window's contents:

```
1 main_window.add (label);  
2 main_window.show_all ();
```

5. Build "Application.vala" just to make sure it all works. If something goes wrong here, feel free to refer back to the last chapter and remember to check your terminal output for any hints.
6. Initialize the branch, add your files to the project, and write a commit message using what you learned in the last chapter. Lastly, make sure you've created a new repository for your project on GitHub push your first revision with `git` :

```
1 git remote add origin git@github.com:yourusername/yourrepositoryname.g  
2 git push -u origin master
```

Everything working as expected? Good. Now, let's get our app ready for other people to use.

The .desktop File

Every app comes with a .desktop file. This file contains all the information needed to display your app in the Applications Menu and in the Dock. Let's go ahead and make one:

1. In your project's root, create a new folder called "data".
2. Create a new file in Code and save it in the "data" folder as "hello-again.desktop".
3. Type the following into your .desktop file. Like before, try to guess what each line does.

```
1 [Desktop Entry]  
2 Name=Hello Again  
3 GenericName=Hello World App
```



```
4 Comment=Proves that we can use Vala and Gtk
5 Categories=Utility;Education;
6 Exec=com.github.yourusername.yourrepositoryname
7 Icon=com.github.yourusername.yourrepositoryname
8 Terminal=false
9 Type=Application
10 Keywords=Hello;World;Example;
```

The first line declares that this file is a "Desktop Entry" file. The next three lines are descriptions of our app: The branded name of our app, a generic name for our app, and a comment that describes our app's function. Next, we categorize our app. Then, we say what command will execute it. Finally, we give our app an icon and let the OS know that this isn't a command line app. For more info about crafting .desktop files, check out [this HIG entry](#).

4. Finally, let's add this file to `git` and commit a revision:

```
1 git add data/hello-again.desktop
2 git commit -am "Add a .desktop file"
3 git push
```

AppData.xml

Every app also comes with an .appdata.xml file. This file contains all the information needed to list your app in AppCenter.

1. In your data folder, create a new file called "hello-again.appdata.xml"
2. Type the following into your .appdata.xml file

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- Copyright 2019 Your Name <you@email.com> -->
3 <component type="desktop">
4   <id>com.github.yourusername.yourrepositoryname</id>
5   <metadata_license>CC0</metadata_license>
6   <name>Your App's Name</name>
7   <summary>A Catchy Tagline</summary>
```

```
8   <description>
9     <p>A quick summary of your app's main selling points and features.
10  </description>
11 </component>
```


These are all the mandatory fields for displaying your app in AppCenter. There are plenty of other [optional fields](#) that you can read about.

There are also some special custom fields for AppCenter to further brand your listing. Specifically, you can set a background color and a text color for your app's header and banner. You can do so by adding the following keys inside the `component` tag:

```
1 <custom>
2   <value key="x-appcenter-color-primary">#603461</value>
3   <value key="x-appcenter-color-primary-text">rgb(255, 255, 255)</value>
4   <value key="x-appcenter-suggested-price">5</value>
5 </custom>
```

You can specify colors here in either Hexadecimal or RGB. The background color will automatically be given a slight gradient in your app's banner.


You can also specify a suggested price in whole USD.

 Remember that AppCenter is a pay-what-you-want store. A suggested price is not a price floor. Users will still be able to choose any price they like, including 0.

Legal Stuff

Since we're going to be putting our app out into the wild, we should include some information about who wrote it and the legal usage of its source code. For this we need a new file in our project's root folder: the `LICENSE` file. This file contains a copy of the license that your code is released under. For elementary OS apps this is typically the [GNU General](#)

Public License (GPL). Remember the header we added to our source code? That header reminds people that your app is licensed and it belongs to you. GitHub has a built-in way to add several popular licenses to your repository. Read their [documentation for adding software licenses](#) and add a `LICENSE` file to your repository.

 If you'd like to better understand software licensing, the Linux Foundation offers a [free online course on open source licensing](#)

Mark Your Progress

Did you remember to add these files to `git` and commit a revision? Each time we add a new file or make a significant change it's a good idea to commit a new revision and push to GitHub. Keep in mind that this acts as a backup system as well; when we push our work to GitHub, we know it's safe and we can always revert to a known good revision if we mess up later.

Now that we've got all these swanky files laying around, we need a way to tell the computer what to do with them. Ready for the next chapter? Let's do this!

The Build System


The next thing we need is a build system. The build system that we're going to be using is called **Meson**. We already installed the `meson` program at the beginning of this book when we installed `elementary-sdk`. What we're going to do in this step is create the files that tell Meson how to install your program. This includes all the rules for building your source code as well as correctly installing your icons, .desktop, and appdata files and the binary app that results from the build process.

Create a new file in your project's root folder called "meson.build". We've included some comments along the way to explain what each section does. You don't have to copy those, but type the rest into that file:

```
1 # project name and programming language
2 project('com.github.yourusername.yourrepositoryname', 'vala', 'c')
3
4 # Create a new executable, list the files we want to compile, list the dep
5 executable(
6     meson.project_name(),
7     'src' / 'Application.vala',
8     dependencies: [
9         dependency('gtk+-3.0')
10    ],
11    install: true
12 )
13
14 # Install our .desktop file so the Applications Menu will see it
15 install_data(
16     'data' / 'hello-again.desktop',
17     install_dir: get_option('datadir') / 'applications',
18     rename: meson.project_name() + '.desktop'
19 )
20
21 # Install our .appdata.xml file so AppCenter will see it
22 install_data(
23     'data' / 'hello-again.appdata.xml',
24     install_dir: get_option('datadir') / 'metainfo',
25     rename: meson.project_name() + '.appdata.xml'
26 )
```

Notice that in each of our `install_data` methods, we rename our files using our project name. By using our project name—and its RDNN scheme—we will ensure that our files are installed under a unique name that won't cause conflicts with other apps.

And you're done! Your app now has a real build system. This is a major milestone in your app's development!

 Don't forget to add these files to `git` and push to GitHub.

Building and Installing with Meson

Now that we have a build system, let's try it out. Configure the build directory using the Meson command in Terminal:

```
meson build --prefix=/usr
```

This command tells Meson to get ready to build our app using the prefix `"/usr"` and that we want to build our app in a clean directory called `"build"`. The `meson` command defaults to installing our app locally, but we want to install our app for all users on the computer.

Change into the build directory and use `ninja` to build. Then, if the build is successful, install with `ninja install`:

```
1 cd build
2 ninja
3 ninja install
```

If all went well, you should now be able to open your app from the Applications Menu and pin it to the Dock. We'll revisit Meson again later to add some more complicated behavior,

but for now this is all you need to know to give your app a proper build system. If you want to explore Meson a little more on your own, you can always check out [Meson's documentation](#).

❗ If you were about to add the "build" folder to your git repository and push it, stop! This binary was built for your computer and we don't want to redistribute it. In fact, we built your app in a separate folder like this so that we can easily delete or ignore the "build" folder and it won't mess up our app's source code.

Review

Let's review all we've learned to do:

- Create a new Gtk app using `Gtk.Window` , `Gtk.Button` , and `Gtk.Label`
- Keep our projects organized into branches
- License our app under the GPL and declare our app's authors in a standardized manner
- Create a .desktop file using RDNN that tells the computer how to display our app in the Applications Menu and the Dock
- Set up a Meson build system that contains all the rules for building our app and installing it cleanly

That's a lot! You're well on your way to becoming a bona fide app developer for elementary OS. Give yourself a pat on the back, then take some time to play around with this example. Change the names of files and see if you can still build and install them properly. Ask another developer to clone your repo from GitHub and see if it builds and installs cleanly on their computer. If so, you've just distributed your first app! When you're ready, we'll move onto the next section: Translations.

i If you're having trouble, you can view the full example code [here on GitHub](#)

Translations

Now that you've learned about Meson, the next step is to make your app able to be translated to different languages. The first thing you need to know is how to mark strings in your code as translatable. Here's an example:

```
1 stdout.printf ("Not Translatable string");
2 stdout.printf (_("Translatable string!"));
3
4 string normal = "Another non-translatable string";
5 string translated = _("Another translatable string");
```

See the difference? We marked the string as translatable by adding `_()` around it. Go back to your project and make all your strings translatable by adding `_()`.

Now we have to make some changes to our Meson build system and add a couple new files to describe which files we want to translate and which languages we want to translate into.

1. Open up your "meson.build" file and add these lines below your project declaration:

```
1 # Include the translations module
2 i18n = import('i18n')
3
4 # Set our translation domain
5 add_global_arguments('-DGETTEXT_PACKAGE="@@"'.format (meson.project_name(),
```

2. Remove the lines that install your .desktop and appdata files and replace them with the following:

```
1 #Translate and install our .desktop file
2 i18n.merge_file(
3     input: 'data' / 'hello-again.desktop.in',
4     output: meson.project_name() + '.desktop',
```

```

5     po_dir: meson.source_root() / 'po',
6     type: 'desktop',
7     install: true,
8     install_dir: get_option('datadir') / 'applications'
9 )
10
11 #Translate and install our .appdata file
12 i18n.merge_file(
13     input: 'data' / 'hello-again.appdata.xml.in',
14     output: meson.project_name() + '.appdata.xml',
15     po_dir: meson.source_root() / 'po',
16     install: true,
17     install_dir: get_option('datadir') / 'metainfo'
18 )

```

The `merge_file` method combines translating and installing files, similarly to how the `executable` method combines building and installing your app. You might have noticed that this method has both an `input` argument and an `output` argument. We can use this instead of the `rename` argument from the `install_data` method.

3. Still in this file, add the following as the last line:

```
subdir('po')
```

4. You might have noticed in step 2 that the `merge_file` method has an `input` and `output`. We're going to append the additional extension `.in` to our `.desktop` and `.appdata.xml` files so that this method can take the untranslated files and produce translated files with the correct names.

```

1 git mv data/hello-again.desktop data/hello-again.desktop.in
2 git mv data/hello-again.appdata.xml data/hello-again.appdata.xml.in

```

We use the `git mv` command here instead of renaming in the file manager or with `mv` so that `git` can keep track of the file rename as part of our revision history.

5. Now, Create a directory named "po" in the root folder of your project. Inside of your po directory you will need to create another "meson.build" file. This time, its contents will

be:

```
1 i18n.gettext(meson.project_name(),
2             args: '--directory=' + meson.source_root(),
3             preset: 'glib'
4 )
```

6. Inside of "po" create another file called "POTFILES" that will contain paths to all of the files you want to translate. For us, this looks like:

```
1 src/Application.vala
2 data/hello-again.desktop.in
3 data/hello-again.appdata.xml.in
```

7. We have one more file to create in the "po" directory. This file will be named "LINGUAS" and it should contain the two-letter language codes for all languages you want to provide translations for. As an example, let's add German and Spanish

```
1 de
2 es
```

8. Now it's time to go back to your build directory and generate some new files using Terminal! The first one is our translation template or `.pot` file:

```
ninja com.github.yourusername.yourrepositoryname-pot
```

After running this command you should notice a new file in the po directory containing all of the translatable strings for your app.

9. Now we can use this template file to generate translation files for each of the languages we listed in the LINGUAS file with the following command:


```
ninja com.github.yourusername.yourrepositoryname-update-po
```

You should notice two new files in your po directory called `de.po` and `es.po`. These files are now ready for translators to localize your app!

10. Last step. Don't forget to add all of the new files we created in the po directory to git:

```
1 git add src/Application.vala meson.build po/ data/
2 git commit -am "Add translations"
3 git push
```

That's it! Your app is now fully ready to be translated. Remember that each time you add new translatable strings or change old ones, you should regenerate your .pot and po files using the `-pot` and `-update-po` build targets from the previous two steps. If you want to support more languages, just list them in the LINGUAS file and generate the new po file with the `-update-po` target. Don't forget to add any new po files to git!

 If you're having trouble, you can view the full example code [here on GitHub](#)

Translators Comments


Sometimes detailed descriptions in the context of translatable strings are necessary for disambiguation or to help in the creation of accurate translations. For these situations use

```
/// TRANSLATORS: comments.
```

```
1 /// TRANSLATORS: The first %s is search term, the second is the name of de
2 title = _("Search for %s in %s").printf (query, get_default_browser_name (
```

Icons

The last thing we need for a minimum-viable-app is to provide app icons. Apps on elementary OS provide icons hinted in 5 sizes: 16px, 32px, 48px, 64px, and 128px. These icons will be shown in AppCenter, the applications menu, the dock, Notifications, System Settings, and many other places throughout the system.

 To help you provide the necessary sizes—and for this tutorial—Micah Ilbery maintains an icon template project [here on GitHub](#)

Place your icons in the data directory and name them after their pixel sizes, such as `32.svg` , `64.svg` , etc. The file structure should look like this:


```
1 hello-again
2   data
3     16.svg
4     32.svg
5     48.svg
6     64.svg
7     128.svg
```

Now that you have icon files in the data directory, add the following lines to the end of `meson.build` to install them .

```
1 # Install our icons in all the required sizes
2 icon_sizes = ['16', '32', '48', '64', '128']
3
4 foreach i : icon_sizes
5     install_data(
6         'data' / i + '.svg',
7         install_dir: get_option('datadir') / 'icons' / 'hicolor' / i + 'x'
8         rename: meson.project_name() + '.svg'
9     )
10    install_data(
11        'data' / i + '.svg',
```

```
12         install_dir: get_option('datadir') / 'icons' / 'hicolor' / i + '.x'
13         rename: meson.project_name() + '.svg'
14     )
15 endforeach
```

You'll notice the section for installing app icons is a little more complicated than installing other files. In this example, we're providing SVG icons in all of the required sizes for AppCenter and, since we're using SVG, we're installing them for both LoDPI and HiDPI. If you're providing PNG icons instead, you'll need to tweak this part a bit to handle assets exported for use on HiDPI displays.

 For more information about creating and hinting icons, check out the [Human Interface Guidelines](#)

Packaging

While having a build system is great, our app still isn't ready for regular users. We want to make sure our app can be built and installed without having to use Terminal. What we need to do is package our app. To do this, we use Flatpak on elementary OS. This section will teach you how to package your app as a Flatpak, which is required to publish apps in AppCenter. This will allow normal people to install your app and even get updates for it when you publish them.

Practice Makes Perfect

If you want to get really good really fast, you're going to want to practice. Repetition is the best way to commit something to memory. So let's recreate our entire Hello World app again *from scratch*:

1. Create a new branch folder "hello-packaging"
2. Set up our directory structure including the "src" and "data" folders.
3. Add your Copying, .desktop, .appdata.xml, icons, and source code.
4. Now set up the Meson build system and translations.
5. Test everything!

Did you commit and push to GitHub for each step? Keep up these good habits and let's get to packaging this app!

Flatpak Manifest

The Flatpak manifest file describes your app's build dependencies and required permissions. Create a `com.github.yourusername.yourrepositoryname.yml` file in your project root with the following contents:

```
1 # This is the same ID that you've used in meson.build and other files
```

```

2  app-id: com.github.yourusername.yourrepositoryname
3
4  # Instead of manually specifying a long list of build and runtime dependen
5  # we can use a convenient pre-made runtime and SDK. For this example, we'll
6  # using the runtime and SDK provided by elementary.
7  runtime: io.elementary.Platform
8  runtime-version: 'daily'
9  sdk: io.elementary.Sdk
10
11 # This should match the exec line in your .desktop file and usually is the
12 # as your app ID
13 command: com.github.yourusername.yourrepositoryname
14
15 # Here we can specify the kinds of permissions our app needs to run. Since
16 # not using hardware like webcams, making sound, or reading external files
17 # only need permission to draw our app on screen using either X11 or Wayla
18 finish-args:
19   - '--share=ipc'
20   - '--socket=fallback-x11'
21   - '--socket=wayland'
22
23 # This section is where you list all the source code required to build you
24 # If we had external dependencies that weren't included in our SDK, we wou
25 # them here.
26 modules:
27   - name: yourrepositoryname
28     buildsystem: meson
29     sources:
30       - type: dir
31         path: .

```

Note that we're using the `daily` version of the SDK for now, as there has not yet been a stable release. To run a test build and install your app, we can use `flatpak-builder` with a few arguments:

```
flatpak-builder build com.github.yourusername.yourrepositoryname.yml --user
```

This tells Flatpak Builder to build the manifest we just wrote into a clean `build` folder the same as we did for Meson. Plus, we install the built Flatpak package locally for our user. If all goes well, congrats! You've just built and installed your app as a Flatpak.

That wasn't too bad, right? We'll set up more complicated packaging in the future, but this is all that is required to submit your app to AppCenter Dashboard for it to be built, packaged, and distributed. If you'd like you can always read [more about Flatpak](#).

Creating Layouts

Now that you know how to code, build, and package an app using Vala, Gtk, Meson, and Flatpak, it's time to learn a little bit more about how to build out your app into something really useful. The first thing we need to learn is how to lay out widgets in our window. But we have a fundamental problem: We can only add one widget (one "child") to `Gtk.Window`. So how do we get around that to create complex layouts in a Window? We have to add a widget that can contain multiple children. One of those widgets is `Gtk.Grid`.

Widgets Subclass Other Widgets

Before we get into `Gtk.Grid`, let's stop for a second and take some time to understand Gtk a little better. At the lower level, Gtk has classes that define some pretty abstract traits of widgets such as `Gtk.Container` and `Gtk.Orientable`. These aren't widgets that we're going to use directly in our code, but they're used as building blocks to create the widgets that we do use. It's important that we understand this, because it means that when we understand how to add children to a `Gtk.Container` like `Gtk.Grid`, we also understand how to add children to a `Gtk.Container` like `Gtk.Toolbar`. Both Grid and Toolbar are widgets that are subclasses of the more abstract class `Gtk.Container`.

If you want to understand more about these widgets and the parts of Gtk that they subclass, jump over to [Valadoc](#) and search for a widget like `Gtk.Grid`. See that big tree at the top of the page? It shows you every component of Gtk that `Gtk.Grid` subclasses and even what those components subclass. Having a lower level knowledge of Gtk will help you to implement widgets you haven't worked with before since you will understand how their parent classes work.

Gtk.Grid

Now that we've gotten that out of the way, let's get back to our Window and `Gtk.Grid`. Since you're a master developer now, you can probably set up a new project complete with

Meson, push it to GitHub, and create a Flatpak manifest in your sleep. If you want the practice, go ahead and do all of that again. Otherwise, it's probably convenient for our testing purposes to just play around locally and build from Terminal. So code up a nice `Gtk.Window` without anything in it and make sure that builds. Ready? Let's add a Grid.

Just like when we add a Button or Label, we need to create our `Gtk.Grid`. As always, don't copy and paste! Practice makes perfect. We create a new `Gtk.Grid` like this:

```
1 var grid = new Gtk.Grid () {  
2     orientation = Gtk.Orientation.VERTICAL  
3 };
```

Remember that Button and Label accepted an argument (a String) in the creation method (that's the stuff in parentheses and quotes). As shown above, `Gtk.Grid` doesn't accept any arguments in the creation method. However, you can still change the grid's properties (like `orientation`) as we did on the second line. Here, we've declared that when we add widgets to our grid, they should stack vertically.

Let's add some stuff to the Grid:

```
1 grid.add (new Gtk.Label (_("Label 1")));  
2 grid.add (new Gtk.Label (_("Label 2")));
```

We can add the grid to our window using the same method that we just used to add widgets to our grid:

```
main_window.add (grid);
```

Now build your app and see what it looks like. Since we've given our grid a `Gtk.Orientation` of `VERTICAL` the labels stack up on top of each other. Try creating a

`Gtk.Grid` without giving it an orientation. By default, `Gtk.Grid` 's orientation is horizontal. You really only ever have to give it an orientation if you need it to be vertical.

Functionality in Gtk.Grid

Okay, so you know all about using a `Gtk.Grid` to pack multiple children into a Window. What about using it to lay out some functionality in our app? Let's try building an app that shows a message when we click a button. Remember in our first "Hello World" how we changed the label of the button with `button.clicked.connect` ? Let's use that method again, but instead of just changing the label of the button, we're going to use it to change an empty label to a message.

Let's create a Window with a vertical Grid that contains a Button and a Label:

```
1 var button = new Gtk.Button.with_label (_("Click me!"));
2
3 var label = new Gtk.Label (null);
4
5 var grid = new Gtk.Grid () {
6     orientation = Gtk.Orientation.VERTICAL,
7     row_spacing = 6
8 };
9 grid.add (button);
10 grid.add (label);
11
12 main_window.add (grid);
```

This time when we created our grid, we gave it another property: `row_spacing` . We can also add `column_spacing` , but since we're stacking widgets vertically we'll only see the effect of `row_spacing` . Notice how we can create new widgets outside the grid and then pack them into the grid by name. This is really helpful when you start using different methods to change the properties of your widgets.

Now, let's hook up the button to change that label. To keep our code logically separated, we're going to add it below `main_window.add (grid);` . In this way, the first portion of our

code defines the UI and the next portion defines the functions that we associated with the UI:

```
1 button.clicked.connect (() => {  
2     label.label = _("Hello World!");  
3     button.sensitive = false;  
4 });
```

Remember, we set the button as insensitive here because clicking it again has no effect. Now compile your app and marvel at your newfound skills. Play around with orientation and spacing until you feel comfortable.

The Attach Method

While we can use `Gtk.Grid` to create single row or single column layouts with the `add` method, we can also use it to create row-and-column-based layouts with the `attach` method. First we're going to create all the widgets we want to attach to our grid, then we'll create a new `Gtk.Grid` and set both column and row spacing, and finally we'll attach our widgets to the grid.

```
1 var hello_button = new Gtk.Button.with_label (_("Say Hello"));  
2 var hello_label = new Gtk.Label (null);  
3  
4 var rotate_button = new Gtk.Button.with_label (_("Rotate"));  
5 var rotate_label = new Gtk.Label (_("Horizontal"));  
6  
7 var grid = new Gtk.Grid () {  
8     column_spacing = 6,  
9     row_spacing = 6  
10 };
```

Make sure to give the Grid, Buttons, and Labels unique names that you'll remember. It's best practice to use descriptive names so that people who are unfamiliar with your code can understand what a widget is for without having to know your app inside and out.

```

1 // add first row of widgets
2 grid.attach (hello_button, 0, 0, 1, 1);
3 grid.attach_next_to (hello_label, hello_button, Gtk.PositionType.RIGHT, 1,
4
5 // add second row of widgets
6 grid.attach (rotate_button, 0, 1);
7 grid.attach_next_to (rotate_label, rotate_button, Gtk.PositionType.RIGHT,
8
9 main_window.add (grid);

```

Notice that the attach method takes 5 arguments:

1. The widget that you want to attach to the grid.
2. The column number to attach to starting at 0.
3. The row number to attach to starting at 0.
4. The number of columns the widget should span.
5. The number of rows the widget should span.

You can also use `attach_next_to` to place a widget next to another one on [all four sides](#).

Note also that providing the number of rows and columns the widget should span is optional. If you supply only the column and row numbers, Gtk will assume that the widget will span 1 column and 1 row.

Don't forget to add the functionality associated with our buttons:

```

1 hello_button.clicked.connect (() => {
2     hello_label.label = _("Hello World!");
3     hello_button.sensitive = false;
4 });
5
6 rotate_button.clicked.connect (() => {
7     rotate_label.angle = 90;
8     rotate_label.label = _("Vertical");
9     rotate_button.sensitive = false;
10 });

```

You'll notice in the example code above that we've created a 2 x 2 grid with buttons on the left and labels on the right. The top label goes from blank to "Hello World!" and the button

label is rotated 90 degrees. Notice how we gave the buttons labels that directly call out what they do to the other labels.

Review

Let's recap what we learned in this section:

- We learned about the building blocks of Gtk and the importance of subclasses
- We packed multiple children into a Window using `Gtk.Grid`
- We set the properties of `Gtk.Grid` including its orientation and spacing
- We added multiple widgets into a single `Gtk.Grid` using the `attach` method to create complex layouts containing Buttons and Labels that did cool stuff.

Now that you understand more about Gtk, Grids, and using Buttons to alter the properties of other widgets, try packing other kinds of widgets into a window like a Toolbar and changing other properties of **Labels** like `width_chars` and `ellipsize`. Don't forget to play around with the `attach` method and widgets that span across multiple rows and columns. Remember that Valadoc is super helpful for learning more about the methods and properties associated with widgets.

Code Style

Internally, elementary uses the following code style guide to ensure that code is consistently formatted both internally and across projects. Consistent and easily-legible code makes it easier for newcomers to learn and contribute. We'd like to recommend that in the spirit of Open Source collaboration, all Vala apps written in the wider ecosystem also follow these guidelines.

Whitespace

White space comes before opening parentheses or braces:

```
1 public string get_text () {}
2 if (a == 5) {
3     return 4;
4 }
5
6 for (i = 0; i < maximum; i++) {}
7 my_function_name ();
8 var my_instance = new Object ();
```

Whitespace goes between numbers and operators in all math-related code.

```
c = (n * 2) + 4;
```

Lines consisting of closing brackets (`}` or `)`) should be followed by an empty line, except when followed by another closing bracket or an `else` statement.

```
1 if (condition) {
2     // ...
3 } else {
4     // ...
```

```
5  }  
6  
7  // other code
```

Indentation

Vala

Vala code is indented using 4 spaces for consistency and readability.

In classes, functions, loops and general flow control, the first brace is on the end of the first line ("One True Brace Style"), followed by the indented code, and a line closing the function with a brace:

```
1  public int my_function (int a, string b, long c, int d, int e) {  
2      if (a == 5) {  
3          b = 3;  
4          c += 2;  
5          return d;  
6      }  
7  
8      return e;  
9  }
```

On conditionals and loops, always use braces even if there's only one line of code:

```
1  if (my_var > 2) {  
2      print ("hello\n");  
3  }
```

Cuddled else and else if:

```

1  if (a == 4) {
2      b = 1;
3      print ("Yay");
4  } else if (a == 3) {
5      b = 3;
6      print ("Not so good");
7  }

```

If you are checking the same variable more than twice, use switch/case instead of multiple else/if:

```

1  switch (week_day) {
2      case "Monday":
3          message ("Let's work!");
4          break;
5      case "Tuesday":
6      case "Wednesday":
7          message ("What about watching a movie?");
8          break;
9      default:
10         message ("You don't have any recommendation.");
11         break;
12 }

```

Markup

Markup languages like HTML, XML, and YAML should use two-space indentation since they are much more verbose and likely to hit line-length issues sooner.

```

1  <component type="desktop">
2      <name>Calendar</name>
3      <description>
4          <p>A slim, lightweight calendar app that syncs and manages multiple ca
5      </description>
6      <releases>
7          <release version="5.0" date="2019-02-28" urgency="medium">
8              <description>
9                  <p>Add a search entry for calendars</p>
10             </description>

```



```
11     </release>
12 </releases>
13 <screenshots>
14     <screenshot type="default">
15         <image>https://raw.githubusercontent.com/elementary/calendar/master/
16     </screenshot>
17 </screenshots>
18 </component>
```

Classes and Files

A file should only contain one public class.

All files have the same name as the class in them. For example, a file containing the class `AbstractAppGrid` should be called "AbstractAppGrid.vala"

Classes should be named in a descriptive way, and include the name of any parent classes. For example, a class that subclasses `Gtk.ListBoxRow` and displays the names of contacts should be called `ContactRow`.

Comments

Comments are either on the same line as the code they reference or in a special line.

Comments are indented alongside the code, and obvious comments do more harm than good.

```
1  /* User chose number five */
2  if (c == 5) {
3      a = 3;
4      b = 4;
5      d = -1; // Values larger than 5 are undefined
6  }
```

Sometimes detailed descriptions in the context of translatable strings are necessary for disambiguation or to help in the creation of accurate translations. For these situations use

```
/// TRANSLATORS: comments.
```

```
1  /// TRANSLATORS: The first %s is search term, the second is the name of de
2  title = _("Search for %s in %s").printf (query, get_default_browser_name (
```

Variable, Class, and Function Names

Variable and function names are all lower case and separated by underscores:

```
1  var my_variable = 5;
2
3  public void my_function_name () {
4      do_stuff ();
5  }
```

Classes and enums are Upper Camel Case (aka Pascal Case):

```
1  public class UserListBox : Gtk.ListBox {
2      private enum OperatingSystem {
3          ELEMENTARY_OS,
4          UBUNTU
5      }
6  }
```

Constants and enum members should be all uppercase and separated by underscores:

```
1  public const string UBUNTU = "ubuntu";
2
3  private enum OperatingSystem {
```

```
4     ELEMENTARY_OS,  
5     UBUNTU  
6 }
```

The values of constant strings (such as when used with `GLib.Action`) should be lowercase and separated with dashes:

```
public const string ACTION_GO_BACK = "action-go-back";
```

Casting

Avoid using `as` keyword when casting as it might give `null` as result, which could be forgotten to check.

```
1  /* OK */  
2  ((Gtk.Entry) widget).max_width_chars  
3  
4  /* NOT OK as this approach requires a check for null */  
5  (widget as Gtk.Entry).max_width_chars
```

Prefer Properties Over Get/Set Methods

In places or operations where you would otherwise use `get` or `set`, you should make use of `=` instead.

For example, instead of using

```
set_can_focus (false);
```

you should use

```
can_focus = false;
```

Initialize Objects with Properties

This is especially clearer when initializing an object with many properties. Avoid the following

```
1 var label = new Gtk.Label ("Test Label");
2 label.set_ellipsize (Pango.EllipsizeMode.END);
3 label.set_valign (Gtk.Align.END);
4 label.set_width_chars (33);
5 label.set_xalign (0);
```

and instead do this

```
1 var label = new Gtk.Label ("Test Label") {
2     ellipsize = Pango.EllipsizeMode.END,
3     valign = Gtk.Align.END,
4     width_chars = 33,
5     xalign = 0
6 };
```

Create Classes with Properties

This goes for creating methods inside of classes as well. Instead of

```
1 private int _number;
2
3 public int get_number () {
4     return _number;
5 }
6
```

```
7 public void set_number (int value) {  
8     _number = value;  
9 }
```

you should use

```
public int number { get; set; }
```

or, where you need some extra logic in the getters and setters:

```
1 private int _number;  
2 public int number {  
3     get {  
4         // We can run extra code here before returning the property. For e  
5         // we can multiply it by 2  
6         return _number * 2;  
7     }  
8     set {  
9         // We can run extra code here before/after updating the value. For  
10        // we could check the validity of the new value or trigger some ot  
11        // changes  
12        _number = value;  
13    }  
14 }
```

Preferring properties in classes enables the use of `GLib.Object.bind_property ()` between classes instead of needing to create signals and handle changing properties manually.

Vala Namespaces

Referring to GLib is not necessary. If you want to print something instead of:

```
GLib.print ("Hello World");
```

You can use

```
print ("Hello World");
```

String Formatting

Avoid using literals when formatting strings:

```
var string = @"Error parsing config: ${config_path}";
```

Instead, prefer printf style placeholders:

```
var string = "Error parsing config: %s".printf (config_path);
```

Warnings in Vala use printf syntax by default:

```
critical ("Error parsing config: %s", config_path);
```

GTK events

Gtk widgets are intended to respond to click events that can be described as "press + release" instead of "press". Usually it is better to respond to `toggle` and `release` events instead of `press` .

Columns Per Line

Ideally, lines should have no more than 80 characters per line, because this is the default terminal size. However, as an exception, more characters could be added, because most people have wide-enough monitors nowadays. The hard limit is 120 characters.

Splitting Arguments Into Lines

For methods that take multiple arguments, it is not uncommon to have very long line lengths. In this case, treat parenthesis like brackets and split lines at commas like so:

```
1 var message_dialog = new Granite.MessageDialog.with_image_from_icon_name (  
2     "Basic Information and a Suggestion",  
3     "Further details, including information that explains any unobvious co  
4     "phone",  
5     Gtk.ButtonType.CANCEL  
6 );
```

EditorConfig

If you are using elementary Code or your code editor supports [EditorConfig](https://EditorConfig.org), you can use this as a default `.editorconfig` file in your projects:

```
1 # EditorConfig <https://EditorConfig.org>  
2 root = true  
3  
4 # elementary defaults  
5 [*]
```

```
6  charset = utf-8
7  end_of_line = lf
8  indent_size = tab
9  indent_style = space
10 insert_final_newline = true
11 max_line_length = 80
12 tab_width = 4
13
14 # Markup files
15 [{*.html,*.xml,*.xml.in,*.yaml}]
16 tab_width = 2
```


APIs

Actions

GTK and GLib have a powerful API called `GLib.Action` which can be used to define the primary actions of your app, assign them keyboard shortcuts, and tie them to `Actionable` widgets in your app like Buttons and Menu Items. In this section, we're going to create a Quit action for your app with an assigned keyboard shortcut and a Button that shows that shortcut in a tooltip.

Gtk.HeaderBar

Begin by creating a `Gtk.Application` with a `Gtk.ApplicationWindow` as you've done in previous examples. Once you have that set up, let's create a new `Gtk.HeaderBar`. Typically your app will have a HeaderBar, at the top of the window, which will contain tool items that users will interact with to trigger your app's actions.

```
1  protected override void activate () {
2      var headerbar = new Gtk.HeaderBar () {
3          show_close_button = true
4      };
5
6      var main_window = new Gtk.ApplicationWindow (this) {
7          default_height = 300,
8          default_width = 300,
9          title = "Actions"
10     };
11     main_window.set_titlebar (headerbar);
12     main_window.show_all ();
13 }
```


Since we're using this HeaderBar as our app's main titlebar, we need to set `show_close_button` to `true` so that GTK knows to include window controls. We can then override our Window's built-in titlebar with the `set_titlebar ()` method.

Now, still in the activate function, let's create a new `Gtk.Button` with a big colorful icon and add it to our headerbar:

```

1  protected override void activate () {
2      var button = new Gtk.Button.from_icon_name ("process-stop", Gtk.IconSi
3
4      var headerbar = new Gtk.HeaderBar () {
5          show_close_button = true
6      };
7      headerbar.add (button);
8
9      [...]
10 }

```

 elementary OS ships with a large set of system icons that you can use in your app for actions, status, and more. You can browse the full set of named icons using the app [LookBook](#), available in AppCenter.

If you compile your app, you can see that it now has a custom HeaderBar with a big red icon in it. But when you click on it, nothing happens.

GLib.SimpleAction

Let's define a new Quit action by adding the following to the beginning of the `activate ()` method

```

1  var quit_action = new SimpleAction ("quit", null);
2
3  add_action (quit_action);
4  set_accels_for_action ("app.quit", {"<Control>q", "<Control>w"});

```

and this to the end of the `activate ()` method:

```

1  quit_action.activate.connect (() => {

```

```
2     main_window.destroy ();
3 });
```


You'll notice that we do a few things here:

- Instantiate a new `GLib.SimpleAction` with the name "quit"
- Add the action to our `Gtk.Application`'s `ActionMap`
- Set the "accelerators" (keyboard shortcuts) for "app.quit" to `<Control>q` and `<Control>w`. Notice that the action name is prefixed with `app`; this refers to the `ActionMap` built in to `Gtk.Application`
- Connect to the `activate` signal of our `SimpleAction` and call `destroy ()` on `main_window`. This must be at the end of `activate ()` because of that reference to `main_window`

and now we can tie the action to the HeaderBar Button by assigning the `action_name` property of our Button:

```
1 var button = new Gtk.Button.from_icon_name ("process-stop", Gtk.IconSize.L
2     action_name = "app.quit"
3 });
```

Compile your app again and see that you can now quit the app either through the defined keyboard shortcuts or by clicking the Button in the HeaderBar.

 Accelerator strings follow a format defined by `Gtk.accelerator_parse`. You can find a list of key values [on Valadoc](#)

Granite.markup_accel_tooltip

There's one more thing we can do here to help improve your app's usability. You may have noticed that in elementary apps you can hover your pointer over tool items to see a description of the button and any available keyboard shortcuts associated with it. We can add the same thing to our Button with `Granite.markup_accel_tooltip ()`.

First, make sure you've included Granite in the build dependencies declared in your meson.build file:


```
1 executable(
2     meson.project_name(),
3     'src/Application.vala',
4     dependencies: [
5         dependency('granite'),
6         dependency('gtk+-3.0')
7     ],
8     install: true
9 )
```

Then, set the `tooltip_markup` property of your HeaderBar Button:

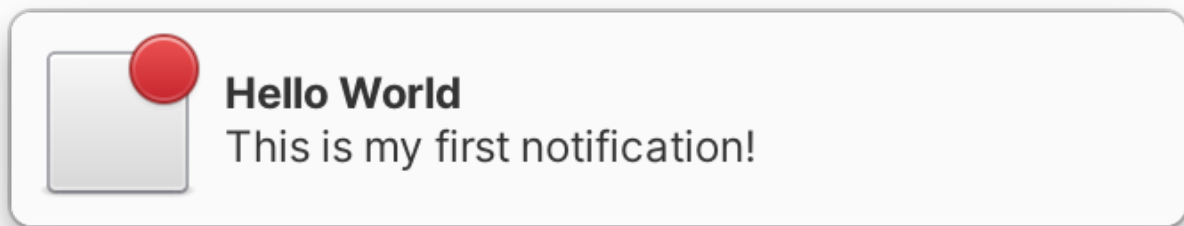
```
1 var button = new Gtk.Button.from_icon_name ("process-stop", Gtk.IconSize.L
2     action_name = "app.quit",
3     tooltip_markup = Granite.markup_accel_tooltip (
4         get_accels_for_action ("app.quit"),
5         "Quit"
6     )
7 };
```

It may now be clear why we needed to declare our action at the beginning of `activate ()`: before we can get a list of the accelerators associated with the action, we have to define those accelerations and add them to the Application.

Compile your app one last time and hover over the HeaderBar Button to see its description and associated keyboard shortcuts.

 If you're having trouble, you can view the full example code [here on GitHub](#)

Notifications



By now you've probably already seen the notification bubbles that appear on the top right of the screen. Notifications are a way to update someone about the state of your app. For example, they can inform the user that a long running background process has been completed or a new message has arrived. In this section we are going to show you just how to get them to work in your app. Let's begin by making a new project!

Making Preparations

1. Create a new folder inside of "~/Projects" called "notifications-app"
2. Create a new folder inside of that folder called "src" and add a file inside of it called `Application.vala`
3. Create a `meson.build` file. If you don't remember how to set up Meson, go back to the [previous section](#) and review.
4. Remember how to [make a .desktop file](#)? Excellent! Make one for this project, but this time, since your app will be displaying notifications, add `X-GNOME-UsesNotifications=true` to the end of the file. This is needed so that users will be able to set notification preferences for your app in the system's notification settings.

When using notifications, it's important that your desktop file has the same name as your application's ID. This is because elementary OS uses desktop files to find extra information about the app who sends the notification such as a default icon, or the name of the app. To keep things simple, we'll be using the same RDNN everywhere.



If you don't have a desktop file whose name matches the application id, your notification might not be displayed.

Yet Another Application

In order to display notifications, you're going to need another `Gtk.Application` with a `Gtk.ApplicationWindow`. Remember what we learned in the last few sections and set up a new `Gtk.Application` !

Now that we have an empty window, let's use what we learned in [creating layouts](#) and make a grid containing one button that will show a notification.

In between `var main_window...` and `main_window.show ();` , write the following lines of code:

```
1 var title_label = new Gtk.Label (_("Notifications"));
2 var show_button = new Gtk.Button.with_label (_("Show"));
3
4 var grid = new Gtk.Grid ();
5 grid.orientation = Gtk.Orientation.VERTICAL;
6 grid.row_spacing = 6;
7 grid.add (title_label);
8 grid.add (show_button);
9
10 main_window.add (grid);
11 main_window.show_all ();
```

Since we're adding translatable strings, don't forget to update your translation template by running `ninja com.github.yourusername.yourrepositoryname-pot` .

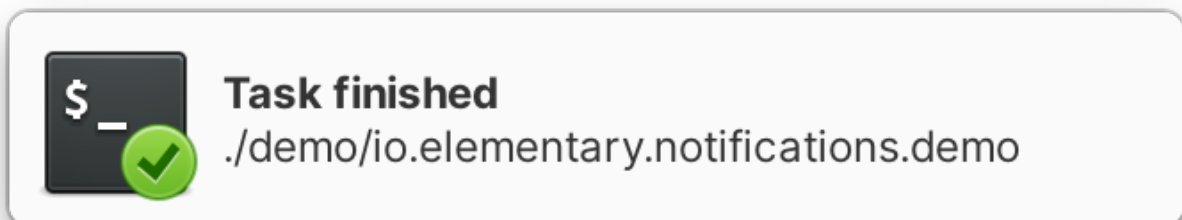
Sending Notifications

Now that we have a `Gtk.Application` we can send notifications. Let's connect a function to the button we created and use it to send a notification:

```
1 show_button.clicked.connect (() => {  
2     var notification = new Notification (_("Hello World"));  
3     notification.set_body (_("This is my first notification!"));  
4  
5     send_notification ("com.github.yourusername.yourrepositoryname", notif  
6 });
```

Okay, now compile your new app. if everything works, you should see your new app. Click the "Send" button. Did you see the notification? Great! Don't forget to commit and push your project in order to save your branch for later.

Badge Icons




Notifications will automatically contain your app's icon, but you can add additional context by setting a badge icon. Right after the line containing

```
var notification = New Notification , add:
```

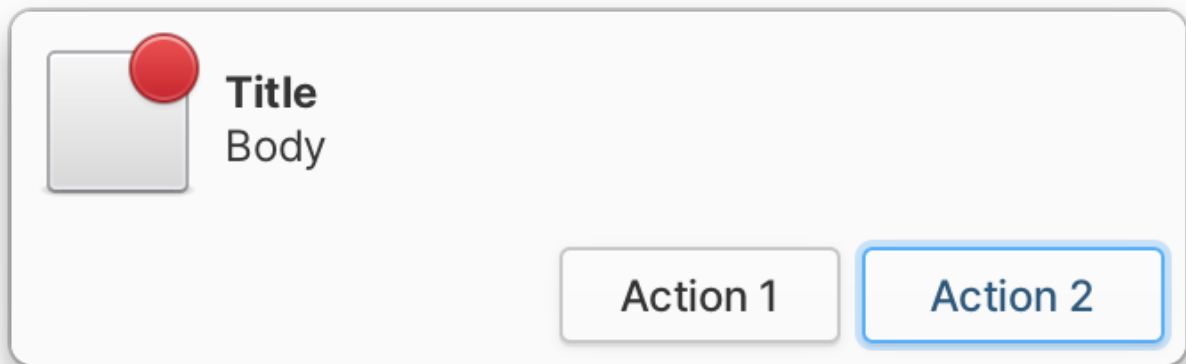
```
notification.set_icon (new ThemedIcon ("process-completed"));
```

Compile your app again, and press the "Send" button. As you can see, the notification now has a smaller badged icon placed over your app's icon. Using this method, you can set the

icon to any of the named icons shipped with elementary OS.

 You can browse the full set of named icons using the app [LookBook](#), available in AppCenter.

Buttons




You can also add buttons to notifications that will trigger actions defined in the `app` namespace. To add a button, first define an action in your Application class as we did in [the actions section](#).

```
1 var quit_action = new SimpleAction ("quit", null);
2
3 add_action (quit_action);
4
5 quit_action.activate.connect (() => {
6     main_window.destroy ();
7 });
```

Now, we can add a button to the notification with a translatable label and the action ID.

```
notification.add_button (_("Quit"), "app.quit");
```

Compile your app again, and press the "Send" button. Notice that the notification now has a button with the label "Quit" and clicking it will close your app.

 Remember that `SimpleAction` s added in the `Application` class with `add_action ()` are automatically added in the `app` namespace. Notifications can't trigger actions defined in other namespaces like `win` .

Priority

Notifications also have priority. When a notification is set as `URGENT` it will stay on the screen until either the user interacts with it, or you withdraw it. To make an urgent notification, add the following line before the `send_notification ()` function

```
notification.set_priority (NotificationPriority.URGENT);
```

`URGENT` notifications should really only be used on the most extreme cases. There are also [other notification priorities](#).


Replace

We now know how to send a notification, but what if you need to update it with new information? Thanks to the notification ID, we can replace a notification with a matching ID. This ID can be anything, but for the purposes of this demo, we're using our app ID.

Let's make the replace button. This button will replace the current notification with one with different information. Let's create a new button for it, and add it to the grid:

```
1 var replace_button = new Gtk.Button.with_label (_("Replace"));
2
3 grid.add (replace_button);
4
5 replace_button.clicked.connect (() => {
6     var notification = new Notification (_("Hello Again"));
7     notification.set_body (_("This is my second Notification!"));
8
9     send_notification ("com.github.yourusername.yourrepositoryname", notif
10 });
```

Compile and run your app again. Click on the buttons, first on "Show", then "Replace". See how the text on your notification changes without making a new one appear?

 You can replace the contents of specific types of notifications your app sends by assigning them a unique ID per category. For example, you can replace the contents of an urgent notification with the ID `alert`, without replacing the contents of a regular notification with a different ID `update`

Review

Let's review what all we've learned:

- We built an app that sends and updates notifications.
- Notifications automatically get our app's icon, but we can also add a badge icon
- We can add buttons that trigger actions in the `app` namespace
- Notification can have a priority which affects their behavior
- We can replace outdated notifications by setting a replaces ID

As you can see, notifications have a number of advanced features and can automatically inherit some information from `Gtk.Application` . If you need some further reading on notifications, Check out the page about `Glib.Notification` on [Valadoc](#).

Launchers

Applications can show additional information in the dock as well as the application menu. This makes the application feel more integrated into the system and give user it's status at a glance. See [HIG for Dock integration](#) for what you should do and what you shouldn't.

For this integration you can use the [Granite.Services.Application API](#). Since it uses the same D-Bus path as the [Unity Launcher API](#), the API can work across many different distributions as it is widely supported by third party applications.

Current API support:

Service	Badge Counter	Progress Bar	Static Quicklist
Application menu	Yes	No	Yes
Dock	Yes	Yes	Yes

Setting Up

Before writing any code, you must add the library Granite to your build system. We already installed this library during [The Basic Setup](#) when we installed `elementary-sdk`. Open your `meson.build` file and add the new dependency to the `executable` method.

```
1 executable(  
2     meson.project_name(),  
3     'src/Application.vala',  
4     dependencies: [  
5         dependency('gtk+-3.0'),  
6         dependency('granite', version: '>=5.2.4') # 5.2.4 is the first rel  
7     ],  
8     install: true  
9 )
```

Your app must also be a `Gtk.Application` with a correctly set `application_id` as we previously set up in Hello World.

Though we haven't made any changes to our source code yet, change into your build directory and run `ninja` to build your project. It should still build without any errors. If you do encounter errors, double check your changes and resolve them before continuing.

Once you've set up `granite` in your build system and created a new `Gtk.Application` with an `application_id`, it's time to write some code.

Badges

Showing a badge in the dock and Applications Menu with the number `12` can be done with the following lines:

```
1 Granite.Services.Application.set_badge_visible.begin (true);  
2 Granite.Services.Application.set_badge.begin (12);
```

Keep in mind you have to set the `set_badge_visible` property to true, and use an int64 type for the `set_badge` property. The suffix `.begin` is required here since these are asynchronous methods.

Progress Bars

The same goes for showing a progress bar, here we show a progress bar showing 20% progress:

```
1 Granite.Services.Application.set_progress_visible.begin (true);  
2 Granite.Services.Application.set_progress.begin (0.2f);
```

As you can see, the method `set_progress` takes a `double` value type and is a range between `0` and `1` : from 0% to 100%. As with badges, Don't forget that `set_progress_visible` must be `true` and `.begin` is required for asynchronous methods.

Static Quicklists

Static quicklists do not involve writing any code or using any external dependencies.

Static quicklists are stored within your `.desktop` file. These are so called "actions". You can define many actions in your desktop file that will always show as an action in the application menu as well as in the dock.

The format is as follows:

```
1 [Desktop Action ActionID]
2 Name=The name of the action
3 Icon=The icon of the action (optional)
4 Exec=The path to application executable and it's command line arguments (o
```

Let's take a look at an example of an action that will open a new window of your application:

```
1 [Desktop Entry]
2 Name=Application name
3 Exec=application-executable
4 ...
5
6 [Desktop Action NewWindow]
7 Name=New Window
8 Exec=application-executable -n
```


Note that adding `-n` or any other argument will not make your application magically open a new window. It is up to your application to handle and interpret command line arguments. The [GLib.Application API](#) provides many examples and an extensive documentation on how to handle these arguments, particularly the [command_line signal](#).

Please take a look at a [freedesktop.org Additional applications actions section](#) for a detailed description of what keys are supported and what they do.

State Saving

Apps should automatically save their state in elementary OS, allowing a user to re-open a closed app and pick up right where they left off. To do so, we utilize `GSettings` via `GLib.Settings`. `GSettings` allows your app to save certain stateful information in the form of booleans, strings, arrays, and more. It's a great solution for window size and position as well as whether certain modes are enabled or not. Note that `GSettings` is ideal for small amounts of configuration or stateful data, but user data (i.e. documents) should be stored on the disk.

For the simplest example, let's create a switch in your app, and save its state.

Define Settings Keys

First, you need to define what settings your app will use so they can be accessed by your app. In your `data/` folder, create a new file named `gschema.xml`. Inside it, let's define a key for the switch:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <schemalist>
3   <schema path="/com/github/yourusername/yourrepositoryname/" id="com.github
4     <key name="useless-setting" type="b">
5       <default>false</default>
6       <summary>Useless Setting</summary>
7       <description>Whether the switch is toggled</description>
8     </key>
9   </schema>
10 </schemalist>
```

The schema's `path` and `id` attributes are your app's ID (`path` in a `/` format while `id` is the standard dot-separated format). Note the key's `name` and `type` attributes: the `name` is a string to reference the setting, while in this case `type="b"` defines the setting as a boolean. The key's `summary` and `description` are developer-facing and are exposed in developer tools like `dconf Editor`.

Use The Settings Object

In order to interact with the settings key we just defined, we need to create a `GLib.Settings` object in our application. Building on previous examples, you can create a `Gtk.Switch`, add it to your window, create a `GLib.Settings` object, and bind the switch to a settings key like so:

```
1  protected override void activate () {
2      var switch = new Gtk.Switch () {
3          halign = Gtk.Align.CENTER,
4          valign = Gtk.Align.CENTER
5      };
6
7      var main_window = new Gtk.ApplicationWindow (this) {
8          default_height = 300,
9          default_width = 300
10     };
11     main_window.add (switch);
12     main_window.show_all ();
13
14     var settings = new GLib.Settings ("com.github.yourusername.yourrepository");
15     settings.bind ("useless-setting", switch, "active", GLib.SettingsBindFlagsNone);
16 }
```

You can read more about `GLib.Settings.bind ()` on [Valadoc](#), but for now this will bind the `active` property of the switch to the value of `useless-setting` in `GSettings`. When one changes, the other will stay in sync to reflect it.

Install and Compile Schemas with Meson

We need to add the new `GSchemas` XML file to our build system so it is included at install time. Create a file `data/meson.build` and type the following:

```
1  install_data (
2      'gschema.xml',
3      install_dir: get_option('datadir') / 'glib-2.0' / 'schemas',
4      rename: meson.project_name() + '.gschema.xml'
```

```
5 )
```

This ensures your gschema.xml file is installed, and renames it with your app's ID to avoid filename conflicts.

We'll also need to add a small Python script to tell Meson to compile our schemas. Create a new folder and a file `meson/post_install.py` that contains the following:

```
1  #!/usr/bin/env python3
2
3  import os
4  import subprocess
5
6  schemadir = os.path.join(os.environ['MESON_INSTALL_PREFIX'], 'share', 'gli
7
8  if not os.environ.get('DESTDIR'):
9      print('Compiling gsettings schemas...')
10     subprocess.call(['glib-compile-schemas', schemadir])
```

Be sure to add the following lines to the end of the `meson.build` file in your source root so that Meson knows where to find the additional instructions we've added:

```
1  meson.add_install_script('meson/post_install.py')
2
3  subdir('data')
```

Compile and install your app to see it in action!



GSettings are installed at install time, not compile time. So you'll need to run `ninja install` to avoid crashes from not-yet-installed settings.

To see the state saving functionality in action, you can open your app, toggle the switch, close it, then re-open it. The switch should retain its previous state. To see it under the hood,

open dconf Editor, navigate to your app's settings at `com.github.yourusername.yourrepositoryname`, and watch the `useless-setting` update in real time when you toggle your switch.



If you're having trouble, you can view the full example code [here on GitHub](#)

Custom Resources

You can include additional resources with your app like icons or CSS files using GResource. When using GResource, these files will be compiled into your app's binary, ensuring they're available and loaded when your app launches. Regardless of which type of resource you'd like to include, you'll need to create a `gresource.xml` file and include it in your build system. Make sure to start with a `Gtk.Application` as described in the [previous section](#)

In the `data` directory, create a new file called `gresource.xml` with the following contents:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <gresources>
3   <gresource prefix="com/github/yourusername/yourrepositoryname">
4   </gresource>
5 </gresources>
```

For now, this resource doesn't include any files. We'll come back to that in the following sections. Make sure to update the resource prefix to match your app's RDNN.

Now add the following lines to `meson.build`, somewhere before the `executable` step:

```
1 # Include the GNOME module
2 gnome = import('gnome')
3
4 # Tell meson where to find our resources file and to compile it as a GResource
5 gresource = gnome.compile_resources(
6     'gresource',
7     'data' / 'gresource.xml',
8     source_dir: 'data'
9 )
```

and finally, add the gresource to your `executable` step:

```
1 executable(
```

```

2     meson.project_name(),
3     gresource,
4     'src/Application.vala',
5     dependencies: [
6         dependency('gtk+-3.0')
7     ],
8     install: true
9 )

```

Now that we have a `gresource.xml` file and have included it in the build system, we can add files and reference them in our app.

Icons

As we saw in the section on `GLib.Action`, GTK has a baked-in set of icon sizes defined under the namespace `Gtk.IconSize`. When creating icons, it is important to know which of these sizes will be used and to design and hint the icon at that size. For more information about creating and hinting icons, check out the [Human Interface Guidelines](#). Add your custom icon to the `data` directory, and then update your `gresource.xml` file to reference it:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <gresources>
3   <gresource prefix="com/github/yourusername/yourrepositoryname">
4     <file compressed="true" preprocess="xml-stripblanks">custom-icon.svg</file>
5   </gresource>
6 </gresources>

```

If you want to use the same icon name in multiple sizes in your app, you can `alias` the icon to paths in `hicolor` and GTK will automatically load the correct version when its size is referenced:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <gresources>

```

```

3   <gresource prefix="com/github/yourusername/yourrepositoryname">
4     <file alias="24x24/actions/custom-icon.svg" compressed="true" preproc
5     <file alias="24x24@2/actions/custom-icon.svg" compressed="true" prepro
6     <file alias="32x32/actions/custom-icon.svg" compressed="true" preproc
7     <file alias="32x32@2/actions/custom-icon.svg" compressed="true" prepro
8   </gresource>
9 </gresources>

```

Now we can add the GResource path to the system's built-in icon theme in the `Application` class' `activate ()` function. This will let us reference the icon name without using long paths, and automatically handle icon sizing as previously mentioned. Again, don't forget to use the same RDN'd path that was defined in `gresource.xml` :

```

1  protected override void activate () {
2      Gtk.IconTheme.get_default ().add_resource_path ("com/github/yourusername/yourrepositoryname");
3
4      var main_window = new Gtk.ApplicationWindow (this);
5      main_window.show_all ();
6  }

```

The last step is to create a `Gtk.Image` or `Gtk.Button` using your custom icon and add it to the main window:

```

1  protected override void activate () {
2      Gtk.IconTheme.get_default ().add_resource_path ("com/github/yourusername/yourrepositoryname");
3
4      // This will create an image with an icon size of 32px
5      var image = new Gtk.Image.from_icon_name ("custom-icon", Gtk.IconSize.Default);
6
7      // This will create a button with an icon size of 24px
8      var button = new Gtk.Button.from_icon_name ("custom-icon", Gtk.IconSize.Default);
9
10     var grid = new Gtk.Grid () {
11         column_spacing = 12
12     };
13     grid.add (image);
14     grid.add (button);
15
16     var main_window = new Gtk.ApplicationWindow (this);
17     main_window.show_all ();

```


AppCenter

Publishing Requirements

We have a few requirements and suggestions for publishing your app to AppCenter. For a quick intro on writing apps for elementary OS, check out our [Getting Started](#) guide. For more developer-oriented tips, see the [Tips series on our blog](#).

Hard Requirements

The following are **hard requirements for all apps submitted to AppCenter**. Both automated and human reviews will check these requirements before each app and app update is approved and released to users.

Technical Requirements

Your app must be hosted in a [GitHub repository](#). AppCenter Dashboard works by importing source code from a GitHub repository and building it in a clean environment. To ensure reproducible builds, transparency, and auditability, binaries cannot be uploaded or included alongside the source code to be installed on users' devices.

Your app may be written in any language, but the front-end must be a native Gtk3 app. Web, Electron, Qt, Java, and other non-native app front-ends will be rejected during the review process. A game may be excepted from this requirement so long as it uses native window decorations and is generally usable on both loDPI and HiDPI displays.

Your app must not attempt to modify, replace, or append software sources on the target system.

Your app must not attempt to modify other apps or system programs.

Packaging

Your app must come with a [Flatpak manifest](#). elementary OS uses Flatpak packages to manage software installation and updates. Your app cannot be downloaded without it.

Extensions & Plugins

AppCenter supports publishing apps that meet these requirements; extensions or plugins (eg. to the Panel, System Settings, or other apps) are not currently supported.

Metadata Requirements

In general, your app's metadata should not refer to "elementary" or "elementary OS" in user-facing strings—it is assumed that all apps submitted to AppCenter are designed for elementary OS, and users don't need to be reminded of this. If there is a rare, legitimate reason for mentioning elementary, it must abide by the [elementary Brand Guidelines](#).

AppData and OARS

Your app must install an [appdata.xml file](#) to `/usr/share/metainfo`. AppCenter uses this metadata to create a listing for your app. It cannot be displayed in AppCenter without it.

Your appdata.xml file must contain a `screenshot` tag that references a screenshot of your app with elementary OS default settings including the GTK stylesheet, icons, window button position, etc. Screenshots referenced in your AppData should not contain marketing copy, illustrations, or other elements aside from a full-window screenshot of your app in use.

Your appdata.xml file must contain a `developer_name` tag that references your name or the name of your organization.

Your appdata.xml must include accurate [Open Age Rating Service \(OARS\)](#) data. OARS uses a short, self-reported survey that only takes a few moments to output the required XML. Reviewers will check this data for accuracy in order for your app to be published.

For more information about the appdata.xml format see, see the [appstream specification](#).

Desktop File

Your app must install a [.desktop file](#) to `/usr/share/applications`. elementary OS uses a .desktop file to display your app in the applications launcher and the dock. Without this file, your app will not be visible to users.

Your .desktop file must not contain `NoDisplay=true` or anything else that prevents it from showing up in the applications launcher.

Icons

Your app must install icons to `/usr/share/icons/hicolor` in 32px, 48px, 64px, and 128px. These are the icon sizes that will be displayed in the Applications menu's search, grid, and category views, Multitasking View, the dock, and in AppCenter.

Legal Requirements

You must have permission to redistribute any software you attempt to publish through AppCenter. If you are not the copyright holder or the source code is not openly licensed, you likely do not have permission to redistribute the software.

You must have permission to use any trademarks in your software or its metadata. You may not publish your app using trademarks reserved by others.

Your app must comply with all applicable laws as well as the terms of any services your app utilizes.

Other Publishing Requirements

Naming

To protect both users and developers, your app's name must not be the same as or confusingly similar to an existing app in AppCenter or other brand, as determined by app reviewers.

Your app should not be named generically like "Web Browser" or "File Manager".

Your app should not use "elementary", "Pantheon", or other elementary brand names in its naming scheme. It should also not be formatted with a leading lowercase "e", such as "eApp".

Launching

Your app should display its own graphical user interface on launch; it should not open another app or system component without user interaction inside your app UI. For example, if your app operates on a given file, it should display its own UI with an "Open File" button (or similar) before throwing an Open File dialog. If your app can request elevated permissions for certain actions, those permissions should be requested after your app's UI is shown and preferably only on-demand when actions requiring those permissions are performed.

App Stores

Your app cannot be an "app store," as ultimately determined by app reviewers. This includes but is not limited to apps that look and function confusingly similarly to AppCenter, provide software from other sources, link to an online app store, and/or facilitate payments for apps in the system repositories.

Suggestions

For the best experience, we strongly suggest the following before you publish your app:

Use a GitHub Organization

Once your app is submitted **you cannot change the GitHub account** that it is linked to. We recommend you use a GitHub organization as the owner of the repository so it can be transferred in the future if you so choose. This also makes it easier to manage community contributions.

Note that anyone with access to the GitHub organization will be able to manage releases and monetization.

Repository and App ID

While AppCenter Dashboard supports repositories with dashes and underscores, it vastly simplifies things to omit them from your GitHub repository name and subsequently your app ID. Some components in the desktop require workarounds for ID matching with dashes

and underscores, so it is typically simpler to stick to all lowercase with no dashes or underscores.

Human Interface Guidelines

Apps should generally abide by the [elementary Human Interface Guidelines](#), especially with regards to [Desktop Integration](#). Guidelines around [Notifications](#) may be strictly enforced, as it is important that apps behave as expected by users.

Tips for Games

If your game UI properly scales when the window is resized, one way to support both HiDPI and LoDPI (even if the engine doesn't out of the box) is to launch your game maximized. That way the UI will scale up no matter the resolution or scaling factor of the display.


Submission Process

The submission process for AppCenter is broken down into a few short steps. The step that you're currently on is indicated by an icon inside of a colored bubble. Hovering over that bubble with your cursor will display a tooltip with more information.

Install the AppCenter Dashboard GitHub Integration


Before AppCenter Dashboard can import and publish your repos, you'll need to install the [AppCenter Dashboard GitHub Integration](#).

Create a New Release

 This step is denoted by the tag icon in a light grey bubble.


Before you can publish your app, you'll need to [release it on GitHub](#). It is important that you use a [Semantic Versioning Number](#) without a pre-release tag when releasing. AppCenter Dashboard will try to sanitize your version number if it is not a proper Semver, but this may not succeed or have unintended results.

Submit for Review

 This step is denoted by an up arrow icon in a green bubble.


If a new release is available on GitHub, you'll be able to submit this release for publishing. Please make sure to review [the publishing requirements](#) before submitting your app. After clicking the button, AppCenter Dashboard will automatically import your repository and begin testing. It is important to [make changes to your app's monetization status](#) before completing this step.

Wait for Testing

 This step is denoted by a clock, rotating arrows, or a person icon in a yellow bubble.

Depending on demand, you may have to wait while AppCenter Dashboard is running test on other apps. When space is available, AppCenter Dashboard will perform automated tests on your app. After automated testing has completed, your app will await review by a human. Depending on demand, this may take several days.

Resolve Issues

 This status is denoted by an exclamation mark inside a triangle.

If your app fails either human or automated testing, new issues reports will be generated in your issue tracker on GitHub. You will need to correct these issues, create a new release of your app, and return to AppCenter Dashboard to begin the submission process again.

Available on AppCenter



This status is denoted by a check mark icon in a green bubble.

When your app passes automated testing and human review, it will become available in AppCenter in elementary OS. If you wish to [publish an update](#), you can restart the submission process by creating a new release on GitHub.

Monetizing Your App

Marking Your App for Monetization

At any time, you can select the "\$" icon in your dashboard to link your [Stripe account](#) and enable payments for your app in AppCenter. However, this change will only apply to your app when a new release is submitted; to link a new account or unlink an account you will need to create a new release and submit it for publishing.

Why Monetize?

Monetizing your app allows you to accept payments from users when they download your app. AppCenter displays the suggested price as the download button, along with a dropdown where users can choose their own price.



Remember that AppCenter is pay-what-you-want; the suggested price is *not a price floor*, and users can always enter a different price, including \$0

Even if you want to give your app away for free, you should consider monetizing and setting the suggested price to \$0. This lists your app as free throughout AppCenter, but allows users to send you optional payments when installing and from your app's listing.

Platform Fees

Each payment is split between you and elementary 70/30, with a minimum payment to elementary of \$0.50 to cover distribution and payment processing. For example: with a \$10 payment, \$3 is paid to elementary and \$7 is paid to you. For a \$1 payment, \$0.50 is paid to elementary to cover distribution and payment processing while \$0.50 is paid to you.

There are no enrollment fees or subscription costs to publish your app in AppCenter.

Publishing Updates

After your app has been published in AppCenter, you can issue a new update at any time by [creating a new release](#). This will begin the submission process again for your app.

Updating Screenshots

Screenshots are uploaded to the AppCenter screenshot server at publication time. The URL referenced in your AppData.xml file will be automatically replaced during publication and any changes you make to images hosted at the remote URL will not be reflected in AppCenter.



If you change the UI of your app in an update, you **must** update your screenshots

Release Descriptions

You must add accurate release descriptions to your appdata.xml file when publishing an update. For more information on formatting, see the [FreeDesktop.org AppStream documentation](#) for the `<releases />` tag. The version in your release should match the latest version submitted to AppCenter.

Release descriptions should be accurate, concise, and written for a typical user of your app —focus on user-visible and user-understandable changes, and simplify or omit technical "under the hood" changes. For example, a good release description might be:

New features:

- Support for WebP images
- Option to maintain aspect ratio when cropping

Other improvements:

- Main view now loads twice as fast
- Updated French translations
- Other code cleaning

...while a poor one would be:

- Fix bug #1234; rewrote method to be async
- Linted all the code, removed trailing whitespace
- Refactored utils
- Updated VAPIs

Previous release descriptions can remain in your AppData in perpetuity; AppCenter may use these to display cumulative descriptions between an older installed version and the latest version.

GitHub User/Org Name Changes

AppCenter **does not support changing your GitHub username or organization name**, as we rely on the unique RDNN of your app to avoid conflicts—and there is not an automated way to change the RDNN throughout the whole stack from AppCenter Dashboard down to the .deb files built and hosted in the AppCenter repository. **If you change your GitHub username or organization name associated with your app, you will no longer be able to publish updates to your app** without changing its name and branding. If you have ended up in this situation, you will receive an issue filed against your app with more details and with some options.

Continuous Integration

When you submit your app for publishing in AppCenter, a suite of automated tests is run to check for things like metadata validity, file naming and installation, and successful package builds. Houston CI is a version of this test suite that can be run continuously on your GitHub projects using [Travis CI](#).

Setting up Houston CI for your app can save you time and give you reassurance that when you submit your app for publishing it will pass the automated testing phase in AppCenter Dashboard. Keep in mind however, that there is also a human review portion of the submission process, so an app that passes Houston CI may still need fixing before being published in AppCenter.

Configuring Travis

If you've never used Travis CI before, refer to their [Getting Started guide](#).

Standard .travis.yml

In order to build your app using Houston CI, Travis requires a configuration file called `.travis.yml` in the root directory of your repo with the following contents:

```
1  ---
2
3  language: node_js
4
5  node_js:
6    - 10.17.0
7
8  sudo: required
9
10 services:
11   - docker
12
13 addons:
14   apt:
15     sources:
```

```
16     - ubuntu-toolchain-r-test
17     packages:
18     - libstdc++-5-dev
19
20     install:
21     - npm i -g @elementaryos/houston
22
23     script:
24     - houston ci
```

Testing with Houston CI

Depending on how you've configured Travis, Houston CI will automatically run its suite of tests on your master branch as well as any branches submitted via pull request.

Markdown Badges

elementary provides badges for your GitHub README. Badges will automatically open AppCenter or redirect to elementary.io based on OS detection. They look like this:



Markdown

```
[![Get it on AppCenter](https://appcenter.elementary.io/badge.svg)](https://
```

HTML

```
<a href="https://appcenter.elementary.io/com.github.USER.REPO"><img src="htt
```



Make sure to replace `com.github.USER.REPO` in the URL with the ID for your app