Briefcase Documentation

Release 0.3.18.dev271+g6d7424a0

Russell Keith-Magee

Apr 30, 2024

CONTENTS

Briefcase is a tool for converting a Python project into a standalone native application. It supports producing binaries for:

- macOS, as a standalone .app
- Windows, as an MSI installer
- Linux, as a native system package
- iOS, as an Xcode project
- Android, as a Gradle project
- the Web, as a static web site using PyScript for client-side Python

It is also extensible, allowing for additional platforms and installation formats to be produced.

TABLE OF CONTENTS

1.1 Tutorial

Get started with a hands-on introduction for beginners.

1.2 How-to guides

Guides and recipes for common problems and tasks, including how to contribute.

1.3 Background

Explanation and discussion of key topics and concepts.

1.4 Reference

Technical reference - commands, modules, classes, methods.

COMMUNITY

Briefcase is part of the BeeWare suite. You can talk to the community through:

- @beeware@fosstodon.org on Mastodon

- Discord

- The Briefcase Github Discussions forum

2.1 Tutorial

Briefcase is a packaging tool - but first you need something to package. The best way to learn about Briefcase is to see it working with the rest of the BeeWare suite of tools.

The BeeWare tutorial walks you through the process of building a native Python application from scratch.

Once you've done that tutorial, the *Briefcase How-To Guides* provide details on performing specific tasks with Briefcase.

2.2 How-to guides

How-to guides are recipes that take the user through steps in key subjects. They are more advanced than tutorials and assume a lot more about what the user already knows than tutorials do, and unlike documents in the tutorial they can stand alone.

2.2.1 Obtaining a Code Signing identity

If you are intending to distribute an application, it is advisable (and, on some platforms, necessary) to code sign your application. This is a cryptographic process that identifies you as a developer, and identifies your application as something that has been distributed by you.

The process of obtaining a code signing identity is slightly different on every platform. The following are guides for every platform that Briefcase supports:

Android

Overview

Android requires that all apps be digitally signed with a certificate before they are installed on a device or updated. Android phones enforce a policy that updates to an app must come from the same signing key to validate. This allows the phone to be sure an update is fundamentally the same app, i.e., has the same author.

This documentation covers one way to sign your app where the Google Play Store maintains the authoritative key for your app. This approach is called App Signing by Google Play.

You will need to generate a key on your development workstation to sign an app package before sending it to the Google Play store. If you use app signing by Google Play, the key on your workstation is called the upload key.

Generate a key

You will need to decide where to store the upload key. A good default is to use one keystore file per app you are creating and to store it in the `.android` folder within your home folder. The folder is automatically created by the Android tools; but if it doesn't exist, create it.

We recommend using a separate keystore file per app. Below, we use the **upload-key-helloworld.jks** filename. This assumes you are building an app called "Hello World"; use the (lowercase, no spaces) app name, `helloworld` in the filename for the keystore.

Try not to lose this key; make backups if needed. If you do lose this key, you can contact Google Play support to reset it. If you choose not to use app signing by Google Play, it is absolutely essential that you not lose this key. For this reason, we recommend using App Signing by Google Play.

macOS

Linux

Windows (cmd)

Windows (PowerShell)

```
$ mkdir -p ~/.android
$ ~/Library/Caches/org.beeware.briefcase/tools/java/Contents/Home/bin/keytool -keyalg↩
↪RSA -deststoretype pkcs12 -genkey -v -storepass android -keystore ~/.android/upload-↩
↪key-helloworld.jks -keysize 2048 -dname "cn=Upload Key" -alias upload-key -validity↩
↪10000
```

```
$ mkdir -p ~/.android
$ ~/.cache/briefcase/tools/java/bin/keytool -keyalg RSA -deststoretype pkcs12 -genkey -v↩
↪-storepass android -keystore ~/.android/upload-key-helloworld.jks -keysize 2048 -dname↩
↪"cn=Upload Key" -alias upload-key -validity 10000
```

```
C:\...>IF not exist %HOMEPATH%\.android mkdir %HOMEPATH%\.android
C:\...>%LOCALAPPDATA%\BeeWare\briefcase\Cache\tools\java\bin\keytool.exe -keyalg RSA -↩
↪deststoretype pkcs12 -genkey -v -storepass android -keystore %HOMEPATH%\.android\↩
↪upload-key-helloworld.jks -keysize 2048 -dname "cn=Upload Key" -alias upload-key -↩
↪validity 10000
```

```
PS C:\...> If (-Not (Test-Path "$env:HOMEPATH/.android")) { New-Item -Path "↩
↪$env:HOMEPATH\.android" -ItemType Directory }
```

```
PS C:\...> & "$env:LOCALAPPDATA\BeeWare\briefcase\Cache\tools\java\bin\keytool.exe" -
→keyalg RSA -deststoretype pkcs12 -genkey -v -storepass android -keystore "
→$env:HOMEPATH\.android\upload-key-helloworld.jks" -keysize 2048 -dname "cn=Upload Key"␣
→-alias upload-key -validity 10000
```

This creates a 2048-bit key and stores it in a Java keystore located in the `.android` folder within your home folder.
Since the key's purpose is to be used as an upload key for the Google Play store, we call the key "upload-key".

We use a password of `android`. This is the default password for common Android keystores. You can change the
password if you like. It is more important to limit who has access to the keystore file than to change the password.

See *Publishing your app* for instructions on using this key to upload an app to the Google Play store.

macOS

Overview

In this tutorial, we'll learn how to generate a macOS code signing identity, which is required to distribute your application across MacOS and iOS devices.

We will specifically focus on generating a Developer ID Application identity, which is used to distribute a *macOS application outside of the Mac App store*. However, the procedure for creating all other types of identities is exactly the same. Once you familiarize yourself with the general process, you'll be able to create identities required to upload applications to the Mac or iOS App stores without much trouble.

Getting the code signing identity will require five main steps, which you will be guided through in this tutorial:

1. Enrolling in the Apple Developer program
2. Generating a Certificate Signing Request on Keychain Access
3. Creating a Developer ID Application Certificate
4. Accessing the details of the Certificate on your Terminal
5. Anticipating potential issues with the identity in the future

Enrolling in the Apple Developer program

You can enroll in the Apple Developer program either as an individual, or as an organization. In both cases, you'll have to follow the instructions on the Apple Developer website.

Once you click "Start Enrollment Now" at the bottom of the page, you can either sign in with your existing Apple ID or alternatively, create a new one:

There are two types of Apple Developer account - a *personal* account, and a *business* account. If you use your personal Apple ID to create an Apple Developer account, converting it to a business account later can be painful. If you use your personal Apple ID to create a business account, extracting your personal credentials later can also be painful.

As part of the registration procedure, you'll have to pay a **$99 fee**, which will be charged on an annual basis.

Fee waivers

If you're registering as a non-profit organization, an educational institution or a government entity, you may be eligible for a fee waiver, which you can read more about here.
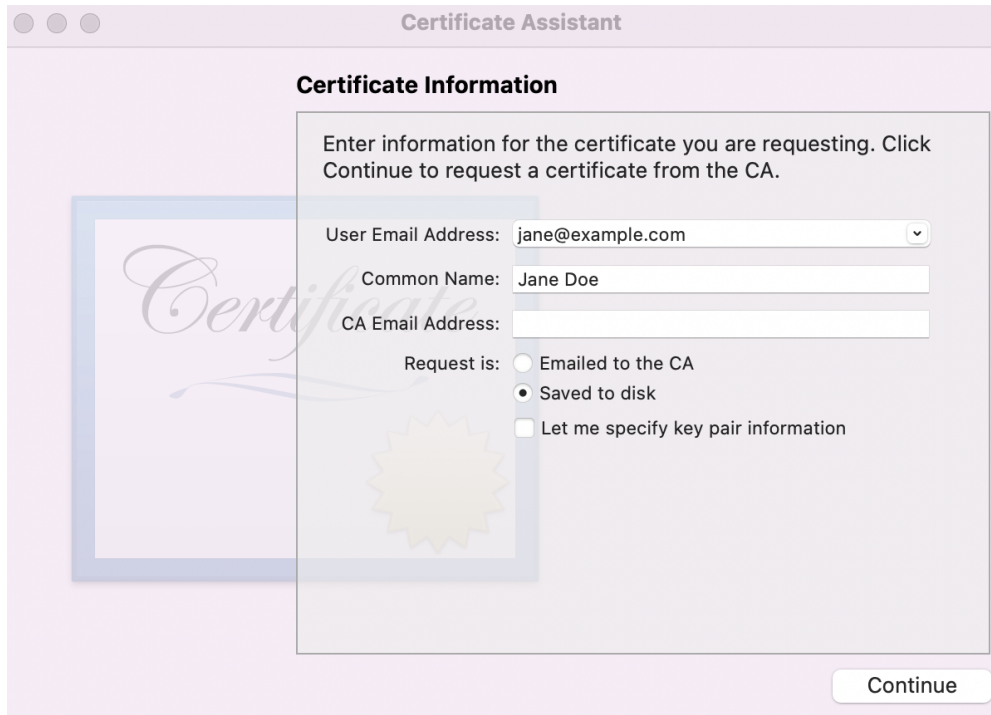
Generating a certificate request on Keychain Access

Now that you're set up with an Apple Developer ID, it's time to create a *certificate request*, which you'll then use to generate a valid Developer ID certificate.

First, open the Keychain Access application on your Mac. At the top left of your screen, click `Keychain Access >` `Certificate Assistant > Request a Certificate From a Certificate Authority`:
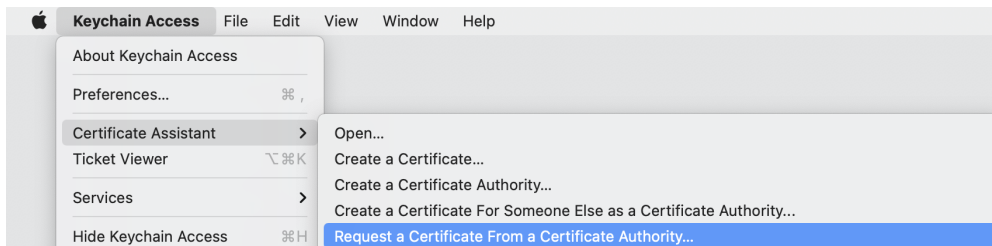
A Certificate Assistant window should open up, looking similar to this one:

- In the field `User Email Address`, type the email address associated with your Apple Developer Account (e.g. `jane@example.com`).

- `Common Name` should refer to the name with which you registered to the Apple Developer program (e.g. `Jane Doe`).

- The field `CA Email Address` can be left empty.

- Make sure that you choose `Saved to Disk` in the `Request is` field.

- Click "Continue", and save your Certificate Signing Request somewhere on your local machine. The saved certificate request should be of the format `example.certSigningRequest`.

As documented by Apple, this procedure creates not only the file you have just saved, but also a private key in your Keychain, which will establish the validity of your actual Developer ID Application certificate later on.

Creating a Developer ID Application Certificate

Once you have saved the certificate request, head to the Apple Developer website, log in, and click "Certificates, Identifiers and Profiles":
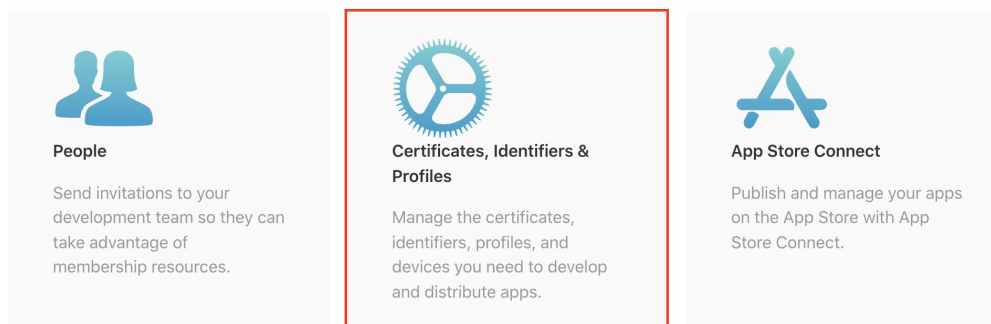
When you land in the Certificates section, click the "+" symbol to create a new certificate:

Certificates

Certificates ➕

In the next page, you'll have to choose the type of certificate you want to generate. In the Software section, choose the option of **"Developer ID Application"**. **It's very important you choose the right type of certificate**.

Later on, if you want to generate another code signing certificate for other purposes, such as uploading your application the App store, you'll simply have to choose a different type of a certificate on this page.

‹ All Certificates

Create a New Certificate   Continue

Software

○ **Apple Development**
Sign development versions of your iOS, macOS, tvOS, and watchOS apps. For use in Xcode 11 or later.

○ **Apple Distribution**
Sign your apps for submission to the App Store or for Ad Hoc distribution. For use with Xcode 11 or later.

○ **iOS App Development**
Sign development versions of your iOS app.

○ **iOS Distribution (App Store and Ad Hoc)**
Sign your iOS app for submission to the App Store or for Ad Hoc distribution.

○ **Mac Development**
Sign development versions of your Mac app.

○ **Mac App Distribution**
This certificate is used to code sign your app and configure a Distribution Provisioning Profile for submission to the Mac App Store.

○ **Mac Installer Distribution**
This certificate is used to sign your app's Installer Package for submission to the Mac App Store.

○ **Developer ID Installer**
This certificate is used to sign your app's Installer Package for distribution outside of the Mac App Store.

⊙ **Developer ID Application**
This certificate is used to code sign your app for distribution outside of the Mac App Store.

Note: If you've been registered as an organization, there's a chance that the option to choose the Developer ID Application certificate is unavailable. This may happen if you're not assigned the role of the Account Holder. You can access and change these roles using App Store Connect.

Select "Developer ID Application" and click "Continue". In the next window, click "Choose file" and upload the Certificate Signing Request you have just generated on your Keychain:

Create a New Certificate   Back   Continue

Upload a Certificate Signing Request
To manually generate a Certificate, you need a Certificate Signing Request (CSR) file from your Mac.
Learn more ›

Choose File   CertificateSigningRequest.certSigningRequest

Once you click "Continue", Apple will generate your Developer ID Application Certificate. Click the "Download" button and save the certificate on your local machine:

‹ All Certificates

Download Your Certificate   Revoke   Download

The certificate should be of the format `example.cer`. Once you download it, double-click to install it in your Keychain Access.

If you now open your Keychain, selected the `login` keychain on the left-hand side, and select the `My Certificates` tab, you should see a certificate with the title "Developer ID Application: <your name>".

Click on the certificate and make sure you see a note that reads `This certificate is valid`.
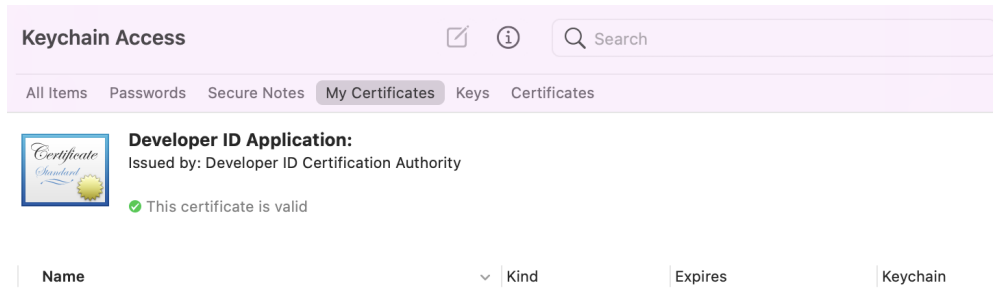
Note: In this screenshot, the certificate details have been redacted. Your certificate should show expiration details, trust chains, and other details about you, the certificate issuer (Apple), and the certificate.

Congratulations! You've just successfully installed the Developer ID Application certificate.

Keep this certificate safe!

The *specific type* of the certificate you have just created is quite precious, and you should make sure to keep it safe. A single Developer ID Application Certificate can be used to sign, notarize and distribute multiple applications outside of the Mac App store, which is why a very limited number of them can be created on a particular Developer Account. You should consider making a backup copy, which will require you to export the certificate together with the associated private key from the Keychain. The procedure for doing so is documented by Apple.

Next steps

Now you can use the certificate to sign and notarize your application with the *briefcase package* command.

When you invoke *briefcase package*, you will be prompted to select the code signing certificate you want to use from the certificates that are installed. Once you select a certificate, Briefcase will output the command line invocation to select that certificate for unattended installation.

Windows

Overview

While Windows does not require applications to be code signed, doing so can help ensure the authenticity and integrity of your application when a user is trying to install it. This includes preventing Windows from showing warnings to users that your application is untrusted and may be dangerous.

To code sign your application with Briefcase, you must obtain a code signing certificate from a Certificate Authority (CA) and install it in a Windows certificate store.

Obtain a Certificate

Microsoft manages a collection of Certificate Authorities that are trusted by default in Windows. A code signing certificate from any of these included entities should be trusted by a modern Windows machine.

Of note, a certificate for code signing is different from a certificate for other cryptographic purposes. For instance, a certificate used by a website to create encrypted HTTP connections cannot be used for code signing. Therefore, a request for a code signing certificate must specifically be for code signing. Certificate Authorities offering these services will make this clear during the request process.

Additionally, it is possible to create a code signing certificate directly within Windows. However, when you create a certificate yourself, it will likely be considered "self-signed". Using such a certificate to code sign an application will not imbue it with the trust that a certificate from a Certificate Authority provides. Therefore, a self-signed certificate should not be used to code sign applications for distribution.

Install the Certificate

Once a code signing certificate is requested, Certificate Authorities may vary is how the certificate is actually delivered to you. In general, though, you'll likely receive an encrypted file containing both the certificate and its private key. Depending on the exact nature of the file format, Windows provides several commands to import certificates in to one of its certificate stores.

For instance, this will a import PFX file in to the `Personal` certificate store of `Current User`:

CMD

PowerShell

```
C:\...>certutil.exe -user -importpfx -p MySecretPassword My .\cert.pfx
```

```
PS C:\...> Import-PfxCertificate -FilePath .\cert.pfx -CertStoreLocation Cert:\
→CurrentUser\My -Password MySecretPassword
```

Refer to your Certificate Authority's documentation for specific instructions.

Certificate's SHA-1 Thumbprint

On Windows, *briefcase package* cannot retrieve the list of installed code signing certificates automatically. You need to retrieve the identity manually, and provide the certificates identity as a command line argument.

The certificates installed on the machine are available in the Certificate Manager. Search for "User Certificates" in the Start Menu to access certificates for `Current User` or search for "Computer Certificates" for certificates for `Local Machine`. Alternatively, the command `certmgr.msc` will open the manager for `Current User` and `certlm.msc` for `Local Machine`.

Once you locate your certificate in the certificate store it was installed in to, double-click it to access information about it. Near the bottom of the list on the Details tab, you'll find the `Thumbprint` field with the 40 character SHA-1 hash to use as the `identity` in the Briefcase package command.

```
(venv) C:\...>briefcase package --identity <certificate thumbprint>
```

2.2.2 Upgrading from Briefcase v0.2

Briefcase v0.2 was built as a setuptools extension. The configuration for your project was contained in a `setup.py` or `setup.cfg` file, and you invoked Briefcase using `python setup.py <platform>`.

Briefcase v0.3 represents a significant change in the development of Briefcase. Briefcase is now a PEP518-compliant build tool. It uses `pyproject.toml` for configuration, and is invoked using a standalone `briefcase` command. This change gives significantly improved flexibility in configuring Briefcase apps, and much better control over the development process.

However, this change is also **backwards incompatible**. If you have a project that was using Briefcase v0.2, you'll need to make some major changes to your configuration and processes as part of upgrading to v0.3.

Configuration

To port your application's configuration to Briefcase v0.3, you'll need to add a `pyproject.toml` file (in, as the extension suggests, TOML format). This file contains similar content to your `setup.py` or `setup.cfg` file.

The following is a minimal starting point for your `pyproject.toml` file:

```
[tool.briefcase]
project_name = "My Project"
bundle = "com.example"
version = "0.1"
author = "Jane Developer"
author_email = "jane@example.com"
requires = []

[tool.briefcase.app.myapp]
formal_name = "My App"
description = "My first Briefcase App"
requires = []
sources = ['src/myapp']

[tool.briefcase.app.myapp.macOS]
requires = ['toga-cocoa==0.3.0.dev15']

[tool.briefcase.app.myapp.windows]
requires = ['toga-winforms==0.3.0.dev15']

[tool.briefcase.app.myapp.linux]
requires = ['toga-gtk==0.3.0.dev15']

[tool.briefcase.app.myapp.iOS]
requires = ['toga-iOS==0.3.0.dev15']
```

The configuration sections are tool specific, and start with the prefix `tool.briefcase`. Additional dotted paths define the specificity of the settings that follow.

Most of the keys in your `setup.py` will map directly to the same key in your `pyproject.toml` (e.g., `version`, `description`). However, the following pointers may help port other values.

- Briefcase v0.2 assumed that a `setup.py` file described a single app. Briefcase v0.3 allows a project to define multiple distributable applications. The `project_name` is the name for the collection of apps described by this `pyproject.toml`; `formal_name` is the name for a single app. If your project defines a single app, your formal name and project name will probably be the same.

- There is no explicit definition for the app's `name` - the app name is derived from the section header name (i.e., `[tool.briefcase.app.myapp]` defines the existence of an app named `myapp`).

- `version` *must* be defined as a string in your `pyproject.toml` file. If you need to know the version of your app (or the value of any other app metadata specified in `pyproject.toml`) at runtime, you should use importlib.metadata. Briefcase will create `myapp.dist-info` for your application (using your app name instead of `myapp`).

- Briefcase v0.3 configuration files are hierarchical. `[tool.briefcase]` describes configuration arguments for the entire project; `[tool.briefcase.app.myapp]` describes configuration arguments for the application named `myapp`; `[tool.briefcase.app.myapp.macOS]` describes configuration arguments for macOS deployments of `myapp`, and `[tool.briefcase.app.myapp.macOS.dmg]` describes configuration arguments for DMG deployments of `myapp` on macOS. The example above doesn't contain a `dmg` section; generally, you won't need one unless you're packaging for multiple output formats on a single platform.

  For most keys, the "most specific" value wins - so, a value for `description` defined at the platform level will override any value at the app level, and so on. The two exceptions are `requires` and `sources`, which are cumulative - the values defined at the platform level will be *appended* to the values at the app level and the project level.

- The `install_requires` and `app_requires` keys in `setup.py` are replaced by `requires` in your `pyproject.toml`. `requires` can be specified at the project level, the app level, the platform level, or the output format level.

- The `packages` (and other various source code and data-defining attributes) in `setup.py` have been replaced with a single `sources` key. The paths specified in sources will be copied in their entirety into the packaged application.

Once you've created and tested your `pyproject.toml`, you can delete your `setup.py` file. You may also be able to delete your `setup.cfg` file, depending on whether it defines any tool configurations (e.g., `flake8` or `pytest` configurations).

Invocation

In Briefcase v0.2, there was only one entry point: `python setup.py <platform>`. This would generate a complete output artefact; and, if you provided the `-s` argument, would also start the app.

Briefcase v0.3 uses its own `briefcase` entry point, with *subcommands* to perform specific functions:

- `briefcase new` - Bootstrap a new project (generating a `pyproject.toml` and other stub content).

- `briefcase dev` - Run the app in developer mode, using the current virtual environment.

- `briefcase create` - Use the platform template to generate the files needed to build a distributable artefact for the platform.

- `briefcase update` - Update the source code of the application in the generated project.

- `briefcase build` - Run whatever compilation process is necessary to produce an executable file for the platform.

- `briefcase run` - Run the executable file for the platform.

- `briefcase package` - Perform whatever post-processing is necessary to wrap the executable into a distributable artefact (e.g., an installer).

When using these commands, there is no need to specify the platform (i.e. `macOS` when on a Mac). The current platform will be detected and the appropriate output format will be selected.

If you want to target a different platform, you can specify that platform as an argument. This will be required when building for mobile platforms (since you'll never be running Briefcase where the mobile platform is "native"). For

example, if you're on a Mac, `briefcase create macOS` and `briefcase create` would perform the same task; `briefcase create iOS` would build an iOS project.

The exceptions to this platform specification are `briefcase new` and `briefcase dev`. These two commands are platform agnostic.

The Briefcase sub-commands will also detect if previous steps haven't been executed, and invoke any prior steps that are required. For example, if you execute `briefcase run` on clean project, Briefcase will detect that there are no platform files, and will automatically run `briefcase create` and `briefcase build`. This won't occur on subsequent runs.

Briefcase v0.3 also allows for multiple output formats on a single platform. The only platform that currently exposes capability is macOS, which supports both `app` and `dmg` output formats (with `dmg` being the platform default).

To use a different output format, add the format as an extra argument to each command after the platform. For example, to create a `app` file for macOS, you would run:

```
$ briefcase create macOS app
$ briefcase build macOS app
$ briefcase run macOS app
$ briefcase package macOS app
```

In the future, we hope to add other output formats for other platforms - Snap and FlatPak on Linux; NSIS installers on Windows, and possibly others. If you're interested in adding support for one of these platforms, please get in touch (or, submit a pull request!)

2.2.3 Accessing Briefcase packaging metadata at runtime

When Briefcase installs your app, it adds a PEP566 metadata file containing information about your app, and Briefcase itself. You can retrieve this information at runtime using the Python builtin library 'importlib.metadata'. `importlib.metadata` was added in Python 3.8; however, it has been backported and published on PyPI as 'importlib_metadata' for older versions of Python.

To access application metadata at runtime, you can use the following code:

```python
import sys
try:
    from importlib import metadata as importlib_metadata
except ImportError:
    # Backwards compatibility - importlib.metadata was added in Python 3.8
    import importlib_metadata

# Find the name of the module that was used to start the app
app_module = sys.modules['__main__'].__package__
# Retrieve the app's metadata
metadata = importlib_metadata.metadata(app_module)
```

The `metadata` returned by this code will be a dictionary-like object that contains the following identifying keys:

- **Metadata-Version** - The syntax version of the metadata file itself (as defined in PEP566).

- **Briefcase-Version** - The version of Briefcase used to package the app. The existence of this key in app metadata can be used to identify if your application code is running in a Briefcase container; it will only exist if the app has been packaged by Briefcase.

It will also have the following keys, derived from your application's `pyproject.toml` configuration:

- **Name** - app_name

- **Formal-Name** - `formal_name`

- **App-ID** - `bundle` and `app_name`, joined with a `.`

- **Version** - `version`

- **Summary** - `description`

The metadata may also contain the following keys, if they have been defined in your app's `pyproject.toml` configuration:

- **Home-page** - `url`

- **Author** - `author`

- **Author-email** - `author_email`

For example, the metadata for the app constructed by the BeeWare Tutorial would contain:

```
Metadata-Version: 2.1
Briefcase-Version: 0.3.1
Name: helloworld
Formal-Name: Hello World
App-ID: com.example.helloworld
Version: 0.0.1
Home-page: https://example.com/helloworld
Author: Jane Developer
Author-email: jane@example.com
Summary: My first application
```

2.2.4 Contributing code to Briefcase

If you experience problems with Briefcase, log them on GitHub. If you want to contribute code, please fork the code and submit a pull request.

tl;dr

Set up the dev environment by running:

macOS

Linux

Windows

```
$ git clone https://github.com/beeware/briefcase.git
$ cd briefcase
$ python3 -m venv venv
$ . venv/bin/activate
(venv) $ python -m pip install -Ue ".[dev]"
(venv) $ pre-commit install
```

```
$ git clone https://github.com/beeware/briefcase.git
$ cd briefcase
$ python3 -m venv venv
$ . venv/bin/activate
```

(continues on next page)

```
(venv) $ python -m pip install -Ue ".[dev]"
(venv) $ pre-commit install
```

```
C:\...>git clone https://github.com/beeware/briefcase.git
C:\...>cd briefcase
C:\...>py -m venv venv
C:\...>venv\Scripts\activate
(venv) C:\...>python -m pip install -Ue .[dev]
(venv) C:\...>pre-commit install
```

Invoke CI checks and tests by running:

macOS

Linux

Windows

```
(venv) $ tox p -m ci
```

```
(venv) $ tox p -m ci
```

```
(venv) C:\...>tox p -m ci
```

Setting up your development environment

The recommended way of setting up your development environment for Briefcase is to use a virtual environment, and then install the development version of Briefcase and its dependencies:

Clone Briefcase and create virtual environment

macOS

Linux

Windows

```
$ git clone https://github.com/beeware/briefcase.git
$ cd briefcase
$ python3 -m venv venv
$ . venv/bin/activate
(venv) $ python -m pip install -Ue ".[dev]"
```

```
$ git clone https://github.com/beeware/briefcase.git
$ cd briefcase
$ python3 -m venv venv
$ . venv/bin/activate
(venv) $ python -m pip install -Ue ".[dev]"
```

```
C:\...>git clone https://github.com/beeware/briefcase.git
C:\...>cd briefcase
```

```
C:\...>py -m venv venv
C:\...>venv\Scripts\activate
(venv) C:\...>python -m pip install -Ue .[dev]
```

Install pre-commit

Briefcase uses a tool called pre-commit to identify simple issues and standardize code formatting. It does this by installing a git hook that automatically runs a series of code linters prior to finalizing any git commit. To enable pre-commit, run:

macOS

Linux

Windows

```
(venv) $ pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

```
(venv) $ pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

```
(venv) C:\...>pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

Pre-commit automatically runs during the commit

With pre-commit installed as a git hook for verifying commits, the pre-commit hooks configured in `.pre-commit-config.yaml` for Briefcase must all pass before the commit is successful. If there are any issues found with the commit, this will cause your commit to fail. Where possible, pre-commit will make the changes needed to correct the problems it has found:

macOS

Linux

Windows

```
(venv) $ git add some/interesting_file.py
(venv) $ git commit -m "Minor change"
check toml...........................................(no files to check)Skipped
check yaml...........................................(no files to check)Skipped
check for case conflicts.................................................Passed
check docstring is first.................................................Passed
fix end of files.........................................................Passed
trim trailing whitespace.................................................Passed
isort....................................................................Passed
pyupgrade................................................................Passed
docformatter.............................................................Passed
black....................................................................Failed
- hook id: black
- files were modified by this hook
```

```
reformatted some/interesting_file.py

All done!
1 file reformatted.

flake8...................................................................Passed
```

```
(venv) $ git add some/interesting_file.py
(venv) $ git commit -m "Minor change"
check toml...........................................(no files to check)Skipped
check yaml...........................................(no files to check)Skipped
check for case conflicts................................................Passed
check docstring is first................................................Passed
fix end of files........................................................Passed
trim trailing whitespace................................................Passed
isort...................................................................Passed
pyupgrade...............................................................Passed
docformatter............................................................Passed
black...................................................................Failed
- hook id: black
- files were modified by this hook

reformatted some/interesting_file.py

All done!
1 file reformatted.

flake8..................................................................Passed
```

```
(venv) C:\...>git add some/interesting_file.py
(venv) C:\...>git commit -m "Minor change"
check toml...........................................(no files to check)Skipped
check yaml...........................................(no files to check)Skipped
check for case conflicts................................................Passed
check docstring is first................................................Passed
fix end of files........................................................Passed
trim trailing whitespace................................................Passed
isort...................................................................Passed
pyupgrade...............................................................Passed
docformatter............................................................Passed
black...................................................................Failed
- hook id: black
- files were modified by this hook

reformatted some/interesting_file.py

All done!
1 file reformatted.

flake8..................................................................Passed
```

You can then re-add any files that were modified as a result of the pre-commit checks, and re-commit the change.

macOS

Linux

Windows

```
(venv) $ git add some/interesting_file.py
(venv) $ git commit -m "Minor change"
check toml...........................................(no files to check)Skipped
check yaml...........................................(no files to check)Skipped
check for case conflicts.................................................Passed
check docstring is first.................................................Passed
fix end of files.........................................................Passed
trim trailing whitespace.................................................Passed
isort....................................................................Passed
pyupgrade................................................................Passed
docformatter.............................................................Passed
black....................................................................Passed
flake8...................................................................Passed
[bugfix daedd37a] Minor change
 1 file changed, 2 insertions(+)
 create mode 100644 some/interesting_file.py
```

```
(venv) $ git add some/interesting_file.py
(venv) $ git commit -m "Minor change"
check toml...........................................(no files to check)Skipped
check yaml...........................................(no files to check)Skipped
check for case conflicts.................................................Passed
check docstring is first.................................................Passed
fix end of files.........................................................Passed
trim trailing whitespace.................................................Passed
isort....................................................................Passed
pyupgrade................................................................Passed
docformatter.............................................................Passed
black....................................................................Passed
flake8...................................................................Passed
[bugfix daedd37a] Minor change
 1 file changed, 2 insertions(+)
 create mode 100644 some/interesting_file.py
```

```
(venv) C:\...>git add some\interesting_file.py
(venv) C:\...>git commit -m "Minor change"
check toml...........................................(no files to check)Skipped
check yaml...........................................(no files to check)Skipped
check for case conflicts.................................................Passed
check docstring is first.................................................Passed
fix end of files.........................................................Passed
trim trailing whitespace.................................................Passed
isort....................................................................Passed
pyupgrade................................................................Passed
docformatter.............................................................Passed
black....................................................................Passed
```

```
flake8...........................................................Passed
[bugfix daedd37a] Minor change
 1 file changed, 2 insertions(+)
 create mode 100644 some/interesting_file.py
```

Create a new branch in git

When you clone Briefcase, it will default to checking out the default branch, `main`. However, your changes should be committed to a new branch instead of being committed directly in to `main`. The branch name should be succinct but relate to what's being changed; for instance, if you're fixing a bug in Windows code signing, you might use the branch name `fix-windows-signing`. To create a new branch, run:

macOS

Linux

Windows

```
(venv) $ git checkout -b fix-windows-signing
```

```
(venv) $ git checkout -b fix-windows-signing
```

```
(venv) C:\...>git checkout -b fix-windows-signing
```

Running tests and coverage

Briefcase uses [tox](#) to manage the testing process and [pytest](#) for its own test suite.

The default `tox` command includes running:

- pre-commit hooks
- towncrier release note check
- documentation linting
- test suite for available Python versions
- code coverage reporting

To run the full test suite, run:

macOS

Linux

Windows

```
(venv) $ tox
```

```
(venv) $ tox
```

```
(venv) C:\...>tox
```

The full test suite can take a while to run. You can speed it up considerably by running tox in parallel, by running `tox p` (or `tox run-parallel`). When you run the test suite in parallel, you'll get less feedback on the progress of the test suite as it runs, but you'll still get a summary of any problems found at the end of the test run.

Run tests for multiple versions of Python

By default, many of the `tox` commands will attempt to run the test suite multiple times, once for each Python version supported by Briefcase. To do this, though, each of the Python versions must be installed on your machine and available to tox's Python discovery process. In general, if a version of Python is available via `PATH`, then tox should be able to find and use it.

Run only the test suite

If you're rapidly iterating on a new feature, you don't need to run the full test suite; you can run *just* the unit tests. To do this, run:

macOS

Linux

Windows

```
(venv) $ tox -e py
```

```
(venv) $ tox -e py
```

```
(venv) C:\...>tox -e py
```

Run a subset of tests

By default, tox will run all tests in the unit test suite. To restrict the test run to a subset of tests, you can pass in any pytest specifier as an argument to tox. For example, to run only the tests in a single file, run:

macOS

Linux

Windows

```
(venv) $ tox -e py -- tests/path/to/test_some_test.py
```

```
(venv) $ tox -e py -- tests/path/to/test_some_test.py
```

```
(venv) C:\...>tox -e py -- tests/path/to/test_some_test.py
```

Run the test suite for a specific Python version

By default `tox -e py` will run using whatever interpreter resolves as `python3` on your machine. If you have multiple Python versions installed, and want to test a specific Python version, you can specify a specific python version to use. For example, to run the test suite on Python 3.10, run:

macOS

Linux

Windows

```
(venv) $ tox -e py310
```

```
(venv) $ tox -e py310
```

```
(venv) C:\...>tox -e py310
```

A *subset of tests* can be run by adding `--` and a test specification to the command line.

Run the test suite without coverage (fast)

By default, tox will run the pytest suite in single threaded mode. You can speed up the execution of the test suite by running the test suite in parallel. This mode does not produce coverage files due to complexities in capturing coverage within spawned processes. To run a single python version in "fast" mode, run:

macOS

Linux

Windows

```
(venv) $ tox -e py-fast
```

```
(venv) $ tox -e py-fast
```

```
(venv) C:\...>tox -e py-fast
```

A *subset of tests* can be run by adding `--` and a test specification to the command line; a *specific Python version* can be used by adding the version to the test target (e.g., `py310-fast` to run fast on Python 3.10).

Code coverage

Briefcase maintains 100% branch coverage in its codebase. When you add or modify code in the project, you must add test code to ensure coverage of any changes you make.

However, Briefcase targets macOS, Linux, and Windows, as well as multiple versions of Python, so full coverage cannot be verified on a single platform and Python version. To accommodate this, several conditional coverage rules are defined in the `tool.coverage.coverage_conditional_plugin.rules` section of `pyproject.toml` (e.g., `no-cover-if-is-windows` can be used to flag a block of code that won't be executed when running the test suite on Windows). These rules are used to identify sections of code that are only covered on particular platforms or Python versions.

Of note, coverage reporting across Python versions can be a bit quirky. For instance, if coverage files are produced using one version of Python but coverage reporting is done on another, the report may include false positives for

missed branches. Because of this, coverage reporting should always use the oldest version Python used to produce the coverage files.

Coverage report for host platform and Python version

You can generate a coverage report for your platform and version of Python. For example, to run the test suite and generate a coverage report on Python3.11, run:

macOS

Linux

Windows

```
(venv) $ tox -m test311
```

```
(venv) $ tox -m test311
```

```
(venv) C:\...>tox -m test311
```

Coverage report for host platform

If all supported versions of Python are available to tox, then coverage for the host platform can be reported by running:

macOS

Linux

Windows

```
(venv) $ tox p -m test-platform
```

```
(venv) $ tox p -m test-platform
```

```
(venv) C:\...>tox p -m test-platform
```

Coverage reporting in HTML

A HTML coverage report can be generated by appending `-html` to any of the coverage tox environment names, for instance:

macOS

Linux

Windows

```
(venv) $ tox -e coverage-platform-html
```

```
(venv) $ tox -e coverage-platform-html
```

```
(venv) C:\...>tox -e coverage-platform-html
```

Add change information for release notes

Briefcase uses towncrier to automate building release notes. To support this, every pull request needs to have a corresponding file in the changes/ directory that provides a short description of the change implemented by the pull request.

This description should be a high level summary of the change from the perspective of the user, not a deep technical description or implementation detail. It should also be written in past tense (i.e., "Added an option to enable X" or "Fixed handling of Y").

See News Fragments for more details on the types of news fragments you can add. You can also see existing examples of news fragments in the changes/ folder.

Simulating GitHub CI checks locally

To run the same checks that run in CI for the platform, run:

macOS

Linux

Windows

```
(venv) $ tox p -m ci
```

```
(venv) $ tox p -m ci
```

```
(venv) C:\...>tox p -m ci
```

Now you are ready to start hacking! Have fun!

2.2.5 Contributing to the documentation

Here are some tips for working on this documentation. You're welcome to add more and help us out!

First of all, you should check the reStructuredText (reST) Primer to learn how to write your .rst file.

Create a .rst file

Look at the structure and choose the best category to put your .rst file. Make sure that it is referenced in the index of the corresponding category, so it will show on in the documentation. If you have no idea how to do this, study the other index files for clues.

Build documentation locally

To build the documentation locally, *set up a development environment*.

You'll also need to install the Enchant spell checking library.

macOS

Linux

Windows

Enchant can be installed using Homebrew:

```
(venv) $ brew install enchant
```

If you're on an M1 machine, you'll also need to manually set the location of the Enchant library:

```
(venv) $ export PYENCHANT_LIBRARY_PATH=/opt/homebrew/lib/libenchant-2.2.dylib
```

Enchant can be installed as a system package:

Ubuntu 20.04+ / Debian 10+

```
$ sudo apt-get update
$ sudo apt-get install enchant-2
```

Fedora

```
$ sudo dnf install enchant
```

Arch, Manjaro

```
$ sudo pacman -Syu enchant
```

Enchant is installed automatically when you set up your development environment.

Once your development environment is set up, run:

macOS

Linux

Windows

```
(venv) $ tox -e docs
```

```
(venv) $ tox -e docs
```

```
(venv) C:\...>tox -e docs
```

The output of the file should be in the `docs/_build/html` folder. If there are any markup problems, they'll raise an error.

Live documentation preview

To support rapid editing of documentation, Briefcase also has a "live preview" mode:

macOS

Linux

Windows

```
(venv) $ tox -e docs-live
```

```
(venv) $ tox -e docs-live
```

```
(venv) C:\...>tox -e docs-live
```

This will build the documentation, start a web server to serve the build documentation, and watch the file system for any changes to the documentation source. If a change is detected, the documentation will be rebuilt, and any browser viewing the modified page will be automatically refreshed.

Documentation linting

Before committing and pushing documentation updates, run linting for the documentation:

macOS

Linux

Windows

```
(venv) $ tox -e docs-lint
```

```
(venv) $ tox -e docs-lint
```

```
(venv) C:\...>tox -e docs-lint
```

This will validate the documentation does not contain:

- invalid syntax and markup
- dead hyperlinks
- misspelled words

If a valid spelling of a word is identified as misspelled, then add the word to the list in docs/spelling_wordlist. This will add the word to the spellchecker's dictionary.

Rebuilding all documentation

To force a rebuild for all of the documentation:

macOS

Linux

Windows

```
(venv) $ tox -e docs-all
```

```
(venv) $ tox -e docs-all
```

```
(venv) C:\...>tox -e docs-all
```

The documentation should be fully rebuilt in the docs/_build/html folder. If there are any markup problems, they'll raise an error.

2.2.6 Internal How-to guides

These guides are for the maintainers of the Briefcase project, documenting internal project procedures.

How to cut a Briefcase release

The release infrastructure for Briefcase is semi-automated, using GitHub Actions to formally publish releases.

This guide assumes that you have an `upstream` remote configured on your local clone of the Briefcase repository, pointing at the official repository. If all you have is a checkout of a personal fork of the Briefcase repository, you can configure that checkout by running:

```
$ git remote add upstream https://github.com/beeware/briefcase.git
```

The procedure for cutting a new release is as follows:

1. Check the contents of the upstream repository's main branch:

   ```
   $ git fetch upstream
   $ git checkout --detach upstream/main
   ```

   Check that the HEAD of release now matches upstream/main.

2. Ensure that the release notes are up to date. Run:

   ```
   $ tox -e towncrier -- --draft
   ```

   to review the release notes that will be included, and then:

   ```
   $ tox -e towncrier
   ```

   to generate the updated release notes.

3. Build the documentation to ensure that the new release notes don't include any spelling errors or markup problems:

   ```
   $ tox -e docs-lint,docs
   ```

4. Ensure that there is a version branch for the new Briefcase version in every template that Briefcase will use at runtime:

   - `briefcase-template`
   - `briefcase-macOS-app-template`
   - `briefcase-macOS-Xcode-template`
   - `briefcase-windows-app-template`
   - `briefcase-windows-VisualStudio-template`
   - `briefcase-linux-appimage-template`
   - `briefcase-linux-flatpak-template`
   - `briefcase-linux-system-template`
   - `briefcase-iOS-Xcode-template`
   - `briefcase-android-gradle-template`
   - `briefcase-web-static-template`

5. Tag the release, and push the branch and tag upstream:

```
$ git tag v1.2.3
$ git push upstream HEAD:main
$ git push upstream v1.2.3
```

6. Pushing the tag will start a workflow to create a draft release on GitHub. You can follow the progress of the workflow on GitHub; once the workflow completes, there should be a new draft release, and an entry on the Test PyPI server.

   Confirm that this action successfully completes. If it fails, there's a couple of possible causes:

   a. The final upload to Test PyPI failed. Test PyPI is not have the same service monitoring as PyPI-proper, so it sometimes has problems. However, it's also not critical to the release process; if this step fails, you can perform Step 6 by manually downloading the "packages" artifact from the GitHub workflow instead.

   b. The test apps fail to build. This is likely because you forgot to branch one (or more) of the templates mentioned in Step 3. If this happens, you can correct the missing template, and re-run the action through the GitHub Actions GUI.

   c. Something else fails in the build process. If the problem can be fixed without a code change to the Briefcase repository (e.g., a transient problem with build machines not being available), you can re-run the action that failed through the GitHub Actions GUI. If the fix requires a code change, delete the old tag, make the code change, and re-tag the release.

7. Create a clean virtual environment, install the new release from Test PyPI, and perform any pre-release testing that may be appropriate:

```
$ python3 -m venv testvenv
$ . ./testvenv/bin/activate
(testvenv) $ pip install --extra-index-url https://test.pypi.org/simple/␣
↪briefcase==1.2.3
(testvenv) $ briefcase --version
briefcase 1.2.3
(testvenv) $ #... any other manual checks you want to perform ...
```

8. Log into ReadTheDocs, visit the Versions tab, and activate the new version. Ensure that the build completes; if there's a problem, you may need to correct the build configuration, roll back and re-tag the release.

9. Edit the GitHub release to add release notes. You can use the text generated by towncrier, but you'll need to update the format to Markdown, rather than ReST. If necessary, check the pre-release checkbox.

10. Double check everything, then click Publish. This will trigger a publication workflow on GitHub.

11. Wait for the package to appear on PyPI.

Congratulations, you've just published a release!

If anything went wrong during steps 4-10, you will need to re-start from step 4 with a new version number. Once the release has successfully appeared on PyPI (or Test PyPI), it cannot be changed; if you spot a problem in a published package, you'll need to tag a completely new release.

Docker X11 Passthrough

Briefcase can use Docker to build apps for Linux distributions other than the distribution you're currently using. Docker can also be used to *run* the app on the foreign distribution, exporting the graphical aspects of the app to your local display. This document describes how to configure your system to do this.

X Window System Background

Linux distributions use either the X Window System (sometimes called X or X11) or Wayland to manage their graphical displays. X11 is the older of the two; Wayland maintains compatibility with the X11 protocol for backwards compatibility.

The X11 protocol operates in a client/server framework; any application that wishes to display a window or receive user input will send and receive commands with an X server.

X Configuration

The location of the X server is declared by the `DISPLAY` environment variable, normally in the form `HOSTNAME:DISPLAYNUMBER.SCREENNUMBER`. If `HOSTNAME` is absent, it is assumed to be the machine the client is running on. The `SCREENNUMBER` is largely historical since all monitors are normally collapsed together in to a single screen now. Therefore, an expected setting for `DISPLAY` is `:0` for many installations.

While security is relatively weak for X11, there are basic facilities to mitigate unauthenticated access. The `XAUTHORITY` environment variable can specify a file path to an `xauth` database; if `XAUTHORITY` is not set, there will normally be a system default file path configured for each user. The `xauth` database itself is protected by file system access controls and will contain "cookies" for individual displays that are assigned by the X server for clients to use to facilitate authentication.

An `xauth` database authorizing a host named `jupiter` might look like:

```
$ xauth list
jupiter/unix:  MIT-MAGIC-COOKIE-1  9e9a67185b1fdc0c46e00dc400559873
#ffff#6a757069746572#:  MIT-MAGIC-COOKIE-1  9e9a67185b1fdc0c46e00dc400559873
```

Along with cookie-based authentication, it is also possible to add entities to an allowlist for a display. For instance, some distributions are configured to allow any process owned by the logged in user access to the display. This is configured by `xhost`.

An `xhost` configuration authorizing a user named `brutus` might look like:

```
$ xhost
access control enabled, only authorized clients can connect
SI:localuser:brutus
```

X Operation

While authentication is normally enabled for X access, security is mostly bolstered by the allowed methods for clients to connect. Most systems will only open a UNIX socket to which clients should connect to send and receive messages. By virtue of file system design, only users on the host machine will have access to this socket. With the introduction of abstract sockets in Linux, such a socket is also typically made available in tandem with the UNIX socket. The advantages of an abstract socket is beyond this discussion, though.

Such a UNIX socket connection will be configured for each display for the machine. As with many things in Linux, UNIX sockets are exposed in the machine via a file in the `/tmp/.X11-unix/` directory. The socket files are named for the number of the DISPLAY they are connected to; so, the file for Display 0 is `X0`. Therefore, any display can be found at `/tmp/.X11-unix/X#`.

Along with a UNIX socket connection, X servers can also listen on a TCP socket on the machine's network interfaces. However, since a network connection can easily be reached by other machines, listening on a TCP socket is normally disabled on most Linux distributions. That said, the X11 standard reserves port numbers starting at port 6000 for X displays. Therefore, Display 0 is available at port 6000 while Display 99 would be available at port 6099 and so on.

Docker

From the design of X11, it is clear that a Docker container needs access to the socket for the display and the `xauth` database for the user.

An example of a well-published method to accomplish this:

```
$ docker run --rm -it --net host --env DISPLAY ubuntu xeyes
```

This method assumes:

- the user in the container is `root` or its ID matches the host user
- `xhost` is configured to allow any connections from the user
- the X server is running an abstract socket for the display

By virtue of how access control is managed for abstract sockets, the `--net host` configuration allows access to the abstract socket for the display inside the container and passing the `DISPLAY` environment variable through lets the `xeyes` application know which display to connect to.

While this strategy works in many environments, the generalized solution is more complicated to accommodate variations in Docker implementations as well as whether the current display is actually being proxied.

Docker Desktop

The original implementation of Docker is referred to as Docker Engine and leverages many advanced features of Linux to run processes in highly containerized environments. Docker Desktop, however, effectively runs Docker Engine inside a lightweight Linux virtual machine (VM) running on the host machine. Therefore, when Docker Desktop runs a container, it is running it inside of a VM and not directly on the host system as Docker Engine does (albeit in isolation via containerization).

As an outcome of the design of Docker Desktop, the behavior of interactions between the host machine and Docker containers can be significantly different. For instance, it is not possible to expose the host machine's network to a container via `--net host` like you can with Docker Engine. While this does change the exact network configuration that's exposed to the container in the Docker Desktop VM, it is much different than Docker Engine and abstract sockets on the host are not available to the container.

Along with not being possible to expose abstract sockets on the host to a container running via Docker Desktop, it is also not possible to expose arbitrary UNIX sockets either. Therefore, attempting to bind mount `/tmp/.X11-unix/X0`, for instance, in to a Docker Desktop container will not allow processes inside the container to successfully communicate with the socket. (The Docker team has added support to pass specific sockets such as the socket Docker itself uses, as well as the SSH agent socket; but exposing arbitrary sockets has been deemed out of scope for now.)

Therefore, since it is not possible to expose a socket for an X display to a container running in Docker Desktop, the X display will need to be exposed over the network shared by the host and container.

Docker Networking

In Docker Engine, networking is relatively straightforward. On the host, a network interface bridge called `docker0` is installed. This bridge serves to mediate communication among containers as well as between containers and the host. If the host would like to expose a network-based service to a container, it can bind to a port on `docker0` and containers can connect to it.

In Docker Desktop, however, the Linux VM in which containers run complicates matters. Inside the Linux VM, it's largely a similar configuration with a network bridge but the host machine cannot directly interact with this bridge interface. Instead, the host's network interface is assigned an address on the bridge similar to how other containers are. In this way, containers can still connect to network-based services on the host but not through a shared network interface called `docker0`.

To help simplify the configurations for applications running inside Docker Desktop containers, the hostname `host.docker.internal` will always resolve to an IP address for the host's network interface and thereby allow access to network-based services on the host.

Unlike Docker Desktop, Docker Engine cannot intercept DNS requests from containers; therefore, `host.docker.internal` must be configured when the container is started. This is accomplished via the `--add-host` option which allows mapping a hostname to an arbitrary address for the hostname. This mapping is applied by writing it in the container's `/etc/hosts` file. Using `--add-host`, `host.docker.internal` is mapped to the keyword `host-gateway`. This keyword is a special value that the Docker server will replace with an address from which the host will be reachable within a container whether it is Docker Engine or Docker Desktop starting it.

In conclusion, we can add `--add-host host.docker.internal:host-gateway` to the options to start a container and the host network interface will be reachable at `host.docker.internal`.

Exposing an X Display to a Container

Given the knowledge of the operation of the Docker implementations, we finally have the pieces to expose an X display to a container. Since it is not possible to expose the display's socket directly to a container, a TCP proxy is configured to pass X messages on the network from the container to the socket on the host machine for the display.

TCP Proxy

The socat tool is a widely available program to relay bi-directional data transfer between independent data channels. It allows running a process on the host to listen on a network port and send any received data to a socket connected to an X display on the other side.

Creating a TCP proxy for the X display effectively creates a spoofed X display. The proxy is configured to listen on the TCP port for an unallocated display; the port number will be 6000 + the number of the display. Additionally, the proxy is configured to listen on all network interfaces since identifying the exact interface that will be available within the container is non-trivial.

The other side of the proxy is connected to the socket for the X display. The socket, though, for the display may actually be another TCP socket; this will be the case if the environment is currently configured for X11 forwarding over SSH, as

discussed below. In most cases, though, the socket will be the UNIX socket for the display in the `/tmp/.X11-unix/` directory.

X Authentication for the Proxied X Display

Authentication for X displays is managed in `xauth` database files. The `xauth` program allows for reading and writing the database. The database is relatively simple mapping of display information to a cookie. When a client wants to establish a connection for a display, it queries the database for the display and receives the cookie back.

Since the proxy creates a spoofed display, a new `xauth` database needs to be created for the spoofed display using the authentication afforded to the user for the current display.

To create a new database, you need to:

- Extract the cookie for the current display

- Create a new database file

- Add an entry for the spoofed display using the extracted cookie to the new database

- Rewrite the hostname of the entry that was just created to be "FamilyWild" (a name reserved by the Xauth specification to match all displays)

This new `xauth` database file is set in the `XAUTHORITY` environment variable for the container so any X connections use it.

The hostname must be updated in the new database file because when the new entry is created, the `xauth` program will associate the host machine's hostname with the display. In the container, though, the `DISPLAY` variable will be using `host.docker.internal` as the hostname for the display. If it is not updated, then the authentication cannot be used. Furthermore, the `xauth` program will not allow creating authentication entries for displays that do not actually exist. So, we manually update the hostname of the entry to a wildcard value such that queries for the display number will return the authentication regardless of the hostname of the query.

X11 Forwarding over SSH

A common practice is to forward X11 communication from a remote machine to the local machine when using SSH. Therefore, when someone establishes an SSH connection to another machine and runs Briefcase, this X11 passthrough mechanism should passthrough the X11 forwarding for SSH in to the Docker container.

When X11 forwarding is configured for SSH, there are multiple channels established between the local and remote machine. The primary channel facilitates the interactive shell session; additionally, though, SSH sets up another channel for the X communication.

It accomplishes X11 forwarding in much the same way that Briefcase is proxying X communication from the Docker container to the host. On the remote machine, the X11 channel is bound to the TCP port for a spoofed display and creates a new entry in the user's `xauth` database for the display. Unlike Briefcase's proxy, the SSH proxy actively modifies some of the X messages; it will verify connection attempts use the authentication created in the database by SSH and will replace it with the actual authentication used on the local machine.

Since Briefcase will first connect to a TCP socket for a display, it will find the spoofed display created by SSH and create the proxy such that it connects to that TCP socket. In this way, the container sends X messages to the proxy, the proxy send them to the SSH X11 channel, and SSH translates them for the X display on the local machine.

2.2.7 Publishing your app

Some Briefcase platforms are linked to app distribution systems. This documentation covers how to publish your app to the appropriate distribution system.

Android

The Google Play Store is the most widely-used Android app store. This guide focuses on how to distribute a BeeWare app on the Google Play Store.

Build the app in release mode

Use Briefcase to build a release bundle for your application:

macOS

Linux

Windows

```
(venv) $ briefcase package android
[hello-world] Building Android App Bundle and APK in release mode...
...
[hello-world] Packaged dist/Hello World-1.0.0.aab
```

```
(venv) $ briefcase package android
[hello-world] Building Android App Bundle and APK in release mode...
...
[hello-world] Packaged dist/Hello World-1.0.0.aab
```

```
(venv) C:\...>briefcase package android
[hello-world] Building Android App Bundle and APK in release mode...
...
[hello-world] Packaged dist\Hello World-1.0.0.aab
```

This will result in an Android App Bundle file being generated. An Android App Bundle is a publishing format that includes all your app's compiled code and resources.

AAB and APK

APK (Android Package) files can be directly installed on a device. AAB is a newer format that simplifies the process of uploading your app to the Play Store, allows Google to manage the signing process, and allows the APK that is installed on your end-user's device to be smaller.

Sign the Android App Bundle

Note: Before you sign the APK files, you need to *create a code signing identity.*

The Google Play Store requires that the Android App Bundle is signed before it is uploaded, using the Java jarsigner tool.

In this example below, we assume your code signing identity is stored in **upload-key-helloworld.jks** under `.android` within your home folder. We also assume that the app's formal name is Hello World. You will need to change the path to the AAB file based on your app's formal name.

macOS

Linux

Windows (cmd)

Windows (PowerShell)

```
$ ~/Library/Caches/org.beeware.briefcase/tools/java/Contents/Home/bin/jarsigner -verbose␣
↪-sigalg SHA1withRSA -digestalg SHA1 -keystore ~/.android/upload-key-helloworld.jks
↪"dist/Hello World-1.0.0.aab" upload-key -storepass android
  adding: META-INF/MANIFEST.MF
  adding: META-INF/UPLOAD-K.SF
  adding: META-INF/UPLOAD-K.RSA
 signing: BundleConfig.pb
 signing: BUNDLE-METADATA/com.android.tools.build.libraries/dependencies.pb
 signing: base/assets/python/app/README
...
 signing: base/manifest/AndroidManifest.xml
 signing: base/assets.pb
 signing: base/native.pb
 signing: base/resources.pb
>>> Signer
  X.509, CN=Upload Key
  [trusted certificate]

jar signed.

Warning:
The signer's certificate is self-signed.
```

```
$ ~/.cache/briefcase/tools/java/bin/jarsigner -verbose -sigalg SHA1withRSA -digestalg␣
↪SHA1 -keystore ~/.android/upload-key-helloworld.jks "dist/Hello World-1.0.0.aab"␣
↪upload-key -storepass android
  adding: META-INF/MANIFEST.MF
  adding: META-INF/UPLOAD-K.SF
  adding: META-INF/UPLOAD-K.RSA
 signing: BundleConfig.pb
 signing: BUNDLE-METADATA/com.android.tools.build.libraries/dependencies.pb
 signing: base/assets/python/app/README
...
 signing: base/manifest/AndroidManifest.xml
 signing: base/assets.pb
```

(continues on next page)

```
   signing: base/native.pb
   signing: base/resources.pb
>>> Signer
   X.509, CN=Upload Key
   [trusted certificate]

jar signed.

Warning:
The signer's certificate is self-signed.
```

```
C:\...>%LOCALAPPDATA%\BeeWare\briefcase\Cache\tools\java\bin\jarsigner.exe -verbose -
→sigalg SHA1withRSA -digestalg SHA1 -keystore %HOMEPATH%\.android\upload-key-helloworld.
→jks "dist\Hello World-1.0.0.aab" upload-key -storepass android
   adding: META-INF/MANIFEST.MF
   adding: META-INF/UPLOAD-K.SF
   adding: META-INF/UPLOAD-K.RSA
  signing: BundleConfig.pb
  signing: BUNDLE-METADATA/com.android.tools.build.libraries/dependencies.pb
  signing: base/assets/python/app/README
...
  signing: base/manifest/AndroidManifest.xml
  signing: base/assets.pb
  signing: base/native.pb
  signing: base/resources.pb
>>> Signer
   X.509, CN=Upload Key
   [trusted certificate]

jar signed.

Warning:
The signer's certificate is self-signed.
```

```
PS C:\...> & "$env:LOCALAPPDATA\BeeWare\briefcase\Cache\tools\java\bin\jarsigner.exe" -
→verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore "$env:HOMEPATH\.android\upload-
→key-helloworld.jks" "android\gradle\Hello World\app\build\outputs\bundle\release\app-
→release.aab" upload-key -storepass android
   adding: META-INF/MANIFEST.MF
   adding: META-INF/UPLOAD-K.SF
   adding: META-INF/UPLOAD-K.RSA
  signing: BundleConfig.pb
  signing: BUNDLE-METADATA/com.android.tools.build.libraries/dependencies.pb
  signing: base/assets/python/app/README
...
  signing: base/manifest/AndroidManifest.xml
  signing: base/assets.pb
  signing: base/native.pb
  signing: base/resources.pb
>>> Signer
   X.509, CN=Upload Key
```

```
  [trusted certificate]

jar signed.

Warning:
The signer's certificate is self-signed.
```

You can safely ignore the warning about the signer's certificate being self-signed. Google will manage the process of signing the app with a verified certificate when you upload your app for distribution.

Add the app to the Google Play store

To publish to the Google Play store, you will need a Google Play Developer account, which costs 25 USD. You will then need to provide information for your app's store listing including an icon and screenshots, upload the app to Google, and finally roll the app out to production.

Register for a Google Play Developer account

Registering for a Google Play Developer account requires a Google Account. You will need to pay registration fee and accept an agreement in the process.

To check if you already have a Google Play Developer account, you can visit the Google Play console. If you see a button to **Publish an Android App on Google Play** or a button to **Create Application**, you can skip this step.

To create your Google Play developer account, pay the fee, and review the agreements, follow Google's documentation.

Create a listing

Visit the Google Play console. and log in. You will see a button labeled **Create App**; click this button.

Fill out the details for your app. We suggest using your app's formal name (as defined in `pyproject.toml` as the App name; the other details relate to the listing and legal compliance. At the bottom of of the page, press **Create App**.

This will take you to **Store Listing** section of your app. You will need to provide a short app description (up to 80 characters) and a full description (up to 4000 characters). Your app metadata may be helpful here.

You will also need to provide a collection of assets that will be used to promote your application:

- **A 512x512px icon.** This will be the icon that appears in the Play Store. It should match the icon you set on the application itself.

- **At least 2 screen screenshots of the app.** Google recommends using a screenshot without framing. One way to capture such a screenshot is with the Android emulator's screenshot functionality (the camera icon on the simulator controls). This allows your screenshot to contain just what appears on the screen rather than a picture of the virtual device. This will store a file in your Desktop folder.

  Screenshots must be at least 320px on their smallest dimension, no larger than 3480px on their largest dimension, and can't have an aspect ratio more extreme than 2:1. A screenshot from the Android emulator typically fulfills these requirements.

- **A 1024x500px feature graphic.** A feature graphic visually represents the purpose of the app or your logo and can optionally include a screenshot of the app in use, typically including device framing.

Google Play supports optional graphic assets including promo videos, TV banners, and 360 degree stereoscopic images. See also Google's advice on graphic assets.

Once you've completed the store listing, you'll need to fill out a range of other details about your app, including the category where it should appear in the Play Store, pricing details, details about the app's content and its suitability for children, and contact details for you as a developer. The navigation pane (typically on the left side of the screen) contains grayed out check marks covering all the sections with required details. Visit each of these sections in turn; when you have met the requirements of each section, the check mark will turn green. Once all the checkmarks are green, you're ready to release your app.

Create a release

In the left navigation bar, select **Production** (in the "Release" grouping), Then select **Create Release.** If prompted to enable App Signing by Google Play, click **Continue**.

Non-production releases

The Play Store also supports releasing your app for internal, alpha and beta testing. Google's documentation contains more details about creating test releases.

In an earlier section of this tutorial, we used `briefcase publish` and `jarsigner` to create a signed Android App Bundle file. It is stored in the `dist` folder of your project. Upload this file to the Google Play console in the **App Bundles** section, fill out the **Release notes** section of the app, and click **Next**.

Google will then check that you've filled out all the necessary compliance details for your app; if there are any missing, you'll be prompted to complete those details.

Once you've completed those details, select **Publishing Overvie** from the navigation sidebar. You should see "Changes ready to send for review", and a button marked **Send for review**. Click this button.

The Google Play Store will now review your app. You will be emailed if any updates are required; otherwise, after a day or two, your app will be rolled out to the Play Store.

Publish an update

At some point, you'll want to publish an updated version of your application. Generate a fresh AAB file, signed with the *same* certificate as your original release. Then log into the Play Store console, and select your application. Select **Release Management** in the navigation bar, then **App Releases**.

At this point, the release process is the same as it was for your initial release; create a release, upload your AAB file, and submit the application for roll out.

iOS

This guide will walk you through the process of publishing an app to the Apple App Store.

To distribute an app on the iOS App Store, you'll need to *enroll in the Apple Developer Program*. You don't need to generate any of the certificates described on that page - you just need an Apple ID registered in the developer program.

Once you've signed up for an Apple ID account, open the Xcode Settings dialog, and add your account under the "Accounts" tab.

Open the app in Xcode

Use Briefcase to open the Xcode project associated with your project.

```
(venv) $ briefcase open iOS
```

Run the app in the simulator

In order to submit your app to the App Store, you will need to provide a range of screenshots:

- 3-5 screenshots running on a 6.5" iPhone (e.g., iPhone 14 Plus)
- 3-5 screenshots running on a 5.5" iPhone (e.g., iPhone 8 Plus)
- 3-5 screenshots running on an 12.9" iPad Pro (Gen 6; without a physical home button)
- 3-5 screenshots running on an 12.9" iPad Pro (Gen 2; with a physical home button)
- (optionally) 3-5 screenshots running on a 6.7" iPhone (e.g., iPhone 14 Pro Max)

The iOS and iPad simulators have a "Save Screen" button in their title bar; this will capture screenshots of the necessary size.

You can change simulator device by clicking on the device target int the top bar of the Xcode window. Click on the device name, and select the device from the list. If the device you need doesn't exist on this list, click on "Manage Run Destinations" to add a simulator for that device type.

Produce an App archive

Select the root node of the Xcode project browser (it should be the formal name of your app), then select the **Signing & Capabilities** tab from configuration options that are displayed. The "Team" option under "Signing" will be listed as "None"; select the name of the development team that will sign the app. If there's no team listed, select "Add an Account", and choose one of the teams that is associated with your Apple ID.

In the top bar of the Xcode window, change the target device from a simulator to "Any iOS device". Clean the build products folder (select "Clean Build Folder…" from the Product menu), then build an archive by selecting "Archive" from the Product menu. This will perform a clean build of your application, build an archive, and open a new window, called the Organizer. It should list a freshly created archive of your app, with the current version number.

Select the archive, and click the "Distribute App" on the right side of the Organizer window. This will display a wizard that will ask details about your app; accept the default values; once the wizard completes, your app binary has been sent to the App Store for inclusion in a release.

After a few minutes, you should receive an email notifying you that the binary has been processed.

Create an App Store entry

Log into App Store Connect, click on "My Apps", then on + to add an app.

Fill out the form for a new app. If you've run the app in Xcode, the Bundle ID for your app should be listed; select it from the list. You must also create an SKU for your app - we suggest `ios-<appname>`, substituting the short app name that you selected when you initially created your app. So, if you've created an app with a formal name of "Hello World", with an app name of `helloworld`, and a bundle of `org.beeware`, you should have a Bundle ID of `org.beeware.helloworld`; we'd suggest an SKU of `ios-helloworld`.

You'll then be shown another page for app details, including:

- Primary and Secondary Category.

- Screenshots

- Promotional Text

- Description

- Keywords

- Support URL

- Marketing URL

- A URL for your app's privacy policy

- Version number

- The name of the copyright holder

Under the "Build" section, you'll be able to select the archive that you uploaded through Xcode.

The "App Review Information" section allows you to provide contact details in case Apple has questions during the review process. If your app requires a login, you *must* provide a set of credentials so that Apple can log in. You can also provide any additional notes to assist the reviewer.

Click on "Pricing and Availability" tab on the sidebar, and set up the pricing schedule and availability for your app.

Then, click on "App Privacy", and click on "Get Started"; this will ask you a series of questions about the information about users that your app collects.

Once these details have all been provided, click on the "1.0 Prepare for Submission" link in the sidebar. On the right of the screen, click on "Add for Review"; this will ask some final questions, and provide one more button "Submit for Review". Click that button, and you're done!

2.3 About Briefcase

2.3.1 Frequently Asked Questions

What version of Python does Briefcase support?

Python 3.8 or higher.

What platforms does Briefcase support?

Briefcase currently has support for:

- macOS (producing DMG files, or raw .app files)

- Linux (producing system packages, AppImage files or Flatpaks)

- Windows (producing MSI installers)

- iOS (producing Xcode projects)

- Android (producing Gradle projects)

Support for other some other operating systems (e.g., tvOS, watchOS, WearOS, and the web) are on our road map.

Briefcase's platform support is built on a plugin system, so if you want to add support for a custom platform, you can do so; or, you can contribute the backend to Briefcase itself.

How do I detect if my app is running in a Briefcase-packaged container?

Briefcase adds a PEP566 metadata file when it installs your app's code. The metadata can be retrieved at runtime as described in the *Accessing Briefcase packaging metadata at runtime* how-to. You can determine if your app was packaged with Briefcase by testing for the existence of the `Briefcase-Version` tag:

```
in_briefcase = 'Briefcase-Version' in metadata
```

Can I use third-party Python packages in my app?

Yes! Briefcase uses `pip` to install third-party packages into your app bundle. As long as the package is available on PyPI, or you can provide a wheel file for the package, it can be added to the `requires` declaration in your `pyproject. toml` file and used by your app at runtime.

If the package is pure-Python (i.e., it does not contain a binary library), that's all you need to do.

If the package contains a binary component, you'll need to ensure that a binary wheel is available for the platform you're targeting:

- **macOS, Linux, Windows**: Binary wheels are hosted on PyPI.
- **Android**: See the *Android platform documentation*.
- **iOS**: See the *iOS platform documentation*.
- **Web**: Binary wheel support is currently limited to those provided by the Pyodide project.

2.3.2 The Briefcase Developer and User community

Briefcase is part of the BeeWare suite. You can talk to the community through:

- @beeware@fosstodon.org on Mastodon
- Discord
- BeeWare Getting Help page

Code of Conduct

The BeeWare community has a strict Code of Conduct. All users and developers are expected to adhere to this code.

If you have any concerns about this code of conduct, or you wish to report a violation of this code, please contact the project founder Russell Keith-Magee.

Contributing

If you experience problems with Briefcase, log them on GitHub. If you want to contribute code, please fork the code and submit a pull request.

2.3.3 Success Stories

Want to see examples of Briefcase in use? Here's some:

- Travel Tips is an app in the iOS App Store that was packaged for distribution using Briefcase.

- Mu is a simple code editor for beginner programmers. It uses Briefcase to prepare a macOS installer.

- Napari is a multi-dimensional image viewer for python. It uses Briefcase to prepare bundled Windows, MacOS, and Linux installers.

2.3.4 Release History

0.3.17 (2024-02-06)

Features

- Android apps are now able to customize the libraries included in the app at build time. (#485)

- App permissions can now be declared as part of an app's configuration. (#547)

- The `-C/--config` option can now be used to override app settings from the command line. (#1115)

- The verbosity flag, `-v`, was expanded to support three levels of logging verbosity. (#1501)

- Briefcase now supports GUI bootstrap plugins to customize how new projects are created. (#1524)

- GitPython's debug logging is now included in deep debug output. (#1529)

- RCEdit v2.0.0 is now used to build Windows apps. Run `briefcase upgrade` to use this latest version. (#1543)

- The Flatpak runtimes for new projects were updated. `org.freedesktop.Platform` will now default to 23.08; `org.gnome.Platform` will now default to 45; and `org.kde.Platform` will now default to 6.6. (#1545)

- When creating new projects with the `briefcase new` command, project configuration overrides can be specified via the `-Q` command line argument. For instance, a specific license can be specified with `-Q "license=MIT license"`. (#1552)

- New virtual devices for the Android emulator are created using the Pixel 7 Pro skin. (#1554)

- The web server for running static web projects now falls back to a system allocated port if the requested port is already in use. (#1561)

- Flatpaks are now created with permissions to access the GPU and sound devices. (#1563)

- AppImages can now be built for the ARM architecture. (#1564)

- Apps can now specify a primary color (for both light and dark modes), and an accent color. If the platform allows apps to customize color use, these colors will be used to style the app's presentation. (#1566)

- The version of PursuedPyBear for new projects was bumped from 1.1 to 3.2.0. (#1592)

- Python 3.12 is now supported on Android. (#1596)

- Android apps can now specify the base theme used to style the application. (#1610)

- The Java JDK was upgraded from 17.0.8.1+1 to 17.0.10+7. Run `briefcase upgrade java` to upgrade existing Briefcase installations. (#1611)

- When the Android emulator fails to start up properly, users are now presented with additional resources to help resolve any issues. (#1630)

Bugfixes

- When a custom Briefcase template from a git repository is used to create an app, Briefcase now ensures that git repository is always used. (#1158)

- The filter for iOS build warnings was improved to catch messages from Xcode 15.0.1. (#1507)

- When merging dependencies on macOS, file permissions are now preserved. (#1510)

- `flatpak-builder` 1.3+ can now be correctly identified. (#1513)

- The BeeWare icon of Brutus is now used as the runtime icon for new projects created with PyGame. (#1532)

- Linux System RPM packaging for openSUSE Tumbleweed no longer errors with `FileNotFoundError`. (#1595)

- Any ANSI escape sequences or console control codes are now stripped in all output captured in the Briefcase log file. (#1604)

- The detection of physical Android devices on macOS was made more resilient. (#1627)

Backward Incompatible Changes

- The use of AppImage as an output format now generates a warning. (#1500)

- Support for creating new projects using PySide2 has been removed. Briefcase's release testing will no longer explicitly verify compatibility with PySide2. (#1524)

- The Flatpak build process no longer strips binaries included in third-party (e.g. PyPI) packages that are bundled with the app. (#1540)

- New projects will now use `manylinux_2_28` instead of `manylinux2014` to create AppImages in Docker. (#1564)

- It is highly recommended that Android applications add a definition for `build_gradle_dependencies` to their app configuration. A default value will be used if this option is not explicitly provided. Refer to the Android documentation for the default value that will be used. (#1610)

Documentation

- The common options available to every command have now been documented. (#1517)

Misc

- #1504, #1505, #1506, #1515, #1516, #1518, #1519, #1526, #1527, #1533, #1534, #1535, #1536, #1538, #1541, #1548, #1549, #1550, #1551, #1555, #1556, #1557, #1560, #1562, #1567, #1568, #1569, #1571, #1575, #1576, #1579, #1582, #1585, #1586, #1589, #1590, #1597, #1606, #1607, #1613, #1614, #1615, #1618, #1621, #1622, #1623, #1624, #1628, #1632, #1633

0.3.16 (2023-10-20)

Features

- Support for less common environments, such as Linux on ARM, has been improved. Error messages for unsupported platforms are now more accurate. (#1360)

- Tool verification for Java, Android SDK, and WiX have been improved to provide more informative errors and debug logging. (#1382)

- A super verbose logging mode was added (enabled using `-vv`). This turns on all Briefcase internal logging, but also enables verbose logging for all the third-party tools that Briefcase invokes. (#1384)

- Briefcase now uses Android SDK Command-Line Tools v9.0. If an externally-managed Android SDK is being used, it must provide this version of Command-Line Tools. Use the SDK Manager in Android Studio to ensure it is installed. (#1397)

- Support for OpenSuSE Linux distributions was added. (#1416)

- iOS apps are no longer rejected by the iOS App Store for packaging reasons. (#1439)

- The Java JDK version was upgraded to 17.0.8.1+1. (#1462)

- macOS apps can now be configured to produce single platform binaries, or binaries that will work on both x86_64 and ARM64. (#1482)

Bugfixes

- Build warnings caused by bugs in Xcode that can be safely ignored are now filtered out of visible output. (#377)

- The run command now ensures Android logging is shown when the datetime on the device is different from the host machine. (#1146)

- Briefcase will detect if you attempt to launch an Android app on a device whose OS doesn't meet minimum version requirements. (#1157)

- macOS apps are now guaranteed to be universal binaries, even when dependencies only provide single-architecture binary wheels. (#1217)

- The ability to build AppImages in Docker on macOS was restored. (#1352)

- Error reporting has been improved when the target Docker image name is invalid. (#1368)

- Creating Debian packages no longer fails due to a permission error for certain `umask` values (such as `0077`). (#1369)

- Inside of Docker containers, the Briefcase data directory is now mounted at `/briefcase` instead of `/home/brutus/.cache/briefcase`. (#1374)

- The console output from invoking Python via a subprocess call is now properly decoded as UTF-8. (#1407)

- The command line arguments used to configure the Python environment for `briefcase dev` no longer leak into the runtime environment on macOS. (#1413)

Backward Incompatible Changes

- AppImage packaging requires a recent release of LinuxDeploy to continue creating AppImages. Run `briefcase upgrade linuxdeploy` to install the latest version. (#1361)

- The size of iOS splash images have changed. iOS apps should now provide 800px, 1600px and 2400px images (previously, this as 1024px, 2048px and 3072px). This is because iOS 14 added a hard limit on the size of image resources. (#1371)

- Support for AppImage has been reduced to "best effort". We will maintain unit test coverage for the AppImage backend, but we no longer build AppImages as part of our release process. We will accept bug reports related to AppImage support, and we will merge PRs that address AppImage support, but the core team no longer considers addressing AppImage bugs a priority, and discourages the use of AppImage for new projects. (#1449)

Documentation

- Documentation on the process of retrieving certificate identities on macOS and Windows was improved. (#1473)

Misc

- #1136, #1290, #1363, #1364, #1365, #1372, #1375, #1376, #1379, #1388, #1394, #1395, #1396, #1398, #1400, #1401, #1402, #1403, #1408, #1409, #1410, #1411, #1412, #1418, #1419, #1420, #1421, #1427, #1429, #1431, #1433, #1435, #1436, #1437, #1438, #1442, #1443, #1444, #1445, #1446, #1447, #1448, #1454, #1455, #1456, #1457, #1464, #1465, #1466, #1470, #1474, #1476, #1477, #1478, #1481, #1485, #1486, #1487, #1488, #1489, #1490, #1492, #1494

0.3.15 (2023-07-10)

Features

- Windows apps can now be packaged as simple ZIP files. (#457)

- An Android SDK specified in `ANDROID_HOME` is respected now and will take precedence over the setting of `ANDROID_SDK_ROOT`. (#463)

- Android support was upgraded to use Java 17 for builds. (#1065)

- On Linux, Docker Desktop and rootless Docker are now supported. (#1083)

- The company/author name in the installation path for Windows MSI installers is now optional. (#1199)

- macOS code signing is now multi-threaded (and therefore much faster!) (#1201)

- Briefcase will now honor PEP-621 project fields where they map to Briefcase configuration items. (#1203)

Bugfixes

- XML compatibility warnings generated by the Android build have been cleaned up. (#827)

- Non ASCII characters provided in the `briefcase new` wizard are quoted before being put into `pyproject.toml`. (#1011)

- Requests to the web server are now recorded in the log file. (#1090)

- An "Invalid Keystore format" error is no longer raised when signing an app if the local Android keystore was generated with a recent version of Java. (#1112)

- Content before a closing square bracket (`]`) or `.so`) is no longer stripped by the macOS and iOS log filter. (#1179)

- The option to run Linux system packages through Docker was removed. (#1207)

- Error handling for incomplete or corrupted Github clones of templates has been improved. (#1210)

- Application/Bundle IDs are normalized to replace underscores with dashes when possible (#1234)

- Filenames and directories in RPM package definitions are quoted in order to include filenames that include white space. (#1236)

- Briefcase will no longer display progress bars if the `FORCE_COLOR` environment variable is set. (#1267)

- When creating a new Briefcase project, the header line in `pyproject.toml` now contains the version of Briefcase instead of "Unknown". (#1276)

- Android logs no longer include timestamp and PID, making them easier to read on narrow screens. (#1286)

- An warning is no longer logged if the Java identified by macOS is not usable by Briefcase. (#1305)

- Incompatibilities with Cookiecutter 2.2.0 have been resolved. (#1347)

Backward Incompatible Changes

- Names matching modules in the Python standard library, and `main`, can no longer be used as an application name. (#853)

- The `--no-sign` option for packaging was removed. Briefcase will now prompt for a signing identity during packaging, falling back to adhoc/no signing as a default where possible. (#865)

- The version of OpenJDK for Java was updated from 8 to 17. Any Android apps generated on previous versions of Briefcase must be re-generated by running `briefcase create android gradle`. If customizations were made to files within the generated app, they will need to be manually re-applied after re-running the create command. (#1065)

- Flatpak apps no longer default to using the Freedesktop runtime and SDK version 21.08 when a runtime is not specified. Instead, the runtime now must be explicitly defined in the application configuration. (#1272)

Documentation

- All code blocks were updated to add a button to copy the relevant contents on to the user's clipboard. (#1213)
- The limitations of using WebKit2 in AppImage were documented. (#1322)

Misc

- #856, #1093, #1178, #1181, #1186, #1187, #1191, #1192, #1193, #1195, #1197, #1200, #1204, #1205, #1206, #1215, #1226, #1228, #1232, #1233, #1239, #1241, #1242, #1243, #1244, #1246, #1248, #1249, #1253, #1254, #1255, #1257, #1258, #1262, #1263, #1264, #1265, #1273, #1274, #1279, #1282, #1283, #1284, #1293, #1294, #1295, #1299, #1300, #1301, #1310, #1311, #1316, #1317, #1323, #1324, #1333, #1334, #1335, #1336, #1339, #1341, #1350, #1351

0.3.14 (2023-04-12)

Features

- Added support for code signing Windows apps. (#366)
- The base image used to build AppImages is now user-configurable. (#947)
- Support for Arch `.pkg.tar.zst` packaging was added to the Linux system backend. (#1064)
- Pygame was added as an explicit option for a GUI toolkit. (#1125)
- AppImage and Flatpak builds now use indygreg's Python Standalone Builds to provide Python support. (#1132)
- BeeWare now has a presence on Mastodon. (#1142)

Bugfixes

- When commands produce output that cannot be decoded to Unicode, Briefcase now writes the bytes as hex instead of truncating output or canceling the command altogether. (#1141)
- When `JAVA_HOME` contains a path to a file instead of a directory, Briefcase will now warn the user and install an isolated copy of Java instead of logging a `NotADirectoryError`. (#1144)
- If the Docker `buildx` plugin is not installed, users are now directed by Briefcase to install it instead of Docker failing to build the image. (#1153)

Misc

- #1133, #1138, #1139, #1140, #1147, #1148, #1149, #1150, #1151, #1156, #1162, #1163, #1168, #1169, #1170, #1171, #1172, #1173, #1177

0.3.13 (2023-03-10)

Features

- Distribution artefacts are now generated into a single `dist` folder. (#424)

- When installing application sources and dependencies, any `__pycache__` folders are now automatically removed. (#986)

- A Linux System backend was added, supporting `.deb` as a packaging format. (#1062)

- Support for `.rpm` packaging was added to the Linux system backend. (#1063)

- Support for passthrough arguments was added to the `dev` and `run` commands. (#1077)

- Users can now define custom content to include in their `pyscript.toml` configuration file for web deployments. (#1089)

- The `new` command now allows for specifying a custom template branch, as well as a custom template. (#1101)

Bugfixes

- Spaces are no longer used in the paths for generated app templates. (#804)

- The stub executable used by Windows now clears the threading mode before starting the Python app. This caused problems with displaying dialogs in Qt apps. (#930)

- Briefcase now prevents running commands targeting Windows platforms when not on Windows. (#1010)

- The command to store notarization credentials no longer causes Briefcase to hang. (#1100)

- macOS developer tool installation prompts have been improved. (#1122)

Misc

- #1070, #1074, #1075, #1076, #1080, #1084, #1085, #1086, #1087, #1094, #1096, #1097, #1098, #1103, #1109, #1110, #1111, #1119, #1120, #1130

0.3.12 (2023-01-30)

Features

- Briefcase is more resilient to file download failures by discarding partially downloaded files. (#753)

- All warnings from the App and its dependencies are now shown when running `briefcase dev` by invoking Python in development mode. (#806)

- The Dockerfile used to build AppImages can now include user-provided container setup instructions. (#886)

- It is no longer necessary to specify a device when building an iOS project. (#953)

- Briefcase apps can now provide a test suite. `briefcase run` and `briefcase dev` both provide a `--test` option to start the test suite. (#962)

- Initial support for Python 3.12 was added. (#965)

- Frameworks contained added to a macOS app bundle are now automatically code signed. (#971)

- The `build.gradle` file used to build Android apps can now include arbitrary additional settings. (#973)

- The run and build commands now have full control over the update of app requirements resources. (#983)

- Resources that require variants will now use the variant name as part of the filename by default. (#989)

- `briefcase open linux appimage` now starts a shell session in the Docker context, rather than opening the project folder. (#991)

- Web project configuration has been updated to reflect recent changes to PyScript. (#1004)

Bugfixes

- Console output of Windows apps is now captured in the Briefcase log. (#787)

- Android emulators configured with `_no_skin` will no longer generate a warning. (#882)

- Briefcase now exits normally when CTRL-C is sent while tailing logs for the App when using `briefcase run`. (#904)

- Backslashes and double quotes are now safe to be used for formal name and description (#905)

- The console output for Windows batch scripts in now captured in the Briefcase log. (#917)

- When using the Windows Store version of Python, Briefcase now ensures the cache directory is created in `%LOCALAPPDATA%` instead of the sandboxed location enforced for Windows Store apps. (#922)

- An Android application that successfully starts, but fails quickly, no longer stalls the launch process. (#936)

- The required Visual Studio Code components are now included in verification errors for Visual Studio Apps. (#939)

- It is now possible to specify app configurations for macOS Xcode and Windows VisualStudio projects. Previously, these sections of configuration files would be ignored due to a case discrepancy. (#952)

- Development mode now starts apps in PEP540 UTF-8 mode, for consistency with the stub apps. (#985)

- Local file references in requirements no longer break AppImage builds. (#992)

- On macOS, Rosetta is now installed automatically if needed. (#1000)

- The way dependency versions are specified has been modified to make Briefcase as accommodating as possible with end-user environments, but as stable as possible for development environments. (#1041)

- To prevent console corruption, dynamic console elements (such as the Wait Bar) are temporarily removed when output streaming is disabled for a command. (#1055)

Improved Documentation

- Release history now contains links to GitHub issues. (#1022)

Misc

- #906, #907, #918, #923, #924, #925, #926, #929, #931, #951, #959, #960, #964, #967, #969, #972, #981, #984, #987, #994, #995, #996, #997, #1001, #1002, #1003, #1012, #1013, #1020, #1021, #1023, #1028, #1038, #1042, #1043, #1044, #1045, #1046, #1047, #1048, #1049, #1051, #1052, #1057, #1059, #1061, #1068, #1069, #1071

0.3.11 (2022-10-14)

Features

- Added support for deploying an app as a static web page using PyScript. (#3)

- Briefcase log files are now stored in the `logs` subdirectory and only when the current directory is a Briefcase project. (#883)

Bugfixes

- Output from spawned Python processes, such as when running `briefcase dev`, is no longer buffered and displays in the console immediately. (#891)

Misc

- #848, #885, #887, #888, #889, #893, #894, #895, #896, #897, #899, #900, #908, #909, #910, #915

0.3.10 (2022-09-28)

Features

- iOS and Android now supports the installation of binary packages. (#471)

- Apps can now selectively remove files from the final app bundle using the `cleanup_paths` attribute. (#550)

- The Docker image for AppImage builds is created or updated for all commands instead of just `create`. (#796)

- The performance of Briefcase's tool verification process has been improved. (#801)

- Briefcase templates are now versioned by the Briefcase version, rather than the Python version. (#824)

- Android commands now start faster, as they only gather a list of SDK packages when needed to write a log file. (#832)

- Log messages can be captured on iOS if they originate from a dynamically loaded module. (#842)

- Added an "open" command that can be used to open projects in IDEs. (#846)

Bugfixes

- The Wait Bar is disabled for batch scripts on Windows to prevent hiding user prompts when CTRL+C is pressed. (#811)

- Android emulators that don't provide a model identifier can now be used to launch apps. (#820)

- All `linuxdeploy` plugins are made executable and ELF headers for AppImage plugins are patched for use in `Docker`. (#829)

- The RCEdit plugin can now be upgraded. (#837)

- When verifying the existence of the Android emulator, Briefcase now looks for the actual binary, not the folder that contains the binary. This was causing false positives on some Android SDK setups. (#841)

- When CTRL+C is entered while an external program is running, `briefcase` will properly abort and exit. (#851)

- An issue with running *briefcase dev* on projects that put their application module in the project root has been resolved. (#863)

Improved Documentation

- Added FAQ entries on the state of binary package support on mobile. (#471)

Misc

- #831, #834, #840, #844, #857, #859, #867, #868, #874, #878, #879

0.3.9 (2022-08-17)

Features

- Linux apps can now be packaged in Flatpak format. (#359)

- SDKs, tools, and other downloads needed to support app builds are now stored in an OS-native user cache directory instead of `~/.briefcase`. (#374)

- Windows MSI installers can now be configured to ask the user whether they want a per-user or per-machine install. (#382)

- The console output of Windows apps is now captured and displayed during `briefcase run`. (#620)

- Windows apps are now packaged with a stub application. This ensures that Windows apps present with the name and icon of the app, rather than the `pythonw.exe` name and icon. It also allows for improvements in logging and error handling. (#629)

- Temporary docker containers are now cleaned up after use. The wording of Docker progress messages has also been improved. (#774)

- Users can now define a `BRIEFCASE_HOME` environment variable. This allows you to specify the location of the Briefcase tool cache, allowing the user to avoid issues with spaces in paths or disk space limitations. (#789)

- Android emulator output is now printed to the console if it fails to start properly. (#799)

- `briefcase android run` now shows logs from only the current process, and includes all log tags except some particularly noisy and useless ones. It also no longer clears the `logcat` buffer. (#814)

Bugfixes

- Apps now have better isolation against the current working directory. This ensures that code in the current working directory isn't inadvertently included when an app runs. (#662)

- Windows MSI installers now install in `Program Files`, rather than `Program Files (x86)`. (#688)

- Linuxdeploy plugins can now be used when building Linux AppImages; this resolves many issues with GTK app deployment. (#756)

- Collision protection has been added to custom support packages that have the same name, but are served by different URLs. (#797)

- Python 3.7 and 3.8 on Windows will no longer deadlock when CTRL+C is sent during a subprocess command. (#809)

Misc

- #778, #783, #784, #785, #786, #787, #794, #800, #805, #810, #813, #815

0.3.8 (2022-06-27)

Features

- macOS apps are now notarized as part of the packaging process. (#365)
- Console output now uses Rich to provide visual highlights and progress bars. (#740)
- The macOS log streamer now automatically exits using the run command when the app exits. (#742)
- A verbose log is written to file when a critical error occurs or –log is specified. (#760)

Bugfixes

- Updating an Android app now forces a re-install of the app. This corrects a problem (usually seen on physical devices) where app updates wouldn't be deployed if the app was already on the device. (#395)
- The iOS simulator is now able to correctly detect the iOS version when only a device name is provided. (#528)
- Windows MSI projects are now able to support files with non-ASCII filenames. (#749)
- The existence of an appropriate Android system image is now verified independently to the existence of the emulator. (#762)
- The error message presented when the Xcode Command Line Tools are installed, but Xcode is not, has been clarified. (#763)
- The METADATA file generated by Briefcase is now UTF-8 encoded, so it can handle non-Latin-1 characters. (#767)
- Output from subprocesses is correctly encoded, avoiding errors (especially on Windows) when tool output includes non-ASCII content. (#770)

Improved Documentation

- Documented a workaround for ELF load command address/offset errors seen when using manylinux wheels. (#718)

Misc

- #743, #744, #755

0.3.7 (2022-05-17)

Features

- Apps can be updated as part of a call to package. (#473)

- The Android emulator can now be used on Apple M1 hardware. (#616)

- Names that are reserved words in Python (or other common programming languages) are now prevented when creating apps. (#617)

- Names that are invalid on Windows as filenames (such as CON and LPT0) are now invalid as app names. (#685)

- Verbose logging via `-v` and `-vv` now includes the return code, output, and environment variables for shell commands (#704)

- When the output of a wrapped command cannot be parsed, full command output, and failure reason is now logged. (#728)

- The iOS emulator will now run apps natively on M1 hardware, rather than through Rosetta emulation. (#739)

Bugfixes

- Bundle identifiers are now validated to ensure they don't contain reserved words. (#460)

- The error reporting when the user is on an unsupported platform or Python version has been improved. (#541)

- When the formal name uses non-Latin characters, the suggested Class and App names are now valid. (#612)

- The code signing process for macOS apps has been made more robust. (#652)

- macOS app binaries are now adhoc signed by default, ensuring they can run on M1 hardware. (#664)

- Xcode version checks are now more robust. (#668)

- Android projects that have punctuation in their formal names can now build without error. (#696)

- Bundle name validation no longer excludes valid country identifiers (like `in.example`). (#709)

- Application code and dist-info is now fully replaced during an update. (#720)

- Errors related to Java JDK detection now properly contain the value of JAVA_HOME instead of the word None (#727)

- All log entries will now be displayed for the run command on iOS and macOS; previously, initial log entries may have been omitted. (#731)

- Using CTRL+C to stop showing Android emulator logs while running the app will no longer cause the emulator to shutdown. (#733)

Misc

- #680, #681, #699, #726, #734

0.3.6 (2022-02-28)

Features

- On macOS, iOS, and Android, `briefcase run` now displays the application logs once the application has started. (#591)

- Xcode detection code now allows for Xcode to be installed in locations other than `/Applications/Xcode.app`. (#622)

- Deprecated support for Python 3.6. (#653)

Bugfixes

- Existing app packages are now cleared before reinstalling dependencies. (#644)

- Added binary patch tool for AppImages to increase compatibility. (#667)

Improved Documentation

- Documentation on creating macOS/iOS code signing identities has been added (#641)

Misc

- #587, #588, #592, #598, #621, #643, #654, #670

0.3.5 (2021-03-06)

Features

- macOS projects can now be generated as an Xcode project. (#523)

Bugfixes

- macOS apps are now built as an embedded native binary, rather than a shell script invoking a Python script. This was necessary to provide better support for macOS app notarization and sandboxing. (#523)

- Fixed the registration of setuptools entry points caused by a change in case sensitivity handling in Setuptools 53.1.0. (#574)

Misc

- #562

0.3.4 (2021-01-03)

Features

- Added signing options for all platforms. App signing is only implemented on macOS, but `--no-sign` can now be used regardless of your target platform. (#486)
- Windows MSI installers can be configured to be per-machine, system-wide installers. (#498)
- Projects can specify a custom branch for the template used to generate the app. (#519)
- Added the *–no-run* flag to the *dev* command. This allows developers to install app dependencies without running the app. (#522)
- The new project wizard will now warn users when they select a platform that doesn't support mobile deployment. (#539)

Bugfixes

- Modified the volume mounting process to allow for SELinux. (#500)
- Fixed missing signature for Python executable in macOS app bundle. This enables the packaged dmg to be notarized by Apple. (#514)
- Modified the Windows tests to allow them to pass on 32-bit machines. (#521)
- Fixed a crash when running with verbose output. (#532)

Improved Documentation

- Clarified documentation around system_requires dependencies on Linux. (#459)

Misc

- #465, #475, #496, #512, #518

0.3.3 (2020-07-18)

Features

- WiX is now auto-downloaded when the MSI backend is used. (#389)
- The `upgrade` command now provides a way to upgrade tools that Briefcase has downloaded, including WiX, Java, Linuxdeploy, and the Android SDK. (#450)

Bugfixes

- Binary modules in Linux AppImages are now processed correctly, ensuring that no references to system libraries are retained in the AppImage. (#420)

- If pip is configured to use a per-user site_packages, this no longer clashes with the installation of application packages. (#441)

- Docker-using commands now check whether the Docker daemon is running and if the user has permission to access it. (#442)

0.3.2 (2020-07-04)

Features

- Added pytest coverage to CI/CD process. (#417)

- Application metadata now contains a `Briefcase-Version` indicator. (#425)

- The device list returned by `briefcase run android` now uses the Android device model name and unique ID e.g. a Pixel 3a shows up as `Pixel 3a (adbDeviceId)`. (#433)

- Android apps are now packaged in Android App Bundle format. This allows the Play Store to dynamically build the smallest APK appropriate to a device installing an app. (#438)

- PursuedPyBear is now included in the new project wizard. (#440)

Bugfixes

- iOS builds will now warn if the Xcode command line tools are the active. (#397)

- Linux Docker builds no longer use interactive mode, allowing builds to run on CI (or other TTY-less devices). (#439)

Improved Documentation

- Documented the process of signing Android apps & publishing them to the Google Play store. (#342)

Misc

- #428

0.3.1 (2020-06-13)

Features

- The Linux AppImage backend has been modified to use Docker. This ensures that the AppImage is always built in an environment that is compatible with the support package. It also enables Linux AppImages to be built on macOS and Windows. "Native" builds (i.e., builds that *don't* use Docker) can be invoked using the `--no-docker` argument. (#344)

- A `PYTHONPATH` property has been added to `AppConfig` that describes the `sys.path` changes needed to run the app. (#401)

- Ad-hoc code signing is now possible on macOS with `briefcase package --adhoc-sign`. (#409)

- Android apps can now use use `-` in their bundle name; we now convert `-` to `_` in the resulting Android package identifier and Java package name. (#415)

- Mobile applications now support setting the background color of the splash screen, and setting a build identifier. (#422)

- Android now has a `package` command that produces the release APK. After manually signing this APK, it can be published to the Google Play Store. (#423)

Bugfixes

- Some stray punctuation in the Android device helper output has been removed. (#396)

- An explicit version check for Docker is now performed. (#402)

- The Linux build process ensures the Docker user matches the UID/GID of the host user. (#403)

- Briefcase now ensures that the Python installation ecosystem tools (`pip`, `setuptools`, and `wheel`), are all present and up to date. (#421)

Improved Documentation

- Documented that Windows MSI builds produce per-user installable MSI installers, while still supporting per-machine installs via the CLI. (#382)

- `CONTRIBUTING.md` has been updated to link to Briefcase-specific documentation. (#404)

- Removed references to the `build-system` table in `pyproject.toml`. (#410)

Misc

- #380, #384

0.3.0 (2020-04-18)

Features

- Converted Briefcase to be a PEP518 tool, rather than a setuptools extension. (#266)

0.2.10

- Improved pre-detection of Xcode and related tools

- Improved error handling when starting external tools

- Fixed iOS simulator integration

0.2.9

- Updated mechanism for starting the iOS simulator
- Added support for environment markers in `install_requires`
- Improved error handling when WiX isn't found

0.2.8

- Corrects packaging problem with `urllib3`, caused by inconsistency between `requests` and `boto3`.
- Corrected problems with Start menu targets being created on Windows.

0.2.7

- Added support for launch images for iPhone X, Xs, Xr, Xs Max and Xr Max
- Completed removal of internal pip API dependencies.

0.2.6

- Added support for registering OS-level document type handlers.
- Removed dependency on an internal pip API.
- Corrected invocation of gradlew on Windows
- Addressed support for support builds greater than b9.

0.2.5

- Restored download progress bars when downloading support packages.

0.2.4

- Corrected a bug in the iOS backend that prevented iOS builds.

0.2.3

- Bugfix release, correcting the fix for pip 10 support.

0.2.2

- Added compatibility with pip 10.
- Improved Windows packaging to allow for multiple executables
- Added a `--clean` command line option to force a refresh of generated code.
- Improved error handling for bad builds

0.2.1

- Improved error reporting when a support package isn't available.

0.2.0

- Added `-s` option to launch projects
- Switch to using AWS S3 resources rather than GitHub Files.

0.1.9

- Added a full Windows installer backend

0.1.8

- Modified template roll out process to avoid API limits on GitHub.

0.1.7

- Added check for existing directories, with the option to replace existing content.
- Added a Linux backend.
- Added a Windows backend.
- Added a splash screen for Android

0.1.6

- Added a Django backend (@glasnt)

0.1.5

- Added initial Android template
- Force versions of pip (>= 8.1) and setuptools (>=27.0)
- Drop support for Python 2

0.1.4

- Added support for tvOS projects
- Moved to using branches in the project template repositories.

0.1.3

- Added support for Android projects using VOC.

0.1.2

- Added support for having multi-target support projects. This clears the way for Briefcase to be used for watchOS and tvOS projects, and potentially for Python-OSX-support and Python-iOS-support to be merged into a single Python-Apple-support.

0.1.1

- Added support for app icons and splash screens.

0.1.0

- Initial public release.

2.4 Reference

This is the technical reference for public APIs provided by Briefcase.

2.4.1 Briefcase configuration options

Environment variables

BRIEFCASE_HOME

When briefcase runs, it will download the support files, tools, and SDKs necessary to support building and packaging apps. By default, it will store the files in a platform-native cache folder:

- macOS: `~/Library/Caches/org.beeware.briefcase`
- Windows: `%LOCALAPPDATA%\BeeWare\briefcase\Cache`
- Linux: `~/.cache/briefcase`

If you want to use a different folder to store the Briefcase resources, you can define a `BRIEFCASE_HOME` environment variable.

There are three restrictions on this path specification:

1. The path must already exist. If it doesn't exist, you should create it manually.

2. It *must not* contain any spaces.

3. It *must not* be on a network drive.

The second two restrictions both exist because some of the tools that Briefcase uses (in particular, the Android SDK) do not work in these locations.

2.4.2 Project configuration options

Briefcase is a PEP518-compliant build tool. It uses a `pyproject.toml` file, in the root directory of your project, to provide build instructions for the packaged file.

If you have an application called "My App", with source code in the `src/myapp` directory, the simplest possible `pyproject.toml` Briefcase configuration file would be:

```
[tool.briefcase]
project_name = "My Project"
bundle = "com.example"
version = "0.1"

[tool.briefcase.app.myapp]
formal_name = "My App"
description = "My first Briefcase App"
sources = ['src/myapp']
```

The configuration sections are tool specific, and start with the prefix `tool.briefcase`.

The location of the `pyproject.toml` file is treated as the root of the project definition. Briefcase should be invoked in a directory that contains a `pyproject.toml` file, and all relative file path references contained in the `pyproject.toml` file will be interpreted relative to the directory that contains the `pyproject.toml` file.

Configuration sections

A project that is packaged by Briefcase can declare multiple *applications*. Each application is a distributable product of the build process. A simple project will only have a single application. However, a complex project may contain multiple applications with shared code.

Each setting can be specified:

- At the level of an output format (e.g., settings specific to building macOS DMGs);
- At the level of an platform for an app (e.g., macOS specific settings);
- At the level of an individual app; or
- Globally, for all applications in the project.

When building an application in a particular output format, Briefcase will look for settings in the same order. For example, if you're building a macOS DMG for an application called `myapp`, Briefcase will look for macOS DMG settings for `myapp`, then for macOS settings for `myapp`, then for `myapp` settings, then for project-level settings.

[tool.briefcase]

The base `[tool.briefcase]` section declares settings that project specific, or are are common to all applications in this repository.

`[tool.briefcase.app.<app name>]`

Configuration options for a specific application.

<app name> must adhere to a valid Python distribution name as specified in PEP508. The app name must also *not* be a reserved word in Python, Java or JavaScript (i.e., app names like `switch` or `pass` would not be valid); and it may not include any of the filenames prohibited by Windows (i.e., CON, PRN, or LPT1).

`[tool.briefcase.app.<app name>.<platform>]`

Configuration options for an application that are platform specific. The platform must match a name for a platform supported by Briefcase (e.g., `macOS` or `windows`). A list of the platforms supported by Briefcase can be obtained by running `briefcase -h`, and inspecting the help for the `platform` option

`[tool.briefcase.app.<app name>.<platform>.<output format>]`

Configuration options that are specific to a particular output format. For example, macOS applications can be generated in `app` or `dmg` format.

This section can contain additional layers. for example, an app targeting the Linux `system` backend can define a `tool.briefcase.app.<app name>.linux.system.ubuntu.jammy` section to provide configurations specific to Ubuntu 22.04 "Jammy" deployments. See the documentation for each backend for more details.

Project configuration

Required values

bundle

A reverse-domain name that can be used to identify resources for the application e.g., `com.example`. The bundle identifier will be combined with the app name to produce a unique application identifier - e.g., if the bundle identifier is `com.example` and the app name is `myapp`, the application will be identified as `com.example.myapp`.

project_name

The project is the collection of all applications that are described by the briefcase configuration. For projects with a single app, this may be the same as the formal name of the solitary packaged app.

version

A PEP440 compliant version string.

Examples of valid version strings:

- `1.0`
- `1.2.3`
- `1.2.3.dev4` - A development release
- `1.2.3a5` - An alpha pre-release

- `1.2.3b6` - A Beta pre-release
- `1.2.3rc7` - A release candidate
- `1.2.3.post8` - A post-release

Optional values

author

The person or organization responsible for the project.

author_email

The contact email address for the person or organization responsible for the project.

url

A URL where more details about the project can be found.

Application configuration

Required

description

A short, one-line description of the purpose of the application.

sources

A list of paths, relative to the `pyproject.toml` file, where source code for the application can be found. The contents of any named files or folders will be copied into the application bundle. Parent directories in any named path will not be included. For example, if you specify `src/myapp` as a source, the contents of the `myapp` folder will be copied into the application bundle; the `src` directory will not be reproduced.

Unlike most other keys in a configuration file, `sources` is a *cumulative* setting. If an application defines sources at the global level, application level, *and* platform level, the final set of sources will be the *concatenation* of sources from all levels, starting from least to most specific.

Optional values

accent_color

A hexadecimal RGB color value (e.g., `#D81B60`) for a subtle secondary color to be used throughout an application to call attention to key elements. This setting is only used if the platform allows color modification, otherwise it is ignored.

---

author

The person or organization responsible for the application.

author_email

The contact email address for the person or organization responsible for the application.

build

A build identifier. An integer, used in addition to the version specifier, to identify a specific compiled version of an application.

cleanup_paths

A list of strings describing paths that will be *removed* from the project after the installation of the support package and app code. The paths provided will be interpreted relative to the app bundle folder (e.g., the `macOS/app/My App` folder in the case of a macOS app).

Paths can be:

- An explicit reference to a single file

- An explicit reference to a single directory

- Any file system glob accepted by `pathlib.glob` (See the Python documentation for details)

Paths are treated as format strings prior to glob expansion. You can use Python string formatting to include references to configuration properties of the app (e.g., `app.formal_name`, `app.version`, etc).

For example, the following `cleanup_paths` specification:

```
cleanup_paths = [
    "path/to/unneeded_file.txt",
    "path/to/unneeded_directory",
    "path/**/*.exe",
    "{app.formal_name}/content/extra.doc"
]
```

on an app with a formal name of "My App" would remove:

1. The file `path/to/unneeded_file.txt`

2. The directory `path/to/unneeded_directory`

3. Any `.exe` file in `path` or its subdirectories.

4. The file `My App/content/extra.doc`.

exit_regex

A regular expression that will be executed against the console output generated by an application. If/when the regular expression find match, the application will be terminated; the line matching the regular expression will *not* be output to the console. Used by Briefcase to monitor test suites; however, the filter will also be honored on normal `run` invocations.

The regular expression should capture a single group named `returncode`, capturing the integer exit status that should be reported for the process. The default value for this regular expression is `^>>>>>>>>>> EXIT (?P<returncode>.*) <<<<<<<<<<$` The regex will be compiled with the `re.MULTILINE` flag enabled.

formal_name

The application name as it should be displayed to humans. This name may contain capitalization and punctuation. If it is not specified, the `name` will be used.

icon

A path, relative to the directory where the `pyproject.toml` file is located, to an image to use as the icon for the application. The path should *exclude* the extension; Briefcase will append a platform appropriate extension when configuring the application. For example, an icon specification of `icon = "resources/icon"` will use `resources/icon.icns` on macOS, and `resources/icon.ico` on Windows.

Some platforms require multiple icons, at different sizes; these will be handled by appending the required size to the provided icon name. For example, iOS requires multiple icon sizes (ranging from 20px to 1024px); Briefcase will look for `resources/icon-20.png`, `resources/icon-1024.png`, and so on. The sizes that are required are determined by the platform template.

installer_icon

A path, relative to the directory where the `pyproject.toml` file is located, to an image to use as the icon for the installer. As with `icon`, the path should *exclude* the extension, and a platform-appropriate extension will be appended when the application is built.

installer_background

A path, relative to the directory where the `pyproject.toml` file is located, to an image to use as the background for the installer. The path should *exclude* the extension, and a platform-appropriate extension will be appended when the application is built.

long_description

A longer description of the purpose of the application. This description can be multiple paragraphs, if necessary. The long description *must not* be a copy of the `description`, or include the `description` as the first line of the long_description.

primary_color

A hexadecimal RGB color value (e.g., `#008577`) to use as the primary color for the application. This setting is only used if the platform allows color modification, otherwise it is ignored.

primary_color_dark

A hexadecimal RGB color value (e.g., `#008577`) used alongside the primary color. This setting is only used if the platform allows color modification, otherwise it is ignored.

requires

A list of packages that must be packaged with this application.

Unlike most other keys in a configuration file, `requires` is a *cumulative* setting. If an application defines requirements at the global level, application level, *and* platform level, the final set of requirements will be the *concatenation* of requirements from all levels, starting from least to most specific.

revision

An identifier used to differentiate specific builds of the same version of an app. Defaults to 1 if not provided.

splash_background_color

A hexadecimal RGB color value (e.g., `#6495ED`) to use as the background color for splash screens.

If the platform output format does not use a splash screen, this setting is ignored.

support_package

A file path or URL pointing at a tarball containing a Python support package. (i.e., a precompiled, embeddable Python interpreter for the platform)

If this setting is not provided, Briefcase will use the default support package for the platform.

support_revision

The specific revision of a support package that should be used. By default, Briefcase will use the support package revision nominated by the application template. If you specify a support revision, that will override the revision nominated by the application template.

If you specify an explicit support package (either as a URL or a file path), this argument is ignored.

supported

Indicates that the platform is not supported. For example, if you know that the app cannot be deployed to Android for some reason, you can explicitly prevent deployment by setting `supported=False` in the Android section of the app configuration file.

If `supported` is set to `false`, the create command will fail, advising the user of the limitation.

template

A file path or URL pointing at a cookiecutter template for the output format.

If this setting is not provided, Briefcase will use a default template for the output format and Python version.

template_branch

The branch of the project template to use when generating the app. If the template is a local file, this attribute will be ignored. If not specified, Briefcase will use a branch matching the version of Briefcase that is being used (i.e., if you're using Briefcase 0.3.9, Briefcase will use the `v0.3.9` template branch when generating the app). If you're using a development version of Briefcase, Briefcase will use the `main` branch of the template.

test_requires

A list of packages that are required for the test suite to run.

Unlike most other keys in a configuration file, `test_requires` is a *cumulative* setting. If an application defines requirements at the global level, application level, *and* platform level, the final set of requirements will be the *concatenation* of requirements from all levels, starting from least to most specific.

test_sources

A list of paths, relative to the `pyproject.toml` file, where test code for the application can be found. The contents of any named files or folders will be copied into the application bundle. Parent directories in any named path will not be included. For example, if you specify `src/myapp` as a source, the contents of the `myapp` folder will be copied into the application bundle; the `src` directory will not be reproduced.

As with `sources`, `test_sources` is a *cumulative* setting. If an application defines sources at the global level, application level, *and* platform level, the final set of sources will be the *concatenation* of test sources from all levels, starting from least to most specific.

url

A URL where more details about the application can be found.

Permissions

Applications may also need to declare the permissions they require. Permissions are specified as sub-attributes of a `permission` property, defined at the level of an project, app, or platform. Permission declarations are *cumulative*; if an application defines permissions at the global level, application level, *and* platform level, the final set of permissions will be the *merged* set of all permissions from all levels, starting from least to most specific, with the most specific taking priority.

Briefcase maintains a set of cross-platform permissions:

- `permission.camera` - permission to access to the camera to take photos or video.

- `permission.microphone` - permission to access the microphone.

- `permission.coarse_location` - permission to determine a rough GPS location.

- `permission.fine_location` - permission to determine a precise GPS location.

- `permission.background_location` - permission to track GPS location while in the background.

- `permission.photo_library` - permission to access to the user's photo library.

If a cross-platform permission is used, it will be mapped to platform-specific values in whatever files are used to define permissions on that platform.

Permissions can also be configured by adding platform-specific configuration items. See the documentation for the the platform backends to see the options that are available.

The value for each permission is a short description of why that permission is required. If the platform requires, the value may be displayed to the user as part of an authorization dialog. This description should describe *why* the app requires the permission, rather than a generic description of the permission being requested.

The use of permissions may also imply other settings in your app. See the individual platform backends for details on how cross-platform permissions are mapped.

Document types

Applications in a project can register themselves with the operating system as handlers for specific document types by adding a `document_type` configuration section for each document type the application can support. This section follows the format:

```
[tool.briefcase.app.<app name>.document_type.<extension>]
```

or, for a platform specific definition:

```
[tool.briefcase.app.<app name>.<platform>.document_type.<extension>]
```

where `extension` is the file extension to register. For example, `myapp` could register as a handler for PNG image files by defining the configuration section `[tool.briefcase.app.myapp.document_type.png]`.

The document type declaration requires the following settings:

description

A short, one-line description of the document format.

icon

A path, relative to the directory where the `pyproject.toml` file is located, to an image for an icon to register for use with documents of this type. The path should *exclude* the extension; Briefcase will append a platform appropriate extension when configuring the application. For example, an icon specification of:

```
icon = "resources/icon"
```

will use `resources/icon.icns` on macOS, and `resources/icon.ico` on Windows.

Some platforms also require different *variants* (e.g., both square and round icons). These variants can be specified by qualifying the icon specification:

```
icon.round = "resource/round-icon"
icon.square = "resource/square-icon"
```

Some platforms require multiple icons, at different sizes; these will be handled by appending the required size to the provided icon name. For example, iOS requires multiple icon sizes (ranging from 20px to 1024px); Briefcase will look for `resources/icon-20.png`, `resources/icon-1024.png`, and so on. The sizes that are required are determined by the platform template.

If a platform requires both different sizes *and* variants, the variant handling and size handling will be combined. For example, Android requires round and square icons, in sizes ranging from 48px to 192px; Briefcase will look for `resource/round-icon-42.png`, `resource/square-icon-42.png`, `resource/round-icon-192.png`, and so on.

url

A URL for help related to the document format.

PEP621 compatibility

Many of the keys that exist in Briefcase's configuration have analogous settings in PEP621 project metadata. If your `pyproject.toml` defines a `[project]` section, Briefcase will honor those settings as a top level definition. Any `[tool.briefcase]` definitions will override those in the `[project]` section.

The following PEP621 project metadata keys will be used by Briefcase if they are available:

- `version` maps to the same key in Briefcase.
- `authors` The `email` and `name` keys of the first value in the `authors` setting map to `author` and `author_email`.
- `dependencies` maps to the Briefcase `requires` setting. This is a cumulative setting; any packages defined in the `requires` setting at the `[tool.briefcase]` level will be appended to the packages defined with `dependencies` at the `[project]` level.
- `description` maps to the same key in Briefcase.
- `test` in an `[project.optional-dependencies]` section maps to `test_requires.`, As with `dependencies/requires`, this is a cumulative setting.
- `text` in a `[project.license]` section will be mapped to `license`.

- homepage in a [project.urls] section will be mapped to url.

2.4.3 Command reference

new

Start a new Briefcase project. Runs a wizard to ask questions about your new application, and creates a stub project using the details provided.

Usage

To start a new application, run:

```
$ briefcase new
```

Options

The following options can be provided at the command line.

-t <template> / --template <template>

A local directory path or URL to use as a cookiecutter template for the new project.

--template-branch <branch>

The branch of the cookiecutter template repository to use for the new project. If not specified, Briefcase will attempt to use a template branch matching the version of Briefcase that is being used (i.e., if you're using Briefcase 0.3.14, Briefcase will use the v0.3.14 template branch when generating the app). If you're using a development version of Briefcase, Briefcase will use the main branch of the template.

-Q <KEY=VALUE>

Override the answer to a new project prompt with the provided value.

For instance, if -Q "license=MIT license is specified, then the question to choose a license will not be presented to the user and the MIT license will be automatically used for the project. When used in conjunction with --no-input, the provided value overrides the default answer.

The expected keys are specified by the cookiecutter template being used to create the new project. Therefore, the set of possible keys is not listed here but should be expected to remain consistent for any specific version of Briefcase; with version changes, though, the keys may change.

convert

Convert an existing project with a `pyproject.toml` configuration into a project that can be deployed with Briefcase. Runs a wizard to ask questions about your existing application (reading from the `pyproject.toml` file where applicable), adds the necessary files to the project, and updates the `pyproject.toml` file to include the `briefcase` section.

Usage

To convert your application, run the following command in the application root directory (where the `pyproject.toml` file is located):

```
$ briefcase convert
```

Options

The following options can be provided at the command line.

-t <template> / --template <template>

A local directory path or URL to use as a cookiecutter template for the project.

--template-branch <branch>

The branch of the cookiecutter template repository to use for the project. If not specified, Briefcase will attempt to use a template branch matching the version of Briefcase that is being used (i.e., if you're using Briefcase 0.3.14, Briefcase will use the `v0.3.14` template branch when generating the app). If you're using a development version of Briefcase, Briefcase will use the `main` branch of the template.

-Q <KEY=VALUE>

Override the answer to a prompt with the provided value.

For instance, if `-Q "license=MIT license"` is specified, then the question to choose a license will not be presented to the user and the MIT license will be automatically used for the project. When used in conjunction with `--no-input`, the provided value overrides the default answer.

The expected keys are specified by the cookiecutter template being used to create a new project (the same cookiecutter template is used here). Therefore, the set of possible keys is not listed here but should be expected to remain consistent for any specific version of Briefcase; with version changes, though, the keys may change.

dev

Run the application in developer mode.

Usage

To run the app, run:

```
$ briefcase dev
```

The first time the application runs in developer mode, any requirements listed in a `requires` configuration item in `pyproject.toml` will be installed into the current environment.

Options

The following options can be provided at the command line.

-a <app name> / --app <app name

Run a specific application target in your project. This argument is only required if your project contains more than one application target. The app name specified should be the machine-readable package name for the app.

-r / --update-requirements

Update application requirements.

--no-run

Do not run the application; only install application requirements.

--test

Run the test suite in the development environment.

Passthrough arguments

If you want to pass any arguments to your app's command line, you can specify them using the `--` marker to separate Briefcase's arguments from your app's arguments. For example:

```
$ briefcase dev -- --wiggle --test
```

will run the app in normal mode, passing the `--wiggle` and `--test` flags to the app's command line. The app will *not* run in *Briefcase's* test mode; the `--test` flag will be left for your own app to interpret.

create

Create a scaffold for an application installer. By default, targets the current platform's default output format.

Usage

To create a scaffold for the default output format for the current platform:

```
$ briefcase create
```

To create a scaffold for a different platform:

```
$ briefcase create <platform>
```

To create a scaffold for a specific output format:

```
$ briefcase create <platform> <output format>
```

If a scaffold for the nominated platform already exists, you'll be prompted to delete and regenerate the app.

Options

There are no additional command line options.

update

While you're developing an application, you may need to rapidly iterate on the code, making small changes and then re-building. The update command applies any changes you've made to your code to the packaged application code.

It will *not* update requirements or installer resources unless specifically requested.

Usage

To repackage your application's code for the current platform's default output format:

```
$ briefcase update
```

To repackage your application's code for a different platform:

```
$ briefcase update <platform>
```

To repackage your application's code for a specific output format:

```
$ briefcase update <platform> <output format>
```

Options

The following options can be provided at the command line.

-r / --update-requirements

Update application requirements.

--update-resources

Update application resources such as icons.

--update-support

Update application support package.

build

Compile/build an application. By default, targets the current platform's default output format.

This will only compile the components necessary to *run* the application. It won't necessarily result in the generation of an installable artefact.

Usage

To build the application for the default output format for the current platform:

```
$ briefcase build
```

To build the application for a different platform:

```
$ briefcase build <platform>
```

To build the application for a specific output format:

```
$ briefcase build <platform> <output format>
```

Build tool requirements

Building for some platforms depends on the build tools for the platform you're targeting being available on the platform you're using. For example, you will only be able to create iOS applications on macOS. Briefcase will check for any required tools, and will report an error if the platform you're targeting is not supported.

Options

The following options can be provided at the command line.

-u / --update

Update the application's source code before building. Equivalent to running:

```
$ briefcase update
$ briefcase build
```

-r / --update-requirements

Update application requirements before building. Equivalent to running:

```
$ briefcase update -r
$ briefcase build
```

--update-resources

Update application resources such as icons before building. Equivalent to running:

```
$ briefcase update --update-resources
$ briefcase build
```

--update-support

Update application support package before building. Equivalent to running:

```
$ briefcase update --update-resources
$ briefcase build
```

--test

Build the app in test mode in the bundled app environment. Running `build --test` will also cause an update to ensure that the packaged application contains the current test code. To prevent this update, use the `--no-update` option.

If you have previously run the app in "normal" mode, you may need to pass `-r` / `--update-requirements` the first time you build in test mode to ensure that your testing requirements are present in the test app.

`--no-update`

Prevent the automated update of app code that is performed when specifying by the `--test` option.

run

Starts the application, using the packaged version of the application code. By default, targets the current platform's default output format.

If the output format is an executable (e.g., a macOS .app file), the `run` command will start that executable. If the output is an installer, `run` will attempt to replicate as much as possible of the runtime environment that would be installed, but will not actually install the app. For example, on Windows, `run` will use the interpreter that will be included in the installer, and the versions of code and requirements that will be installed, but *won't* run the installer to produce Start Menu items, registry records, etc.

Test mode

The `run` command can also be used to execute your app's test suite, in the packaged environment (e.g., on the iOS simulator, or from within a Linux Flatpak). When running in test mode (using the `--test` option), a different entry point will be used for the app: if your app is contained in a Python module named `myapp`, test mode will attempt to launch `tests.myapp`. Your app is responsible for providing the logic to discover and start the test suite.

The code for your test suite can specified using the `test_sources` setting; test-specific requirements can be specified with `test_requires`. Test sources and requirements will only be included in your app when running in test mode.

Briefcase will monitor the log output of the test suite, looking for the output corresponding to test suite completion. Briefcase has built-in support for pytest and unittest test suites; support for other test frameworks can be added using the `test_success_regex` and `test_failure_regex` settings.

Usage

To run your application on the current platform's default output format:

```
$ briefcase run
```

To run your application for a different platform:

```
$ briefcase run <platform>
```

To run your application using a specific output format:

```
$ briefcase run <platform> <output format>
```

Options

The following options can be provided at the command line.

-a <app name> / --app <app name

Run a specific application target in your project. This argument is only required if your project contains more than one application target. The app name specified should be the machine-readable package name for the app.

-u / --update

Update the application's source code before running. Equivalent to running:

```
$ briefcase update
$ briefcase build
$ briefcase run
```

-r / --update-requirements

Update application requirements before running. Equivalent to running:

```
$ briefcase update -r
$ briefcase build
$ briefcase run
```

--update-resources

Update application resources such as icons before running. Equivalent to running:

```
$ briefcase update --update-resources
$ briefcase build
$ briefcase run
```

--update-support

Update application support package before running. Equivalent to running:

```
$ briefcase update --update-resources
$ briefcase build
$ briefcase run
```

`--test`

Run the app in test mode in the bundled app environment. Running `run --test` will also cause an update and build to ensure that the packaged application contains the most recent test code. To prevent this update and build, use the `--no-update` option.

`--no-update`

Prevent the automated update and build of app code that is performed when specifying by the `--test` option.

Passthrough arguments

If you want to pass any arguments to your app's command line, you can specify them using the `--` marker to separate Briefcase's arguments from your app's arguments. For example:

```
briefcase run -- --wiggle --test
```

will run the app in normal mode, passing the `--wiggle` and `--test` flags to the app's command line. The app will *not* run in *Briefcase's* test mode; the `--test` flag will be left for your own app to interpret.

package

Compile/build an application installer. By default, targets the current platform's default output format.

This will produce an installable artefact.

Usage

To build an installer of the default output format for the current platform:

```
$ briefcase package
```

To build an installer for a different platform:

```
$ briefcase package <platform>
```

To build an installer for a specific output format:

```
$ briefcase package <platform> <output format>
```

Packaging tool requirements

Building installers for some platforms depends on the build tools for the platform you're targeting being available on the platform you're using. For example, you will only be able to create iOS applications on macOS. Briefcase will check for any required tools, and will report an error if the platform you're targeting is not supported.

Options

The following options can be provided at the command line.

-u / --update

Update and recompile the application's code before running. Equivalent to running:

```
$ briefcase update
$ briefcase package
```

-p <format>, --packaging-format <format>

The format to use for packaging. The available packaging formats are platform dependent.

--adhoc-sign

Perform the bare minimum signing that will result in a app that can run on your local machine. This may result in no signing, or signing with an ad-hoc signing identity. The `--adhoc-sign` option may be useful during development and testing. However, care should be taken using this option for release artefacts, as it may not be possible to distribute an ad-hoc signed app to others.

-i <identity> / --identity <identity>

The *code signing identity* to use when signing the app.

publish

COMING SOON

Uploads your application to a publication channel. By default, targets the current platform's default output format, using that format's default publication channel.

You may need to provide additional configuration details (e.g., authentication credentials), depending on the publication channel selected.

Usage

To publish the application artefacts for the current platform's default output format to the default publication channel:

```
$ briefcase publish
```

To publish the application artefacts for a different platform:

```
$ briefcase publish <platform>
```

To publish the application artefacts for a specific output format:

```
$ briefcase publish <platform> <output format>
```

Options

The following options can be provided at the command line.

-c <channel> / **--channel <channel>**

Nominate a publication channel to use.

upgrade

Briefcase uses external tools to manage the process of packaging apps. Where possible, Briefcase will manage the process of obtaining those tools. This is currently done for

- **WiX** (used by the Windows MSI backend)
- **linuxdeploy** (used by the Linux AppImage backend)
- **Java JDK** (used by the Android backed)
- **Android SDK** (used by the Android backend)

Over time, it may be necessary to upgrade these tools. The `upgrade` command provides a way to perform these upgrades.

If you are managing your own version of these tools (e.g., if you have downloaded a version of WiX and have set the `WIX_HOME` environment variable), you must manage any upgrades on your own.

Usage

To see what tools are currently being managed by Briefcase:

```
$ briefcase upgrade --list
```

To upgrade all the tools that are currently being managed by Briefcase:

```
$ briefcase upgrade
```

To upgrade a specific tool:

```
$ briefcase upgrade <tool_name>
```

Options

The following options can be provided at the command line.

-l / --list

List the tools that are currently being managed by Briefcase.

Common options

The following options are available to all commands:

-C <KEY=VALUE> / --config <KEY=VALUE>

Override the value of an app's configuration in `pyproject.toml` with the provided value.

The key will be applied *after* (and will therefore take precedence over) any platform or backend-specific configuration has been merged into the app's configuration. The key must be a "top level" TOML key. The use of dotted keys to define nested configuration structures is not permitted.

The value passed to the setting should be valid TOML. If the value being overridden is a string, this means you must quote the value. This may require the use of escape sequences at the command line to ensure the value provided to Briefcase by the shell includes the quotes.

For example, to override the template used by the create command, you can use `-C template=...`, but the value must be quoted:

```
briefcase create -C template=\"https://example.com/template\"
```

The only app key that *cannot* be overridden with `-C` is `app_name`, as it is used to identify apps.

-h / --help

Display the available options for the command, and a description of usage.

--log

Always generate a log file. A log file will *always* be generated if Briefcase doesn't exit normally; however, it can sometimes be useful to generate a log file when a command has succeeded unsuccessfully. The log file will always contain debug level output, regardless of the `-v` / `--verbosity` option specified.

`--no-input`

Do not ask the user for input, and use default values. If a safe default behavior exists, that default will be used; otherwise, the command will raise an error.

`-v / --verbosity`

The verbosity of output generated by Briefcase. `-v` can be specified multiple times to increase the verbosity:

- If no flag is specified, Briefcase will generate **default** output - enough detail to see that progress in a build is being made without being overwhelming.

- `-v` generates **verbose** output. For example, if a command processes a list of files, "normal" verbosity will show a progress bar as those files are processed, but verbose mode will output the name of each file being modified.

- `-vv` generates **debug** output. This will output more internal details of Briefcase operation. This includes the command line, environment, output and return value of every tool and command that Briefcase executes as part of a build.

- `-vvv` generates **deep debug** output. This will include the verbose output of all commands that Briefcase has invoked.

`-V / --version`

Print the version of Briefcase being used.

2.4.4 Platform support

macOS

The default output format for macOS is an *app*.

Briefcase also supports creating an *Xcode project* which in turn can be used to build an app.

Both output formats support packaging as a macOS DMG or as a standalone signed app bundle.

.app bundle

Host Platform Support (*Platform support*)								
macOS		Windows			Linux			
x86-64	arm64	x86	x86-64	arm64	x86	x86-64	arm	arm64

A macOS `.app` bundle is a collection of directory with a specific layout, and with some key metadata. If this structure and metadata exists, macOS treats the folder as an executable file, giving it an icon.

`.app` bundles can be copied around as if they are a single file. They can also be compressed to reduce their size for transport.

By default, apps will be both signed and notarized when they are packaged.

Packaging format

Briefcase supports two packaging formats for a macOS `.app` bundle:

1. A DMG that contains the `.app` bundle (the default output of `briefcase package macOS`, or by using `briefcase package macOS -p dmg`); or

2. A zipped `.app` folder (using `briefcase package macOS -p app`).

Icon format

macOS `.app` bundles use `.icns` format icons.

macOS `.app` bundles do not support splash screens or installer images.

Additional options

The following options can be provided at the command line when packaging macOS apps.

--no-notarize

Do not submit the application for notarization. By default, apps will be submitted for notarization unless they have been signed with an ad-hoc signing identity.

Application configuration

The following options can be added to the `tool.briefcase.app.<appname>.macOS.app` section of your `pyproject.toml` file.

entitlement

A property whose sub-attributes define keys that will be added to the app's `Entitlements.plist` file. Each entry will be converted into a key in the entitlements file. For example, specifying:

```
entitlement."com.apple.vm.networking" = true
```

will result in an `Entitlements.plist` declaration of:

```
<key>com.apple.vm.networking</key><true/>
```

Any Boolean or string value can be used for an entitlement value.

All macOS apps are automatically granted the following entitlements:

- `com.apple.security.cs.allow-unsigned-executable-memory`
- `com.apple.security.cs.disable-library-validation`

You can disable these default entitlements by defining them manually. For example, to enable library validation, you could add the following to your `pyproject.toml`:

```
entitlement."com.apple.security.cs.disable-library-validation" = false
```

info

A property whose sub-attributes define keys that will be added to the app's `Info.plist` file. Each entry will be converted into a key in the entitlements file. For example, specifying:

```
info."NSAppleScriptEnabled" = true
```

will result in an `Info.plist` declaration of:

```
<key>NSAppleScriptEnabled</key><true/>
```

Any Boolean or string value can be used for an `Info.plist` value.

universal_build

A Boolean, indicating whether Briefcase should build a universal app (i.e, an app that can target both x86_64 and ARM64). Defaults to `true`; if `false`, the binary will only be executable on the host platform on which it was built - i.e., if you build on an x86_64 machine, you will produce an x86_65 binary; if you build on an ARM64 machine, you will produce an ARM64 binary.

Permissions

Briefcase cross platform permissions map to a combination of `info` and `entitlement` keys:

- `microphone`: an `entitlement` of `com.apple.security.device.audio-input`

- `camera`: an `entitlement` of `com.apple.security.device.camera`

- `coarse_location`: an `info` entry for `NSLocationUsageDescription` (ignored if `background_location` or `fine_location` is defined); plus an entitlement of `com.apple.security.personal-information.location`

- `fine_location`: an `info` entry for `NSLocationUsageDescription``(ignored if ``background_location` is defined); plus an entitlement of `com.apple.security.personal-information.location`

- `background_location`: an `info` entry for `NSLocationUsageDescription`; plus an `entitlement` of `com.apple.security.personal-information.location`

- `photo_library`: an `entitlement` of `com.apple.security.personal-information.photos-library`

Platform quirks

Packaging with `--adhoc-sign`

Using the `--adhoc-sign` option on macOS produces an app that will be able to run on your own machine, but won't run on any other computer. In order to distribute your app to other users, you will need to sign the app with a full signing identity.

macOS Xcode project

Host Platform Support (*Platform support*)								
macOS		Windows			Linux			
x86-64	arm64	x86	x86-64	arm64	x86	x86-64	arm	arm64

Briefcase supports creating a full Xcode project for a macOS app. This project can then be used to build an actual app bundle, with the `briefcase build` command or directly from Xcode.

By default, apps will be both signed and notarized when they are packaged.

Packaging format

Briefcase supports two packaging formats for a macOS Xcode project:

1. A DMG that contains the `.app` bundle (the default output of `briefcase package macOS Xcode`, or by using `briefcase package macOS Xcode -p dmg`); or

2. A zipped `.app` folder (using `briefcase package macOS Xcode -p app`).

Icon format

macOS Xcode projects use `.png` format icons. An application must provide icons of the following sizes:

- 16px
- 32px
- 64px
- 128px
- 256px
- 512px
- 1024px

macOS Xcode projects do not support splash screens.

Additional options

The following options can be provided at the command line when packaging macOS apps.

`--no-notarize`

Do not submit the application for notarization. By default, apps will be submitted for notarization unless they have been signed with an ad-hoc signing identity.

Application configuration

The following options can be added to the `tool.briefcase.app.<appname>.macOS.Xcode` section of your `pyproject.toml` file.

entitlement

A property whose sub-attributes define keys that will be added to the app's `Entitlements.plist` file. Each entry will be converted into a key in the entitlements file. For example, specifying:

```
entitlement."com.apple.vm.networking" = true
```

will result in an `Entitlements.plist` declaration of:

```
<key>com.apple.vm.networking</key><true/>
```

Any Boolean or string value can be used for an entitlement value.

All macOS apps are automatically granted the following entitlements:

- `com.apple.security.cs.allow-unsigned-executable-memory`
- `com.apple.security.cs.disable-library-validation`

You can disable these default entitlements by defining them manually. For example, to enable library validation, you could add the following to your `pyproject.toml`:

```
entitlement."com.apple.security.cs.disable-library-validation" = false
```

info

A property whose sub-attributes define keys that will be added to the app's `Info.plist` file. Each entry will be converted into a key in the entitlements file. For example, specifying:

```
info."NSAppleScriptEnabled" = true
```

will result in an `Info.plist` declaration of:

```
<key>NSAppleScriptEnabled</key><true/>
```

Any Boolean or string value can be used for an `Info.plist` value.

universal_build

A Boolean, indicating whether Briefcase should build a universal app (i.e, an app that can target both x86_64 and ARM64). Defaults to `true`; if `false`, the binary will only be executable on the host platform on which it was built - i.e., if you build on an x86_64 machine, you will produce an x86_65 binary; if you build on an ARM64 machine, you will produce an ARM64 binary.

Permissions

Briefcase cross platform permissions map to a combination of `info` and `entitlement` keys:

- `microphone`: an `entitlement` of `com.apple.security.device.audio-input`

- `camera`: an `entitlement` of `com.apple.security.device.camera`

- `coarse_location`: an `info` entry for `NSLocationUsageDescription` (ignored if `background_location` or `fine_location` is defined); plus an entitlement of `com.apple.security.personal-information.location`

- `fine_location`: an `info` entry for `NSLocationUsageDescription``(ignored if ``background_location` is defined); plus an `entitlement` of `com.apple.security.personal-information.location`

- `background_location`: an `info` entry for `NSLocationUsageDescription`; plus an `entitlement` of `com.apple.security.personal-information.location`

- `photo_library`: an `entitlement` of `com.apple.security.personal-information.photos-library`

Platform quirks

Packaging with `--adhoc-sign`

Using the `--adhoc-sign` option on macOS produces an app that will be able to run on your own machine, but won't run on any other computer. In order to distribute your app to other users, you will need to sign the app with a full signing identity.

Windows

The default output format for Windows is an *app*.

Briefcase also supports creating a *Visual Studio project* which in turn can be used to build an app.

Both output formats support packaging as a Microsoft Software Installer (MSI) or as a ZIP file.

Windows App

Host Platform Support (*Platform support*)									
macOS		Windows			Linux				
x86-64	arm64	x86	x86-64	arm64	x86	x86-64	arm	arm64	

A Windows app is a stub binary, allow with a collection of folders that contain the Python code for the app and the Python runtime libraries.

Packaging format

Briefcase supports two packaging formats for a Windows app:

1. As an MSI installer (the default output of `briefcase package windows`, or by using `briefcase package windows -p msi`); or

2. As a ZIP file containing all files needed to run the app (by using `briefcase package windows -p zip`).

Briefcase uses the WiX Toolset to build an MSI installer for a Windows App. WiX, in turn, requires that .NET Framework 3.5 is enabled. To ensure .NET Framework 3.5 is enabled:

1. Open the Windows Control Panel

2. Traverse to Programs -> Programs and Features

3. Select "Turn Windows features On or Off"

4. Ensure that ".NET framework 3.5 (includes .NET 2.0 and 3.0)" is selected.

Icon format

Windows apps installers use multi-format `.ico` icons; these icons should contain images in the following sizes:

- 16px
- 32px
- 48px
- 64px
- 256px

Windows Apps do not support splash screens or installer images.

Additional options

The following options can be provided at the command line when packaging Windows apps.

`--file-digest <digest>`

The digest algorithm to use for code signing files in the project. Defaults to `sha256`.

`--use-local-machine-stores`

By default, the certificate for code signing is assumed to be in the Current User's certificate stores. Use this flag to indicate the certificate is in the Local Machine's certificate stores.

`--cert-store <store>`

The internal Windows name for the certificate store containing the certificate for code signing. Defaults to `My`.

Common Stores:

Personal	My
Intermediate Certification Authorities	CA
Third-Party Root Certification Authorities	AuthRoot
Trusted People	TrustedPeople
Trusted Publishers	TrustedPublisher
Trusted Root Certification Authorities	Root

`--timestamp-url <url>`

The URL of the Timestamp Authority server to timestamp the code signing. Defaults to `http://timestamp.digicert.com`.

`--timestamp-digest <url>`

The digest algorithm to request the Timestamp Authority server uses for the timestamp for code signing. Defaults to `sha256`.

Application configuration

The following options can be added to the `tool.briefcase.app.<appname>.windows` section of your `pyproject.toml` file.

`system_installer`

Controls whether the app will be installed as a per-user or per-machine app. Per-machine apps are "system" apps, and require admin permissions to run the installer; however, they are installed once and shared between all users on a computer.

If `true` the installer will attempt to install the app as a per-machine app, available to all users. If `false`, the installer will install as a per-user app. If undefined the installer will ask the user for their preference.

use_full_install_path

Controls whether the app will be installed using a path which includes both the application name *and* the company or developer's name. If `true` (the default), the app will be installed to `Program Files\<Author Name>\<Project Name>`. If `false`, it will be installed to `Program Files\<Project Name>`. Using the full path makes sense for larger companies with multiple applications, but less so for a solo developer.

version_triple

Python and Briefcase allow any valid PEP440 version number as a `version` specifier. However, MSI installers require a strict integer triple version number. Many PEP440-compliant version numbers, such as "1.2", "1.2.3b3", and "1.2.3.4", are invalid for MSI installers.

Briefcase will attempt to convert your `version` into a valid MSI value by extracting the first three parts of the main series version number (excluding pre, post and dev version indicators), padding with zeros if necessary:

- `1.2` becomes `1.2.0`

- `1.2b4` becomes `1.2.0`

- `1.2.3b3` becomes `1.2.3`

- `1.2.3.4` becomes `1.2.3`.

However, if you need to override this default value, you can define `version_triple` in your app settings. If provided, this value will be used in the MSI configuration file instead of the auto-generated value.

Platform quirks

Use caution with `--update-support`

Care should be taken when using the `--update-support` option to the `update`, `build` or `run` commands. Support packages in Windows apps are overlaid with app content, so it isn't possible to remove all old support files before installing new ones.

Briefcase will unpack the new support package without cleaning up existing support package content. This *should* work; however, ensure a reproducible release artefacts, it is advisable to perform a clean app build before release.

Packaging with `--adhoc-sign`

Using the `--adhoc-sign` option on Windows results in no signing being performed on the packaged app. This will result in your application being flagged as coming from an unverified publisher. This may limit who is able to install your app.

Visual Studio project

Host Platform Support (*Platform support*)								
macOS		Windows			Linux			
x86-64	arm64	x86	x86-64	arm64	x86	x86-64	arm	arm64

Briefcase supports creating a full Visual Studio project for a Windows App. This project can then be used to build the stub app binary with the `briefcase build` command, or directly from Visual Studio.

Building the Visual Studio project requires that you install Visual Studio 2022 or later. Visual Studio 2022 Community Edition can be downloaded for free from Microsoft. You can also use the Professional or Enterprise versions if you have them.

Briefcase will auto-detect the location of your Visual Studio installation, provided one of the following three things are true:

1. You install Visual Studio in the standard location in your Program Files folder.

2. `MSBuild.exe` is on your path.

3. You define the environment variable `MSBUILD` that points at the location of your `MSBuild.exe` executable.

When you install Visual Studio, there are many optional components. You should ensure that you have installed the following:

- .NET Desktop Development - All default packages

- Desktop Development with C++ - All default packages - C++/CLI support for v143 build tools

Packaging format

Briefcase supports two packaging formats for a Windows app:

1. As an MSI installer (the default output of `briefcase package windows VisualStudio`, or by using `briefcase package windows VisualStudio -p msi`); or

2. As a ZIP file containing all files needed to run the app (by using `briefcase package windows VisualStudio -p zip`).

Briefcase uses the WiX Toolset to build an MSI installer for a Windows App. WiX, in turn, requires that .NET Framework 3.5 is enabled. To ensure .NET Framework 3.5 is enabled:

1. Open the Windows Control Panel

2. Traverse to Programs -> Programs and Features

3. Select "Turn Windows features On or Off"

4. Ensure that ".NET framework 3.5 (includes .NET 2.0 and 3.0)" is selected.

Icon format

Windows apps installers use multi-format `.ico` icons; these icons should contain images in the following sizes:

- 16px

- 32px

- 48px

- 64px

- 256px

Windows Apps do not support splash screens or installer images.

Additional options

The following options can be provided at the command line when packaging Windows apps.

`--file-digest <digest>`

The digest algorithm to use for code signing files in the project. Defaults to `sha256`.

`--use-local-machine-stores`

By default, the certificate for code signing is assumed to be in the Current User's certificate stores. Use this flag to indicate the certificate is in the Local Machine's certificate stores.

`--cert-store <store>`

The internal Windows name for the certificate store containing the certificate for code signing. Defaults to `My`.

Common Stores:

Personal	My
Intermediate Certification Authorities	CA
Third-Party Root Certification Authorities	AuthRoot
Trusted People	TrustedPeople
Trusted Publishers	TrustedPublisher
Trusted Root Certification Authorities	Root

`--timestamp-url <url>`

The URL of the Timestamp Authority server to timestamp the code signing. Defaults to `http://timestamp.digicert.com`.

`--timestamp-digest <url>`

The digest algorithm to request the Timestamp Authority server uses for the timestamp for code signing. Defaults to `sha256`.

Application configuration

The following options can be added to the `tool.briefcase.app.<appname>.windows` section of your `pyproject.toml` file.

system_installer

Controls whether the app will be installed as a per-user or per-machine app. Per-machine apps are "system" apps, and require admin permissions to run the installer; however, they are installed once and shared between all users on a computer.

If `true` the installer will attempt to install the app as a per-machine app, available to all users. If `false`, the installer will install as a per-user app. If undefined the installer will ask the user for their preference.

use_full_install_path

Controls whether the app will be installed using a path which includes both the application name *and* the company or developer's name. If `true` (the default), the app will be installed to `Program Files\<Author Name>\<Project Name>`. If `false`, it will be installed to `Program Files\<Project Name>`. Using the full path makes sense for larger companies with multiple applications, but less so for a solo developer.

version_triple

Python and Briefcase allow any valid [PEP440 version number](#) as a `version` specifier. However, MSI installers require a strict integer triple version number. Many PEP440-compliant version numbers, such as "1.2", "1.2.3b3", and "1.2.3.4", are invalid for MSI installers.

Briefcase will attempt to convert your `version` into a valid MSI value by extracting the first three parts of the main series version number (excluding pre, post and dev version indicators), padding with zeros if necessary:

- 1.2 becomes `1.2.0`
- 1.2b4 becomes `1.2.0`
- 1.2.3b3 becomes `1.2.3`
- 1.2.3.4 becomes `1.2.3`.

However, if you need to override this default value, you can define `version_triple` in your app settings. If provided, this value will be used in the MSI configuration file instead of the auto-generated value.

Platform quirks

Use caution with `--update-support`

Care should be taken when using the `--update-support` option to the `update`, `build` or `run` commands. Support packages in Windows apps are overlaid with app content, so it isn't possible to remove all old support files before installing new ones.

Briefcase will unpack the new support package without cleaning up existing support package content. This *should* work; however, ensure a reproducible release artefacts, it is advisable to perform a clean app build before release.

Packaging with `--adhoc-sign`

Using the `--adhoc-sign` option on Windows results in no signing being performed on the packaged app. This will result in your application being flagged as coming from an unverified publisher. This may limit who can (or is willing to) install your app.

Linux

Briefcase supports packaging Linux apps as native system packages, as an AppImage, and in Flatpak format.

The default output format for Linux is *system packages*.

Native System Packages

Host Platform Support (*Platform support*)									
macOS		Windows			Linux				
x86-64	arm64	x86	x86-64	arm64	x86	x86-64	arm	arm64

All modern Linux distributions have a native format for distributing packages that are integrated into their overall operating system:

- `.deb`, used by Debian, Ubuntu, Mint (and others)
- `.rpm`, used by Red Hat, Fedora, CentOS, AlmaLinux, openSUSE (and others)
- `.pkg.tar.zst`, used by Arch Linux and Manjaro Linux

The Briefcase `system` backend provides a way to build your app in these system package formats.

Not all Linux distributions are supported!

Briefcase cannot reliably identify *every* Linux vendor. If your Linux distribution isn't being identified (or isn't being identified correctly), please open a ticket with the contents of your `/etc/os-release` file.

The packaged app includes a stub binary, so that the app will appear in process lists using your app's name. It also includes a FreeDesktop registration so the app will appear in system menus.

When installed from a Briefcase-produced system package, the app will use the system Python install, and the standard library provided by the system. However, the app will be isolated from any packages that have been installed at a system level.

As the app uses the system Python, system packages are highly dependent on the distribution version. It is therefore necessary to build a different system package for every distribution you want to target. To help simplify this process, Briefcase uses Docker to provide build environments. Using these Docker environments, it is possible to build a system package for any target distribution and version, regardless of the host distribution and version - that is, you can build a Debian Buster package on an Ubuntu 20.04 machine, or an Ubuntu 22.04 package on a RHEL8 machine.

The usage of the system Python also means that system packages are different from most other Briefcase-packaged apps. On other target platforms (macOS and Windows apps, Linux AppImage, etc), the version of Python used to run Briefcase will be the version of Python used by the bundled app. However, when building a system package, Briefcase will use the operating system's Python3 installation for system packages, regardless of the host Python version. This means you will need to perform additional platform testing to ensure that your app is compatible with that version of Python.

Icon format

Deb packages uses `.png` format icons. An application must provide icons in the following sizes:

- 16px
- 32px
- 64px
- 128px
- 256px
- 512px

Linux System packages do not support splash screens or installer images.

Additional files

The Linux system app template includes a `LICENSE` and `CHANGELOG` file, with stub content. When the application is generated from template, Briefcase will look in the project root folder (i.e., the folder that contains your `pyproject.toml`) for files with the same name. If these files are found, they will be copied into your project. You should ensure these files are complete and correct before publishing your app.

The Linux system app template also includes an initial draft manfile for your app. This manfile will be populated with the `description` and `long_description` of your app. You may wish to add more details on app usage.

Additional options

The following options can be provided at the command line when producing Deb packages:

`--target`

A Docker base image identifier for the Linux distribution you want to target. The identifier will be in the pattern `<vendor>:<codename>` (e.g., `debian:buster` or `ubuntu:jammy`). You can also use the version number in place of the code name (e.g., `debian:10`, `ubuntu:22.04`, or `fedora:37`). Whichever form you choose, you should be consistent; no normalization of code name and version is performed, so `ubuntu:jammy` and `ubuntu:22.04` will be identified as different versions (even though they the same version).

You can specify any identifier you want, provided the distribution is still supported by the vendor, and system Python is Python 3.8 or later.

The following Linux vendors are known to work as Docker targets:

- Debian (e.g., `debian:bookworm` or `debian:12`)
- Ubuntu (e.g., `ubuntu:jammy` or `ubuntu:22.04`)
- Fedora (e.g, `fedora:39`)
- AlmaLinux (e.g., `almalinux:9`)
- Red Hat Enterprise Linux (e.g., `redhat/ubi9:9`)
- openSUSE Tumbleweed (e.g., `"opensuse/tumbleweed:latest"`)
- Arch Linux (e.g., `archlinux:latest`)
- Manjaro Linux (e.g., `manjarolinux/base:latest`)

`--Xdocker-build=<value>`

A configuration argument to be passed to the Docker build command for the app image. For example, to provide an additional build argument to the Dockerfile, specify `--Xdocker-build=--build-arg=ARG=value`. See the Docker build documentation for details on the full list of options that can be provided.

You may specify multiple `--Xdocker-build` arguments; each one specifies a single argument to pass to Docker, in the order they are specified.

Application configuration

The following options can be added to the `tool.briefcase.app.<appname>.linux.system` section of your `pyproject.toml` file; if defined in this section, the values will apply for *all* Linux distributions for which you build packages.

If you need to override these settings for a specific target vendor, or for a specific distribution version, you can provide increasingly specific sections for vendor and version information. Each distribution is identified by:

- Vendor base (e.g., `debian`, `rhel`, `arch`, `suse`)
- Vendor (e.g, `debian`, `ubuntu`, `rhel`, `fedora`, `opensuse-tumbleweed`, `arch`, `manjaro`). The vendor identifier *may* be the same as the vendor base (e.g, in the case of Debian, Red Hat, or Arch)
- Code name (e.g., a version number, or `jammy`).

For example, a full configuration for `myapp` running on Ubuntu 22.04 (jammy) would consist of the following sections:

- `tool.briefcase.app.myapp` providing global configuration options
- `tool.briefcase.app.myapp.linux` providing definitions common to *all* Linux packaging backends
- `tool.briefcase.app.myapp.linux.system` providing definitions for all Linux system package targets

- `tool.briefcase.app.myapp.linux.system.debian` providing definitions common to all Debian-based packaging targets

- `tool.briefcase.app.myapp.linux.system.ubuntu` providing definitions common to all Ubuntu-based packaging targets

- `tool.briefcase.app.myapp.linux.system.ubuntu.jammy` providing definitions specific to for Ubuntu 22.04 (Jammy).

These configurations will be merged at runtime; any version-specific definitions will override the generic vendor definitions; any vendor definitions will override the vendor-base definitions; and any vendor-base definitions will override generic system package definitions.

system_requires

A list of operating system packages that must be installed for the system package build to succeed. If a Docker build is requested, this list will be passed to the Docker context when building the container for the app build. These entries should be the format the target Linux distribution will accept. For example, if you're using a Debian-derived distribution, you might use:

```
system_requires = ["libgirepository1.0-dev", "libcairo2-dev"]
```

to make the GTK GI and Cairo operating system development packages available to your app. However, if you're on a RedHat-derived distribution, you would use:

```
system_requires = ["gobject-introspection-devel", "python3-cairo-devel"]
```

If you see errors during `briefcase build` of the form:

```
Could not find dependency: libSomething.so.1
```

but the app works under `briefcase dev`, the problem may be an incomplete `system_requires` definition. The `briefcase build` process generates a new environment that is completely isolated from your development environment, so if your app has any operating system dependencies, they *must* be listed in your `system_requires` definition.

`system_requires` are the packages required at *build* time. To specify *runtime* system requirements, use the `system_runtime_requires` setting.

system_runtime_requires

A list of system packages that your app requires at *runtime*. These will be closely related to the `system_requires` setting, but will likely be different; most notably, you will probably need `-dev` packages at build time, but non `-dev` packages at runtime.

`system_runtime_requires` should be specified as system package requirements; they can optionally include version pins. Briefcase will automatically include the dependencies needed for Python. For example:

```
system_runtime_requires = ["libgtk-3-0 (>=3.14)", "libwebkit2gtk-4.1-0"]
```

will specify that your app needs Python 3, a version of `libgtk` >= `3.14`, and any version of `libwebkit2gtk`.

Any problems with installing or running your system package likely indicate an issue with your `system_runtime_requires` definition.

system_section

When an application is published as a `.deb` file, Debian requires that you specify a "section", describing a classification of the application area. The template will provide a default section of `utils`; if you want to override that default, you can specify a value for `system_section`. For details on the allowed values for `system_section`, refer to the Debian Policy Manual.

dockerfile_extra_content

Any additional Docker instructions that are required to configure the container used to build your Python app. For example, any dependencies that cannot be configured with `apt-get` could be installed. `dockerfile_extra_content` is string literal that will be added verbatim to the end of the project Dockerfile.

Any Dockerfile instructions added by `dockerfile_extra_content` will be executed as the `brutus` user, rather than the `root` user. If you need to perform container setup operations as `root`, switch the container's user to `root`, perform whatever operations are required, then switch back to the `brutus` user - e.g.:

```
dockerfile_extra_content = """
RUN <first command run as brutus>

USER root
RUN <second command run as root>

USER brutus
"""
```

AppImage

Host Platform Support (*Platform support*)									
macOS		Windows			Linux				
x86-64	arm64	x86	x86-64	arm64	x86	x86-64	arm	arm64

Best effort support

AppImage has a number of significant issues building images for GUI apps. It is incompatible with the use of binary wheels, and the use of an older Linux base image for compatibility purposes is incompatible with using modern GUI frameworks. Even when a GUI toolkit *can* be installed, the AppImage packaging process frequently introduces bugs related to DBus integration or libraries like WebKit2 that use subprocesses.

Briefcase provides an AppImage backend for historical reasons, but we strongly discourage the use of AppImages for distribution. We maintain unit test coverage for the AppImage backend, but we do not build AppImages as part of our release process. We will accept bug reports related to AppImage support, and we will merge PRs that address AppImage support, but the core team does not consider addressing AppImage bugs a priority.

If you need to distribute a Linux app, *System packages* or *Flatpaks* are much more reliable options.

AppImage provides a way for developers to provide "native" binaries for Linux users. It allow packaging applications for any common Linux based operating system, including Ubuntu, Debian, Fedora, and more. AppImages contain all

the dependencies that cannot be assumed to be part of each target system, and will run on most Linux distributions without further modifications. Briefcase uses Linuxdeploy to build the AppImage in the correct format.

Packaging binaries for Linux is complicated, because of the inconsistent library versions present on each distribution. An AppImage can be executed on *any* Linux distribution with a version of `libc` greater than or equal the version of the distribution where the AppImage was created.

To ensure that an application is built in an environment that is as compatible as possible, Briefcase builds AppImages inside Docker. The Docker base image used by Briefcase can be configured to any manylinux base using the `manylinux` application configuration option; if `manylinux` isn't specified, it falls back to an Ubuntu 18.04 base image. While it is *possible* to build AppImages without Docker, it is highly recommended that you do not, as the resulting AppImages will not be as portable as they could otherwise be.

Note: AppImage works by attempting to autodetect all the libraries that an application requires, copying those libraries into a distribution, and manipulating them to reflect their new locations. This approach *can* work well... but it is also prone to major problems. Python apps (which load their dependencies dynamically) are particularly prone to exposing those flaws.

Briefcase makes a best-effort attempt to use the AppImage tools to build a binary, but sometimes, the problem lies with AppImage itself. If you have problems with AppImage binaries, you should first check whether the problem is a limitation with AppImage.

Icon format

AppImages use `.png` format icons. An application must provide icons in the following sizes:

- 16px
- 32px
- 64px
- 128px
- 256px
- 512px

AppImages do not support splash screens or installer images.

Additional options

The following options can be provided at the command line when producing AppImages.

`--no-docker`

Use native execution, rather than using Docker to start a container.

`--Xdocker-build=<value>`

A configuration argument to be passed to the Docker build command for the app image. For example, to provide an additional build argument to the Dockerfile, specify `--Xdocker-build=--build-arg=ARG=value`. See the Docker build documentation for details on the full list of options that can be provided.

You may specify multiple `--Xdocker-build` arguments; each one specifies a single argument to pass to Docker, in the order they are specified.

Application configuration

The following options can be added to the `tool.briefcase.app.<appname>.linux.appimage` section of your `pyproject.toml` file.

manylinux

The manylinux tag to use as a base image when building the AppImage. Should be one of:

- `manylinux1`
- `manylinux2010`
- `manylinux2014`
- `manylinux2_24`
- `manylinux2_28`

New projects will default to `manylinux2014`. If an application doesn't specify a `manylinux` value, `ubuntu:18.04` will be used as the base image.

manylinux_image_tag

The specific tag of the `manylinux` image to use. Defaults to `latest`.

system_requires

A list of operating system packages that must be installed for the AppImage build to succeed. If a Docker build is requested, this list will be passed to the Docker context when building the container for the app build. By default, entries should be Ubuntu 18.04 `apt` package requirements. For example:

```
system_requires = ['libgirepository1.0-dev', 'libcairo2-dev']
```

would make the GTK GI and Cairo operating system libraries available to your app.

If you see errors during `briefcase build` of the form:

```
Could not find dependency: libSomething.so.1
```

but the app works under `briefcase dev`, the problem may be an incomplete `system_requires` definition. The `briefcase build` process generates a new environment that is completely isolated from your development environment, so if your app has any operating system dependencies, they *must* be listed in your `system_requires` definition.

linuxdeploy_plugins

A list of linuxdeploy plugins that you wish to be included when building the AppImage. This is needed for applications that depend on libraries that have dependencies that cannot be automatically discovered by Linuxdeploy. GTK and Qt both have complex runtime resource requirements that can be difficult for Linuxdeploy to identify automatically.

The `linuxdeploy_plugins` declaration is a list of strings. Briefcase can take plugin definitions in three formats:

1. The name of a plugin known by Briefcase. One of `gtk` or `qt`.

2. A URL where a plugin can be downloaded

3. A path to a local plugin file

If your plugin requires an environment variable for configuration, that environment variable can be provided as a prefix to the plugin declaration, similar to how environment variables can be defined for a shell command.

For example, the `gtk` plugin requires the `DEPLOY_GTK_VERSION` environment variable. To set this variable with the Briefcase-managed GTK Linuxdeploy plugin, you would define:

```
linuxdeploy_plugins = ["DEPLOY_GTK_VERSION=3 gtk"]
```

Or, if you were using a plugin stored as a local file:

```
linuxdeploy_plugins = ["DEPLOY_GTK_VERSION=3 path/to/plugins/linuxdeploy-gtk-plugin.sh"]
```

dockerfile_extra_content

Any additional Docker instructions that are required to configure the container used to build your Python app. For example, any dependencies that cannot be configured with `apt-get` could be installed. `dockerfile_extra_content` is string literal that will be added verbatim to the end of the project Dockerfile.

Any Dockerfile instructions added by `dockerfile_extra_content` will be executed as the `brutus` user, rather than the `root` user. If you need to perform container setup operations as `root`, switch the container's user to `root`, perform whatever operations are required, then switch back to the `brutus` user - e.g.:

```
dockerfile_extra_content = """
RUN <first command run as brutus>

USER root
RUN <second command run as root>

USER brutus
"""
```

Platform quirks

Use caution with --update-support

Care should be taken when using the `--update-support` option to the `update`, `build` or `run` commands. Support packages in Linux AppImages are overlaid with app content, so it isn't possible to remove all old support files before installing new ones.

Briefcase will unpack the new support package without cleaning up existing support package content. This *should* work; however, ensure a reproducible release artefacts, it is advisable to perform a clean app build before release.

Apps using WebKit2 are not supported

WebKit2, the library that provides web widget support, can't currently be deployed with AppImage. WebKit2 uses subprocesses to manage network and rendering requests, but the way it packages and launches these subprocesses isn't currently compatible with AppImage.

In addition, many of the commonly used `manylinux` base images predate the release of WebKit2. As a result, system packages providing WebKit2 are not available on these base images. `manylinux_2_28` is the earliest supported `manylinux` image that provides WebKit2 support.

Runtime issues with AppImages

Packaging on Linux is a difficult problem - especially when it comes to binary libraries. The following are some common problems you may see, and ways that they can be mitigated.

Failure to load `libpango-1.0-so.0`

Older Linux distributions (e.g., Ubuntu 18.04) may not be compatible with AppImages of Toga apps produced by Briefcase, complaining about problems with `libpango-1.0.so.0` and an undefined symbols (`fribidi_get_par_embedding_levels_ex` is a common missing symbol to be reported). This is caused because the version of `fribidi` provided by these distributions. Unfortunately, there's no way to fix this limitation.

Undefined symbol and Namespace not available errors

If you get the error:

```
ValueError: Namespace Something not available
```

or:

```
ImportError: /usr/lib/libSomething.so.0: undefined symbol: some_symbol
```

it is likely that one or more of the libraries you are using in your app requires a Linuxdeploy plugin. GUI libraries, or libraries that do dynamic module loading are particularly prone to this problem.

ELF load command address/offset not properly aligned

Briefcase uses a tool named Linuxdeploy to build AppImages. Linuxdeploy processes all the libraries used by an app so that they can be relocated into the final packaged binary. Building a `manylinux` binary wheel involves a tool named `auditwheel` that performs a very similar process. Unfortunately, processing a binary with Linuxdeploy after it has been processed by `auditwheel` can result in a binary library that cannot be loaded at runtime.

This is particularly common when a module installed as a binary wheel has a dependency on external libraries. For example, Pillow is a Python library that contains a binary submodule; that submodule uses `libpng`, `libtiff`, and other system libraries for image manipulation. If you install Pillow from a `manylinux` wheel, you may see an error similar to the following at runtime:

```
Traceback (most recent call last):
File "/tmp/.mount_TestbewwDi98/usr/app/testbed/app.py", line 54, in main
  test()
```

```
File "/tmp/.mount_TestbewwDi98/usr/app/testbed/linux.py", line 94, in test_pillow
    from PIL import Image
File "/tmp/.mount_TestbewwDi98/usr/app_packages/PIL/Image.py", line 132, in <module>
    from . import _imaging as core
ImportError: libtiff-d0580107.so.5.7.0: ELF load command address/offset not properly␣
→aligned
```

This indicates that one of the libraries that has been included in the AppImage has become corrupted as a result of double processing.

The solution is to ask Briefcase to install the affected library from source. This can be done by adding a `"--no-binary"` entry to the `requires` declaration for your app. For example, if your app includes Pillow as a requirement:

```
requires = ["pillow==9.1.0"]
```

You can force Briefcase to install Pillow from source by adding:

```
requires = [
    "pillow==9.1.0",
    "--no-binary", "pillow",
]
```

Since the library will be installed from source, you also need to add any system requirements that are needed to compile the binary library. For example, Pillow requires the development libraries for the various image formats that it uses:

```
system_requires = [
    ... other system requirements ...
    "libjpeg-dev",
    "libpng-dev",
    "libtiff-dev",
]
```

If you are missing a system requirement, the call to `briefcase build` will fail with an error:

```
error: subprocess-exited-with-error

× pip subprocess to install build dependencies did not run successfully.
│ exit code: 1
╰─> See above for output.

note: This error originates from a subprocess, and is likely not a problem with pip.
>>> Return code: 1

Unable to install requirements. This may be because one of your
requirements is invalid, or because pip was unable to connect
to the PyPI server.
```

You must add a separate `--no-binary` option for every binary library you want to install from source. For example, if your app also includes the `cryptography` library, and you want to install that library from source, you would add:

```
requires = [
    "pillow==9.1.0",
    "cryptography==37.0.2",
    "--no-binary", "pillow",
```

```
    "--no-binary", "cryptography",
]
```

If you want to force *all* packages to be installed from source, you can add a single `:all` declaration:

```
requires = [
    "pillow==9.1.0",
    "cryptography==37.0.2",
    "--no-binary", ":all:",
]
```

The `--no-binary` declaration doesn't need to be added to the same `requires` declaration that defines the requirement. For example, if you have a library that is used on all platforms, the declaration will probably be in the top-level `requires`, not the platform-specific `requires`. If you add `--no-binary` in the top-level requires, the use of a binary wheel would be prevented on *all* platforms. To avoid this, you can add the requirement in the top-level requires, but add the `--no-binary` declaration to the Linux-specific requirements:

```
[tool.briefcase.app.helloworld]
formal_name = "Hello World"
...
requires = [
    "pillow",
]

[tool.briefcase.app.helloworld.linux]
requires = [
    "--no-binary", "pillow"
]
```

Flatpak

Host Platform Support (*Platform support*)								
macOS		Windows			Linux			
x86-64	arm64	x86	x86-64	arm64	x86	x86-64	arm	arm64

Flatpak provides a way for developers to distribute apps to Linux users in a format that is independent of the specific distribution used by the end-user. It allow packaging applications for use on any common Linux distribution, including Ubuntu, Debian, Fedora, and more. There are some system packages needed to run and build Flatpaks; see the Flatpak setup guide for more details.

A Flatpak app is built by compiling against a `runtime`. Runtimes provide the basic dependencies that are used by applications. Each application must be built against a runtime, and this runtime must be installed on a host system in order for the application to run (Flatpak can automatically install the runtime required by an application).

The end user will install the Flatpak into their local app repository; this can be done by installing directly from a single file `.flatpak` bundle, or by installing from a package repository like Flathub. Apps can be installed into user-space, or if the user has sufficient privileges, they can be installed into a system-wide app repository.

Briefcase currently supports creating `.flatpak` single file bundles; end users can install the app bundle by running:

```
$ flatpak install --user App_Name-1.2.3-x86_64.flatpak
```

substituting the name of the flatpak file as appropriate. The `--user` option can be omitted if the user wants to install the app system-wide.

The app can then be run with:

```
$ flatpak run com.example.appname
```

specifying the app bundle identifier as appropriate.

Briefcase *can* be published to Flathub or another Flatpak repository; but Briefcase does not currently support automated publication of apps.

Icon format

Flatpak uses `.png` format icons. An application must provide icons in the following sizes:

- 16px
- 32px
- 64px
- 128px
- 256px
- 512px

Flatpaks do not support splash screens or installer images.

Application configuration

The following options can be added to the `tool.briefcase.app.<appname>.linux.flatpak` section of your `pyproject.toml` file:

finish_arg

The arguments used to configure the Flatpak sandbox. `finish_arg` is an attribute that can have additional sub-attributes; each sub-attribute maps to a single property that will be added to the app's manifest. For example, to add `--allow=bluetooth` as a finish argument, you would specify:

```
finish_arg."allow=bluetooth" = True
```

Briefcase adds the following finish arguments by default:

- `share=ipc`
- `socket=x11`
- `nosocket=wayland`
- `share=network`
- `device=dri`
- `socket=pulseaudio`

- `filesystem=xdg-cache`

- `filesystem=xdg-config`

- `filesystem=xdg-data`

- `filesystem=xdg-documents`

- `socket=session-bus`

These can be disabled by explicitly setting their value to `False`; for example, to disable audio access, you would specify:

```
finish_arg."socket=pulseaudio" = false
```

flatpak_runtime_repo_alias

An alias to use when registering the Flatpak repository that will store the Flatpak runtime used to build the app. By default, Briefcase will use Flathub as its runtime repository, with an alias of `flathub`.

flatpak_runtime_repo_url

The repository URL hosting the runtime and SDK package that the Flatpak will use. By default, Briefcase will use Flathub as its runtime repository.

flatpak_runtime

A string, identifying the runtime to use as a base for the Flatpak app.

The Flatpak runtime and SDK are paired; so, both a `flatpak_runtime` and a corresponding `flatpak_sdk` must be defined.

flatpak_runtime_version

A string, identifying the version of the Flatpak runtime that should be used.

flatpak_sdk

A string, identifying the SDK associated with the platform that will be used to build the Flatpak app.

The Flatpak runtime and SDK are paired; so, both a `flatpak_runtime` and a corresponding `flatpak_sdk` must be defined.

modules_extra_content

Additional build instructions that will be inserted into the Flatpak manifest, *after* Python has been installed and `pip` is guaranteed to exist, but *before* any app code or app packages have been installed into the Flatpak.

Permissions

Permissions are not used for Flatpak packaging.

Compilation issues with Flatpak

Flatpak works by building a sandbox in which to compile the application bundle. This sandbox uses some low-level kernel and file system operations to provide the sandboxing behavior. As a result, Flatpaks cannot be built inside a Docker container, and they cannot be build on an NFS mounted drive.

If you get errors about `renameat` when building an app, similar to the following:

```
[helloworld] Building Flatpak...
Downloading sources
Initializing build dir
Committing stage init to cache
Error: Writing metadata object: renameat: Operation not permitted
Building...

Error while building app helloworld.

Log saved to ...
```

you may be building on an NFS drive. Move your project to local storage, and retry the build.

iOS

Briefcase supports packaging iOS apps using an *Xcode project*.

iOS Xcode project

Host Platform Support (*Platform support*)								
macOS		Windows			Linux			
x86-64	arm64	x86	x86-64	arm64	x86	x86-64	arm	arm64

When generating an iOS project, Briefcase produces an Xcode project.

Icon format

iOS projects use `.png` format icons. An application must provide icons of the following sizes:

- 20px
- 29px
- 40px
- 58px
- 60px

- 76px

- 80px

- 87px

- 120px

- 152px

- 167px

- 180px

- 640px

- 1024px

- 1280px

- 1920px

The icon will also be used to populate the splash screen. You can specify a background color for the splash screen using the `splash_background_color` configuration setting.

iOS projects do not support installer images.

Colors

iOS allows for some customization of the colors used by your app:

- `splash_background_color` is the color of the splash background that displays while an app is loading.

Additional options

The following options can be provided at the command line when producing iOS projects:

run

-d <device> / --device <device>

The device simulator to target. Can be either a UDID, a device name (e.g., `"iPhone 11"`), or a device name and OS version (`"iPhone 11::iOS 13.3"`).

Application configuration

The following options can be added to the `tool.briefcase.app.<appname>.iOS.app` section of your `pyproject.toml` file.

info

A property whose sub-attributes define keys that will be added to the app's `Info.plist` file. Each entry will be converted into a key in the entitlements file. For example, specifying:

```
info."UIFileSharingEnabled" = true
```

will result in an `Info.plist` declaration of:

```
<key>UIFileSharingEnabled</key><true/>
```

Any Boolean or string value can be used for an `Info.plist` value.

Permissions

Briefcase cross platform permissions map to the following `info` keys:

- `camera`: `NSCameraUsageDescription`

- `microphone`: `NSMicrophoneUsageDescription`

- `coarse_location` - `NSLocationDefaultAccuracyReduced=True` - `NSLocationWhenInUseUsageDescription` if `fine_location` is not defined

- `fine_location` - `NSLocationDefaultAccuracyReduced=False` - `NSLocationWhenInUseUsageDescription`

- `background_location`: - `NSLocationAlwaysAndWhenInUseUsageDescription` - `NSLocationWhenInUseUsageDescription` if neither `fine_location` or `coarse_location` is set - `UIBackgroundModes` will include `location` and `processing`

- `photo_library`: `NSPhotoLibraryAddUsageDescription`

Platform quirks

Availability of third-party packages

Briefcase is able to use third-party packages in iOS apps. As long as the package is available on PyPI, or you can provide a wheel file for the package, it can be added to the `requires` declaration in your `pyproject.toml` file and used by your app at runtime.

If the package is pure Python (i.e., it does not contain a binary library), that's all you need to do. To check whether a package is pure Python, look at the PyPI downloads page for the project; if the wheels provided are have a `-py3-none-any.whl` suffix, then they are pure Python wheels. If the wheels have version and platform-specific extensions (e.g., `-cp311-cp311-macosx_11_0_universal2.whl`), then the wheel contains a binary component.

If the package contains a binary component, that wheel needs to be compiled for iOS. PyPI does not currently support uploading iOS-compatible wheels, so you can't rely on PyPI to provide those wheels. Briefcase uses a secondary repository to store pre-compiled iOS wheels.

This repository is maintained by the BeeWare project, and as a result, it does not have binary wheels for *every* package that is available on PyPI, or even every *version* of every package that is on PyPI. If you see any of the following messages when building an app for a mobile platform, then the package (or this version of it) probably isn't supported yet:

- The error "Cannot compile native modules"

- A reference to downloading a `.tar.gz` version of the package

- A reference to `Building wheels for collected packages:  <package>`

It is *usually* possible to compile any binary package wheels for iOS, depending on the requirements of the package itself. If the package has a dependency on other binary libraries (e.g., something like `libjpeg` that isn't written in Python), those libraries will need to be compiled for iOS as well. However, if the library requires build tools that don't support iOS, such as a compiler that can't target iOS, or a PEP517 build system that doesn't support cross-compilation, it may not be possible to build an iOS wheel.

The BeeWare Project provides the Mobile Forge project to assist with cross-compiling iOS binary wheels. This repository contains recipes for building the packages that are stored in the secondary package repository. Contributions of new package recipes are welcome, and can be submitted as pull requests. Or, if you have a particular package that you'd like us to support, please visit the issue tracker and provide details about that package.

Android

Briefcase supports packaging Android apps using a *Gradle project*.

Gradle project

Host Platform Support (*Platform support*)								
macOS		Windows			Linux			
x86-64	arm64	x86	x86-64	arm64	x86	x86-64	arm	arm64

When generating an Android project, Briefcase produces a Gradle project.

Environment

Gradle requires an install of a Java 17 JDK and the Android SDK.

If the methods below fail to find an Android SDK or Java JDK, Briefcase will download and install an isolated copy in its data directory.

Java JDK

If you have an existing install of a Java 17 JDK, it will be used by Briefcase if the `JAVA_HOME` environment variable is set. On macOS, if `JAVA_HOME` is not set, Briefcase will use the `/usr/libexec/java_home` tool to find an existing JDK install.

Android SDK

If you have an existing install of the Android SDK, it will be used by Briefcase if the `ANDROID_HOME` environment variable is set. If `ANDROID_HOME` is not present in the environment, Briefcase will honor the deprecated `ANDROID_SDK_ROOT` environment variable. Additionally, an existing SDK install must have version 9.0 of Command-line Tools installed; this version can be installed in the SDK Manager in Android Studio.

Packaging format

Briefcase supports three packaging formats for an Android app:

1. An AAB bundle (the default output of `briefcase package android`, or by using `briefcase package android -p aab`); or

2. A Release APK (by using `briefcase package android -p apk`); or

3. A Debug APK (by using `briefcase package android -p debug-apk`).

Icon format

Android projects use `.png` format icons, in round, square and adaptive variants. An application must provide the icons in the following sizes, for 3 variants:

- `round`:
    - 48px (`mdpi`; baseline resolution)
    - 72px (`hdpi`; 1.5x scale)
    - 96px (`xhdpi`; 2x scale)
    - 144px (`xxhdpi`; 3x scale)
    - 192px (`xxxhdpi`; 4x scale)
- `square`:
    - 48px (`mdpi`; baseline resolution)
    - 72px (`hdpi`; 1.5x scale)
    - 96px (`xhdpi`; 2x scale)
    - 144px (`xxhdpi`; 3x scale)
    - 192px (`xxxhdpi`; 4x scale)
    - 320px (`mdpi`; baseline resolution for splash screen)
    - 480px (`hdpi`; 1.5x scale for splash screen)
    - 640px (`xhdpi`; 2x scale for splash screen)
    - 960px (`xxhdpi`; 3x scale for splash screen)
    - 1280px (`xxxhdpi`; 4x scale for splash screen)
- `adaptive`:
    - 108px (`mdpi`; baseline resolution; 66px drawable area)
    - 162px (`hdpi`; 1.5x scale; 99px drawable area)
    - 216px (`xhdpi`; 2x scale; 132px drawable area)
    - 324px (`xxhdpi`; 3x scale; 198px drawable area)
    - 432px (`xxxhdpi`; 4x scale; 264px drawable area)

The `round` and `square` icons should include their background color in the image. The `adaptive` icons should have a transparent background; the icon image should be centered in the overall image, and should not exceed the drawable area. The background color of the adaptive icon will be the value specified with `splash_background_color`.

The icon will also be used to populate the splash screen. You can specify a background color for the splash screen using the `splash_background_color` configuration setting.

Android projects do not support installer images.

Colors

Android allows for some customization of the colors used by your app:

- `base_theme` is used to set the base Android theme.
- `accent_color` is used as a subtle highlight throughout your app to call attention to key elements. It's used on things like form labels and inputs.
- `primary_color` is the main branding color of the app and is used to color the app bar in the main window.
- `primary_color_dark` is used alongside the primary color to color the status bar at the top of the screen.
- `splash_background_color` is the color of the splash background that displays while an app is loading.

Additional options

The following options can be provided at the command line when producing Android projects:

run

-d <device> / --device <device>

The device or emulator to target. Can be specified as:

- @ followed by an AVD name (e.g., `@beePhone`); or
- a device ID (a hexadecimal identifier associated with a specific hardware device); or
- a JSON dictionary specifying the properties of a device that will be created. This dictionary must have, at a minimum, an AVD name:

```
$ briefcase run -d '{"avd":"new-device"}'

You may also specify:

- ``device_type`` (e.g., ``pixel``) - the type of device to emulate
- ``skin`` (e.g., ``pixel_3a``) - the skin to apply to the emulator
- ``system_image`` (e.g., ``system-images;android-31;default;arm64-v8a``) - the Android
  system image to use in the emulator.

If any of these attributes are *not* specified, they will fall back
to reasonable defaults.
```

--Xemulator=<value>

A configuration argument to be passed to the emulator on startup. For example, to start the emulator in "headless" mode (i.e., without a display window), specify `--Xemulator=-no-window`. See the Android documentation for details on the full list of options that can be provided.

You may specify multiple `--Xemulator` arguments; each one specifies a single argument to pass to the emulator, in the order they are specified.

--shutdown-on-exit

Instruct Briefcase to shut down the emulator when the run finishes. This is especially useful if you are running in headless mode, as the emulator will continue to run in the background, but there will be no visual manifestation that it is running. It may also be useful as a cleanup mechanism when running in a CI configuration.

Application configuration

The following options can be added to the `tool.briefcase.app.<appname>.android` section of your `pyproject.toml` file.

android_manifest_attrs_extra_content

Additional attributes that will be added verbatim to the `<manifest>` declaration of the `AndroidManifest.xml` of your app.

android_manifest_extra_content

Additional content that will be added verbatim just before the closing `</manifest>` declaration of the `AndroidManifest.xml` of your app.

android_manifest_application_attrs_extra_content

Additional attributes that will be added verbatim to the `<application>` declaration of the `AndroidManifest.xml` of your app.

android_manifest_application_extra_content

Additional content that will be added verbatim just before the closing `</application>` declaration of the `AndroidManifest.xml` of your app.

android_manifest_activity_attrs_extra_content

Additional attributes that will be added verbatim to the `<activity>` declaration of the `AndroidManifest.xml` of your app.

android_manifest_activity_extra_content

Additional content that will be added verbatim just before the closing `</activity>` declaration of the `AndroidManifest.xml` of your app.

base_theme

The base theme for the application. Defaults to `Theme.AppCompat.Light.DarkActionBar`

build_gradle_dependencies

The list of libraries that should be linked into the Android application. Each library should be a versioned Maven package specifier. If unspecified, three libraries will be linked into the app:

- `androidx.appcompat:appcompat:1.0.2`
- `androidx.constraintlayout:constraintlayout:1.1.3`
- `androidx.swiperefreshlayout:swiperefreshlayout:1.1.0`

build_gradle_extra_content

A string providing additional Gradle settings to use when building your app. This will be added verbatim to the end of your `app/build.gradle` file.

feature

A property whose sub-properties define the features that will be marked as required by the final app. Each entry will be converted into a `<uses-feature>` declaration in your app's `AndroidManifest.xml`, with the feature name matching the name of the sub-attribute.

For example, specifying:

```
feature."android.hardware.bluetooth" = true
```

will result in an `AndroidManifest.xml` declaration of:

```
<uses-feature android:name="android.hardware.bluetooth" android:required="true">
```

The use of some cross-platform permissions will imply the addition of features; see *the discussion on Android permissions* for more details.

permission

A property whose sub-properties define the platform-specific permissions that will be marked as required by the final app. Each entry will be converted into a `<uses-permission>` declaration in your app's `AndroidManifest.xml`, with the feature name matching the name of the sub-attribute.

For example, specifying:

```
permission."android.permission.HIGH_SAMPLING_RATE_SENSORS" = true
```

will result in an `AndroidManifest.xml` declaration of:

```
<uses-permission android:name="android.permission.HIGH_SAMPLING_RATE_SENSORS">
```

version_code

In addition to a version number, Android projects require a version "code". This code is an integer version of your version number that *must* increase with every new release pushed to the Play Store.

Briefcase will attempt to generate a version code by combining the version number with the build number. It does this by using each part of the main version number (padded to 3 digits if necessary) and the build number as 2 significant digits of the final version code:

- Version `1.0`, build 1 becomes `1000001` (i.e, 1, `00`, `00`, `01`)
- Version `1.2`, build 37 becomes `1020037` (i.e., 1, `02`, `00`, 37)
- Version `1.2.37`, build 42 becomes `1023742` (i.e, 1, `02`, 37, 42)
- Version `2020.6`, build 4 becomes `2020060004` (i.e., `2020`, `06`, `00`, `04`)

If you want to manually specify a version code by defining `version_code` in your application configuration. If provided, this value will override any auto-generated value.

Permissions

Briefcase cross platform permissions map to `<uses-permission>` declarations in the app's `AppManifest.xml`:

- camera: `android.permission.CAMERA`
- microphone: `android.permission.RECORD_AUDIO`
- coarse_location: `android.permission.ACCESS_COARSE_LOCATION`
- fine_location: `android.permission.ACCESS_FINE_LOCATION`
- background_location: `android.permission.ACCESS_BACKGROUND_LOCATION`
- photo_library: `android.permission.READ_MEDIA_VISUAL_USER_SELECTED`

Every application will be automatically granted the `android.permission.INTERNET` and `android.permission.NETWORK_STATE` permissions.

Specifying a `camera` permission will result in the following non-required `feature` definitions being implicitly added to your app:

- `android.hardware.camera`,
- `android.hardware.camera.any`,
- `android.hardware.camera.front`,

- `android.hardware.camera.external` and
- `android.hardware.camera.autofocus`.

Specifying the `coarse_location`, `fine_location` or `background_location` permissions will result in the following non-required `feature` declarations being implicitly added to your app:

- `android.hardware.location.network`
- `android.hardware.location.gps`

This is done to ensure that an app is not prevented from installing if the device doesn't have the given features. You can make the feature explicitly required by manually defining these feature requirements. For example, to make GPS hardware required, you could add the following to the Android section of your `pyproject.toml`:

```
feature."android.hardware.location.gps" = True
```

Platform quirks

Availability of third-party packages

Briefcase is able to use third-party packages in Android apps. As long as the package is available on PyPI, or you can provide a wheel file for the package, it can be added to the `requires` declaration in your `pyproject.toml` file and used by your app at runtime.

If the package is pure Python (i.e., it does not contain a binary library), that's all you need to do. To check whether a package is pure Python, look at the PyPI downloads page for the project; if the wheels provided are have a `-py3-none-any.whl` suffix, then they are pure Python wheels. If the wheels have version and platform-specific extensions (e.g., `-cp311-cp311-macosx_11_0_universal2.whl`), then the wheel contains a binary component.

If the package contains a binary component, that wheel needs to be compiled for Android. PyPI does not currently support uploading Android-compatible wheels, so you can't rely on PyPI to provide those wheels. Briefcase uses a secondary repository to provide pre-compiled Android wheels.

This repository is maintained by the BeeWare project, and as a result, it does not have binary wheels for *every* package that is available on PyPI, or even every *version* of every package that is on PyPI. If you see any of the following messages when building an app for a mobile platform, then the package (or this version of it) probably isn't supported yet:

- The error "Chaquopy cannot compile native code"
- A reference to downloading a `.tar.gz` version of the package
- A reference to `Building wheels for collected packages:  <package>`

It is *usually* possible to compile any binary package wheels for Android, depending on the requirements of the package itself. If the package has a dependency on other binary libraries (e.g., something like `libjpeg` that isn't written in Python), those libraries will need to be compiled for Android as well. However, if the library requires build tools that don't support Android, such as a compiler that can't target Android, or a PEP517 build system that doesn't support cross-compilation, it may not be possible to build an Android wheel.

The Chaquopy repository contains tools to assist with cross-compiling Android binary wheels. This repository contains recipes for building the packages that are stored in the secondary package repository. Contributions of new package recipes are welcome, and can be submitted as pull requests. Or, if you have a particular package that you'd like us to support, please visit the issue tracker and provide details about that package.

Web

Briefcase supports packaging web apps as *static web deployments*.

Static Web

Host Platform Support (*Platform support*)								
macOS		Windows			Linux			
x86-64	arm64	x86	x86-64	arm64	x86	x86-64	arm	arm64

When generating a web project, Briefcase produces a static folder of HTML, CSS and JavaScript resources that can be deployed as a web site. The static web site is packaged as a `.zip` file for distribution.

Although Briefcase provides a `run` command that can be used to serve the website, this web server is provided as a development convenience. **It should not be used in production**. If you wish to serve your app in production, you can unzip the `.zip` file in the root of any web server that can serve static web content.

Web support is experimental!

PyScript (which forms the base of Briefcase's web backend) is a new project; and Toga's web backend is very new. As a result this web backend should be considered experimental.

Regardless of what Python version you run Briefcase with, the app will use PyScript's current Python version (as of October 2022, this is 3.10).

There are also a number of constraints on what you can do in a web environment. Some of these are fundamental constraints on the web as a platform; some are known issues with PyScript and Pyodide as runtime environments. You shouldn't expect that arbitrary third-party Python packages will "just run" in a web environment.

Icon format

Web projects use a single 32px `.png` format icon as the site icon.

Web projects do not support splash screens or installer images.

Additional options

The following options can be provided at the command line when producing web projects:

run

--host <ip or hostname>

The hostname or IP address that the development web server should be bound to. Defaults to `localhost`.

-p <port> / --port <port>

The port that the development web server should be bound to. Defaults to `8080`. If port `8080` is already in use, an arbitrary available port will be used.

--no-browser

Don't open a web browser after starting the development web server.

Application configuration

The following options can be added to the `tool.briefcase.app.<appname>.web` section of your `pyproject.toml` file:

extra_pyscript_toml_content

Any additional configuration that you wish to add to the `pyscript.toml` file for your deployed site. For example, you can use this to change the runtime used by Briefcase when deploying your site.

`extra_pyscript_toml_content` is a string that defines valid TOML content; this content will be parsed, and used to add values to the `pyscript.toml` generated by Briefcase, overriding any pre-existing keys. For example, to define a custom runtime, and change the default PyScript app name, you could use:

```
extra_pyscript_toml_content = """
name = "My name override"

[[runtimes]]
src = "https://example.com/custom/pyodide.js"
"""
```

| Supported and tested in CI |
| Supported and tested by maintainers |
| Supported but not tested regularly |

Target App Format		Host System macOS		Windows			Linux			
		x86-64	arm64	x86	x86-64	arm64	x86	x86-64	arm	arm64
Android	**Gradle project**									
iOS	**Xcode project**									
Linux	AppImage									
	Flatpak									
	System package									
macOS	**.app bundle**									
	Xcode project									
Web	**Static**									
Win-dows	**Windows app**									
	Visual Stu-dio project									