Briefcase Documentation

Release 0.3.0.dev7

Russell Keith-Magee

Feb 22, 2020

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 an AppImage;
- iOS, as an XCode project; and
- Android, as a Gradle project.

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

CHAPTER 1

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

CHAPTER 2

Community

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

- @pybeeware on Twitter
- beeware/general on Gitter

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:

macOS

TODO

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:

```
[build-system]
requires = ["briefcase"]

[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 `[build-system]` section is preamble required by PEP518, declaring the dependency on Briefcase.

The remaining 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 heirarchical. `[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.

Briecase v0.3 uses it's 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 subcommands 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 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.

Setting up your development environment

The recommended way of setting up your development environment for Briefcase is to use a virtual environment, install the required dependencies and start coding:

macOS

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

Linux

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

Windows

```
C:\...>git clone https://github.com/beeware/briefcase.git
C:\...>cd briefcase
C:\...>py -m venv venv
C:\...>venv\Scripts\activate
```

To install all the development version of Briefcase, along with all it's requirements, run the following commands within your virtual environment:

macOS

```
$ (venv) pip install -e .
```

Linux

```
$ (venv) pip install -e .
```

Windows

```
C:\...>pip install -e .
```

Now you are ready to start hacking! Have fun!

Briefcase uses PyTest for its own test suite. To run the test suite, install PyTest, and run the test suite.

macOS

```
$ (venv) pip install pytest
$ (venv) pytest
```

Linux

```
$ (venv) pip install pytest
$ (venv) pytest
```

Windows

```
C:\...>pip install pytest
C:\...>pytest
```

2.2.4 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 Restructured Text (reST) and Sphinx CheatSheet 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

Go to the documentation folder:

```
$ cd docs
```

Install Sphinx with the helpers and extensions we use:

```
$ pip install -r requirements_rtd.txt
```

Create the static files:

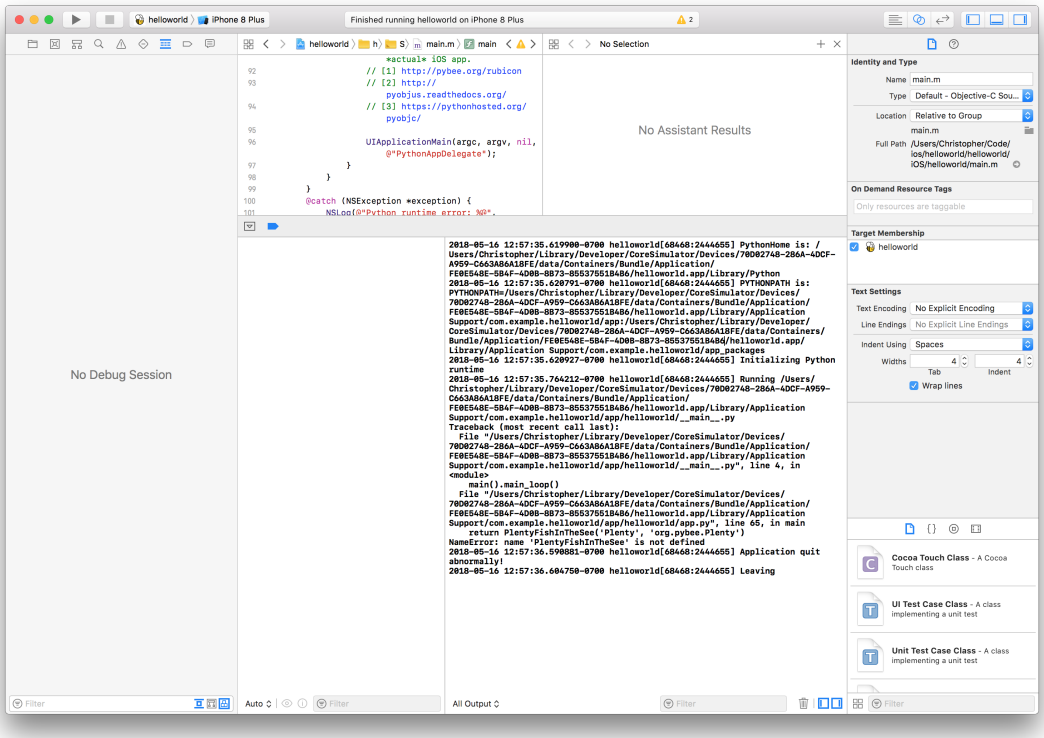
```
$ make html
```

Check for any errors and,if possible, fix them. The output of the file should be in the `_build/html` folder. Open the file you changed in the browser.

2.2.5 See Errors on iOS

If you have a beeware iOS project that has a crash, it can be difficult to see the stacktrace. Here's how to do it -

1. Build your iOS project. You don't have to start it.
2. Open that iOS project in Xcode. Click the Run button (looks like an arrow) and wait for the simulator to open. Cause the app to crash.
3. Your stack trace ought to appear in the 'debugger area' at the bottom of the screen. If you can't see that area, you may have to activate it with `View` > `Debug Area` > `Show Debug Area`



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. It also assumes that you have a local `release` branch that tracks `upstream/master`.

You should have a checkout of a personal fork of the Briefcase repository; to configure this fork for a release, run:

```
$ git remote add upstream git@github.com:beeware/briefcase.git
$ git clone upstream/master release
```

The procedure for cutting a new release is as follows:

1. Refresh your release branch:

   ```
   $ git checkout release
   $ git fetch upstream
   $ git pull
   ```

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

2. Make sure the branch is ready for release. Ensure that:

   1. The version number has been bumped.
   2. The release notes are up to date. If they are, the changes directory should be empty, except for the template.rst. If it isn't empty, run:

      ```
      $ towncrier --draft
      ```

      to review the release notes, and then:

      ```
      $ towncrier
      ```

      to generate the updated release notes.

3. Tag the release, and push the tag upstream:

   ```
   $ git tag v1.2.3
   $ git push upstream --tags
   ```

4. 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.
5. Edit the Github release. Add release notes (you can use the text generated by towncrier). Check the pre-release checkbox (if necessary).
6. Double check everything, then click Publish. This will trigger a publication workflow on Github.

7. Wait for the package to appear on PyPI.

Congratulations, you've just published a release!

If anything went wrong during steps 3 or 5, you will need to delete the draft release from Github, and push an updated tag. Once the release has successfully appeared on PyPI, it cannot be changed; if you spot a problem in a published package, you'll need to tag a completely new release.

2.3 About Briefcase

2.3.1 Frequently Asked Questions

What version of Python does Briefcase support?

Python 3.5 or higher.

What platforms does Briefcase support?

Briefcase currently has support for:

- macOS (producing DMG files, or raw .app files)
- Linux (producing AppImage files)
- Windows (producing MSI installers)
- iOS (producing Xcode projects)

Support for Android will be added in the near future. Support for other some other packaging formats (e.g., NSIS installers for Windows; Snap and Flatpak installers for Linux) and other operating systems (e.g., tvOS, watchOS, WearOS) are on our roadmap.

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.

2.3.2 The Briefcase Developer and User community

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

- @pybeeware on Twitter
- 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.

2.3.4 Release History

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 rollout 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 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:

```
[build-system]
requires = ["briefcase"]

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

(continues on next page)

(continued from previous page)

```
version = "0.1"

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

The `[build-system]` section is preamble required by PEP518, declaring the dependency on Briefcase.

The remaining 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.

[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.

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 *cumlative* 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

author

The person or organization responsible for the application.

author_email

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

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. As with `splash`, the path should *exclude* the extension, and a platform-appropriate extension will be appended when the application is built.

requires

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

Unlike most other keys in a configuration file, `requires` is *cumlative* 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.

splash

A path, relative to the directory where the `pyproject.toml` file is located, to an image to use as the splash screen for the application. The path should *exclude* the extension; Briefcase will append a platform appropriate extension when configuring the application.

Some platforms require multiple splash images, at different sizes; these will be handled by appending the required size to the provided icon name. For example, iOS requires multiple splash screens, (including 1024x768px, 768x1024px, 2048x1536px, and more); Briefcase will look for `resources/splash-1024x768.png`, `resources/splash-768x1024.png`, `resources/splash-2048x1536.png`, and so on. The sizes that are required are determined by the platform template.

If the platform output format does not use a splash screen, the `splash` 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.

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.

url

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

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 applcation. 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.

url

A URL for help related to the document format.

2.4.2 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.

If a URL is provided, the `master` branch of the referenced repository will be used.

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 dependencies 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.

`-d` / `--update-dependencies`

Update application dependencies.

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 codebase to the packaged application code.

It will *not* update dependencies 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.

`-d` / `--update-dependencies`

Update application dependencies.

-r / --update-resources

Update application resources (e.g., icons and splash screens).

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 dependencies

Building for some platforms depends on the build tools for the platform you're targetting 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 targetting is not supported.

Options

The following options can be provided at the command line.

-u / --update

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

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

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 dependencies that will be installed, but *won't* run the installer to produce Start Menu items, registry records, etc.

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 run
```

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 dependencies

Building installers for some platforms depends on the build tools for the platform you're targetting 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 targetting 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
```

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 specifif 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.

2.4.3 Platform support

macOS

The default output format for macOS is a *DMG*.

Briefcase also supports creating *.app bundles*.

macOS DMG

A macOS DMG (Disk iMaGe) is a common format for distributing macOS content. It presents itself to the operating system a disk that can be mounted; the image then contains the files being distributed. For this reason, DMGs are often used for distributing new applications, especially when there

Briefcase's DMG support is an extension of `.app` bundle support. The DMG created by Briefcase contains a `.app` bundle for the application, plus a symbolic link to the user's `/Applications` folder; this enables the user to install their application by mounting the DMG, and dragging the application's icon onto the icon for the `/Applications` folder contained in the DMG. A background image can be provided to provide an additional visual hint for the installation action.

Icon format

macOS DMGs use `.icns` format icons for the application and installer.

Image format

macOS DMGs do not support splash screens. The installer background must be in `.png` format.

Additional options

The following options can be provided at the command line when producing DMGs.

publish

--no-sign

Don't perform code signing on the `.app` bundles in the DMG.

`-i <identity>` / `--identity <identity>`

The code signing identity to use when signing the `.app` bundles in the DMG.

.app bundle

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.

Icon format

`.app` bundles use `.icns` format icons.

Image format

`.app` bundles do not support splash screens or installer images.

Additional options

The following options can be provided at the command line when producing `.app` bundles.

publish

`--no-sign`

Don't perform code signing on the `.app` bundles.

`-i <identity>` / `--identity <identity>`

The code signing identity to use when signing the `.app` bundles.

Windows

The default output format for Windows is an *MSI Installer*.

MSI Installer

An MSI installer is a common format used for the installation, maintenance, and removal of Windows software. It contains the files to be distributed, along with metadata supporting the files to be installed, including details such as registry entries. It includes a GUI installer, and automated generation of the uninstallation sequence.

Briefcase uses the WiX Toolset to build installers. 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

MSI installers use `.ico` format icons.

Image format

MSI installers do not support splash screens or installer images.

Linux

The default output format for Linux is *AppImage*.

Linux AppImage

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.

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 simplify the packaging process, Briefcase provides a pre-compiled Python support library. This support library was compiled on Ubuntu 16.04, which means the AppImages build by Briefcase can be used on *any* Linux distribution of about the same age or newer - but those AppImages *must* be compiled on Ubuntu 16.04.

This means you have four options for using Briefcase to compile a Linux AppImage:

1. Install Ubuntu 16.04 on your own machine.
2. Find a cloud or CI provider that can provide you an Ubuntu 16.04 machine for build purposes. Github Actions, for example, provides Ubuntu 16.04 as a build option.
3. Run Briefcase inside a Docker container. Once you have installed Docker, the command:

   ```
   $ docker run -it -v /path/to/project:/project ubuntu:16.04 /bin/bash
   ```

   will start a Docker container running Ubuntu 16.04, mounting your local project directory (`/path/to/project`) as the `/project` directory in the container. You can then install the requirements necessary to run Briefcase inside the container:

   ```
   $ apt-get update
   $ apt-get install python3-dev
   $ pip install briefcase
   ```

Depending on the application you're packaging, you may need to install additional system libraries (e.g., graphics libraries to support the GUI toolkit). For example, if you're intending to use BeeWare's Toga GUI toolkit, you'll need the following system libraries:

```
$ apt-get install libgirepository1.0-dev libcairo2-dev libpango1.0-dev␣
↪libwebkitgtk-3.0-0 gir1.2-webkit-3.0
```

As an aside, using Docker will also allow you to create Linux packages on Windows or macOS.

4. Build your own version of the BeeWare Python support libraries. If you take this approach, be aware that your AppImage will only be as portable as the version of libc that is available on the distribution you use. If you build using Ubuntu 19.10, for example, you can expect that only people on the most recent versions of another distribution will be able to run your AppImage.

Icon format

AppImages use `.png` format icons.

Image format

AppImages do not support splash screens or installer images.

iOS

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
- 1024px

Image format

iOS projects use `.png` format splash screen iamges. An application must provide splash images of the following sizes:

- 640x1136px
- 640x960px
- 750x1334px
- 768x1004px
- 768x1024px
- 828x1792px
- 1024x748px
- 1024x768px
- 1125x2436px
- 1242x2208px
- 1242x2688px
- 1536x2008px
- 1536x2048px
- 1792x828px
- 2048x1496px
- 2048x1536px
- 2208x1242px
- 2436x1125px
- 2688x1242px

iOS projects do not support installer images.

Additional options

The following options can be provided at the command line when producing iOS projects

build

`-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::13.3"`).

run

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::13.3"`).

Android

Note: The BeeWare project *has* tooling for Android. Unfortunately, due to some recent changes in the Android ecosystem, combined with some changes in Toga itself, means that our Android tooling is currently broken. We're working on it, and hope to have a solution in the very near future.

When generating an Android project, Briefcase produces a Gradle project.