BeeWare Documentation

Release 0.3.0

Russell Keith-Magee

Oct 22, 2023

CONTENTS

Write Python. Run Anywhere.

Welcome to BeeWare! In this tutorial, we're going to build an graphical user interface using Python, and deploy it as a desktop application, as a mobile application, and as a single page web app. We're also going to look at how you can use BeeWare tools to do some of the common tasks that you'll need to do as an app developer, such as testing your app.

CHAPTER

ONE

WHAT IS BEEWARE?

BeeWare is not a single product, or tool, or library - it's a collection of tools and libraries, each of which works together to help you write cross platform Python applications with a native GUI. It includes:

- Toga, a cross platform widget toolkit;
- Briefcase, a tool for packaging Python projects as distributable artefacts that can be shipped to end users;
- Libraries (such as Rubicon ObjC) for accessing platform-native libraries;
- Pre-compiled builds of Python that can be used on platforms where official Python installers aren't available.

In this tutorial, we'll be using all these tools, but as a user, you'll only need to interact with the first two (Toga and Briefcase). However, each of the tools can also be used individually - for example, you can use Briefcase to deploy an app without using Toga as a GUI toolkit.

The BeeWare suite is available on macOS, Windows, Linux (using GTK); on mobile platforms such as Android and iOS; and for the Web.

CHAPTER

TWO

LETS GO!

Ready to try BeeWare for yourself? *Let's build a cross platform application in Python!*

2.1 Tutorial 0 - Let's get set up!

Before we build our first BeeWare app, we have to make sure we've got all the prerequisites for running BeeWare.

2.1.1 Install Python

The first thing we'll need is a working Python interpreter.

macOS

If you're on macOS, a recent version of Python is included with Xcode or the command line developer tools. To check if you already have it, run the following command:

```
$ python3 --version
```

If Python is installed, you'll see its version number. Otherwise, you'll be prompted to install the command line developer tools.

Linux

If you're on Linux, you'll install Python using the system package manager (`apt` on Debian/Ubuntu/Mint, `dnf` on Fedora, or `pacman` on Arch).

You should ensure that the system Python is Python 3.8 or newer; if it isn't (e.g., Ubuntu 18.04 ships with Python 3.6), you'll need to upgrade your Linux distribution to something more recent.

Support for Raspberry Pi is limited at this time.

Windows

If you're on Windows, you can get the official installer from the Python website. You can use any stable version of Python from 3.8 onward. We'd advise avoiding alphas, betas, and release candidates unless you *really* know what you're doing.

Alternative Python distributions

There are lots of different ways of installing Python. You can install Python through homebrew. You can use pyenv to manage multiple Python installs on the same machine. Windows users can install Python from the Windows App Store. Users from a data science background might want to use Anaconda or Miniconda.

If you're on macOS or Windows, it doesn't matter *how* you've installed Python - it only matters that you can run `python3` from your operating system's command prompt/terminal application, and get a working Python interpreter.

If you're on Linux, you should use the system Python provided by your operating system. You will be able to complete *most* of this tutorial using a non-system Python, but you won't be able to package your application for distribution to others.

2.1.2 Install dependencies

Next, install the additional dependencies needed for your operating system:

macOS

Building BeeWare apps on macOS requires:

- **Git**, a version control system. This is included with Xcode or the command line developer tools, which you installed above.

Linux

To support local development, you'll need to install some system packages. The list of packages required varies depending on your distribution:

Ubuntu 20.04+ / Debian 10+

```
$ sudo apt update
$ sudo apt install build-essential git pkg-config python3-dev python3-venv␣
↪libgirepository1.0-dev libcairo2-dev gir1.2-webkit2-4.0 libcanberra-gtk3-module
```

Fedora

```
$ sudo dnf install git pkg-config rpm-build python3-devel gobject-introspection-devel␣
↪cairo-gobject-devel webkit2gtk3 libcanberra-gtk3
```

Arch, Manjaro

```
$ sudo pacman -Syu base-devel git pkgconf python3 gobject-introspection cairo webkit2gtk␣
↪libcanberra
```

Windows

Building BeeWare apps on Windows requires:

- **Git**, a version control system. You can download Git from from git-scm.org.

After installing these tools, you should ensure you restart any terminal sessions. Windows will only expose newly installed tools terminals started *after* the install has completed.

2.1.3 Set up a virtual environment

We're now going to create a virtual environment - a "sandbox" that we can use to isolate our work on this tutorial from our main Python installation. If we install packages into the virtual environment, our main Python installation (and any other Python projects on our computer) won't be affected. If we make a complete mess of our virtual environment, we'll be able to simply delete it and start again, without affecting any other Python project on our computer, and without the need to re-install Python.

macOS

```
$ mkdir beeware-tutorial
$ cd beeware-tutorial
$ python3 -m venv beeware-venv
$ source beeware-venv/bin/activate
```

Linux

```
$ mkdir beeware-tutorial
$ cd beeware-tutorial
$ python3 -m venv beeware-venv
$ source beeware-venv/bin/activate
```

Windows

```
C:\...>md beeware-tutorial
C:\...>cd beeware-tutorial
C:\...>py -m venv beeware-venv
C:\...>beeware-venv\Scripts\activate
```

Errors running PowerShell Scripts

If you're using PowerShell, and you receive the error:

```
File C:\...\beeware-tutorial\beeware-venv\Scripts\activate.ps1 cannot be loaded because␣
↪running scripts is disabled on this system.
```

Your Windows account doesn't have permissions to run scripts. To fix this:

1. Run Windows PowerShell as Administrator.
2. Run `set-executionpolicy RemoteSigned`
3. Select `Y` to change the execution policy.

Once you've done this you can rerun `beeware-venv\Scripts\activate.ps1` in your original PowerShell session (or a new session in the same directory).

If this worked, your prompt should now be changed - it should have a `(beeware-venv)` prefix. This lets you know that you're currently in your BeeWare virtual environment. Whenever you're working on this tutorial, you should make sure your virtual environment is activated. If it isn't, re-run the last command (the `activate` command) to re-activate your environment.

Alternative virtual environments

If you're using Anaconda or miniconda, you may be more familiar with using conda environments. You might also have heard of `virtualenv`, a predecessor to Python's built in `venv` module. As with Python installs - if you're on macOS or Windows, it doesn't matter *how* you create your virtual environment, as long as you have one. If you're on Linux, you should stick to `venv` and the system Python.

2.1.4 Next steps

We've now set up our environment. We're ready to *create our first BeeWare application*.

2.2 Tutorial 1 - Your first app

We're ready to create our first application.

2.2.1 Install the BeeWare tools

First, we need to install **Briefcase**. Briefcase is a BeeWare tool that can be used to package your application for distribution to end users - but it can also be used to bootstrap a new project. Make sure you're in the `beeware-tutorial` directory you created in *Tutorial 0*, with the `beeware-venv` virtual environment activated, and run:

macOS

```
(beeware-venv) $ python -m pip install briefcase
```

Linux

```
(beeware-venv) $ python -m pip install briefcase
```

Possible errors during installation

If you see errors during installation, it's almost certainly because some of the system requirements haven't been installed. Make sure you have *installed all the platform pre-requisites*.

Windows

```
(beeware-venv) C:\...>python -m pip install briefcase
```

Possible errors during installation

It is important that you use `python -m pip`, rather than a bare `pip`. Briefcase needs to ensure that it has an up-to-date version of `pip` and `setuptools`, and a bare invocation of `pip` can't self-update. If you want to know more, Brett Cannon has a detailed blog post about the issue.

One of the BeeWare tools is **Briefcase**. Briefcase can be used to package your application for distribution to end users - but it can also be used to bootstrap a new project.

2.2.2 Bootstrap a new project

Let's start our first BeeWare project! We're going to use the Briefcase `new` command to create an application called **Hello World**. Run the following from your command prompt:

macOS

```
(beeware-venv) $ briefcase new
```

Linux

```
(beeware-venv) $ briefcase new
```

Windows

```
(beeware-venv) C:\...>briefcase new
```

Briefcase will ask us for some details of our new application. For the purposes of this tutorial, use the following:

- **Formal Name** - Accept the default value: `Hello World`.
- **App Name** - Accept the default value: `helloworld`.
- **Bundle** - If you own your own domain, enter that domain in reversed order. (For example, if you own the domain "cupcakes.com", enter `com.cupcakes` as the bundle). If you don't own your own domain, accept the default bundle (`com.example`).
- **Project Name** - Accept the default value: `Hello World`.
- **Description** - Accept the default value (or, if you want to be really creative, come up with your own description!)
- **Author** - Enter your own name here.
- **Author's email** - Enter your own email address. This will be used in the configuration file, in help text, and anywhere that an email is required when submitting the app to an app store.
- **URL** - The URL of the landing page for your application. Again, if you own your own domain, enter a URL at that domain (including the `https://`). Otherwise, just accept the default URL (`https://example.com/helloworld`). This URL doesn't need to actually exist (for now); it will only be used if you publish your application to an app store.
- **License** - Accept the default license (BSD). This won't affect anything about the operation of the tutorial, though - so if you have particularly strong feelings about license choice, feel free to choose another license.
- **GUI framework** - Accept the default option, Toga (BeeWare's own GUI toolkit).

Briefcase will then generate a project skeleton for you to use. If you've followed this tutorial so far, and accepted the defaults as described, your file system should look something like:

```
beeware-tutorial/
    beeware-venv/
        ...
    helloworld/
        CHANGELOG
        LICENSE
        README.rst
        pyproject.toml
        src/
            helloworld/
                resources/
```

(continues on next page)

(continued from previous page)

```
                    helloworld.icns
                    helloworld.ico
                    helloworld.png
                __init__.py
                __main__.py
                app.py
        tests/
            __init__.py
            helloworld.py
            test_app.py
```

This skeleton is actually a fully functioning application without adding anything else. The `src` folder contains all the code for the application, the `tests` folder contains an initial test suite, and the `pyproject.toml` file describes how to package the application for distribution. If you open `pyproject.toml` in an editor, you'll see the configuration details you just provided to Briefcase.

Now that we have a stub application, we can use Briefcase to run the application.

2.2.3 Run the app in developer mode

Move into the `helloworld` project directory and tell briefcase to start the project in Developer (or `dev`) mode:

macOS

```
(beeware-venv) $ cd helloworld
(beeware-venv) $ briefcase dev

[hello-world] Installing requirements...
...

[helloworld] Starting in dev mode...
===========================================================================
```

Linux

```
(beeware-venv) $ cd helloworld
(beeware-venv) $ briefcase dev

[hello-world] Installing requirements...
...

[helloworld] Starting in dev mode...
===========================================================================
```

Windows

```
(beeware-venv) C:\...>cd helloworld
(beeware-venv) C:\...>briefcase dev

[hello-world] Installing requirements...
...

```

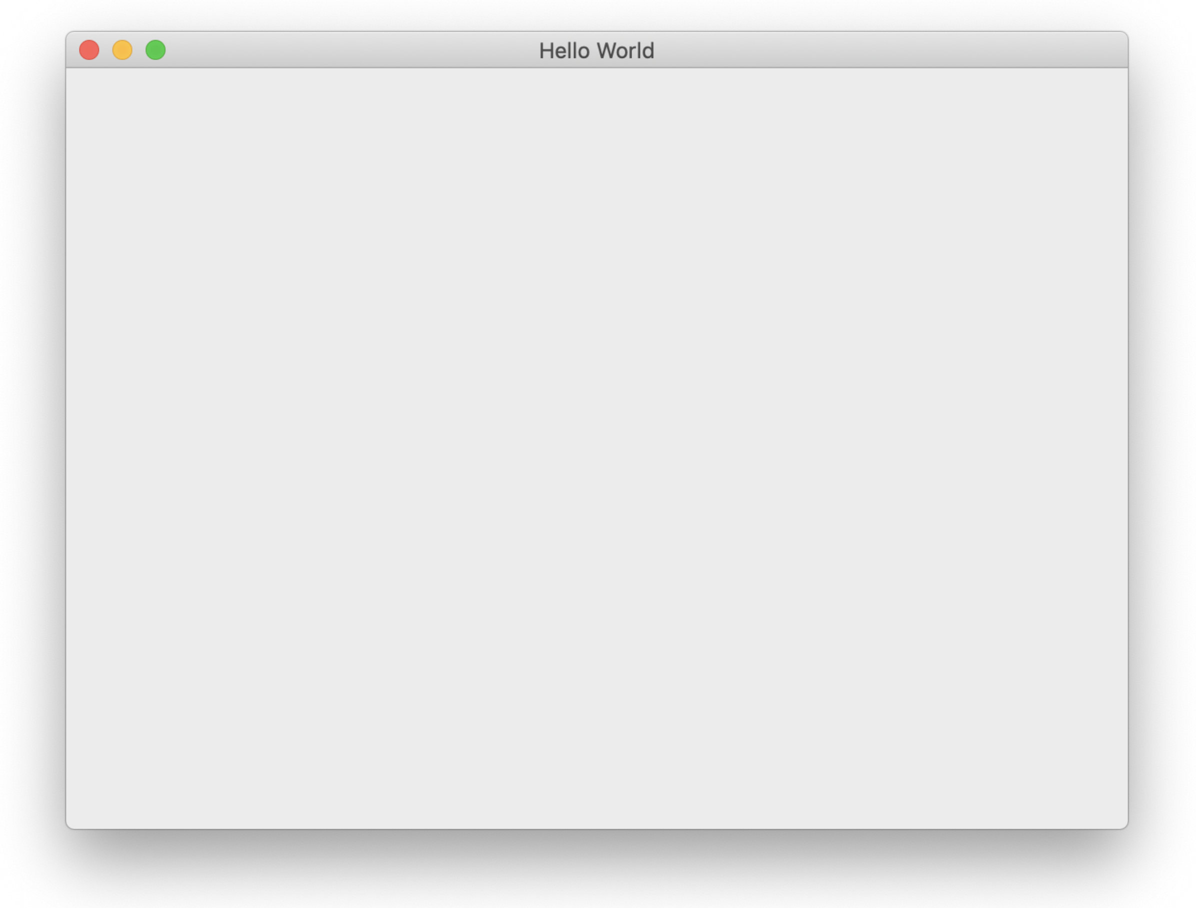
(continues on next page)

(continued from previous page)

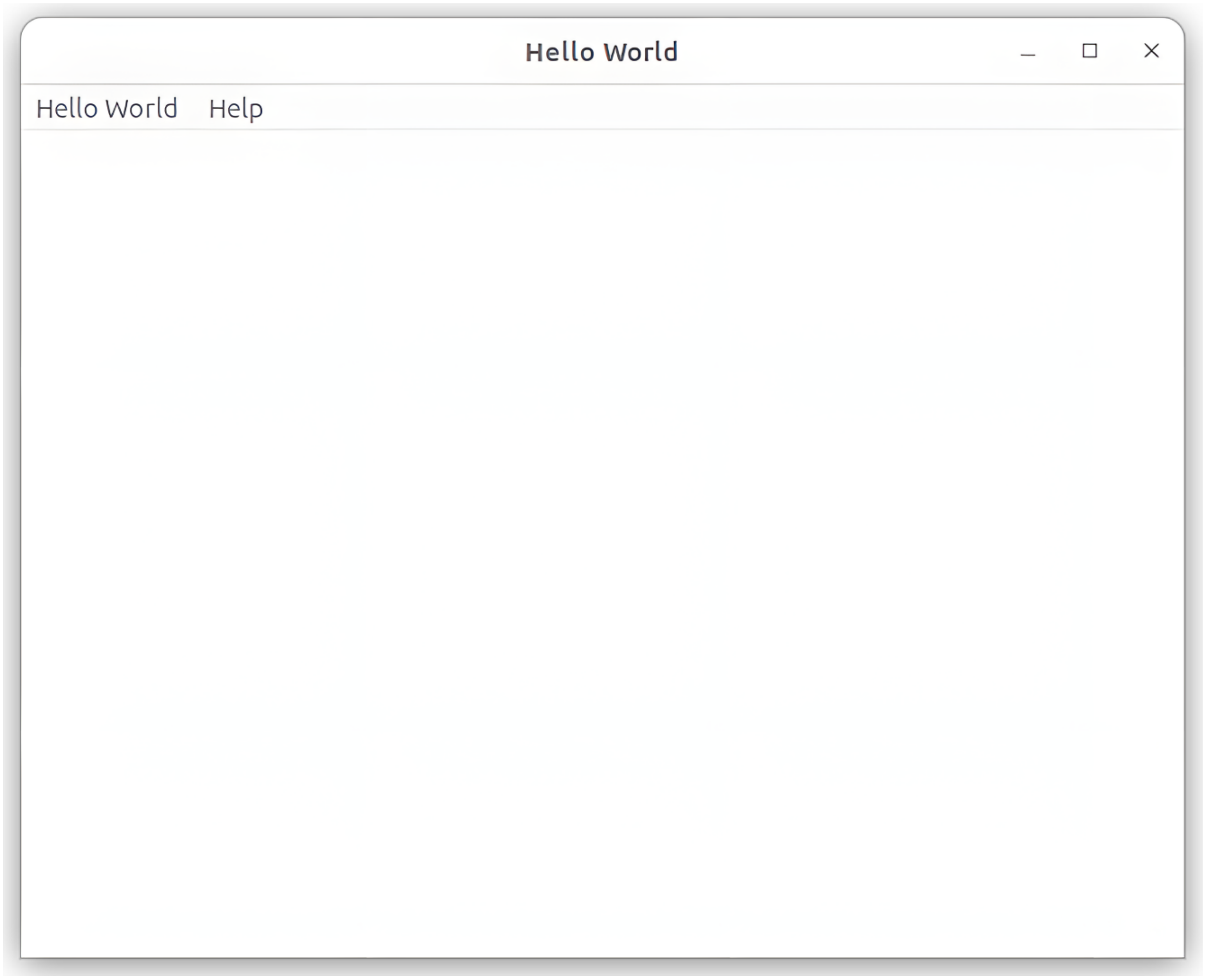
```
[helloworld] Starting in dev mode...
===========================================================================
```

This should open a GUI window:

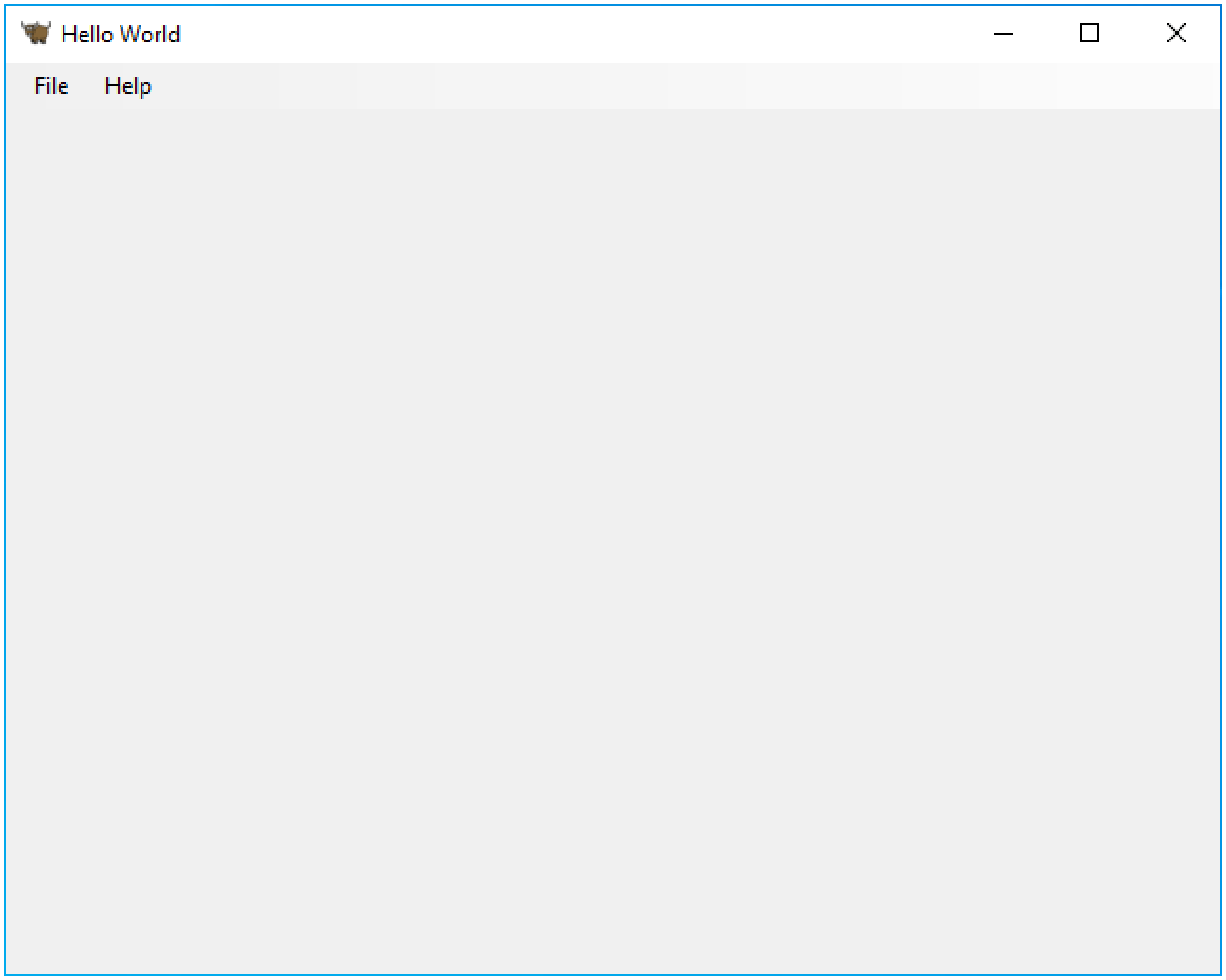
macOS



Linux



Windows



Press the close button (or select Quit from the application's menu), and you're done! Congratulations - you've just written a standalone, native application in Python!

2.2.4 Next steps

We now have a working application, running in developer mode. Now we can add some logic of our own to make our application do something a little more interesting. In *Tutorial 2*, we'll put a more useful user interface onto our application.

2.3 Tutorial 2 - Making it interesting

In *Tutorial 1*, we generated a stub project that was able to run, but we didn't write any code ourselves. Let's take a look at what was generated for us.

2.3.1 What was generated

In the src/helloworld directory, you should see 3 files: __init__.py, __main__.py and app.py.

__init__.py marks the helloworld directory as an importable Python module. It is an empty file; the very fact it exists tells the Python interpreter that the helloworld directory defines a module.

__main__.py marks the helloworld module as a special kind of module - an executable module. If you try to run the helloworld module using python -m helloworld, the __main__.py file is where Python will start executing. The contents of __main__.py is relatively simple:

```
from helloworld.app import main

if __name__ == '__main__':
    main().main_loop()
```

That is - it imports the main method from the helloworld app; and if it's being executed as an entry point, calls the main() method, and starts the application's main loop. The main loop is the way a GUI application listens for user input (like mouse clicks and keyboard presses).

The more interesting file is app.py - this contains the logic that creates our application window:

```
import toga
from toga.style import Pack
from toga.style.pack import COLUMN, ROW

class HelloWorld(toga.App):
    def startup(self):
        main_box = toga.Box()

        self.main_window = toga.MainWindow(title=self.formal_name)
        self.main_window.content = main_box
        self.main_window.show()

def main():
    return HelloWorld()
```

Let's go through this line by line:

```
import toga
from toga.style import Pack
from toga.style.pack import COLUMN, ROW
```

First, we import the toga widget toolkit, as well as some style-related utility classes and constants. Our code doesn't use these yet - but we'll make use of them shortly.

Then, we define a class:

```
class HelloWorld(toga.App):
```

Each Toga application has a single toga.App instance, representing the running entity that is the application. The app may end up managing multiple windows; but for simple applications, there will be a single main window.

Next, we define a startup() method:

```
def startup(self):
    main_box = toga.Box()
```

The first thing the startup method does is to define a main box. Toga's layout scheme behaves similar to HTML. You build an application by constructing a collection of boxes, each of which contains other boxes, or actual widgets. You then apply styles to these boxes to define how they will consume the available window space.

In this application, we define a single box, but we don't put anything into it.

Next, we define a window into which we can put this empty box:

```
self.main_window = toga.MainWindow(title=self.formal_name)
```

This creates an instance of a `toga.MainWindow`, which will have a title matching the application's name. A Main Window is a special kind of window in Toga - it's a window that is closely bound to the life cycle of the app. When the Main Window is closed, the application exits. The Main Window is also the window that has the application's menu (if you're on a platform like Windows where menu bars are part of the window)

We then add our empty box as the content of the main window, and instruct the application to show our window:

```
self.main_window.content = main_box
self.main_window.show()
```

Last of all, we define a `main()` method. This is what creates the instance of our application:

```
def main():
    return HelloWorld()
```

This `main()` method is the one that is imported and invoked by `__main__.py`. It creates and returns an instance of our `HelloWorld` application.

That's the simplest possible Toga application. Let's put some of our own content into the application, and make the app do something interesting.

2.3.2 Adding some content of our own

Modify your `HelloWorld` class inside `src/helloworld/app.py` so that it looks like this:

```
class HelloWorld(toga.App):
    def startup(self):
        main_box = toga.Box(style=Pack(direction=COLUMN))

        name_label = toga.Label(
            "Your name: ",
            style=Pack(padding=(0, 5))
        )
        self.name_input = toga.TextInput(style=Pack(flex=1))

        name_box = toga.Box(style=Pack(direction=ROW, padding=5))
        name_box.add(name_label)
        name_box.add(self.name_input)

        button = toga.Button(
            "Say Hello!",
            on_press=self.say_hello,
            style=Pack(padding=5)
        )
```

(continues on next page)

(continued from previous page)

```
        main_box.add(name_box)
        main_box.add(button)

        self.main_window = toga.MainWindow(title=self.formal_name)
        self.main_window.content = main_box
        self.main_window.show()

    def say_hello(self, widget):
        print(f"Hello, {self.name_input.value}")
```

Note: Don't remove the imports at the top of the file , or the `main()` at the bottom. You only need to update the `HelloWorld` class.

Let's look in detail at what has changed.

We're still creating a main box; however, we are now applying a style:

```
main_box = toga.Box(style=Pack(direction=COLUMN))
```

Toga's builtin layout system is called "Pack". It behaves a lot like CSS. You define objects in a hierarchy - in HTML, the objects are <div>, <span>, and other DOM elements; in Toga, they're widgets and boxes. You can then assign styles to the individual elements. In this case, we're indicating that this is a `COLUMN` box - that is, it is a box that will consume all the available width, and will expand its height as content is added, but it will try to be as short as possible.

Next, we define a couple of widgets:

```
name_label = toga.Label(
    "Your name: ",
    style=Pack(padding=(0, 5))
)
self.name_input = toga.TextInput(style=Pack(flex=1))
```

Here, we define a Label and a TextInput. Both widgets have styles associated with them; the label will have 5px of padding on its left and right, and no padding on the top and bottom. The TextInput is marked as being flexible - that is, it will absorb all available space in its layout axis.

The TextInput is assigned as an instance variable of the class. This gives us easy access to the widget instance - something that we'll use in a moment.

Next, we define a box to hold these two widgets:

```
name_box = toga.Box(style=Pack(direction=ROW, padding=5))
name_box.add(name_label)
name_box.add(self.name_input)
```

The `name_box` is a box just like the main box; however, this time, it's a `ROW` box. That means content will be added horizontally, and it will try to make its width as narrow as possible. The box also has some padding - 5px on all sides.

Now we define a button:

```
button = toga.Button(
    "Say Hello!",
    on_press=self.say_hello,
```

(continues on next page)

(continued from previous page)

```
    style=Pack(padding=5)
)
```

The button also has 5px of padding on all sides. We also define a *handler* - a method to invoke when the button is pressed.

Then, we add the name box and the button to the main box:

```
main_box.add(name_box)
main_box.add(button)
```

This completes our layout; the rest of the startup method is as it was previously - defining a MainWindow, and assigning the main box as the window's content:

```
self.main_window = toga.MainWindow(title=self.formal_name)
self.main_window.content = main_box
self.main_window.show()
```

The last thing we need to do is define the handler for the button. A handler can be any method, generator, or asynchronous co-routine; it accepts the widget that generated the event as an argument, and will be invoked whenever the button is pressed:

```
def say_hello(self, widget):
    print(f"Hello, {self.name_input.value}")
```

The body of the method is a simple print statement - however, it will interrogate the current value of the name input, and use that content as the text that is printed.

Now that we've made these changes we can see what they look like by starting the application again. As before, we'll use developer mode:

macOS

```
(beeware-venv) $ briefcase dev

[helloworld] Starting in dev mode...
===========================================================================
```

Linux

```
(beeware-venv) $ briefcase dev

[helloworld] Starting in dev mode...
===========================================================================
```

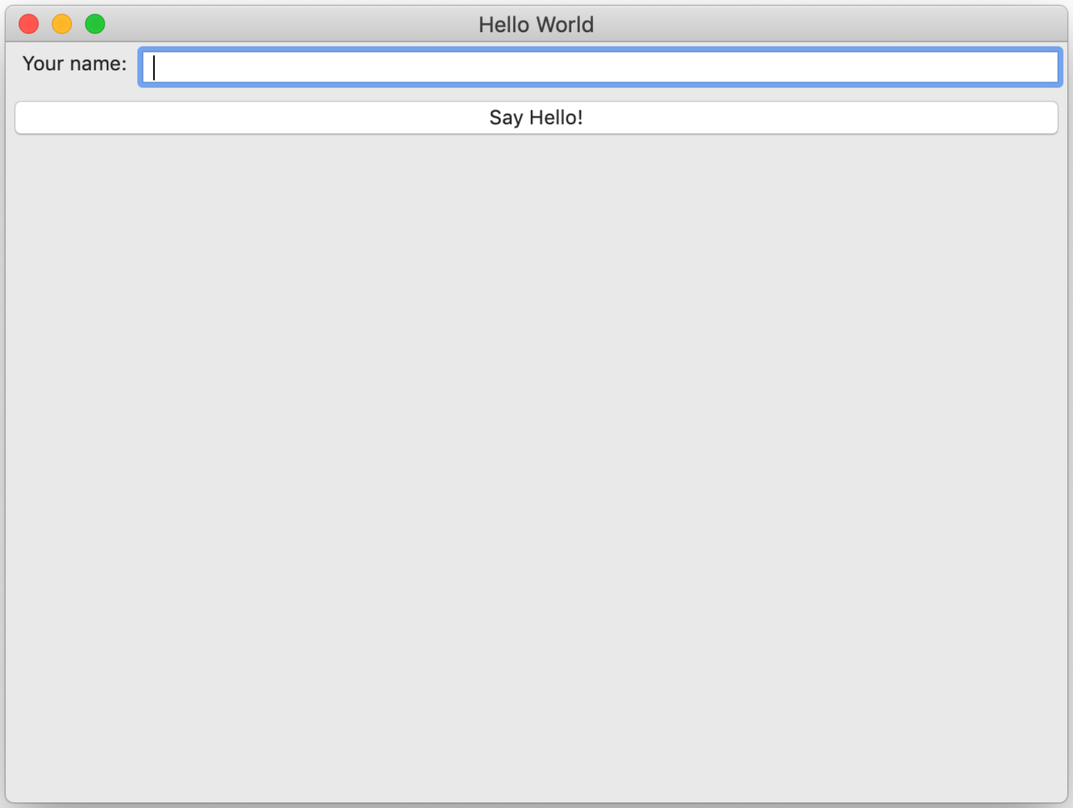
Windows

```
(beeware-venv) C:\...>briefcase dev

[helloworld] Starting in dev mode...
===========================================================================
```

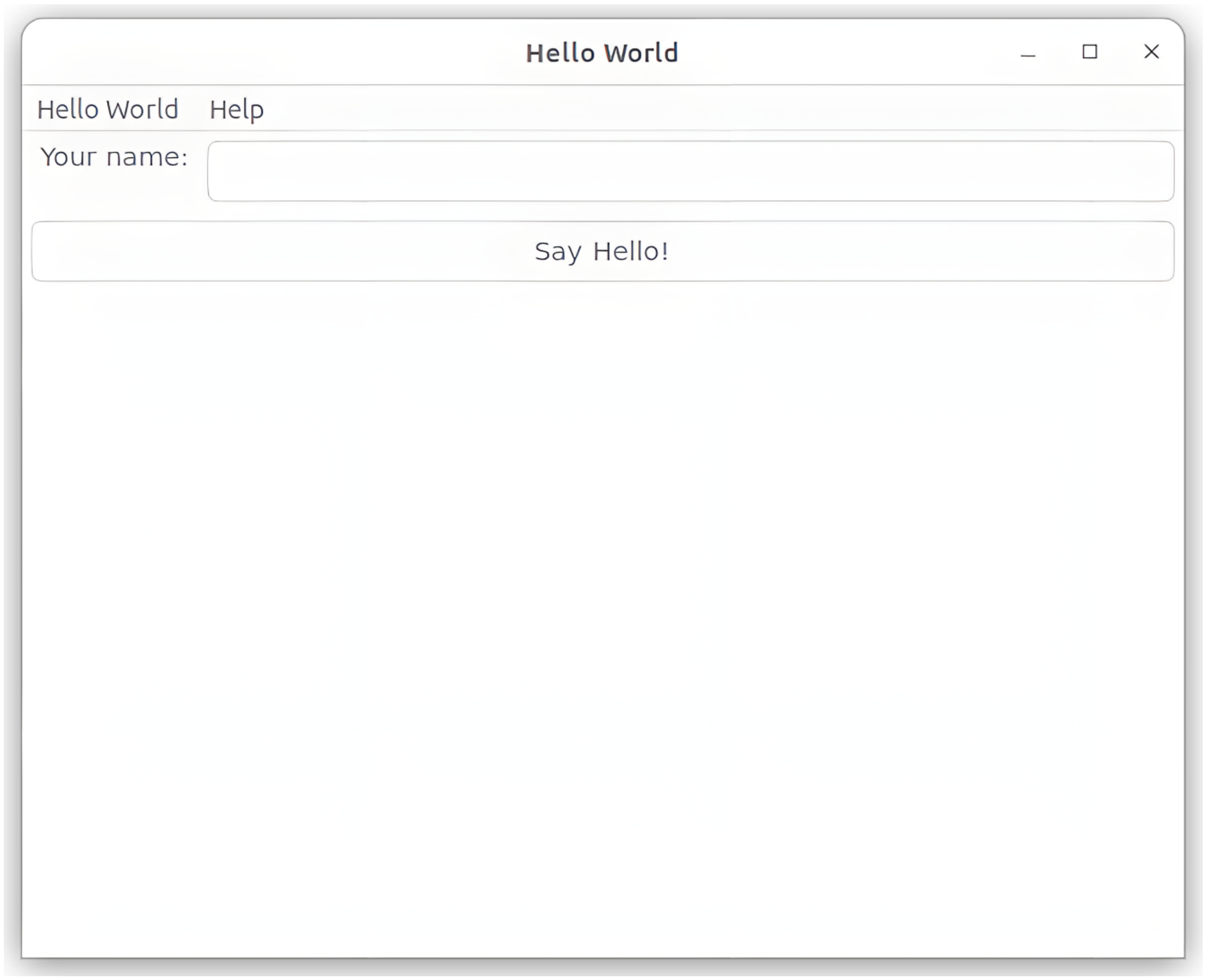
You'll notice that this time, it *doesn't* install dependencies. Briefcase can detect that the application has been run before, and to save time, will only run the application. If you add new dependencies to your app, you can make sure that they're installed by passing in a `-r` option when you run `briefcase dev`.

This should open a GUI window:

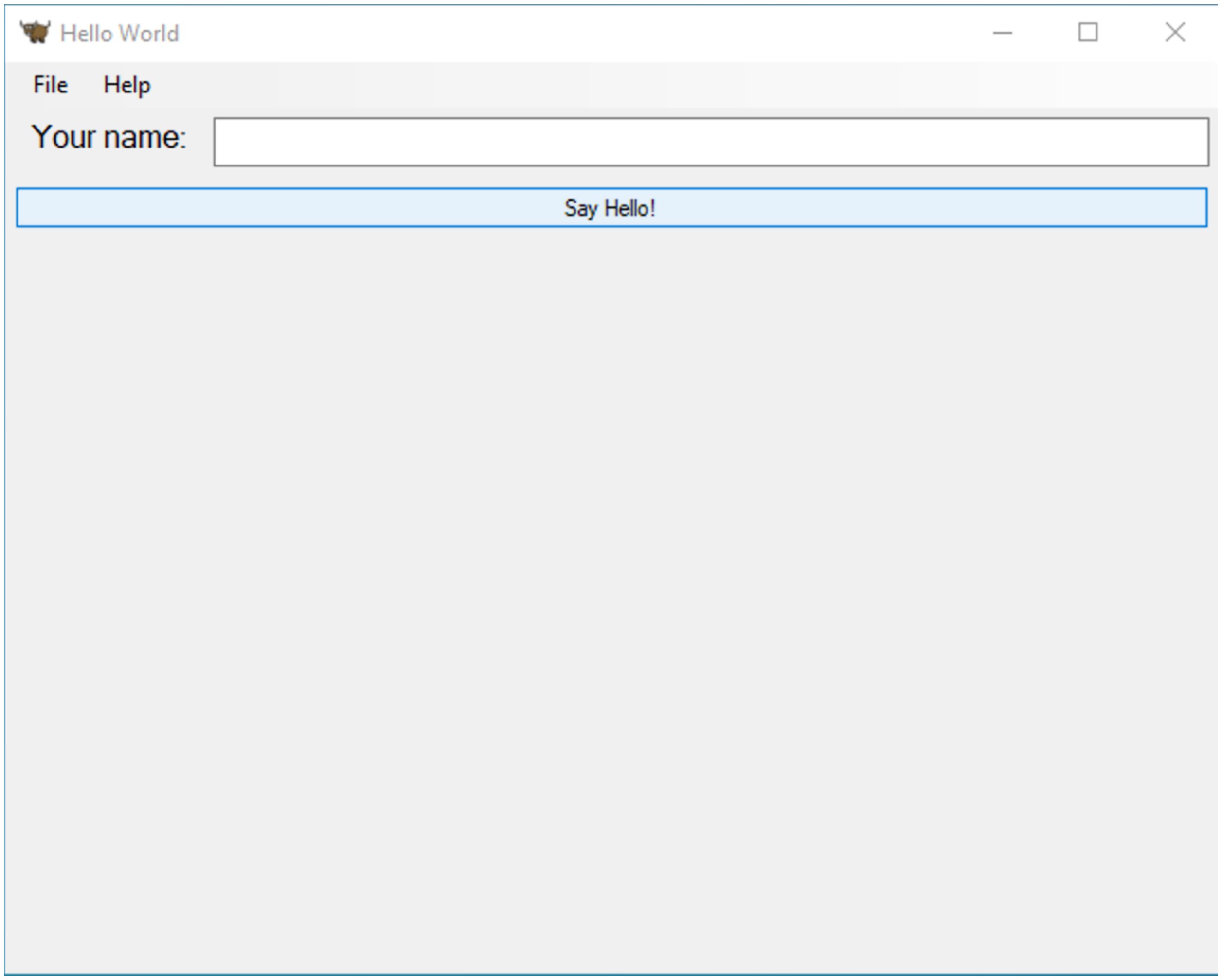
macOS



Linux



Windows



If you enter a name in the text box, and press the GUI button, you should see output appear in the console where you started the application.

2.3.3 Next steps

We've now got an application that does something a little more interesting. But it only runs on our own computer. Let's package this application for distribution. In *Tutorial 3*, we'll wrap our application up as a standalone installer that we could send to a friend, a customer, or upload to an App Store.

2.4 Tutorial 3 - Packaging for distribution

So far, we've been running our application in "Developer mode". This makes it easy for us to run our application locally - but what we really want is to be able to give our application to others.

However, we don't want to have to teach our users how to install Python, create a virtual environment, clone a git repository, and run Briefcase in developer mode. We'd rather just give them an installer, and have the application Just Work.

Briefcase can be used to package your application for distribution in this way.

2.4.1 Creating your application scaffold

Since this is the first time we're packaging our application, we need to create some configuration files and other scaffolding to support the packaging process. From the `helloworld` directory, run:

macOS

```
(beeware-venv) $ briefcase create

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-macOS-app-template.git, branch␣
↪v0.3.14
...

[helloworld] Installing support package...
...

[helloworld] Installing application code...
Installing src/helloworld... done

[helloworld] Installing requirements...
...

[helloworld] Installing application resources...
...

[helloworld] Removing unneeded app content...
...

[helloworld] Created build/helloworld/macos/app
```

Linux

```
(beeware-venv) $ briefcase create

[helloworld] Finalizing application configuration...
Targeting ubuntu:jammy (Vendor base debian)
Determining glibc version... done

Targeting glibc 2.35
Targeting Python3.10

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-linux-AppImage-template.git,␣
↪branch v0.3.14
...

[helloworld] Installing support package...
No support package required.

[helloworld] Installing application code...
Installing src/helloworld... done

[helloworld] Installing requirements...
```

(continues on next page)

(continued from previous page)

```
...

[helloworld] Installing application resources...
...

[helloworld] Removing unneeded app content...
...

[helloworld] Created build/helloworld/linux/ubuntu/jammy
```

Windows

```
(beeware-venv) C:\...>briefcase create

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-windows-app-template.git,␣
↪branch v0.3.14
...

[helloworld] Installing support package...
...

[helloworld] Installing application code...
Installing src/helloworld... done

[helloworld] Installing requirements...
...

[helloworld] Installing application resources...
...

[helloworld] Created build\helloworld\windows\app
```

You've probably just seen pages of content go past in your terminal… so what just happened? Briefcase has done the following:

1. It **generated an application template**. There's a lot of files and configurations required to build a native installer, above and beyond the code of your actual application. This extra scaffolding is almost the same for every application on the same platform, except for the name of the actual application being constructed - so Briefcase provides an application template for each platform it supports. This step rolls out the template, substituting the name of your application, bundle ID, and other properties of your configuration file as required to support the platform you're building on.

   If you're not happy with the template provided by Briefcase, you can provide your own. However, you probably don't want to do this until you've got a bit more experience using Briefcase's default template.

2. It **downloaded and installed a support package**. The packaging approach taken by briefcase is best described as "the simplest thing that could possibly work" - it ships a complete, isolated Python interpreter as part of every application it builds. This is slightly space inefficient - if you have 5 applications packaged with Briefcase, you'll have 5 copies of the Python interpreter. However, this approach guarantees that every application is completely independent, using a specific version of Python that is known to work with the application.

   Again, Briefcase provides a default support package for each platform; if you want, you can provide your own support package, and have that package included as part of the build process. You may want to do this if you have particular options in the Python interpreter that you need to have enabled, or if you want to strip modules out of the standard library that you don't need at runtime.

Briefcase maintains a local cache of support packages, so once you've downloaded a specific support package, that cached copy will be used on future builds.

3. It **installed application requirements**. Your application can specify any third-party modules that are required at runtime. These will be installed using `pip` into your application's installer.
4. It **Installed your application code**. Your application will have its own code and resources (e.g., images that are needed at runtime); these files are copied into the installer.
5. It **installed your resources needed by your application.** Lastly, it adds any additional resources that are needed by the installer itself. This includes things like icons that need to be attached to the final application and splash screen images.

Once this completes, if you look in the project directory, you should now see a directory corresponding to your platform (`macOS`, `linux`, or `windows`) that contains additional files. This is the platform-specific packaging configuration for your application.

2.4.2 Building your application

You can now compile your application. This step performs any binary compilation that is necessary for your application to be executable on your target platform.

macOS

```
(beeware-venv) $ briefcase build

[helloworld] Adhoc signing app...
...
Signing build/helloworld/macos/app/Hello World.app
 100.0% • 00:07

[helloworld] Built build/helloworld/macos/app/Hello World.app
```

On macOS, the `build` command doesn't need to *compile* anything, but it does need to sign the contents of binary so that it can be executed. This signature is an *ad hoc* signature - it will only work on *your* machine; if you want to distribute the application to others, you'll need to provide a full signature.

Linux

```
(beeware-venv) $ briefcase build

[helloworld] Finalizing application configuration...
Targeting ubuntu:jammy (Vendor base debian)
Determining glibc version... done

Targeting glibc 2.35
Targeting Python3.10

[helloworld] Building application...
Build bootstrap binary...
make: Entering directory '/home/brutus/beeware-tutorial/helloworld/build/linux/ubuntu/
↪jammy/bootstrap'
...
make: Leaving directory '/home/brutus/beeware-tutorial/helloworld/build/linux/ubuntu/
```

(continues on next page)

(continued from previous page)

```
↪jammy/bootstrap'
Building bootstrap binary... done
Installing license... done
Installing changelog... done
Installing man page... done
Update file permissions...
...
Updating file permissions... done
Stripping binary... done

[helloworld] Built build/helloworld/linux/ubuntu/jammy/helloworld-0.0.1/usr/bin/
↪helloworld
```

Once this step completes, the `build` folder will contain a `helloworld-0.0.1` folder that contains a mirror of a Linux `/usr` file system. This file system mirror will contain a `bin` folder with a `helloworld` binary, plus `lib` and `share` folders needed to support the binary.

Windows

```
(beeware-venv) C:\...>briefcase build
Setting stub app details... done

[helloworld] Built build\helloworld\windows\app\src\Hello World.exe
```

On Windows, the `build` command doesn't need to *compile* anything, but it does need to write some metadata so that the application knows its name, version, and so on.

Triggering antivirus

Since this metadata is being written directly in to the pre-compiled binary rolled out from the template during the `create` command, this may trigger antivirus software running on your machine and prevent the metadata from being written. In that case, instruct the antivirus to allow the tool (named `rcedit-x64.exe`) to run and re-run the command above.

2.4.3 Running your app

You can now use Briefcase to run your application:

macOS

```
(beeware-venv) $ briefcase run

[helloworld] Starting app...
===========================================================================
Configuring isolated Python...
Pre-initializing Python runtime...
PythonHome: /Users/brutus/beeware-tutorial/helloworld/macOS/app/Hello World/Hello World.
↪app/Contents/Resources/support/python-stdlib
PYTHONPATH:
- /Users/brutus/beeware-tutorial/helloworld/macOS/app/Hello World/Hello World.app/
↪Contents/Resources/support/python311.zip
```

(continues on next page)

(continued from previous page)

```
- /Users/brutus/beeware-tutorial/helloworld/macOS/app/Hello World/Hello World.app/
↪Contents/Resources/support/python-stdlib
- /Users/brutus/beeware-tutorial/helloworld/macOS/app/Hello World/Hello World.app/
↪Contents/Resources/support/python-stdlib/lib-dynload
- /Users/brutus/beeware-tutorial/helloworld/macOS/app/Hello World/Hello World.app/
↪Contents/Resources/app_packages
- /Users/brutus/beeware-tutorial/helloworld/macOS/app/Hello World/Hello World.app/
↪Contents/Resources/app
Configure argc/argv...
Initializing Python runtime...
Installing Python NSLog handler...
Running app module: helloworld
---------------------------------------------------------------------------
```

Linux

```
(beeware-venv) $ briefcase run

[helloworld] Finalizing application configuration...
Targeting ubuntu:jammy (Vendor base debian)
Determining glibc version... done

Targeting glibc 2.35
Targeting Python3.10

[helloworld] Starting app...
===========================================================================
Install path: /home/brutus/beeware-tutorial/helloworld/build/helloworld/linux/ubuntu/
↪jammy/helloworld-0.0.1/usr
Pre-initializing Python runtime...
PYTHONPATH:
- /usr/lib/python3.10
- /usr/lib/python3.10/lib-dynload
- /home/brutus/beeware-tutorial/helloworld/build/helloworld/linux/ubuntu/jammy/
↪helloworld-0.0.1/usr/lib/helloworld/app
- /home/brutus/beeware-tutorial/helloworld/build/helloworld/linux/ubuntu/jammy/
↪helloworld-0.0.1/usr/lib/helloworld/app_packages
Configure argc/argv...
Initializing Python runtime...
Running app module: helloworld
---------------------------------------------------------------------------
```

Windows

```
(beeware-venv) C:\...>briefcase run

[helloworld] Starting app...

===========================================================================
Log started: 2023-04-23 04:47:45Z
PreInitializing Python runtime...
PythonHome: C:\Users\brutus\beeware-tutorial\helloworld\windows\app\Hello World\src
```

(continues on next page)

(continued from previous page)

```
PYTHONPATH:
- C:\Users\brutus\beeware-tutorial\helloworld\windows\app\Hello World\src\python39.zip
- C:\Users\brutus\beeware-tutorial\helloworld\windows\app\Hello World\src
- C:\Users\brutus\beeware-tutorial\helloworld\windows\app\Hello World\src\app_packages
- C:\Users\brutus\beeware-tutorial\helloworld\windows\app\Hello World\src\app
Configure argc/argv...
Initializing Python runtime...
Running app module: helloworld
---------------------------------------------------------------------------
```

This will start to run your native application, using the output of the `build` command.

You might notice some small differences in the way your application looks when it's running. For example, icons and the name displayed by the operating system may be slightly different to those you saw when running under developer mode. This is also because you're using the packaged application, not just running Python code. From the operating system's perspective, you're now running "an app", not "a Python program", and this is reflected in how the application appears.

2.4.4 Building your installer

You can now package your application for distribution, using the `package` command. The package command does any compilation that is required to convert the scaffolded project into a final, distributable product. Depending on the platform, this may involve compiling an installer, performing code signing, or doing other pre-distribution tasks.

macOS

```
(beeware-venv) $ briefcase package --adhoc-sign

[helloworld] Signing app with adhoc identity...
...
Signing build/helloworld/macos/app/Hello World.app
      100.0% • 00:07

[helloworld] Building DMG...
Signing dist/Hello World-0.0.1.dmg

[helloworld] Packaged dist/Hello World-0.0.1.dmg
```

The `dist` folder will contain a file named `Hello World-0.0.1.dmg`. If you locate this file in the Finder, and double click on its icon, you'll mount the DMG, giving you a copy of the Hello World app, and a link to your Applications folder for easy installation. Drag the app file into Applications, and you've installed your application. Send the DMG file to a friend, and they should be able to do the same.

In this example, we've used the `--adhoc-sign` option - that is, we're signing our application with *ad hoc* credentials - temporary credentials that will only work on your machine. We've done this to keep the tutorial simple. Setting up code signing identities is a little fiddly, and they're only *required* if you're intending to distribute your application to others. If we were publishing a real application for others to use, we would need to specify real credentials.

When you're ready to publish a real application, check out the Briefcase How-To guide on Setting up a macOS code signing identity

Linux

The output of the package step will be slightly different depending on your Linux distribution. If you're on a Debian-derived distribution, you'll see:

```
(beeware-venv) $ briefcase package

[helloworld] Finalizing application configuration...
Targeting ubuntu:jammy (Vendor base debian)
Determining glibc version... done

Targeting glibc 2.35
Targeting Python3.10

[helloworld] Building .deb package...
Write Debian package control file... done

dpkg-deb: building package 'helloworld' in 'helloworld-0.0.1.deb'.
Building Debian package... done

[helloworld] Packaged dist/helloworld_0.0.1-1~ubuntu-jammy_amd64.deb
```

The `dist` folder will contain the `.deb` file that was generated.

If you're on a RHEL-based distribution, you'll see:

```
(beeware-venv) $ briefcase package

[helloworld] Finalizing application configuration...
Targeting fedora:36 (Vendor base rhel)
Determining glibc version... done

Targeting glibc 2.35
Targeting Python3.10

[helloworld] Building .rpm package...
Generating rpmbuild layout... done

Write RPM spec file... done

Building source archive... done

Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.Kav9H7
+ umask 022
...
+ exit 0
Building RPM package... done

[helloworld] Packaged dist/helloworld-0.0.1-1.fc36.x86_64.rpm
```

The `dist` folder will contain the `.rpm` file that was generated.

If you're on an Arch-based distribution, you'll see:

```
(beeware-venv) $ briefcase package

[helloworld] Finalizing application configuration...
Targeting arch:rolling (Vendor base arch)
Determining glibc version... done
```

(continues on next page)

(continued from previous page)

```
Targeting glibc 2.37
Targeting Python3.10

[helloworld] Building .pkg.tar.zst package...
...
Building Arch package... done

[helloworld] Packaged dist/helloworld-0.0.1-1-x86_64.pkg.tar.zst
```

The `dist` folder will contain the `.pkg.tar.zst` file that was generated.

Other Linux distributions aren't currently supported for packaging.

If you want to build a package for a Linux distribution other than the one you're using, Briefcase can also help - but you'll need to install Docker.

Official installers for Docker Engine are available for a range of Unix distributions. Follow the instructions for your platform; however, ensure you don't install Docker in "rootless" mode.

Once you've installed Docker, you should be able to start an Linux container - for example:

```
$ docker run -it ubuntu:22.04
```

will show you a Unix prompt (something like `root@84444e31cff9:/#`) inside an Ubuntu 22.04 Docker container. Type Ctrl-D to exit Docker and return to your local shell.

Once you've got Docker installed, you can use Briefcase to build a package for any Linux distribution that Briefcase supports by passing in a Docker image as an argument. For example, to build a DEB package for Ubuntu 22.04 (Jammy), regardless of the operating system you're on, you can run:

```
$ briefcase package --target ubuntu:jammy
```

This will download the Docker image for your selected operating system, create a container that is able to run Briefcase builds, and build the app package inside the image. Once it's completed, the `dist` folder will contain the package for the target Linux distribution.

Windows

```
(beeware-venv) C:\...>briefcase package

[helloworld] Building MSI...
Compiling application manifest...
Compiling... done
Compiling application installer...
helloworld.wxs
helloworld-manifest.wxs
Compiling... done
Linking application installer...
Linking... done

[helloworld] Packaged dist\Hello_World-0.0.1.msi
```

Once this step completes, the `dist` folder will contain a file named `Hello_World-0.0.1.msi`. If you double click on this installer to run it, you should go through a familiar Windows installation process. Once this installation completes, there will be a "Hello World" entry in your start menu.

2.4.5 Next steps

We now have our application packaged for distribution on desktop platforms. But what happens when we need to update the code in our application? How do we get those updates into our packaged application? Turn to *Tutorial 4* to find out. . .

2.5 Tutorial 4 - Updating your application

In the last tutorial, we packaged our application as a native application. If you're dealing with a real-world app, that isn't going to be the end of the story - you'll likely do some testing, discover problems, and need to make some changes. Even if your application is perfect, you'll eventually want to publish version 2 of your application with improvements.

So - how do you update your installed app when you make code changes?

2.5.1 Updating application code

Our application currently prints to the console when you press the button. However, GUI applications shouldn't really use the console for output. They need to use dialogs to communicate with users.

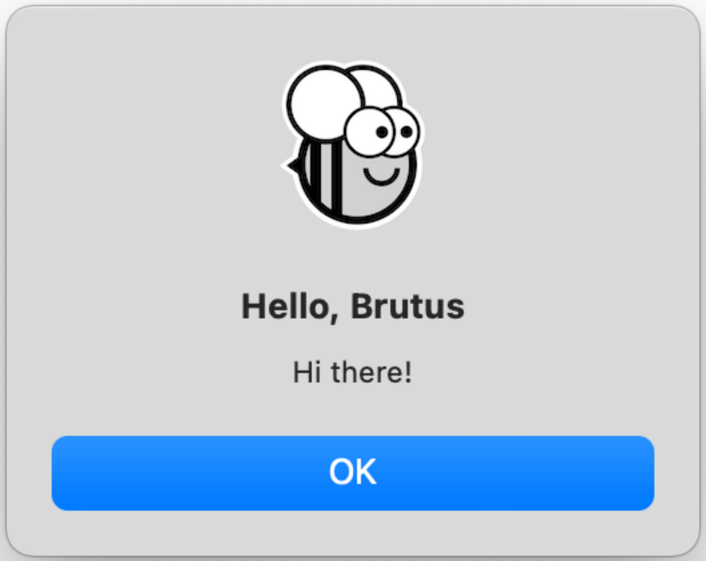
Let's add a dialog box to say hello, instead of writing to the console. Modify the `say_hello` callback so it looks like this:

```
def say_hello(self, widget):
    self.main_window.info_dialog(
        f"Hello, {self.name_input.value}",
        "Hi there!"
    )
```

This directs Toga to open a modal dialog box when the button is pressed.

If you run `briefcase dev`, enter a name, and press the button, you'll see the new dialog box:

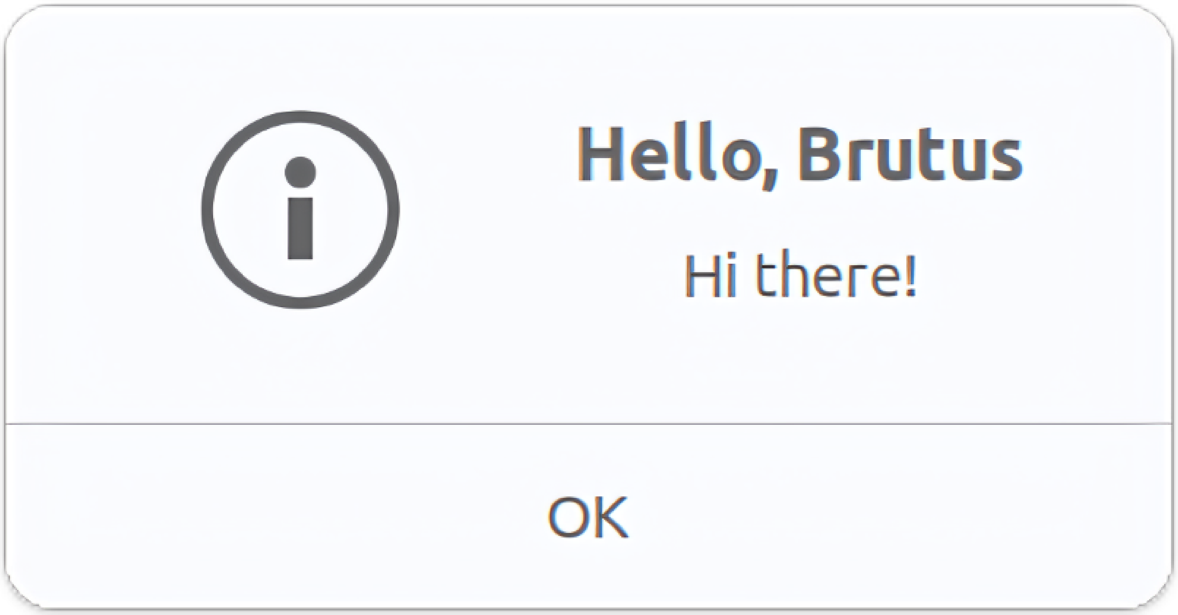
macOS



Hello, Brutus

Hi there!

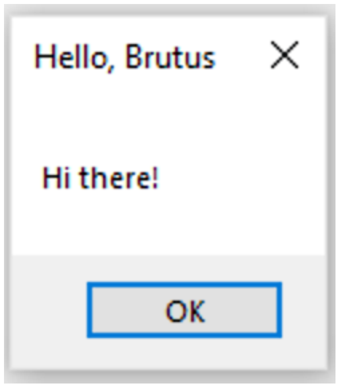
Linux



Hello, Brutus

Hi there!

Windows



However, if you run `briefcase run`, the dialog box won't appear.

Why is this? Well, `briefcase dev` operates by running your code in place - it tries to produce as realistic runtime environment for your code as possible, but it doesn't provide or use any of the platform infrastructure for wrapping your code as an application. Part of the process of packaging your app involves copying your code *into* the application bundle - and at the moment, your application still has the old code in it.

So - we need to tell briefcase to update your app, copying in the new version of the code. We *could* do this by deleting the old platform directory and starting from scratch. However, Briefcase provides an easier way - you can update the code for your existing bundled application:

macOS

```
(beeware-venv) $ briefcase update

[helloworld] Updating application code...
Installing src/helloworld... done

[helloworld] Removing unneeded app content...
Removing unneeded app bundle content... done

[helloworld] Application updated.
```

Linux

```
(beeware-venv) $ briefcase update

[helloworld] Updating application code...
Installing src/helloworld... done

[helloworld] Removing unneeded app content...
Removing unneeded app bundle content... done

[helloworld] Removing unneeded app content...
Removing unneeded app bundle content... done

[helloworld] Application updated.
```

Windows

```
(beeware-venv) C:\...>briefcase update

[helloworld] Updating application code...
```

(continues on next page)

(continued from previous page)

```
Installing src/helloworld... done

[helloworld] Removing unneeded app content...
Removing unneeded app bundle content... done

[helloworld] Application updated.
```

If Briefcase can't find the scaffolded template, it will automatically invoke `create` to generate a fresh scaffold.

Now that we've updated the installer code, we can then run `briefcase build` to re-compile the app, `briefcase run` to run the updated app, and `briefcase package` to repackage the application for distribution.

(macOS users, remember that as noted in *Tutorial 3*, for the tutorial we recommend running `briefcase package` with the `--adhoc-sign` flag to avoid the complexity of setting up a code signing identity and keep the tutorial as simple as possible.)

2.5.2 Update and run in one step

If you're rapidly iterating code changes, you'll likely want to make a code change, update the application, and immediately re-run your application. For most purposes, developer mode (`briefcase dev`) will be the easiest way to do this sort of rapid iteration; however, if you're testing something about how your application runs as a native binary, or hunting a bug that only manifests when your application is in packaged form, you may need to use repeated calls to `briefcase run`. To simplify the process of updating and running the bundled app, Briefcase has a shortcut to support this usage pattern - the `-u` (or `--update`) option on the `run` command.

Let's try making another change. You may have noticed that if you don't type a name in the text input box, the dialog will say "Hello, ". Let's modify the `say_hello` function again to handle this edge case.

At the top of the file, between the imports and the `class HelloWorld` definition, add a utility methods to generate an appropriate greeting depending on the value of the name that has been provided:

```python
def greeting(name):
    if name:
        return f"Hello, {name}"
    else:
        return "Hello, stranger"
```

Then, modify the `say_hello` callback to use this new utility method:

```python
def say_hello(self, widget):
    self.main_window.info_dialog(
        greeting(self.name_input.value),
        "Hi there!",
    )
```

Run your app in development mode (with `briefcase dev`) to confirm that the new logic works; then update, build and run the app with one command:

macOS

```
(beeware-venv) $ briefcase run -u

[helloworld] Updating application code...
Installing src/helloworld... done
```

(continues on next page)

(continued from previous page)

```
[helloworld] Removing unneeded app content...
Removing unneeded app bundle content... done

[helloworld] Application updated.

[helloworld] Building application...
...

[helloworld] Built build/helloworld/macos/app/Hello World.app

[helloworld] Starting app...
```

Linux

```
(beeware-venv) $ briefcase run -u

[helloworld] Finalizing application configuration...
Targeting ubuntu:jammy (Vendor base debian)
Determining glibc version... done

Targeting glibc 2.35
Targeting Python3.10

[helloworld] Updating application code...
Installing src/helloworld... done

[helloworld] Removing unneeded app content...
Removing unneeded app bundle content... done

[helloworld] Application updated.

[helloworld] Building application...
...

[helloworld] Built build/helloworld/linux/ubuntu/jammy/helloworld-0.0.1/usr/bin/
↪helloworld

[helloworld] Starting app...
```

Windows

```
(beeware-venv) C:\...>briefcase run -u

[helloworld] Updating application code...
Installing src/helloworld... done

[helloworld] Removing unneeded app content...
Removing unneeded app bundle content... done

[helloworld] Application updated.

[helloworld] Starting app...
```

The package command also accepts the `-u` argument, so if you make a change to your application code and want to repackage immediately, you can run `briefcase package -u`.

2.5.3 Next steps

We now have our application packaged for distribution on desktop platforms, and we've been able to update the code in our application.

But what about mobile? In *Tutorial 5*, we'll convert our application into a mobile application, and deploy it onto a device simulator, and onto a phone.

2.6 Tutorial 5 - Taking it Mobile

So far, we've been running and testing our application on the desktop. However, BeeWare also supports mobile platforms - and the application we've written can be deployed to your mobile device, too!

iOS iOS applications can only be compiled on macOS.

Let's build our app for iOS!

Android Android applications can be compiled on macOS, Windows or Linux.

Let's build our app for Android!

2.6.1 Tutorial 5 - Taking it mobile: iOS

To compile iOS applications we'll need Xcode, which is available for free from the macOS App Store.

Once we've got Xcode installed, we can take our application and deploy it as an iOS app.

The process of deploying an application to iOS is very similar to the process for deploying as a desktop application. First, you run the `create` command - but this time, we specify that we want to create an iOS application:

```
(beeware-venv) $ briefcase create iOS

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-iOS-Xcode-template.git, branch␣
↪v0.3.14
...

[helloworld] Installing support package...
...

[helloworld] Installing application code...
Installing src/helloworld... done

[helloworld] Installing requirements...
...

[helloworld] Installing application resources...
...

[helloworld] Removing unneeded app content...
```

(continues on next page)

(continued from previous page)

```
...

[helloworld] Created build/helloworld/ios/xcode
```

Once this completes, we'll have a `build/helloworld/ios/xcode` directory containing an Xcode project, as well as the support libraries and the application code needed for the application.

You can then use Briefcase to compile your app using `briefcase build iOS`:

```
(beeware-venv) $ briefcase build iOS

[helloworld] Updating app metadata...
Setting main module... done

[helloworld] Building Xcode project...
...
Building... done

[helloworld] Built build/helloworld/ios/xcode/build/Debug-iphonesimulator/Hello World.app
```

We're now ready to run our application, using `briefcase run iOS`. You'll be prompted to select a device to compile for; if you've got simulators for multiple iOS SDK versions installed, you may also be asked which iOS version you want to target. The options you are shown may differ from the options show in this output - at the very least, the list of devices will likely be different. For our purposes, it doesn't matter which simulator you pick.

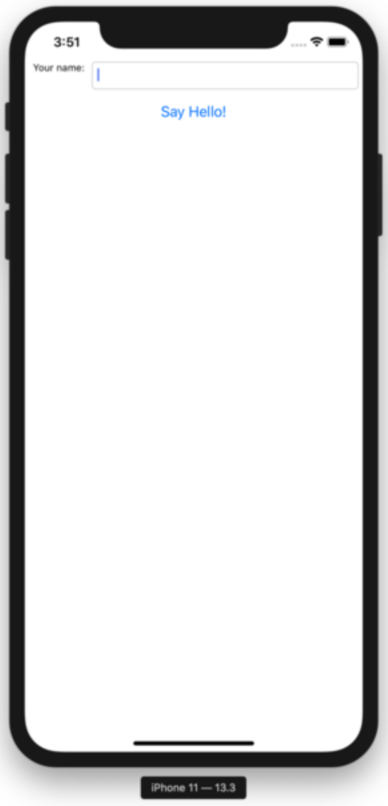
```
(beeware-venv) $ briefcase run iOS

Select simulator device:

   1) iPad (10th generation)
   2) iPad Air (5th generation)
   3) iPad Pro (11-inch) (4th generation)
   4) iPad Pro (12.9-inch) (6th generation)
   5) iPad mini (6th generation)
   6) iPhone 14
   7) iPhone 14 Plus
   8) iPhone 14 Pro
   9) iPhone 14 Pro Max
   10) iPhone SE (3rd generation)

>  10

In the future, you could specify this device by running:

    $ briefcase run iOS -d "iPhone SE (3rd generation)::iOS 16.2"

or:

    $ briefcase run iOS -d 2614A2DD-574F-4C1F-9F1E-478F32DE282E

[helloworld] Starting app on an iPhone SE (3rd generation) running iOS 16.2 (device UDID␣
↪2614A2DD-574F-4C1F-9F1E-478F32DE282E)
Booting simulator... done
```

(continues on next page)

(continued from previous page)

```
Opening simulator... done

[helloworld] Installing app...
Uninstalling any existing app version... done
Installing new app version... done

[helloworld] Starting app...
Launching app... done

[helloworld] Following simulator log output (type CTRL-C to stop log)...
===========================================================================
...
```

This will start the iOS simulator, install your app, and start it. You should see the simulator start, and eventually open your iOS application:



If you know ahead of time which iOS simulator you want to target, you can tell Briefcase to use that simulator by providing a `-d` (or `--device`) option. Using the name of the device you selected when you built your application, run:

```
$ briefcase run iOS -d "iPhone SE (3rd generation)"
```

If you have multiple iOS versions available, Briefcase will pick the highest iOS version; if you want to pick a particular iOS version, you tell it to use that specific version:

```
$ briefcase run iOS -d "iPhone SE (3rd generation)::iOS 15.5"
```

Or, you can name a specific device UDID:

```
$ briefcase run iOS -d 2614A2DD-574F-4C1F-9F1E-478F32DE282E
```

Next steps

We've now got an application on our phone! Is there anywhere else we can deploy a BeeWare app? Turn to *Tutorial 6* to find out…

2.6.2 Tutorial 5 - Taking it mobile: Android

Now, we're going to take our application, and deploy it as an Android application.

The process of deploying an application to Android is very similar to the process for deploying as a desktop application. Briefcase handles installing dependencies for Android, including the Android SDK, the Android emulator, and a Java compiler.

Create an Android app and compile it

First, run the `create` command. This downloads an Android app template and adds your Python code to it.

macOS

```
(beeware-venv) $ briefcase create android

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-android-gradle-template.git,␣
↪branch v0.3.14
...

[helloworld] Installing support package...
No support package required.

[helloworld] Installing application code...
Installing src/helloworld... done

[helloworld] Installing requirements...
Writing requirements file... done

[helloworld] Installing application resources...
...

[helloworld] Removing unneeded app content...
Removing unneeded app bundle content... done

[helloworld] Created build/helloworld/android/gradle
```

Linux

```
(beeware-venv) $ briefcase create android

[helloworld] Generating application template...
```

(continues on next page)

(continued from previous page)

```
Using app template: https://github.com/beeware/briefcase-android-gradle-template.git,␣
↪branch v0.3.14
...

[helloworld] Installing support package...
No support package required.

[helloworld] Installing application code...
Installing src/helloworld... done

[helloworld] Installing requirements...
Writing requirements file... done

[helloworld] Installing application resources...
...

[helloworld] Removing unneeded app content...
Removing unneeded app bundle content... done

[helloworld] Created build/helloworld/android/gradle
```

Windows

```
(beeware-venv) C:\...>briefcase create android

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-android-gradle-template.git,␣
↪branch v0.3.14
...

[helloworld] Installing support package...
No support package required.

[helloworld] Installing application code...
Installing src/helloworld... done

[helloworld] Installing requirements...
Writing requirements file... done

[helloworld] Installing application resources...
...

[helloworld] Removing unneeded app content...
Removing unneeded app bundle content... done

[helloworld] Created build\helloworld\android\gradle
```

When you run `briefcase create android` for the first time, Briefcase downloads a Java JDK, and the Android SDK. File sizes and download times can be considerable; this may take a while (10 minutes or longer, depending on the speed of your Internet connection). When the download has completed, you will be prompted to accept Google's Android SDK license.

Once this completes, we'll have a `build\helloworld\android\gradle` directory in our project, which will contain an Android project with a Gradle build configuration. This project will contain your application code, and a support package containing the Python interpreter.

We can then use Briefcase's `build` command to compile this into an Android APK app file.

macOS

```
(beeware-venv) $ briefcase build android

[helloworld] Updating app metadata...
Setting main module... done

[helloworld] Building Android APK...
Starting a Gradle Daemon
...
BUILD SUCCESSFUL in 1m 1s
28 actionable tasks: 17 executed, 11 up-to-date
Building... done

[helloworld] Built build/helloworld/android/gradle/app/build/outputs/apk/debug/app-debug.
↪apk
```

Linux

```
(beeware-venv) $ briefcase build android

[helloworld] Updating app metadata...
Setting main module... done

[helloworld] Building Android APK...
Starting a Gradle Daemon
...
BUILD SUCCESSFUL in 1m 1s
28 actionable tasks: 17 executed, 11 up-to-date
Building... done

[helloworld] Built build/helloworld/android/gradle/app/build/outputs/apk/debug/app-debug.
↪apk
```

Windows

```
(beeware-venv) C:\...>briefcase build android

[helloworld] Updating app metadata...
Setting main module... done

[helloworld] Building Android APK...
Starting a Gradle Daemon
...
BUILD SUCCESSFUL in 1m 1s
28 actionable tasks: 17 executed, 11 up-to-date
Building... done

[helloworld] Built build\helloworld\android\gradle\app\build\outputs\apk\debug\app-debug.
↪apk
```

Gradle may look stuck

During the `briefcase build android` step, Gradle (the Android platform build tool) will print `CONFIGURING: 100%`, and appear to be doing nothing. Don't worry, it's not stuck - it's downloading more Android SDK components. Depending on your Internet connection speed, this may take another 10 minutes (or longer). This lag should only happen the very first time you run `build`; the tools are cached, and on your next build, the cached versions will be used.

Run the app on a virtual device

We're now ready to run our application. You can use Briefcase's `run` command to run the app on an Android device. Let's start by running on an Android emulator.

To run your application, run `briefcase run android`. When you do this, you'll be prompted with a list of devices that you could run the app on. The last item will always be an option to create a new Android emulator.

macOS

```
(beeware-venv) $ briefcase run android

Select device:

  1) Create a new Android emulator

>
```

Linux

```
(beeware-venv) $ briefcase run android

Select device:

  1) Create a new Android emulator

>
```

Windows

```
(beeware-venv) C:\...>briefcase run android

Select device:

  1) Create a new Android emulator

>
```

We can now choose our desired device. Select the "Create a new Android emulator" option, and accept the default choice for the device name (`beePhone`).

Briefcase `run` will automatically boot the virtual device. When the device is booting, you will see the Android logo:

Once the device has finished booting, Briefcase will install your app on the device. You will briefly see a launcher screen:

The app will then start. You'll see a splash screen while the app starts up:

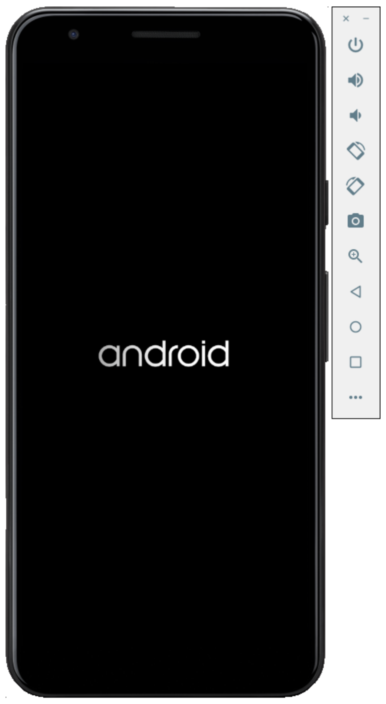


Fig. 1: Android virtual device booting

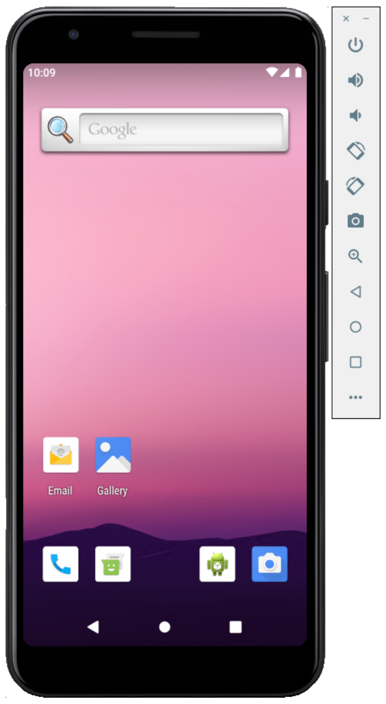


Fig. 2: Android virtual device fully started, on the launcher screen

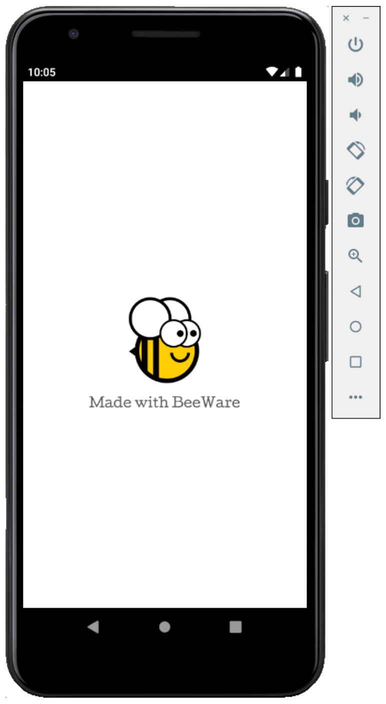


Fig. 3: App splash screen

The emulator didn't start!

The Android emulator is a complex piece of software that relies on a number of hardware and operating system features - features that may not be available or enabled on older machines. If you experience any difficulties starting the Android emulator, consult the Requirements and recommendations section of the Android developer documentation.

The first time the app starts, it needs to unpack itself onto the device. This may take a few seconds. Once it's unpacked, you'll see the Android version of our desktop app:

If you fail to see your app launching, you may need to check your terminal where you ran `briefcase run` and look for any error messages.

In future, if you want to run on this device without using the menu, you can provide the emulator's name to Briefcase, using `briefcase run android -d @beePhone` to run on the virtual device directly.

Run the app on a physical device

If you have a physical Android phone or tablet, you can connect it to your computer with a USB cable, and then use the Briefcase to target your physical device.

Android requires that you prepare your device before it can be used for development. You will need to make 2 changes to the options on your device:

- Enable developer options
- Enable USB debugging

Details on how to make these changes can be found in the Android developer documentation.

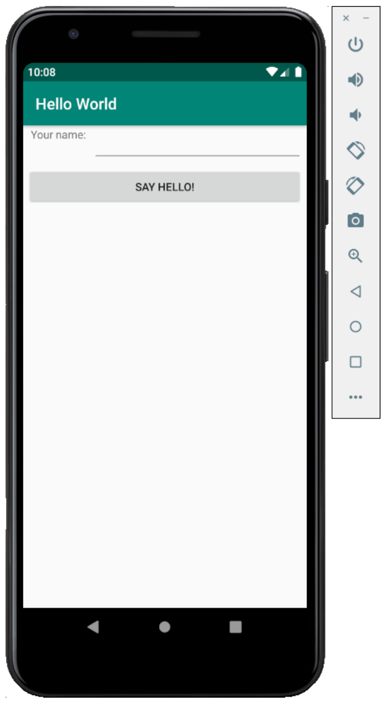


Fig. 4: Demo app fully launched

Once these steps have been completed, your device should appear in the list of available devices when you run `briefcase run android`.

macOS

```
(beeware-venv) $ briefcase run android

Select device:

  1) Pixel 3a (94ZZY0LNE8)
  2) @beePhone (emulator)
  3) Create a new Android emulator

>
```

Linux

```
(beeware-venv) $ briefcase run android

Select device:

  1) Pixel 3a (94ZZY0LNE8)
  2) @beePhone (emulator)
  3) Create a new Android emulator

>
```

Windows

```
(beeware-venv) C:\...>briefcase run android

Select device:

  1) Pixel 3a (94ZZY0LNE8)
  2) @beePhone (emulator)
  3) Create a new Android emulator

>
```

Here we can see a new physical device with it's serial number on the deployment list - in this case, a Pixel 3a. In the future, if you want to run on this device without using the menu, you can provide the phone's serial number to Briefcase (in this case, `briefcase run android -d 94ZZY0LNE8`). This will run on the device directly, without prompting.

My device doesn't appear!

If your device doesn't appear on this list at all, either you haven't enabled USB debugging, (or the device isn't plugged in!).

If your device appears, but is listed as "Unknown device (not authorized for development)", developer mode hasn't been correctly enabled. Re-run the steps to enable developer options, and re-run `briefcase run android`.

Next steps

We've now got an application on our phone! Is there anywhere else we can deploy a BeeWare app? Turn to *Tutorial 6* to find out…

2.7 Tutorial 6 - Put it on the web!

In addition to supporting mobile platforms, the Toga widget toolkit also supports the web! Using the same API that you used to deploy your desktop and mobile applications, you can deploy your application as a single-page web app.

Proof of Concept

The Toga Web backend is the least mature of all the Toga backends. It's mature enough to show off a few features, but it's likely to be buggy, and will be missing many of the widgets that are available on other platforms. At this point in time, Web deployment should be considered a "Proof of Concept" - enough to demonstrate what can be done, but not enough to be relied on for serious development.

If you have problems with this step of the tutorial, you can skip to the next page.

2.7.1 Deploying as a web app

The process of deploying as a single-page web app follows the same familiar pattern - you create the application, then build the application, then run it. However, Briefcase can be a little bit smart; if you attempt to run an application, and Briefcase determines that it hasn't been created or built for the platform being targeted, it will do the create and build steps for you. Since this is our first time running the app for the web, we can perform all three steps with one command:

macOS

```
(beeware-venv) $ briefcase run web

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-web-static-template.git, branch␣
↪v0.3.14
...

[helloworld] Installing support package...
No support package required.

[helloworld] Installing application code...
Installing src/helloworld... done

[helloworld] Installing requirements...
Writing requirements file... done

[helloworld] Installing application resources...
...

[helloworld] Removing unneeded app content...
Removing unneeded app bundle content... done

[helloworld] Created build/helloworld/web/static

[helloworld] Building web project...
...

[helloworld] Built build/helloworld/web/static/www/index.html

[helloworld] Starting web server...
Web server open on http://127.0.0.1:8080

[helloworld] Web server log output (type CTRL-C to stop log)...
===========================================================================
```

Linux

```
(beeware-venv) $ briefcase run web

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-web-static-template.git, branch␣
↪v0.3.14
...

[helloworld] Installing support package...
```

(continues on next page)

(continued from previous page)

```
No support package required.

[helloworld] Installing application code...
Installing src/helloworld... done

[helloworld] Installing requirements...
Writing requirements file... done

[helloworld] Installing application resources...
...

[helloworld] Removing unneeded app content...
Removing unneeded app bundle content... done

[helloworld] Created build/helloworld/web/static

[helloworld] Building web project...
...

[helloworld] Built build/helloworld/web/static/www/index.html

[helloworld] Starting web server...
Web server open on http://127.0.0.1:8080

[helloworld] Web server log output (type CTRL-C to stop log)...
===========================================================================
```

Windows

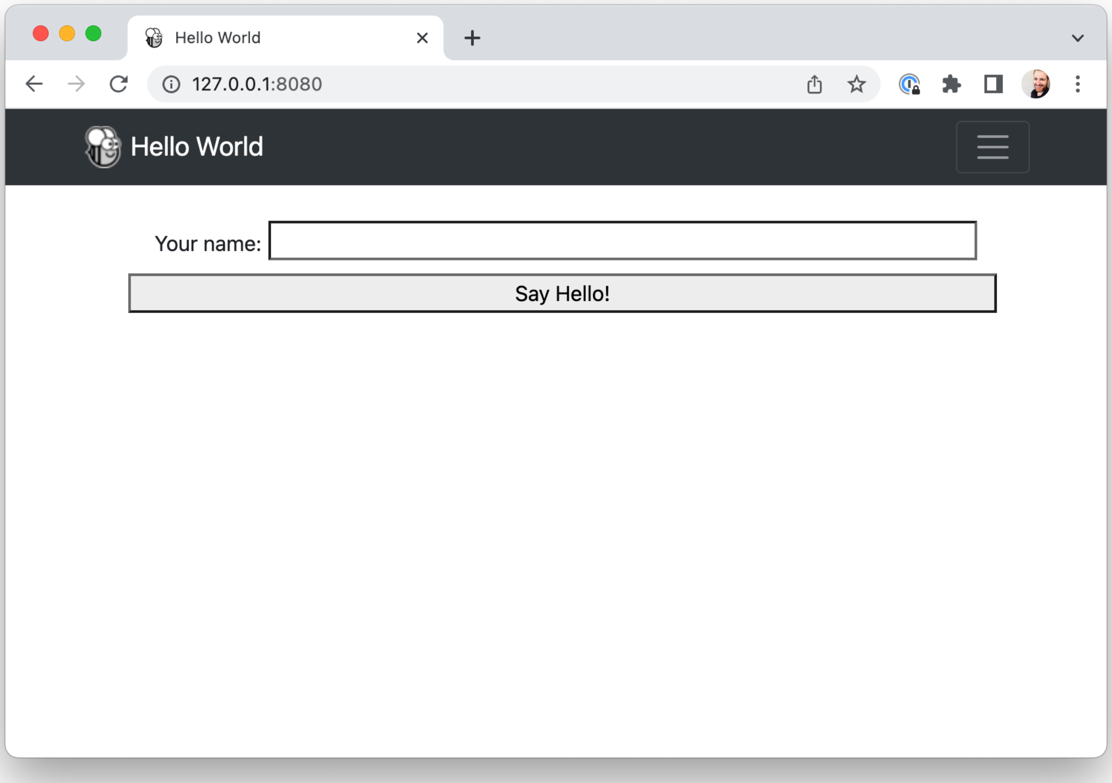
```
(beeware-venv) C:\...>briefcase run web

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-web-static-template.git, branch␣
↪v0.3.14
...

[helloworld] Installing support package...
No support package required.

[helloworld] Installing application code...
Installing src/helloworld... done

[helloworld] Installing requirements...
Writing requirements file... done

[helloworld] Installing application resources...
...

[helloworld] Removing unneeded app content...
Removing unneeded app bundle content... done

[helloworld] Created build\helloworld\web\static
```

(continues on next page)

(continued from previous page)

```
[helloworld] Building web project...
...

[helloworld] Built build\helloworld\web\static\www\index.html

[helloworld] Starting web server...
Web server open on http://127.0.0.1:8080

[helloworld] Web server log output (type CTRL-C to stop log)...
===========================================================================
```

This will open a web browser, pointing at http://127.0.0.1:8080:



If you enter your name and click the button, a dialog will appear.

2.7.2 How does this work?

This web app is a static website - a single HTML source page, with some CSS and other resources. Briefcase has started a local web server to serve this page so your browser can view the page. If you wanted to put this web page into production, you could copy the contents of the `www` folder onto any web server that can serve static content.

But when you press the button, you're running Python code… how does that work? Toga uses PyScript to provide a Python interpreter in the browser. Briefcase packages your app's code as wheels that PyScript can load in the browser. When the page is loaded, the application code runs in the browser, building the UI using the browser DOM. When you click a button, that button runs the event handling code in the browser.

2.7.3 Next steps

Although we've now deployed this app on desktop, mobile and the web, the app is fairly simple, and doesn't involve any third-party libraries. Can we include libraries from the Python Package Index (PyPI) in our app? Turn to *Tutorial 7* to find out…

2.8 Tutorial 7 - Get this (third)-party started

So far, the app we've built has only used our own code, plus the code provided by BeeWare. However, in a real-world app, you'll likely want to use a third-party library, downloaded from the Python Package Index (PyPI).

Let's modify our app to include a third-party library.

2.8.1 Accessing an API

A common task an app will need to perform is to make a request on a web API to retrieve data, and display that data to the user. This is a toy app, so we don't have a *real* API to work with, so we'll use the {JSON} Placeholder API as a source of data.

The {JSON} Placeholder API has a number of "fake" API endpoints you can use as test data. One of those APIs is the `/posts/` endpoint, which returns fake blog posts. If you open `https://jsonplaceholder.typicode.com/posts/42` in your browser, you'll get a JSON payload describing a single post - some Lorum ipsum content for a blog post with ID 42.

The Python standard library contains all the tools you'd need to access an API. However, the built-in APIs are very low level. They are good implementations of the HTTP protocol - but they require the user to manage lots of low-level details, like URL redirection, sessions, authentication, and payload encoding. As a "normal browser user" you're probably used to taking these details for granted, as a browser manages these details for you.

As a result, people have developed third-party libraries that wrap the built-in APIs and provide a simpler API that is a closer match for the everyday browser experience. We're going to use one of those libraries to access the {JSON} Placeholder API - a library called httpx.

Let's add a `httpx` API call to our app. Add an import to the top of the `app.py` to import `httpx`:

```python
import httpx
```

Then modify the `say_hello()` callback so it looks like this:

```python
def say_hello(self, widget):
    with httpx.Client() as client:
        response = client.get("https://jsonplaceholder.typicode.com/posts/42")
```

(continues on next page)

(continued from previous page)

```
    payload = response.json()

    self.main_window.info_dialog(
        greeting(self.name_input.value),
        payload["body"],
    )
```

This will change the say_hello() callback so that when it is invoked, it will:

- make a GET request on the JSON placeholder API to obtain post 42;
- decode the response as JSON;
- extract the body of the post; and
- include the body of that post as the text of the dialog.

Lets run our updated app in Briefcase developer mode to check that our change has worked.

macOS

```
(beeware-venv) $ briefcase dev
Traceback (most recent call last):
File ".../venv/bin/briefcase", line 5, in <module>
    from briefcase.__main__ import main
File ".../venv/lib/python3.9/site-packages/briefcase/__main__.py", line 3, in <module>
    from .cmdline import parse_cmdline
File ".../venv/lib/python3.9/site-packages/briefcase/cmdline.py", line 6, in <module>
    from briefcase.commands import DevCommand, NewCommand, UpgradeCommand
File ".../venv/lib/python3.9/site-packages/briefcase/commands/__init__.py", line 1, in
↪<module>
    from .build import BuildCommand  # noqa
File ".../venv/lib/python3.9/site-packages/briefcase/commands/build.py", line 5, in
↪<module>
    from .base import BaseCommand, full_options
File ".../venv/lib/python3.9/site-packages/briefcase/commands/base.py", line 14, in
↪<module>
    import httpx
ModuleNotFoundError: No module named 'httpx'
```

Linux

```
(beeware-venv) $ briefcase dev
Traceback (most recent call last):
File ".../venv/bin/briefcase", line 5, in <module>
    from briefcase.__main__ import main
File ".../venv/lib/python3.9/site-packages/briefcase/__main__.py", line 3, in <module>
    from .cmdline import parse_cmdline
File ".../venv/lib/python3.9/site-packages/briefcase/cmdline.py", line 6, in <module>
    from briefcase.commands import DevCommand, NewCommand, UpgradeCommand
File ".../venv/lib/python3.9/site-packages/briefcase/commands/__init__.py", line 1, in
↪<module>
    from .build import BuildCommand  # noqa
File ".../venv/lib/python3.9/site-packages/briefcase/commands/build.py", line 5, in
```

(continues on next page)

(continued from previous page)

```
↪<module>
    from .base import BaseCommand, full_options
File ".../venv/lib/python3.9/site-packages/briefcase/commands/base.py", line 14, in
↪<module>
    import httpx
ModuleNotFoundError: No module named 'httpx'
```

Windows

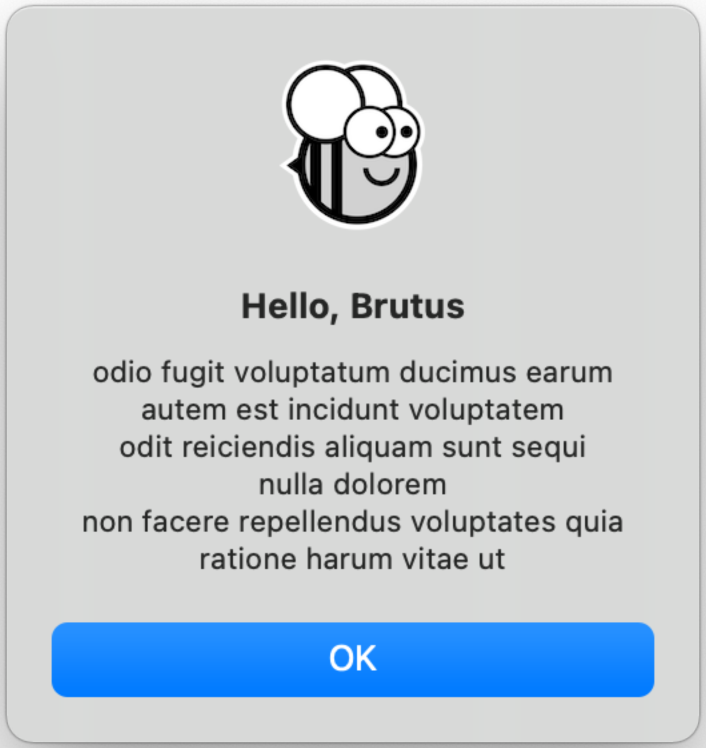
```
(beeware-venv) C:\...>briefcase dev
Traceback (most recent call last):
File "...\venv\bin\briefcase", line 5, in <module>
    from briefcase.__main__ import main
File "...\venv\lib\python3.9\site-packages\briefcase\__main__.py", line 3, in <module>
    from .cmdline import parse_cmdline
File "...\venv\lib\python3.9\site-packages\briefcase\cmdline.py", line 6, in <module>
    from briefcase.commands import DevCommand, NewCommand, UpgradeCommand
File "...\venv\lib\python3.9\site-packages\briefcase\commands\__init__.py", line 1, in
↪<module>
    from .build import BuildCommand  # noqa
File "...\venv\lib\python3.9\site-packages\briefcase\commands\build.py", line 5, in
↪<module>
    from .base import BaseCommand, full_options
File "...\venv\lib\python3.9\site-packages\briefcase\commands\base.py", line 14, in
↪<module>
    import httpx
ModuleNotFoundError: No module named 'httpx'
```

What happened? We've added `httpx` to our *code*, but we haven't added it to our development virtual environment. We can fix this by installing `httpx` with `pip`, and then re-running `briefcase dev`:

macOS

```
(beeware-venv) $ python -m pip install httpx
(beeware-venv) $ briefcase dev
```

When you enter a name and press the button, you should see a dialog that looks something like:



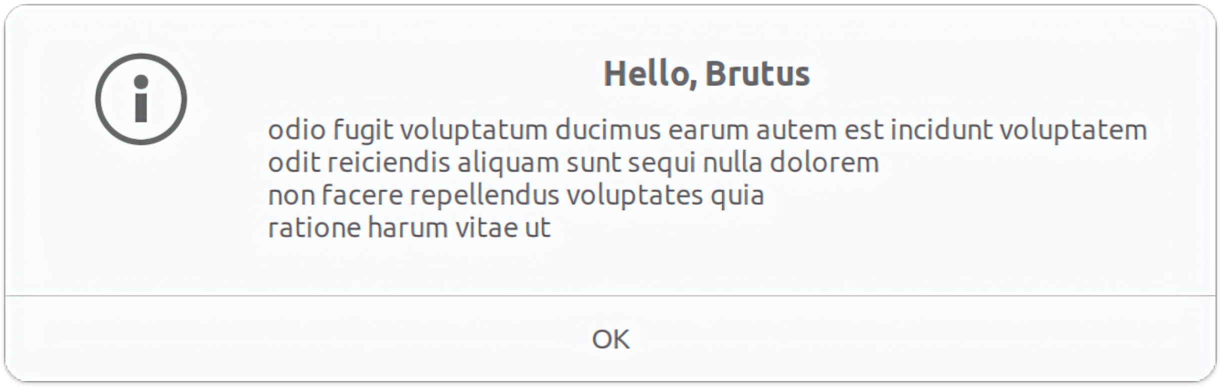
Hello, Brutus

odio fugit voluptatum ducimus earum
autem est incidunt voluptatem
odit reiciendis aliquam sunt sequi
nulla dolorem
non facere repellendus voluptates quia
ratione harum vitae ut

Linux

```
(beeware-venv) $ python -m pip install httpx
(beeware-venv) $ briefcase dev
```

When you enter a name and press the button, you should see a dialog that looks something like:



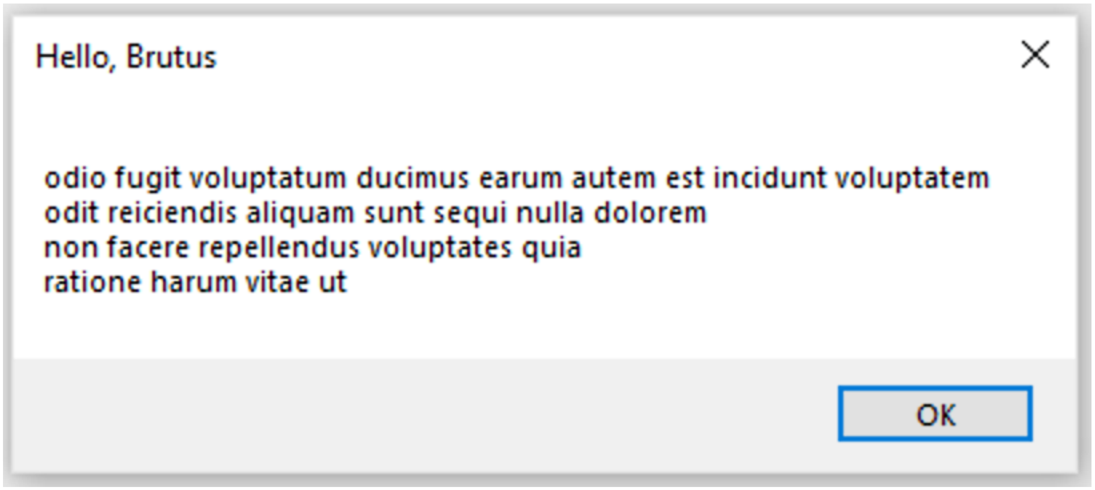
Hello, Brutus

odio fugit voluptatum ducimus earum autem est incidunt voluptatem
odit reiciendis aliquam sunt sequi nulla dolorem
non facere repellendus voluptates quia
ratione harum vitae ut

Windows

```
(beeware-venv) C:\...>python -m pip install httpx
(beeware-venv) C:\...>briefcase dev
```

When you enter a name and press the button, you should see a dialog that looks something like:


Hello, Brutus

odio fugit voluptatum ducimus earum autem est incidunt voluptatem
odit reiciendis aliquam sunt sequi nulla dolorem
non facere repellendus voluptates quia
ratione harum vitae ut

We've now got a working app, using a third party library, running in development mode!

2.8.2 Running the updated app

Let's get this updated application code packaged as a standalone app. Since we've made code changes, we need to follow the same steps as in *Tutorial 4*:

macOS

Update the code in the packaged app:

```
(beeware-venv) $ briefcase update

[helloworld] Updating application code...
...

[helloworld] Application updated.
```

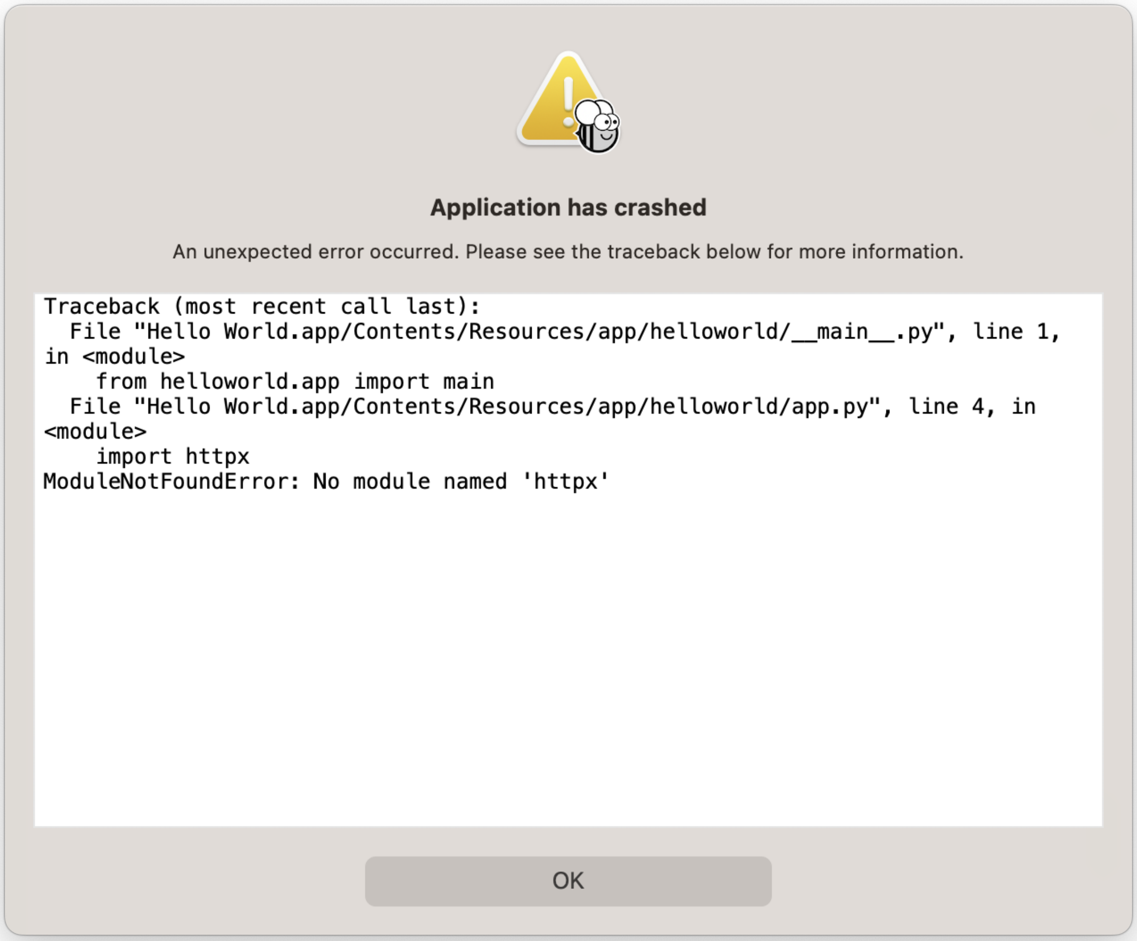
Rebuild the app:

```
(beeware-venv) $ briefcase build

[helloworld] Adhoc signing app...
[helloworld] Built build/helloworld/macos/app/Hello World.app
```

And finally, run the app:

```
(beeware-venv) $ briefcase run

[helloworld] Starting app...
===========================================================================
```

However, when the app runs, you'll see an error in the console, plus a crash dialog:



An unexpected error occurred. Please see the traceback below for more information.

```
Traceback (most recent call last):
  File "Hello World.app/Contents/Resources/app/helloworld/__main__.py", line 1,
in <module>
    from helloworld.app import main
  File "Hello World.app/Contents/Resources/app/helloworld/app.py", line 4, in
<module>
    import httpx
ModuleNotFoundError: No module named 'httpx'
```

Linux

Update the code in the packaged app:

```
(beeware-venv) $ briefcase update

[helloworld] Updating application code...
...

[helloworld] Application updated.
```

Rebuild the app:

```
(beeware-venv) $ briefcase build

[helloworld] Finalizing application configuration...
...

[helloworld] Building application...
...
```

(continues on next page)

(continued from previous page)

```
[helloworld] Built build/helloworld/linux/ubuntu/jammy/helloworld-0.0.1/usr/bin/
↪helloworld
```

And finally, run the app:

```
(beeware-venv) $ briefcase run

[helloworld] Starting app...
===========================================================================
```

However, when the app runs, you'll see an error in the console:

```
Traceback (most recent call last):
  File "/usr/lib/python3.10/runpy.py", line 194, in _run_module_as_main
    return _run_code(code, main_globals, None,
  File "/usr/lib/python3.10/runpy.py", line 87, in _run_code
    exec(code, run_globals)
  File "/home/brutus/beeware-tutorial/helloworld/build/linux/ubuntu/jammy/helloworld-0.0.
↪1/usr/app/hello_world/__main__.py", line 1, in <module>
    from helloworld.app import main
  File "/home/brutus/beeware-tutorial/helloworld/build/linux/ubuntu/jammy/helloworld-0.0.
↪1/usr/app/hello_world/app.py", line 8, in <module>
    import httpx
ModuleNotFoundError: No module named 'httpx'

Unable to start app helloworld.
```

Windows

Update the code in the packaged app:

```
(beeware-venv) C:\...>briefcase update

[helloworld] Updating application code...
...

[helloworld] Application updated.
```

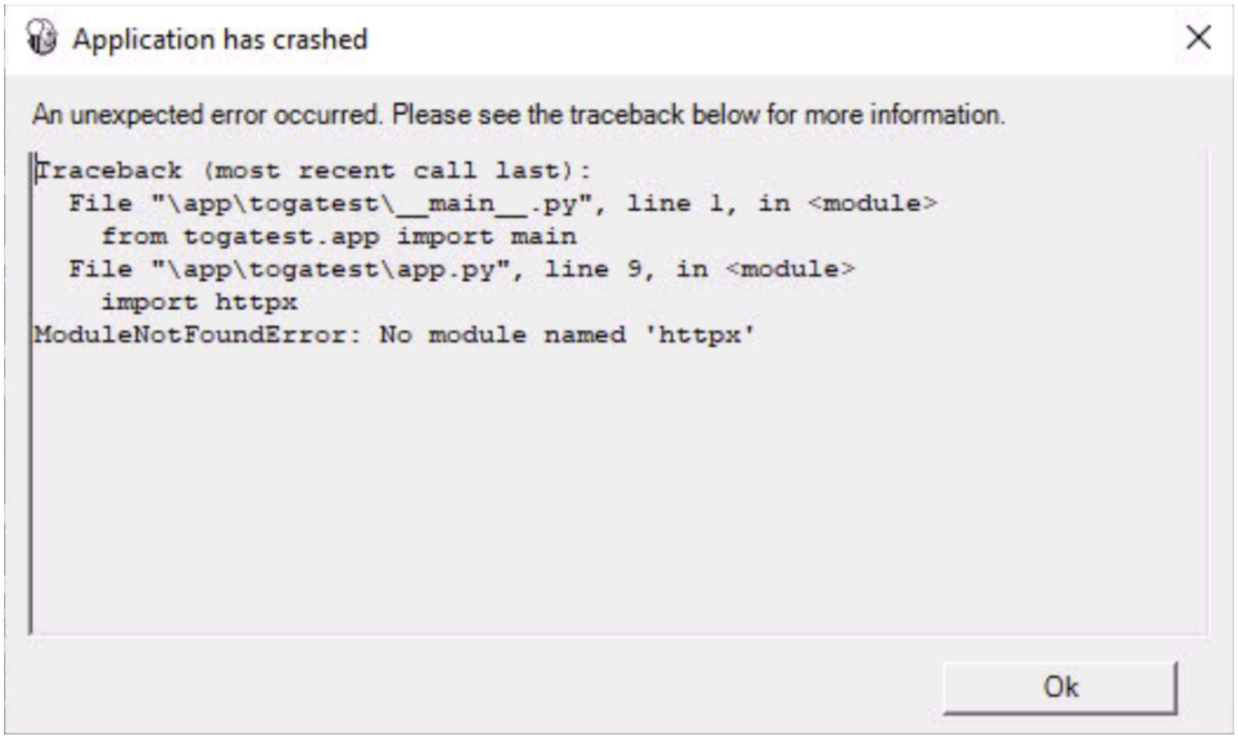
Rebuild the app:

```
(beeware-venv) C:\...>briefcase build
...

[helloworld] Built build\helloworld\windows\app\src\Toga Test.exe
```

And finally, run the app:

```
(beeware-venv) C:\...>briefcase run

[helloworld] Starting app...
===========================================================================
```

However, when the app runs, you'll see an error in the console, plus a crash dialog:



An unexpected error occurred. Please see the traceback below for more information.

```
Traceback (most recent call last):
  File "\app\togatest\__main__.py", line 1, in <module>
    from togatest.app import main
  File "\app\togatest\app.py", line 9, in <module>
    import httpx
ModuleNotFoundError: No module named 'httpx'
```

Once again, the app has failed to start because `httpx` has been installed - but why? Haven't we already installed `httpx`?

We have - but only in the development environment. Your development environment is entirely local to your machine - and is only enabled when you explicitly activate it. Although Briefcase has a development mode, the main reason you'd use Briefcase is to package up your code so you can give it to someone else.

The only way to guarantee that someone else will have a Python environment that contains everything it needs is to build a completely isolated Python environment. This means there's a completely isolated Python install, and a completely isolated set of dependencies. This is what Briefcase is building when you run `briefcase build` - an isolated Python environment. This also explains why `httpx` isn't installed - it has been installed in your *development* environment, but not in the packaged app.

So - we need to tell Briefcase that our app has an external dependency.

2.8.3 Updating dependencies

In the root directory of your app, there is a file named `pyproject.toml`. This file contains all the app configuration details that you provided when you originally ran `briefcase new`.

`pyproject.toml` is broken up into sections; one of the sections describes the settings for your app:

```
[tool.briefcase.app.helloworld]
formal_name = "Hello World"
description = "A Tutorial app"
icon = "src/helloworld/resources/helloworld"
sources = ["src/helloworld"]
requires = []
```

The `requires` option describes the dependencies of our application. It is a list of strings, specifying libraries (and, optionally, versions) of libraries that you want to be included with your app.

Modify the `requires` setting so that it reads:

```
requires = [
    "httpx",
]
```

By adding this setting, we're telling Briefcase "when you build my app, run `pip install httpx` into the application bundle". Anything that would be legal input to `pip install` can be used here - so, you could specify:

- A specific library version (e.g., `"httpx==0.19.0"`);
- A range of library versions (e.g., `"httpx>=0.19"`);
- A path to a git repository (e.g., `"git+https://github.com/encode/httpx"`); or
- A local file path (However - be warned: if you give your code to someone else, this path probably won't exist on their machine!)

Further down in `pyproject.toml`, you'll notice other sections that are operating system dependent, like `[tool.briefcase.app.helloworld.macOS]` and `[tool.briefcase.app.helloworld.windows]`. These sections *also* have a `requires` setting. These settings allow you to define additional platform-specific dependencies - so, for example, if you need a platform-specific library to handle some aspect of your app, you can specify that library in the platform-specific `requires` section, and that setting will only be used for that platform. You will notice that the `toga` libraries are all specified in the platform-specific `requires` section - this is because the libraries needed to display a user interface are platform specific.

In our case, we want `httpx` to be installed on all platforms, so we use the app-level `requires` setting. The app-level dependencies will always be installed; the platform-specific dependencies are installed *in addition* to the app-level ones.

Some binary packages may not be available

On desktop platforms (macOS, Windows, Linux), any `pip`-installable can be added to your requirements. On mobile and web platforms, your options are slightly limited.

In short; any *pure Python* package (i.e., packages that do *not* contain a binary module) can be used without difficulty. However, if your dependency contains a binary component, it must be compiled; at this time, most Python packages don't provide compilation support for non-desktop platforms.

BeeWare can provide binaries for some popular binary modules (including `numpy`, `pandas`, and `cryptography`). It's *usually* possible to compile packages for mobile platforms, but it's not easy to set up – well outside the scope of an introductory tutorial like this one.

Now that we've told Briefcase about our additional requirements, we can try packaging our app again. Ensure that you've saved your changes to `pyproject.toml`, and then update your app again - this time, passing in the `-r` flag. This tells Briefcase to update requirements in the packaged app:

macOS

```
(beeware-venv) $ briefcase update -r

[helloworld] Updating application code...
Installing src/hello_world...

[helloworld] Updating requirements...
Collecting httpx
  Using cached httpx-0.19.0-py3-none-any.whl (77 kB)
...
Installing collected packages: sniffio, idna, travertino, rfc3986, h11, anyio, toga-core,
↪ rubicon-objc, httpcore, charset-normalizer, certifi, toga-cocoa, httpx
```

(continues on next page)

(continued from previous page)

```
Successfully installed anyio-3.3.2 certifi-2021.10.8 charset-normalizer-2.0.6 h11-0.12.0␣
↪httpcore-0.13.7 httpx-0.19.0 idna-3.2 rfc3986-1.5.0 rubicon-objc-0.4.1 sniffio-1.2.0␣
↪toga-cocoa-0.3.0.dev28 toga-core-0.3.0.dev28 travertino-0.1.3

[helloworld] Removing unneeded app content...
...

[helloworld] Application updated.
```

Linux

```
(beeware-venv) $ briefcase update -r

[helloworld] Finalizing application configuration...
Targeting ubuntu:jammy (Vendor base debian)
Determining glibc version... done

Targeting glibc 2.35
Targeting Python3.10

[helloworld] Updating application code...
Installing src/hello_world...

[helloworld] Updating requirements...
Collecting httpx
  Using cached httpx-0.19.0-py3-none-any.whl (77 kB)
...
Installing collected packages: sniffio, idna, travertino, rfc3986, h11, anyio, toga-core,
↪ rubicon-objc, httpcore, charset-normalizer, certifi, toga-cocoa, httpx
Successfully installed anyio-3.3.2 certifi-2021.10.8 charset-normalizer-2.0.6 h11-0.12.0␣
↪httpcore-0.13.7 httpx-0.19.0 idna-3.2 rfc3986-1.5.0 rubicon-objc-0.4.1 sniffio-1.2.0␣
↪toga-cocoa-0.3.0.dev28 toga-core-0.3.0.dev28 travertino-0.1.3

[helloworld] Removing unneeded app content...
...

[helloworld] Application updated.
```

Windows

```
(beeware-venv) C:\...>briefcase update -r

[helloworld] Updating application code...
Installing src/helloworld...

[helloworld] Updating requirements...
Collecting httpx
  Using cached httpx-0.19.0-py3-none-any.whl (77 kB)
...
Installing collected packages: sniffio, idna, travertino, rfc3986, h11, anyio, toga-core,
↪ rubicon-objc, httpcore, charset-normalizer, certifi, toga-cocoa, httpx
Successfully installed anyio-3.3.2 certifi-2021.10.8 charset-normalizer-2.0.6 h11-0.12.0␣
```

(continues on next page)

(continued from previous page)

```
↪httpcore-0.13.7 httpx-0.19.0 idna-3.2 rfc3986-1.5.0 rubicon-objc-0.4.1 sniffio-1.2.0␣
↪toga-cocoa-0.3.0.dev28 toga-core-0.3.0.dev28 travertino-0.1.3

[helloworld] Removing unneeded app content...
...

[helloworld] Application updated.
```

Once you've updated, you can run `briefcase build` and `briefcase run` - and you should see your packaged app, with the new dialog behavior.

Note: The `-r` option for updating requirements is also honored by the `build` and `run` command, so if you want to update, build, and run in one step, you could use `briefcase run -u -r`.

2.8.4 Next steps

We've now got an app that uses a third-party library! However, you may have noticed that when you press the button, the app becomes a little unresponsive. Can we do anything to fix this? Turn to *Tutorial 8* to find out…

2.9 Tutorial 8 - Making it Smooooth

Unless you've got a *really* fast internet connection, you may notice that when you press the button, the GUI for your app locks up for a little bit. This is because the web request we have made is *synchronous*. When our application makes the web request, it waits for the API to return a response before continuing. While it's waiting, it *isn't* allowing the application to redraw - and as a result, the application locks up.

2.9.1 GUI Event Loops

To understand why this happens, we need to dig into the details of how a GUI application works. The specifics vary depending on the platform; but the high level concepts are the same, no matter the platform or GUI environment you're using.

A GUI app is, fundamentally, a single loop that looks something like:

```
while not app.quit_requested():
    app.process_events()
    app.redraw()
```

This loop is called the *Event Loop*. (These aren't actual method names - it's an illustration of what is going on in "pseudo-code").

When you click on a button, or drag a scroll bar, or type a key, you are generating an "event". That "event" is put onto a queue, and the app will process the queue of events when it next has the opportunity to do so. The user code that is triggered in response to the event is called an *event handler*. These event handlers are invoked as part of the `process_events()` call.

Once an app has processed all the available events, it will `redraw()` the GUI. This takes into account any changes that the events have caused to the display of the app, as well as anything else that is going on in the operating system - for example, the windows of another app may obscure or reveal part of our app's window, and our app's redraw will need to reflect the portion of the window that is currently visible.

The important detail to notice: while an application is processing an event, *it can't redraw*, and *it can't process other events*.

This means any user logic contained in an event handler needs to complete quickly. Any delay in completing the event handler will be observed by the user as a slowdown (or stop) in GUI updates. If this delay is long enough, your operating system may report this as a problem - the macOS "beachball" and Windows "spinner" icons are the operating system telling you that your app is taking too long in an event handler.

Simple operations like "update a label", or "recompute the total of the inputs" are easy to complete quickly. However, there are a lot of operations that can't be completed quickly. If you're performing a complex mathematical calculation, or indexing all the files on a file system, or performing a large network request, you can't "just do it quickly" - the operations are inherently slow.

So - how do we perform long-lived operations in a GUI application?

2.9.2 Asynchronous programming

What we need is a way to tell an app in the middle of a long-lived event handler that it is OK to temporarily release control back to the event loop, as long as we can resume where we left off. It's up to the app to determine when this release can occur; but if the app releases control to the event loop regularly, we can have a long-running event handler *and* maintain a responsive UI.

We can do this by using *asynchronous programming*. Asynchronous programming is a way to describe a program that allows the interpreter to run multiple functions at the same time, sharing resources between all the concurrently running functions.

Asynchronous functions (known as *co-routines*) need to be explicitly declared as being asynchronous. They also need to internally declare when an opportunity exists to change context to another co-routine.

In Python, asynchronous programming is implemented using the `async` and `await` keywords, and the asyncio module in the standard library. The `async` keyword allows us to declare that a function is an asynchronous co-routine. The `await` keyword provides a way to declare when an opportunity exists to change context to another co-routine. The asyncio module provides some other useful tools and primitives for asynchronous coding.

2.9.3 Making the tutorial Asynchronous

To make our tutorial asynchronous, modify the `say_hello()` event handler so it looks like this:

```python
async def say_hello(self, widget):
    async with httpx.AsyncClient() as client:
        response = await client.get("https://jsonplaceholder.typicode.com/posts/42")

    payload = response.json()

    self.main_window.info_dialog(
        greeting(self.name_input.value),
        payload["body"],
    )
```

There are only 4 changes in this code from the previous version:

1. The method is defined as `async def`, rather than just `def`. This tells Python that the method is an asynchronous co-routine.

2. The client that is created is an asynchronous `AsyncClient()`, rather than a synchronous `Client()`. This tells `httpx` that it should operate in asynchronous mode, rather than synchronous mode.

3. The context manager used to create the client is marked as `async`. This tells Python that there is an opportunity to release control as the context manager is entered and exited.
4. The `get` call is made with an `await` keyword. This instructs the app that while we are waiting for the response from the network, the app can release control to the event loop.

Toga allows you to use regular methods or asynchronous co-routines as handlers; Toga manages everything behind the scenes to make sure the handler is invoked or awaited as required.

If you save these changes and re-run the app (either with `briefcase dev` in development mode, or by updating and re-running the packaged app), there won't be any obvious changes to the app. However, when you click on the button to trigger the dialog, you may notice a number of subtle improvements:

- The button returns to an "unclicked" state, rather than being stuck in a "clicked" state.
- The "beachball"/"spinner" icon won't appear
- If you move/resize the app window while waiting for the dialog to appear, the window will redraw.
- If you try to open an app menu, the menu will appear immediately.

2.9.4 Next steps

We've now got an application that is slick and responsive, even when it's waiting on a slow API. But how can we make sure that the app keeps working as we continue to develop it further? How do we test our app? Turn to *Tutorial 9* to find out…

2.10 Tutorial 9 - Testing times

Most software development doesn't involve writing new code - it's modifying existing code. Ensuring that existing code continues to work in the way we expect is a key part of the software development process. One way to do ensure the behavior of our app is with a *test suite*.

2.10.1 Running the test suite

It turns out our project already has a test suite! When we originally generated our project, two top-level directories were generated: `src` and `tests`. The `src` folder contains the code for our app; the `tests` folder contains our test suite. Inside the `tests` folder is a file named `test_app.py` with the following content:

```
def test_first():
    "An initial test for the app"
    assert 1 + 1 == 2
```

This is a Pytest *test case* - a block of code that can be executed to verify some behavior of your app. In this case, the test is a placeholder, and doesn't test anything about our app - but it is a test that we can perform.

We can run this test suite using the `--test` option to `briefcase dev`. As this is the first time we are running tests, we also need to pass in the `-r` option to ensure that the test requirements are also installed:

macOS

```
(beeware-venv) $ briefcase dev --test -r

[helloworld] Installing requirements...
...
```

(continues on next page)

(continued from previous page)

```
Installing dev requirements... done

[helloworld] Running test suite in dev environment...
===========================================================================
============================= test session starts ==============================
platform darwin -- Python 3.11.0, pytest-7.2.0, pluggy-1.0.0 -- /Users/brutus/beeware-
↪tutorial/beeware-venv/bin/python3.11
cachedir: /var/folders/b_/khqk71xd45d049kxc_59ltp80000gn/T/.pytest_cache
rootdir: /Users/brutus
plugins: anyio-3.6.2
collecting ... collected 1 item

tests/test_app.py::test_first PASSED                                    [100%]

============================== 1 passed in 0.01s ===============================
```

Linux

```
(beeware-venv) $ briefcase dev --test -r

[helloworld] Installing requirements...
...
Installing dev requirements... done

[helloworld] Running test suite in dev environment...
===========================================================================
============================= test session starts ==============================
platform linux -- Python 3.11.0
pytest==7.2.0
py==1.11.0
pluggy==1.0.0
cachedir: /tmp/.pytest_cache
rootdir: /home/brutus
plugins: anyio-3.6.2
collecting ... collected 1 item

tests/test_app.py::test_first PASSED                                    [100%]

============================== 1 passed in 0.01s ===============================
```

Windows

```
(beeware-venv) C:\...>briefcase dev --test -r

[helloworld] Installing requirements...
...
Installing dev requirements... done

[helloworld] Running test suite in dev environment...
===========================================================================
============================= test session starts ==============================
platform win32 -- Python 3.11.0
```

(continues on next page)

(continued from previous page)

```
pytest==7.2.0
py==1.11.0
pluggy==1.0.0
cachedir: C:\Users\brutus\AppData\Local\Temp\.pytest_cache
rootdir: C:\Users\brutus
plugins: anyio-3.6.2
collecting ... collected 1 item

tests/test_app.py::test_first PASSED                                     [100%]

============================== 1 passed in 0.01s ===============================
```

Success! We've just executed a single test that verifies Python math works in the way we'd expect (What a relief!).

Let's replace this placeholder test with a test to verify that our `greeting()` method behaves the way we'd expect. Replace the contents of `test_app.py` with the following:

```python
from helloworld.app import greeting


def test_name():
    """If a name is provided, the greeting includes the name"""

    assert greeting("Alice") == "Hello, Alice"


def test_empty():
    """If a name is not provided, a generic greeting is provided"""

    assert greeting("") == "Hello, stranger"
```

This defines two new tests, verifying the two behaviors we expect to see: the output when a name is provided, and the output when the name is empty.

We can now re-run the test suite. This time, we don't need to provided the `-r` option, as the test requirements have already been installed; we only need to use the `--test` option:

macOS

```
(beeware-venv) $ briefcase dev --test

[helloworld] Running test suite in dev environment...
===========================================================================
============================= test session starts ==============================
...
collecting ... collected 2 items

tests/test_app.py::test_name PASSED                                      [ 50%]
tests/test_app.py::test_empty PASSED                                     [100%]

============================== 2 passed in 0.11s ===============================
```

Linux

```
(beeware-venv) $ briefcase dev --test

[helloworld] Running test suite in dev environment...
===========================================================================
============================= test session starts ==============================
...
collecting ... collected 2 items

tests/test_app.py::test_name PASSED                                      [ 50%]
tests/test_app.py::test_empty PASSED                                     [100%]

============================== 2 passed in 0.11s ===============================
```

Windows

```
(beeware-venv) C:\...>briefcase dev --test

[helloworld] Running test suite in dev environment...
===========================================================================
============================= test session starts ==============================
...
collecting ... collected 2 items

tests/test_app.py::test_name PASSED                                      [ 50%]
tests/test_app.py::test_empty PASSED                                     [100%]

============================== 2 passed in 0.11s ===============================
```

Excellent! Our `greeting()` utility method is working as expected.

2.10.2 Test driven development

Now that we have a test suite, we can use it to drive the development of new features. Let's modify our app to have a special greeting for one particular user. We can start by adding a test case for the new behavior that we'd like to see to the bottom of `test_app.py`:

```python
def test_brutus():
    """If the name is Brutus, a special greeting is provided"""

    assert greeting("Brutus") == "BeeWare the IDEs of Python!"
```

Then, run the test suite with this new test:

macOS

```
(beeware-venv) $ briefcase dev --test

[helloworld] Running test suite in dev environment...
===========================================================================
============================= test session starts ==============================
...
collecting ... collected 3 items

```

(continues on next page)

(continued from previous page)

```
tests/test_app.py::test_name PASSED                                      [ 33%]
tests/test_app.py::test_empty PASSED                                     [ 66%]
tests/test_app.py::test_brutus FAILED                                    [100%]

=================================== FAILURES ===================================
_________________________________ test_brutus __________________________________

    def test_brutus():
        """If the name is Brutus, a special greeting is provided"""

>       assert greeting("Brutus") == "BeeWare the IDEs of Python!"
E       AssertionError: assert 'Hello, Brutus' == 'BeeWare the IDEs of Python!'
E         - BeeWare the IDEs of Python!
E         + Hello, Brutus

tests/test_app.py:19: AssertionError
=========================== short test summary info ============================
FAILED tests/test_app.py::test_brutus - AssertionError: assert 'Hello, Brutus...
========================= 1 failed, 2 passed in 0.14s ==========================
```

Linux

```
(beeware-venv) $ briefcase dev --test

[helloworld] Running test suite in dev environment...
===========================================================================
============================= test session starts ==============================
...
collecting ... collected 3 items

tests/test_app.py::test_name PASSED                                      [ 33%]
tests/test_app.py::test_empty PASSED                                     [ 66%]
tests/test_app.py::test_brutus FAILED                                    [100%]

=================================== FAILURES ===================================
_________________________________ test_brutus __________________________________

    def test_brutus():
        """If the name is Brutus, provide a special greeting"""

>       assert greeting("Brutus") == "BeeWare the IDEs of Python!"
E       AssertionError: assert 'Hello, Brutus' == 'BeeWare the IDEs of Python!'
E         - BeeWare the IDEs of Python!
E         + Hello, Brutus

tests/test_app.py:19: AssertionError
=========================== short test summary info ============================
FAILED tests/test_app.py::test_brutus - AssertionError: assert 'Hello, Brutus...
========================= 1 failed, 2 passed in 0.14s ==========================

============================== 2 passed in 0.11s ===============================
```

Windows

```
(beeware-venv) C:\...>briefcase dev --test

[helloworld] Running test suite in dev environment...
===========================================================================
============================= test session starts ==============================
...
collecting ... collected 3 items

tests/test_app.py::test_name PASSED                                      [ 33%]
tests/test_app.py::test_empty PASSED                                     [ 66%]
tests/test_app.py::test_brutus FAILED                                    [100%]

=================================== FAILURES ===================================
_________________________________ test_brutus __________________________________

    def test_brutus():
        """If the name is Brutus, provide a special greeting"""

>       assert greeting("Brutus") == "BeeWare the IDEs of Python!"
E       AssertionError: assert 'Hello, Brutus' == 'BeeWare the IDEs of Python!'
E         - BeeWare the IDEs of Python!
E         + Hello, Brutus

tests/test_app.py:19: AssertionError
=========================== short test summary info ============================
FAILED tests/test_app.py::test_brutus - AssertionError: assert 'Hello, Brutus...
========================= 1 failed, 2 passed in 0.14s ==========================
```

This time, we see a test failure - and the output explains the source of the failure: the test is expecting the output “BeeWare the IDEs of Python!”, but our implementation of `greeting()` is returning “Hello, Brutus”. Let’s modify the implementation of `greeting()` in `src/helloworld/app.py` to have the new behavior:

```python
def greeting(name):
    if name:
        if name == "Brutus":
            return "BeeWare the IDEs of Python!"
        else:
            return f"Hello, {name}"
    else:
        return "Hello, stranger"
```

If we run the tests again, we’ll now see our tests pass:

macOS

```
(beeware-venv) $ briefcase dev --test

[helloworld] Running test suite in dev environment...
===========================================================================
============================= test session starts ==============================
...
collecting ... collected 3 items

tests/test_app.py::test_name PASSED                                      [ 33%]
```

(continues on next page)

(continued from previous page)

```
tests/test_app.py::test_empty PASSED                                       [ 66%]
tests/test_app.py::test_brutus PASSED                                      [100%]

============================== 3 passed in 0.15s ===============================
```

Linux

```
(beeware-venv) $ briefcase dev --test

[helloworld] Running test suite in dev environment...
===========================================================================
============================= test session starts ==============================
...
collecting ... collected 3 items

tests/test_app.py::test_name PASSED                                        [ 33%]
tests/test_app.py::test_empty PASSED                                       [ 66%]
tests/test_app.py::test_brutus PASSED                                      [100%]

============================== 3 passed in 0.15s ===============================
```

Windows

```
(beeware-venv) C:\...>briefcase dev --test

[helloworld] Running test suite in dev environment...
===========================================================================
============================= test session starts ==============================
...
collecting ... collected 3 items

tests/test_app.py::test_name PASSED                                        [ 33%]
tests/test_app.py::test_empty PASSED                                       [ 66%]
tests/test_app.py::test_brutus PASSED                                      [100%]

============================== 3 passed in 0.15s ===============================
```

2.10.3 Runtime tests

So far, we've been running the tests in development mode. This is especially useful when you're developing new features, as you can rapidly iterate on adding tests, and adding code to make those tests pass. However, at some point, you'll want to verify that your code also runs correctly when inside the bundle app environment.

The `--test` and `-r` options can also be passed to the `run` command. If you use `briefcase run --test -r`, the same test suite will run, but it will run inside the packaged application bundle rather than your development environment:

macOS

```
(beeware-venv) $ briefcase run --test -r

[helloworld] Updating application code...
Installing src/helloworld... done
```

(continues on next page)

(continued from previous page)

```
Installing tests... done

[helloworld] Updating requirements...
...
[helloworld] Built build/helloworld/macos/app/Hello World.app (test mode)

[helloworld] Starting test suite...
===========================================================================
Configuring isolated Python...
Pre-initializing Python runtime...
PythonHome: /Users/brutus/beeware-tutorial/helloworld/macOS/app/Hello World/Hello World.
↪app/Contents/Resources/support/python-stdlib
PYTHONPATH:
- /Users/brutus/beeware-tutorial/helloworld/macOS/app/Hello World/Hello World.app/
↪Contents/Resources/support/python311.zip
- /Users/brutus/beeware-tutorial/helloworld/macOS/app/Hello World/Hello World.app/
↪Contents/Resources/support/python-stdlib
- /Users/brutus/beeware-tutorial/helloworld/macOS/app/Hello World/Hello World.app/
↪Contents/Resources/support/python-stdlib/lib-dynload
- /Users/brutus/beeware-tutorial/helloworld/macOS/app/Hello World/Hello World.app/
↪Contents/Resources/app_packages
- /Users/brutus/beeware-tutorial/helloworld/macOS/app/Hello World/Hello World.app/
↪Contents/Resources/app
Configure argc/argv...
Initializing Python runtime...
Installing Python NSLog handler...
Running app module: tests.helloworld
---------------------------------------------------------------------------
============================= test session starts ==============================
...
collecting ... collected 3 items

tests/test_app.py::test_name PASSED [ 33%]
tests/test_app.py::test_empty PASSED [ 66%]
tests/test_app.py::test_brutus PASSED [100%]

============================== 3 passed in 0.21s ===============================

[helloworld] Test suite passed!
```

Linux

```
(beeware-venv) $ briefcase run --test -r

[helloworld] Finalizing application configuration...
Targeting ubuntu:jammy (Vendor base debian)
Determining glibc version... done

Targeting glibc 2.35
Targeting Python3.10

[helloworld] Updating application code...
```

(continues on next page)

(continued from previous page)

```
Installing src/helloworld... done
Installing tests... done

[helloworld] Updating requirements...
...
[helloworld] Built build/helloworld/linux/ubuntu/jammy/helloworld-0.0.1/usr/bin/
↪helloworld (test mode)

[helloworld] Starting test suite...
===========================================================================
============================= test session starts ==============================
...
collecting ... collected 3 items

tests/test_app.py::test_name PASSED [ 33%]
tests/test_app.py::test_empty PASSED [ 66%]
tests/test_app.py::test_brutus PASSED [100%]

============================== 3 passed in 0.21s ===============================
```

Windows

```
(beeware-venv) C:\...>briefcase run --test -r

[helloworld] Updating application code...
Installing src/helloworld... done
Installing tests... done

[helloworld] Updating requirements...
...
[helloworld] Built build\helloworld\windows\app\src\Hello World.exe (test mode)

===========================================================================
Log started: 2022-12-02 10:57:34Z
PreInitializing Python runtime...
PythonHome: C:\Users\brutus\beeware-tutorial\helloworld\windows\app\Hello World\src
PYTHONPATH:
- C:\Users\brutus\beeware-tutorial\helloworld\windows\app\Hello World\src\python311.zip
- C:\Users\brutus\beeware-tutorial\helloworld\windows\app\Hello World\src
- C:\Users\brutus\beeware-tutorial\helloworld\windows\app\Hello World\src\app_packages
- C:\Users\brutus\beeware-tutorial\helloworld\windows\app\Hello World\src\app
Configure argc/argv...
Initializing Python runtime...
Running app module: tests.helloworld
---------------------------------------------------------------------------
============================= test session starts ==============================
...
collecting ... collected 3 items

tests/test_app.py::test_name PASSED [ 33%]
tests/test_app.py::test_empty PASSED [ 66%]
tests/test_app.py::test_brutus PASSED [100%]
```

(continues on next page)

(continued from previous page)

```

============================== 3 passed in 0.21s ================================
```

As with `briefcase dev --test`, the `-r` option is only needed the first time you run the test suite to ensure that the test dependencies are present. On subsequent runs, you can omit this option.

You can also use the `--test` option on mobile backends: - so `briefcase run iOS --test` and `briefcase run android --test` will both work, running the test suite on the mobile device you select.

2.10.4 Next steps

This has been a taste for what you can do with the tools provided by the BeeWare project. What you do from here is up to you!

Some places to go from here:

- Tutorials demonstrating features of the Toga widget toolkit.
- Details on the options available when configuring your Briefcase project.