BeeWare Documentation

Version 0.1.dev50+gea6c563

Russell Keith-Magee

23 juin 2025

Table des matières

Écrire en Python. Exécutez n'importe où.

Bienvenue à BeeWare ! Dans ce tutoriel, nous allons construire une interface utilisateur graphique en utilisant Python, et la déployer en tant qu'application de bureau, application mobile et application web à page unique. Nous allons également voir comment vous pouvez utiliser les outils de BeeWare pour effectuer certaines tâches courantes pour les développeur d'applications, telles que tester votre application.

Il s'agit d'une traduction automatique !

Cette version du didacticiel a en partie été générée par une traduction automatique. Nous savons que ce n'est pas la solution idéale, mais nous avons estimé qu'une mauvaise traduction valait mieux que pas de traduction du tout.

Si vous souhaitez nous aider à améliorer la traduction, n'hésitez pas à nous contacter ! Nous avons un canal `#translations` dans Discord ; présentez-vous et nous vous ajouterons à l'équipe de traduction.

CHAPITRE 1

Qu'est-ce que BeeWare ?

BeeWare n'est pas un produit, un outil ou une bibliothèque unique - c'est une collection d'outils et de bibliothèques, chacun d'entre eux fonctionnant ensemble pour vous aider à écrire des applications Python multiplateformes avec une interface graphique native. Il comprend :

- Toga, une boîte à outils de widgets multiplateforme ;
- Briefcase, un outil pour empaqueter les projets Python en tant qu'artefacts distribuables pouvant être envoyés aux utilisateurs finaux ;
- Des bibliothèques (telles que Rubicon ObjC) pour accéder aux bibliothèques natives des diverses plates-formes ;
- Des versions pré-compilées de Python pour les plateformes où les programmes d'installation officiels de Python ne sont pas disponibles.

Dans ce tutoriel, nous utiliserons tous ces outils, mais en tant qu'utilisateur, vous n'aurez besoin d'interagir qu'avec les deux premiers (Toga et Briefcase). Cependant, chacun de ces outils peut également être utilisé séparément. Par exemple, vous pouvez utiliser Briefcase pour déployer une application sans utiliser Toga comme boîte à outils GUI.

La suite BeeWare est disponible pour macOS, Windows, Linux (avec GTK) ; pour des plateformes mobiles telles qu'Android et iOS ; et pour le Web.

CHAPITRE 2

C'est parti !

Prêt à essayer BeeWare par vous-même ? *Créons une application multiplateforme en Python !*

2.1 Tutoriel 0 - Préparons-nous !

Avant de créer notre première application BeeWare, nous devons nous assurer que nous disposons de tous les prérequis pour le fonctionnement de BeeWare.

2.1.1 Installer Python

La première chose dont nous avons besoin est un interpréteur Python fonctionnel.

macOS

Linux

Windows

Si vous utilisez macOS, une version récente de Python est incluse dans Xcode ou dans les outils de développement en ligne de commande. Pour vérifier si vous l'avez déjà, exécutez la commande suivante :

```
$ python3 --version
```

Si Python est installé, vous verrez son numéro de version. Sinon, vous serez invité à installer les outils de développement en ligne de commande.

Si vous êtes sous Windows, vous pouvez obtenir l'installateur officiel à partir du site web de Python. Vous pouvez utiliser n'importe quelle version stable de Python à partir de la 3.8. Nous vous conseillons d'éviter les versions alpha, beta et release candidates à moins que vous ne sachiez *vraiment* ce que vous faites.

Si vous êtes sous Linux, vous installerez Python en utilisant le gestionnaire de paquets du système (`apt` sur Debian/Ubuntu/Mint, `dnf` sur Fedora, ou `pacman` sur Arch).

Vous devez vous assurer que le Python du système est Python 3.8 ou plus récent ; si ce n'est pas le cas (par exemple, Ubuntu 18.04 est livré avec Python 3.6), vous devrez mettre à jour votre distribution Linux vers quelque chose de plus récent.

La prise en charge du Raspberry Pi est limitée pour le moment.

Si vous êtes sous Windows, vous pouvez obtenir l'installateur officiel à partir du site web de Python. Vous pouvez utiliser n'importe quelle version stable de Python à partir de la 3.8. Nous vous conseillons d'éviter les versions alpha, beta et release candidates à moins que vous ne sachiez *vraiment* ce que vous faites.

Distributions alternatives de Python

Il existe de nombreuses façons d'installer Python. Vous pouvez installer Python via homebrew. Vous pouvez utiliser pyenv pour gérer plusieurs installations de Python sur la même machine. Les utilisateurs de Windows peuvent installer Python à partir du Windows App Store. Les utilisateurs ayant une formation en science des données pourraient préfèrer utiliser Anaconda ou Miniconda.

Si vous êtes sous macOS ou Windows, peu importe *comment* vous avez installé Python – ce qui compte c'est que vous puissiez lancer `python3` à partir de l'invite de commande/terminal de votre système d'exploitation, et obtenir un interpréteur Python fonctionnel.

Si vous êtes sous Linux, vous devez utiliser le Python fourni par votre système d'exploitation. Vous pourrez réaliser *la majeure partie* de ce tutoriel en utilisant un Python non système, mais vous ne pourrez pas empaqueter votre application pour la distribuer à d'autres.

2.1.2 Installer les dépendances

Ensuite, installez les dépendances supplémentaires nécessaires à votre système d'exploitation :

macOS

Linux

Windows

La construction d'applications BeeWare sur macOS nécessite :

— **Git**, a version control system. This is included with Xcode or the command line developer tools, which you installed above. You may need to open Xcode for the first time in order for Git to work in your terminal session. If it still doesn't register that Git is installed, you may need to restart your terminal session.

Pour permettre le développement local, vous allez devoir installer certains paquets système. La liste des paquets requis varie en fonction de votre distribution :

Ubuntu 20.04+ / Debian 10+

```
$ sudo apt update
$ sudo apt install git build-essential pkg-config python3-dev python3-venv␣
↪libgirepository1.0-dev libcairo2-dev gir1.2-gtk-3.0 libcanberra-gtk3-module
```

Fedora

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

Arch, Manjaro

```
$ sudo pacman -Syu git base-devel pkgconf python3 gobject-introspection cairo gtk3␣
↪libcanberra
```

OpenSUSE Tumbleweed

```
$ sudo zypper install git patterns-devel-base-devel_basis pkgconf-pkg-config python3-
↪devel gobject-introspection-devel cairo-devel gtk3 'typelib(Gtk)=3.0' libcanberra-gtk3-
↪module
```

La création d'applications BeeWare sur Windows nécessite :

— **Git**, un système de contrôle de version. Vous pouvez télécharger Git à partir de git-scm.com.

Après avoir installé ces outils, assurez-vous de redémarrer toutes les sessions de terminal. Sous Windows, les nouveaux outils installés ne seront accessibles qu'aux terminaux démarrés *après* la fin de l'installation.

2.1.3 Mise en place d'un environnement virtuel

Nous allons maintenant créer un environnement virtuel – un « bac à sable » (sandbox) que nous pouvons utiliser pour isoler notre travail sur ce tutoriel de notre installation Python principale. Si nous installons des paquets dans l'environnement virtuel, notre installation principale de Python (et tout autre projet Python sur notre ordinateur) ne sera pas affectée. Si nous provoquons un désordre complet dans notre environnement virtuel, nous pourrons simplement l'effacer et recommencer, sans affecter aucun autre projet Python sur notre ordinateur, et sans avoir besoin de réinstaller Python.

macOS

Linux

Windows

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

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

```
C:\...>md beeware-tutorial
C:\...>cd beeware-tutorial
C:\...>py -3.12 -m venv beeware-venv
C:\...>beeware-venv\Scripts\activate
```

If you're not using Python 3.12, replace the `-3.12` in these instructions with the version number that you are using.

Erreurs lors de l'exécution de scripts PowerShell

Si vous utilisez PowerShell et que vous recevez l'erreur :

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

Votre compte Windows n'a pas les autorisations nécessaires pour exécuter des scripts. Pour y remédier :

1. Run Windows PowerShell as Administrator.
2. Exécutez `set-executionpolicy RemoteSigned`
3. Sélectionnez « O » pour modifier la politique d'exécution.

Une fois que vous avez fait cela, vous pouvez exécuter à nouveau `beeware-venv\Scripts\activate.ps1` dans votre session PowerShell d'origine (ou dans une nouvelle session dans le même répertoire).

Si cela a fonctionné, votre prompt devrait maintenant être modifié – il devrait avoir un préfixe `(beeware-venv)`. Cela vous permet de savoir que vous êtes actuellement dans votre environnement virtuel BeeWare. Chaque fois que vous travaillerez sur ce tutoriel, vous devrez vous assurer que votre environnement virtuel est activé. Si ce n'est pas le cas, relancez la dernière commande (la commande `activate`) pour réactiver votre environnement.

Environnements virtuels alternatifs

Si vous utilisez Anaconda ou miniconda, vous êtes peut-être plus familier avec l'utilisation des environnements conda. Vous avez peut-être aussi entendu parler de `virtualenv`, un prédécesseur du module intégré `venv` de Python. De même pour les installations de Python – si vous êtes sous macOS ou Windows, peu importe *comment* vous créez votre environnement virtuel, tant que vous en avez un. Si vous êtes sous Linux, vous devriez vous en tenir à `venv` et au Python du système.

2.1.4 Étapes suivantes

Nous avons maintenant mis en place notre environnement. Nous sommes prêts à*créer notre première application BeeWare*.

2.2 Tutoriel 1 - Votre première application

Nous sommes prêts à créer notre première application.

2.2.1 Installer les outils BeeWare

Tout d'abord, nous devons installer **Briefcase**. Briefcase est un outil BeeWare qui peut être utilisé pour empaqueter votre application afin de la distribuer aux utilisateurs finaux – mais il peut également être utilisé pour démarrer un nouveau projet. Assurez-vous d'être dans le répertoire `beeware-tutorial` que vous avez créé lors du*Tutorial 0*, avec l'environnement virtuel `beeware-venv` activé, et exécutez :

macOS

Linux

Windows

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

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

Erreurs possibles lors de l'installation

Si vous rencontrez des erreurs pendant l'installation, c'est très certainement parce que certains des prérequis du système n'ont pas été installés. Assurez-vous d'avoir*installé tous les prérequis de la plateforme*.

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

Erreurs possibles lors de l'installation

Il est important que vous utilisiez `python -m pip`, plutôt qu'un simple `pip`. Briefcase doit s'assurer qu'il a une version à jour de `pip` et de `setuptools`, et une simple utilisation de `pip` ne peut pas se mettre à jour d'elle-même. Si vous voulez en savoir plus, Brett Cannon a publié un billet de blog détaillé sur ce sujet.

2.2.2 Créer un nouveau projet

Commençons notre premier projet BeeWare ! Nous allons utiliser la commande `new` de Briefcase pour créer une application appelée **Hello World**. Exécutez la commande suivante à partir de votre invite de commande :

macOS

Linux

Windows

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

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

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

Briefcase nous demandera quelques détails sur notre nouvelle application. Pour les besoins de ce tutoriel, nous utiliserons les éléments suivants :

- **Formal Name** (Nom formel) – Acceptez la valeur par défaut : `Hello World`.
- **App Name** (Nom de l'application) - Acceptez la valeur par défaut : `helloworld`.
- **Bundle** - Si vous possédez votre propre domaine, entrez ce domaine en ordre inverse. (Par exemple, si vous possédez le domaine « cupcakes.com », entrez `com.cupcakes` comme bundle). Si vous ne possédez pas votre propre domaine, acceptez le bundle par défaut (`com.example`).
- **Project Name** (Nom du projet) - Acceptez la valeur par défaut : `Hello World`.
- **Description** - Acceptez la valeur par défaut (ou, si vous voulez être vraiment créatif, trouvez votre propre description !)
- ** Author** (Auteur) - Entrez votre propre nom ici.
- **Author's email** (adresse email) - Entrez votre propre adresse email. Elle sera utilisée dans le fichier de configuration, dans le texte d'aide, et partout où un email est requis lors de la soumission de l'application à un app store (magasin d'applications).
- **URL** - L'URL de la page d'accueil de votre application. Encore une fois, si vous possédez votre propre domaine, entrez une URL sur ce domaine (y compris le `https://`). Sinon, acceptez l'URL par défaut (*https ://example.com/helloworld* `). Cette URL n'a pas besoin d'exister (pour l'instant) ; elle ne sera utilisée que si vous publiez votre application dans un magasin d'applications.
- **License** (Licence) - Acceptez la licence par défaut (BSD). Cela n'affecte en rien le fonctionnement du tutoriel, donc si vous avez des opinions fortes à ce sujet, n'hésitez pas à en choisir une autre.
- **GUI framework** (framework d'interface utilisateur graphique) - Acceptez l'option par défaut, Toga (la boîte à outils GUI de BeeWare).

Briefcase va alors générer un squelette de projet que vous pourrez utiliser. Si vous avez suivi ce tutoriel jusqu'ici, et accepté les paramètres par défaut tels que décrits, votre système de fichiers devrait ressembler à quelque chose comme ceci :

```
beeware-tutorial/
├── beeware-venv/
│   └── ...
└── helloworld/
    ├── CHANGELOG
    ├── LICENSE
    ├── pyproject.toml
    ├── README.rst
    ├── src/
    │   └── helloworld/
    │       ├── app.py
    │       ├── __init__.py
```

(suite sur la page suivante)

(suite de la page précédente)

```
    │       ├── __main__.py
    │       └── resources/
    │           └── README
    └── tests/
        ├── helloworld.py
        ├── __init__.py
        └── test_app.py
```

Ce squelette est en fait une application pleinement fonctionnelle, sans avoir rien à ajouter d'autre. Le dossier `src` contient tout le code de l'application, le dossier `tests` contient une suite de tests initiale, et le fichier `pyproject.toml` décrit comment empaqueter l'application pour la distribuer. Si vous ouvrez `pyproject.toml` dans un éditeur, vous verrez les détails de la configuration que vous venez de fournir à Briefcase.

Maintenant que nous avons une ébauche d'application, nous pouvons utiliser Briefcase pour l'exécuter.

2.2.3 Exécuter l'application en mode développeur

Entrez dans le répertoire du projet `helloworld` et faites démarrer le projet en mode développeur (ou `dev`) par briefcase :

macOS

Linux

Windows

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

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

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

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

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

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

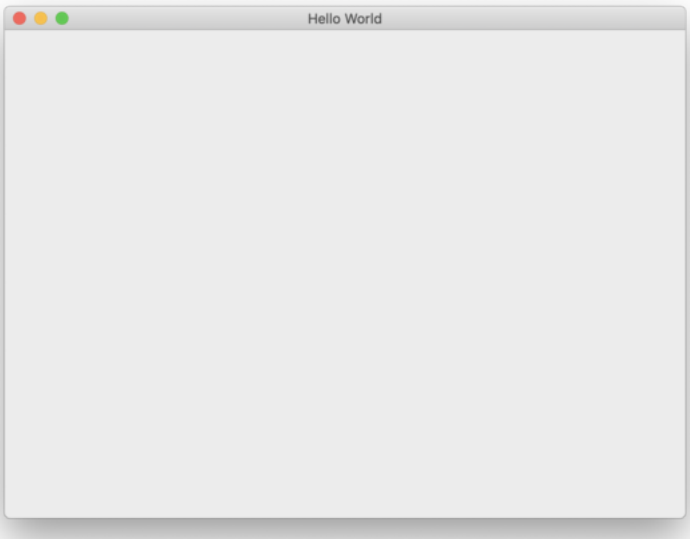
```
(beeware-venv) C:\...>cd helloworld
(beeware-venv) C:\...>briefcase dev

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

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

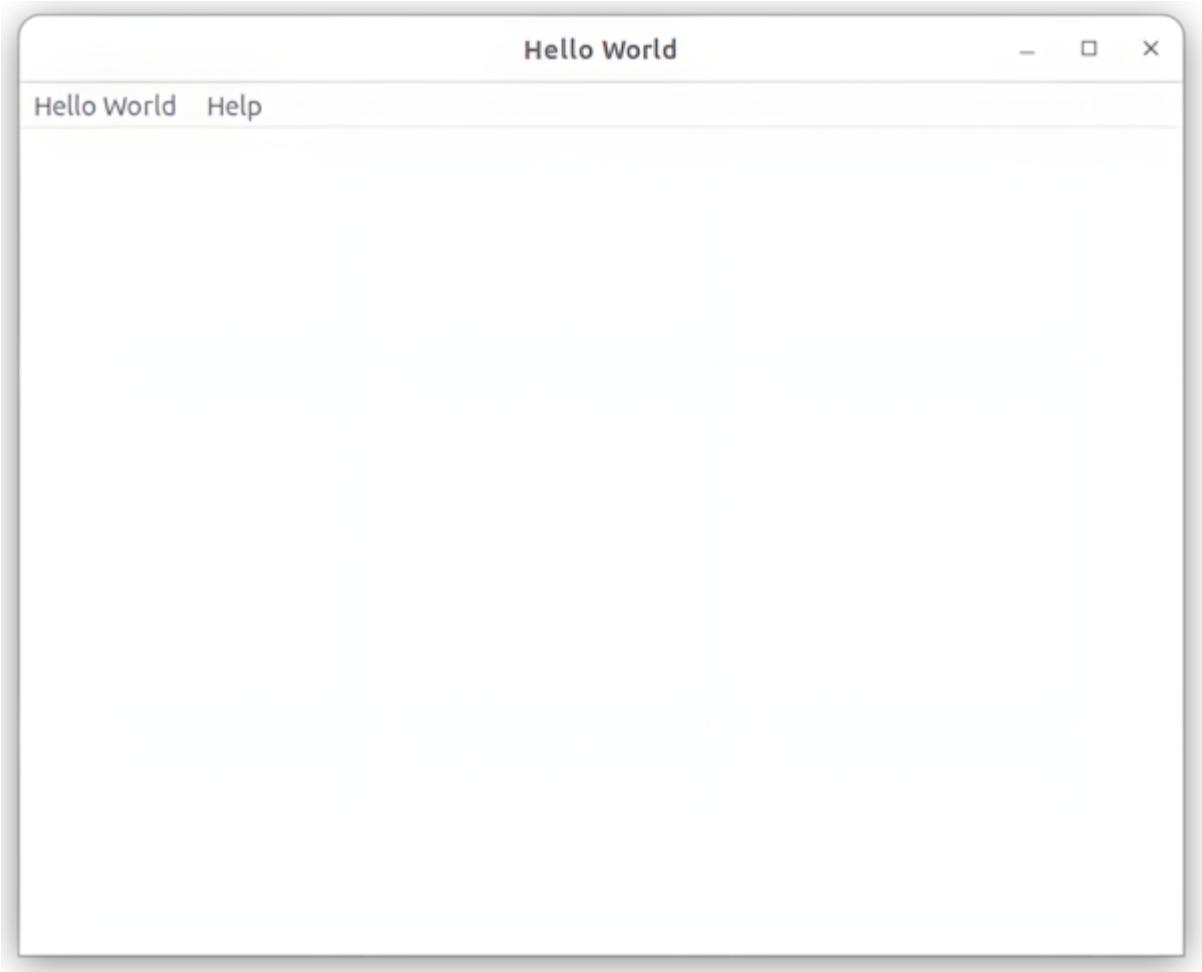
Cela devrait ouvrir une fenêtre GUI :

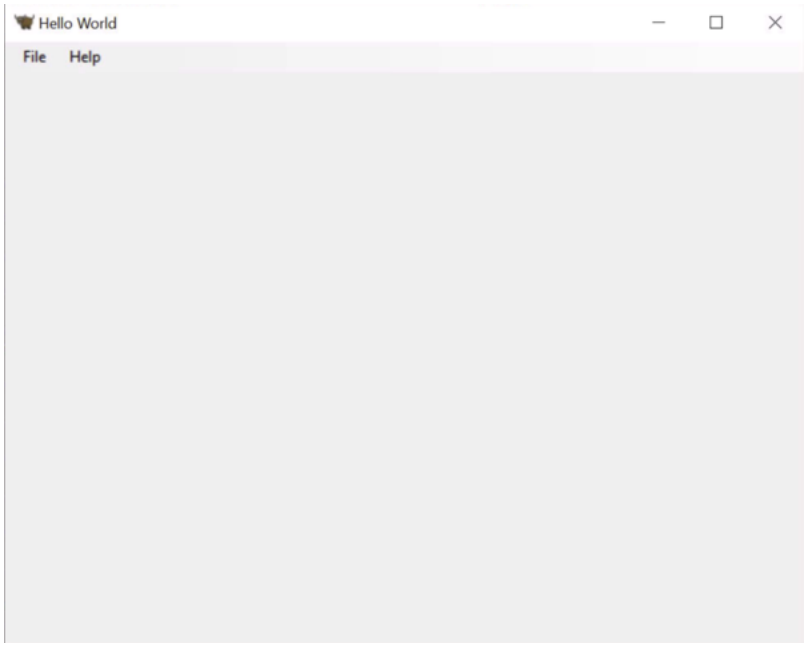
macOS

Linux

Windows







Invalid requirements or pip unable to connect

If you encounter the error :

```
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.
```

Confirm that you are running a *version of python that this tutorial supports*.

If your version of python is *not* a supported version, you will have to restart the tutorial.

Appuyez sur le bouton de fermeture (ou sélectionnez Quitter dans le menu d'application), et c'est terminé ! Félicitations, vous venez d'écrire une application native et autonome en Python !

2.2.4 Étapes suivantes

Nous avons maintenant une application fonctionnelle, s'exécutant en mode développeur. Nous pouvons maintenant ajouter notre propre logique pour que notre application fasse quelque chose d'un peu plus intéressant. Dans*Tutoriel 2*, nous allons ajouter à notre application une interface utilisateur plus utile.

2.3 Tutoriel 2 - Rendre les choses intéressantes

Dans le*Tutoriel 1*, nous avons généré une ébauche de projet capable de s'exécuter, mais nous n'avons pas écrit de code nous-mêmes. Jetons un coup d'œil à ce qui a été généré pour nous.

2.3.1 Ce qui a été généré

Dans le répertoire `src/helloworld`, vous devriez voir 3 fichiers : `__init__.py`, `__main__.py` et `app.py`.

`__init__.py` marque le répertoire `helloworld` comme un module Python importable. C'est un fichier vide ; le simple fait qu'il existe indique à l'interpréteur Python que le répertoire `helloworld` définit un module.

Le fichier `__main__.py` marque le module `helloworld` comme un type spécial de module - un module exécutable. Si vous essayez d'exécuter le module `helloworld'' en utilisant ``python -m helloworld'', le fichier ``__main__.py` est l'endroit où Python commencera à s'exécuter. Le contenu de *__main__.py*` est relativement simple :

```
from helloworld.app import main

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

This file does two things :

— It imports the `main` method from from the `helloworld` app.

— Il importe la méthode `main` de l'application `helloworld`, et s'il est exécuté en tant que point d'entrée, il appelle la méthode main(), et démarre la boucle principale de l'application. La boucle principale est la façon dont une application GUI écoute les entrées de l'utilisateur (comme les clics de souris et les pressions sur le clavier).

Le fichier le plus intéressant est `app.py` - il contient la logique responsable de la création de notre fenêtre d'application :

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

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

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

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

Examinons-le ligne par ligne :

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

Tout d'abord, nous importons la boîte à outils `toga`, ainsi que quelques classes et constantes utilitaires liées au style. Notre code ne les utilise pas encore, mais nous les utiliserons bientôt.

Ensuite, nous définissons une classe :

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

Chaque application Toga possède une seule instance `toga.App`, représentant l'entité en cours d'exécution, c'est-à-dire notre application. L'application peut gérer plusieurs fenêtres, mais pour les applications simples, il n'y aura qu'une seule fenêtre principale.

Ensuite, nous définissons une méthode de démarrage `startup()` :

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

La première chose que fait la méthode de démarrage est de définir une boîte principale. Le schéma de présentation de Toga est similaire à celui du HTML. Vous construisez une application en construisant une collection de boîtes, chacune d'entre elles contenant d'autres boîtes, ou de véritables widgets. Vous appliquez ensuite des styles à ces boîtes pour définir la manière dont elles utiliseront l'espace disponible dans la fenêtre.

Dans cette application, nous définissons une seule boîte, mais nous n'y mettons rien.

Ensuite, nous définissons une fenêtre dans laquelle nous pouvons placer cette boîte vide :

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

Cela crée une instance de `toga.MainWindow`, qui aura un titre correspondant au nom de l'application. Une fenêtre principale est un type spécial de fenêtre dans Toga - c'est une fenêtre qui est étroitement liée au cycle de vie de l'application. Lorsque la fenêtre principale est fermée, l'application se termine. La fenêtre principale est également la fenêtre qui contient le menu de l'application (si vous êtes sur une plateforme comme Windows où les barres de menu font partie des fenêtres).

Where is my window ?

If you have made an error in your code, the main window of the app may not display. If this happens, you can type **Ctrl+C** in the terminal where you started the app. This will stop the app. You can then fix the error and restart the app.

Nous ajoutons ensuite notre boîte vide au contenu de la fenêtre principale et demandons à l'application d'afficher notre fenêtre :

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

Enfin, nous définissons une méthode `main()`. C'est elle qui crée l'instance de notre application :

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

Cette méthode `main()` est celle qui est importée et invoquée par `main__.py`. Elle crée et retourne une instance de notre application `HelloWorld`.

C'est l'application Toga la plus simple possible. Bous allons ajouter notre propre contenu à l'application, et faire en sorte qu'elle fasse quelque chose de plus intéressant.

2.3.2 Ajouter notre propre contenu

Let's do something more interesting with our `HelloWorld` app.

Note

Ne supprimez pas les imports en haut du fichier, ni le `main()` en bas. Vous n'avez besoin de mettre à jour que la classe `HelloWorld`.

Modifiez votre classe `HelloWorld` dans `src/helloworld/app.py` pour qu'elle ressemble à ceci :

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

        name_label = toga.Label(
            "Your name: ",
            margin=(0, 5),
```

(suite sur la page suivante)

(suite de la page précédente)

```
        )
        self.name_input = toga.TextInput(flex=1)

        name_box = toga.Box(direction=ROW, margin=5)
        name_box.add(name_label)
        name_box.add(self.name_input)

        button = toga.Button(
            "Say Hello!",
            on_press=self.say_hello,
            margin=5,
        )

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

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

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

Examinons en détail ce qui a changé.

Nous sommes toujours en train de créer une boîte principale, mais nous appliquons maintenant un style :

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

Le système de mise en page intégré à Toga s'appelle « Pack ». Il se comporte en grande partie comme CSS. Vous définissez des objets dans une hiérarchie - en HTML, les objets sont `<div>`, `<span>`, et d'autres éléments DOM ; dans Toga, ce sont des widgets et des boîtes. Vous pouvez ensuite attribuer des styles aux différents éléments. Dans ce cas, nous indiquons qu'il s'agit d'une boîte `COLUMN` - c'est-à-dire qu'il s'agit d'une boîte qui utilisera toute la largeur disponible, et qui augmentera sa hauteur au fur et à mesure que du contenu sera ajouté, mais qui essaiera d'être aussi courte que possible.

Note

For more advanced uses, Toga also supports a separate style object, which is used like this :

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

Ensuite, nous définissons quelques widgets :

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

Ici, nous définissons un Label (widget de texte) et un TextInput (widget de saisie de texte). Les deux widgets sont associés à des styles ; le Label aura un espace de 5 px à gauche et à droite, et aucun espace en haut et en bas. Le

TextInput est marqué comme étant flexible, c'est-à-dire qu'il absorbera tout l'espace disponible dans son axe de mise en page.

Le TextInput est assigné en tant que variable d'instance d classe. Cela nous permet d'accéder facilement à l'instance du widget, ce que nous utiliserons dans un instant.

Ensuite, nous définissons une boîte pour contenir ces deux widgets :

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

La `name_box` est une boîte comme la boîte principale, mais cette fois, c'est une boîte `ROW` (rangée). Cela signifie que le contenu sera ajouté horizontalement, et qu'il essaiera d'être le moins large possible. La boîte a également un peu de padding (espace) - 5px sur tous les côtés.

Nous définissons maintenant un bouton :

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

Le bouton est également doté d'un espace de 5 px sur tous les côtés. Nous définissons également un *handler* - une méthode à invoquer lorsque le bouton est pressé.

Ensuite, nous ajoutons la boîte de nom et le bouton à la boîte principale :

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

Le reste de la méthode de démarrage est identique à la précédente : définition d'une fenêtre principale et attribution de la boîte principale en tant que contenu de la fenêtre :

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

Notre dernière action est de définir le gestionnaire du bouton. Un gestionnaire peut être n'importe quelle méthode, générateur ou co-routine asynchrone ; il accepte le widget qui a généré l'événement comme argument, et sera invoqué chaque fois que le bouton est pressé :

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

Le corps de la méthode est une simple instruction d'affichage (print). Cependant, elle interroge la valeur actuelle de l'entrée name et utilise ce contenu comme texte à imprimer.

Maintenant que nous avons effectué ces changements, nous pouvons voir à quoi ils ressemblent en relançant l'application. Comme précédemment, nous utiliserons le mode développeur :

macOS

Linux

Windows

```
(beeware-venv) $ briefcase dev

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

```
(beeware-venv) $ briefcase dev

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

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

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

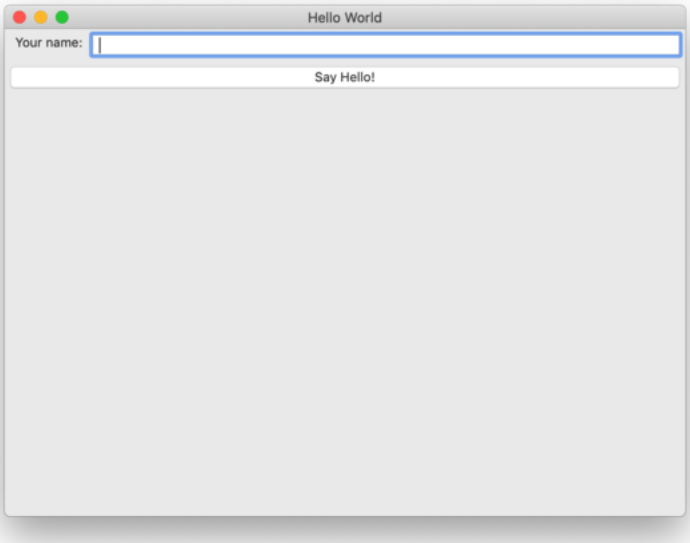
Vous remarquerez que cette fois-ci, briefcase n'installe pas de dépendances. Briefcase peut détecter que l'application a déjà été exécutée auparavant, et pour gagner du temps, il n'exécutera que l'application. Si vous ajoutez de nouvelles dépendances à votre application, vous pouvez vous assurer qu'elles soient installées en passant une option `-r` lorsque vous lancez `briefcase dev`.

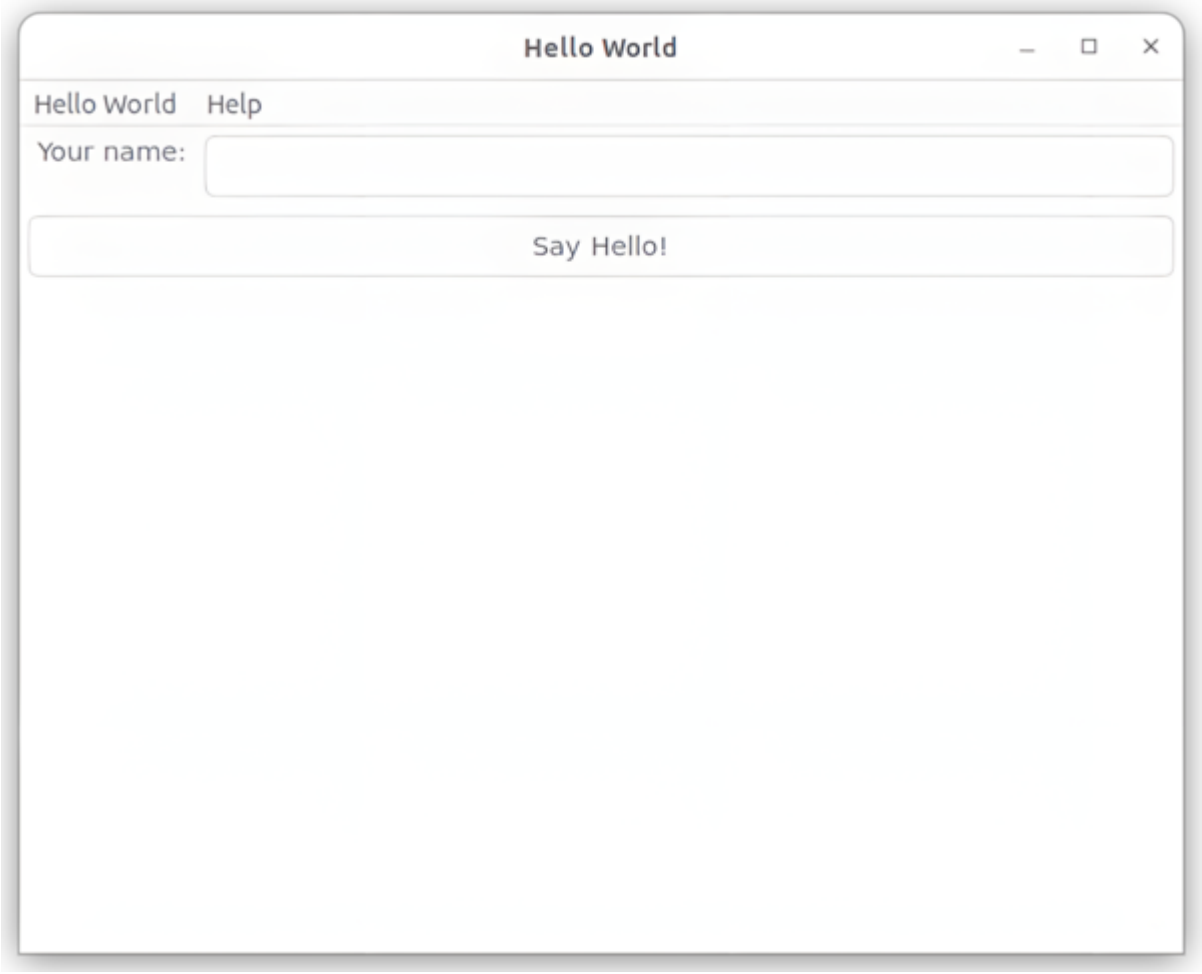
Cela devrait ouvrir une fenêtre GUI :

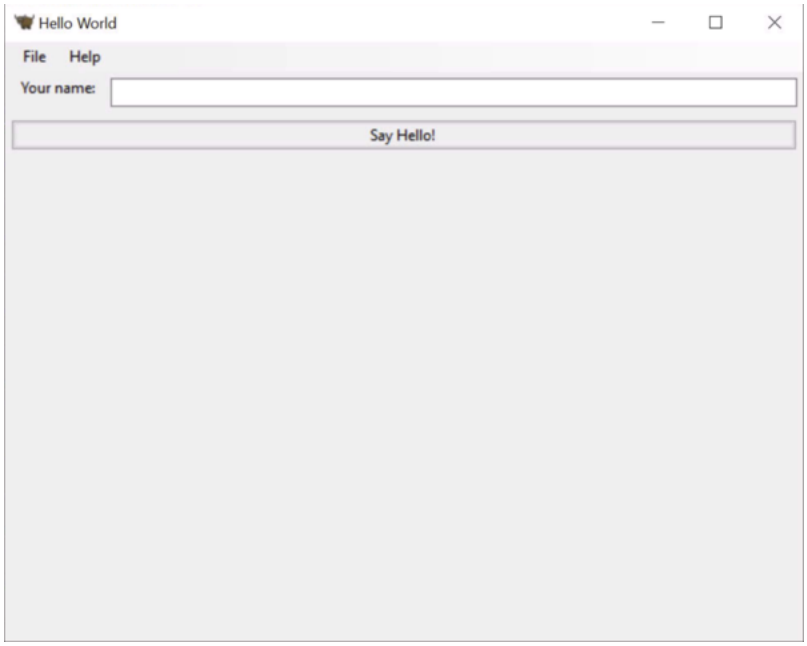
macOS

Linux

Windows







Si vous saisissez un nom dans la zone de texte et que vous appuyez sur le bouton GUI, vous devriez voir apparaître une sortie dans la console où vous avez démarré l'application.

Before continuing, close the app. As with Tutorial 1, you can do this by pressing the close button on the application window, by selecting Quit/Exit from the application's menu, or by typing **Ctrl+C** in the terminal where you ran `briefcase dev`.

2.3.3 Étapes suivantes

Nous avons maintenant une application qui fait quelque chose d'un peu plus intéressant. Mais elle ne fonctionne que sur notre propre ordinateur. Allons empaqueter cette application pour la distribuer. Dans*Tutoriel 3*, nous allons empaqueter notre application sous la forme d'un programme d'installation autonome que nous pourrons envoyer à un ami, un client, ou télécharger sur un App Store.

2.4 Tutoriel 3 - Emballage pour la distribution

Jusqu'à présent, nous avons exécuté notre application en « mode développeur ». Cela nous permet d'exécuter facilement notre application localement - mais ce que nous voulons vraiment, c'est pouvoir donner notre application à d'autres personnes.

Cependant, nous ne voulons pas avoir à apprendre à nos utilisateurs comment installer Python, créer un environnement virtuel, cloner un dépôt git, et lancer Briefcase en mode développeur. Nous préférons leur donner un programme d'installation et faire en sorte que l'application fonctionne tout simplement.

Briefcase peut être utilisé pour empaqueter votre application afin de la distribuer de cette manière.

2.4.1 Création de l'échafaudage de l'application

Puisque c'est la première fois que nous empaquetons notre application, nous devons créer quelques fichiers de configuration et autres échafaudages pour supporter le processus d'empaquetage. Depuis le répertoire `helloworld`, exécutez :

macOS

Linux

Windows

```
(beeware-venv) $ briefcase create

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-macOS-app-template.git, branch␣
↪v0.3.18
...

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

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

[helloworld] Installing requirements...
...

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

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

(suite sur la page suivante)

(suite de la page précédente)

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

```
(beeware-venv) $ briefcase create

[helloworld] Finalizing application configuration...
Targeting ubuntu:jammy (Vendor base debian)
Determining glibc version... done
Targeting glibc 2.35
Targeting Python3.10

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-linux-AppImage-template.git,␣
↪branch v0.3.18
...

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

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

[helloworld] Installing requirements...
...

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

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

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

Errors about Python versions

If you receive an error that reads something like :

> The version of Python being used to run Briefcase (3.12) is not the system python3 (3.10).

You will need to recreate your virtual environment using the system `python3`. Using the system Python is a requirement for packaging your application.

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

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-windows-app-template.git,␣
↪branch v0.3.18
...

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

(suite sur la page suivante)

(suite de la page précédente)

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

[helloworld] Installing requirements...
...

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

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

Vous venez probablement de voir défiler des pages de contenu dans votre terminal… que s'est-il passé ? Briefcase a fait ce qui suit :

1. Il **génère un modèle d'application**. Il y a beaucoup de fichiers et de configurations nécessaires pour construire un installateur natif, en plus du code de votre application réelle. Cet échafaudage supplémentaire est presque le même pour chaque application sur la même plateforme, à l'exception du nom de l'application réelle en cours de construction - Briefcase fournit donc un modèle d'application pour chaque plateforme qu'il supporte. Cette étape déploie le modèle, en substituant le nom de votre application, l'ID du bundle, et d'autres propriétés de votre fichier de configuration comme requis pour supporter la plateforme sur laquelle vous construisez.

   Si vous n'êtes pas satisfait du modèle fourni par Briefcase, vous pouvez créer votre propre modèle. Cependant, vous ne voudrez probablement pas faire cela avant d'avoir un peu plus d'expérience dans l'utilisation du modèle par défaut de Briefcase.

2. Il **télécharge et installe un paquetage de support**. L'approche d'empaquetage adoptée par Briefcase est décrite comme « la chose la plus simple qui puisse fonctionner » - elle fournit un interpréteur Python complet et isolé dans le cadre de chaque application qu'elle construit. Cette approche est légèrement inefficace en termes d'espace - si vous avez 5 applications empaquetées avec Briefcase, vous aurez 5 copies de l'interpréteur Python. Cependant, cette approche garantit que chaque application est complètement indépendante, utilisant une version spécifique de Python connue pour fonctionner avec l'application.

   Encore une fois, Briefcase fournit un paquetage de support par défaut pour chaque plateforme ; si vous le souhaitez, vous pouvez fournir votre propre paquetage de support, et faire en sorte que ce paquetage soit inclus dans le processus de construction. Vous pouvez faire cela si vous avez des options particulières dans l'interpréteur Python que vous devez activer, ou si vous voulez retirer de la bibliothèque standard les modules dont vous n'avez pas besoin au moment de l'exécution.

   Briefcase maintient un cache local des paquets de support, de sorte qu'une fois que vous avez téléchargé un paquet de support spécifique, cette copie en cache sera utilisée dans les futures versions.

   As noted above, when Briefcase packages an app as a native Linux system package (the default package format for Linux), a support package is not included with the app. Instead, the app will use the Python that is provided by the distribution of Linux being targeted.

3. Il **installe les exigences de l'application**. Votre application peut spécifier tous les modules tiers qui sont nécessaires à l'exécution. Ceux-ci seront installés en utilisant `pip` dans l'installeur de votre application.

4. Il **installe le code de votre application**. Votre application aura son propre code et ses propres ressources (par exemple, les images nécessaires à l'exécution) ; ces fichiers sont copiés dans le programme d'installation.

5. Il **installe les ressources nécessaires à votre application.** Enfin, il ajoute toutes les ressources supplémentaires nécessaires à l'installateur lui-même. Cela inclut des choses comme les icônes qui doivent être attachées à l'application finale et les images de l'écran d'accueil.

Une fois cette opération terminée, si vous regardez dans le répertoire du projet, vous devriez voir un répertoire correspondant à votre plateforme (`macOS`, `linux`, ou `windows`) qui contient des fichiers supplémentaires. C'est la configuration de l'empaquetage spécifique à la plate-forme pour votre application.

2.4.2 Construire votre application

Vous pouvez maintenant compiler votre application. Cette étape permet d'effectuer toute compilation binaire nécessaire pour que votre application soit exécutable sur votre plate-forme cible.

macOS

Linux

Windows

```
(beeware-venv) $ briefcase build

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

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

Sous macOS, la commande `build` n'a pas besoin de *compiler* quoi que ce soit, mais elle doit signer le contenu du binaire pour qu'il puisse être exécuté. Cette signature est une signature *ad hoc* - elle ne fonctionnera que sur *votre* machine ; si vous voulez distribuer l'application à d'autres, vous devrez fournir une signature complète.

```
(beeware-venv) $ briefcase build

[helloworld] Finalizing application configuration...
Targeting ubuntu:jammy (Vendor base debian)
Determining glibc version... done
Targeting glibc 2.35
Targeting Python3.10

[helloworld] Building application...
Build bootstrap binary...
make: Entering directory '/home/brutus/beeware-tutorial/helloworld/build/linux/ubuntu/
↪jammy/bootstrap'
...
make: Leaving directory '/home/brutus/beeware-tutorial/helloworld/build/linux/ubuntu/
↪jammy/bootstrap'
Building bootstrap binary... done
Installing license... done
Installing changelog... done
Installing man page... done
Updating file permissions... done
Stripping binary... done

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

Une fois cette étape terminée, le dossier `build` contiendra un dossier `helloworld-0.0.1` qui contient un miroir du système de fichiers Linux `/usr`. Ce miroir du système de fichiers contiendra un dossier `bin` avec un binaire `helloworld`, ainsi que les dossiers `lib` et `share` nécessaires pour supporter le binaire.

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

(suite sur la page suivante)

(suite de la page précédente)

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

Sous Windows, la commande `build` n'a pas besoin de *compiler* quoi que ce soit, mais elle doit écrire quelques métadonnées pour que l'application connaisse son nom, sa version, et ainsi de suite.

Déclenchement de l'antivirus

Puisque ces métadonnées sont écrites directement dans le binaire précompilé déployé à partir du modèle pendant la commande `create`, cela peut déclencher un logiciel antivirus fonctionnant sur votre machine et empêcher les métadonnées d'être écrites. Dans ce cas, demandez à l'antivirus d'autoriser l'outil (nommé `rcedit-x64.exe`) à s'exécuter et relancez la commande ci-dessus.

2.4.3 Exécution de l'application

Vous pouvez maintenant utiliser Briefcase pour exécuter votre application :

macOS

Linux

Windows

```
(beeware-venv) $ briefcase run

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

```
(beeware-venv) $ briefcase run

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

(suite sur la page suivante)

(suite de la page précédente)

```
Targeting glibc 2.35
Targeting Python3.10

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

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

[helloworld] Starting app...

================================================================================
Log started: 2023-04-23 04:47:45Z
PreInitializing Python runtime...
PythonHome: C:\Users\brutus\beeware-tutorial\helloworld\windows\app\Hello World\src
PYTHONPATH:
- C:\Users\brutus\beeware-tutorial\helloworld\windows\app\Hello World\src\python39.zip
- C:\Users\brutus\beeware-tutorial\helloworld\windows\app\Hello World\src
- C:\Users\brutus\beeware-tutorial\helloworld\windows\app\Hello World\src\app_packages
- C:\Users\brutus\beeware-tutorial\helloworld\windows\app\Hello World\src\app
Configure argc/argv...
Initializing Python runtime...
Running app module: helloworld
--------------------------------------------------------------------------------
```

Ceci lancera l'exécution de votre application native, en utilisant la sortie de la commande `build`.

Il se peut que vous remarquiez de petites différences dans l'apparence de votre application lorsqu'elle est en cours d'exécution. Par exemple, les icônes et le nom affichés par le système d'exploitation peuvent être légèrement différents de ceux que vous avez vus lors de l'exécution en mode développeur. Cela s'explique également par le fait que vous utilisez l'application packagée et que vous ne vous contentez pas d'exécuter du code Python. Du point de vue du système d'exploitation, vous exécutez maintenant « une application » et non « un programme Python », ce qui se reflète dans l'apparence de l'application.

Before continuing, close the app. As with previous tutorial steps, you can do this by pressing the close button on the application window, by selecting Quit/Exit from the application's menu, or by typing **Ctrl+C** in the terminal where you ran `briefcase dev`.

2.4.4 Construction du programme d'installation

Vous pouvez maintenant empaqueter votre application pour la distribuer, en utilisant la commande `package`. La commande package effectue toutes les compilations nécessaires pour convertir le projet d'échafaudage en un produit final distribuable. Selon la plateforme, cela peut impliquer la compilation d'un installateur, la signature du code, ou d'autres tâches de pré-distribution.

macOS

Linux

Windows

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

[helloworld] Signing app...

************************************************************************
** WARNING: Signing with an ad-hoc identity                           **
************************************************************************

    This app is being signed with an ad-hoc identity. The resulting
    app will run on this computer, but will not run on anyone else's
    computer.

    To generate an app that can be distributed to others, you must
    obtain an application distribution certificate from Apple, and
    select the developer identity associated with that certificate
    when running 'briefcase package'.

************************************************************************

Signing app with ad-hoc identity...
      100.0% • 00:07

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

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

Le dossier `dist` contiendra un fichier nommé `Hello World-0.0.1.dmg`. Si vous localisez ce fichier dans le Finder, et que vous double-cliquez sur son icône, vous monterez le DMG, ce qui vous donnera une copie de l'application Hello World, et un lien vers votre dossier Applications pour faciliter l'installation. Faites glisser le fichier de l'application dans Applications, et vous aurez installé votre application. Envoyez le fichier DMG à un ami, qui devrait pouvoir faire de même.

Dans cet exemple, nous avons utilisé l'option `--adhoc-sign` - c'est-à-dire que nous signons notre application avec des informations d'identification *ad hoc* - des informations d'identification temporaires qui ne fonctionneront que sur votre machine. Nous avons fait cela pour que le tutoriel reste simple. La mise en place d'identités de signature de code est un peu fastidieuse, et elles ne sont *nécessaires* que si vous avez l'intention de distribuer votre application à d'autres personnes. Si nous publiions une application réelle pour que d'autres puissent l'utiliser, nous devrions spécifier de vraies informations d'identification.

Lorsque vous êtes prêt à publier une application réelle, consultez le guide Briefcase How-To sur Setting up a macOS code signing identity

Le résultat de l'étape du paquetage sera légèrement différent selon votre distribution Linux. Si vous êtes sur une distribution dérivée de Debian, vous verrez :

```
(beeware-venv) $ briefcase package

[helloworld] Finalizing application configuration...
Targeting ubuntu:jammy (Vendor base debian)
Determining glibc version... done
Targeting glibc 2.35
Targeting Python3.10

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

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

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

Le dossier `dist` contiendra le fichier `.deb` qui a été généré.

Si vous utilisez une distribution basée sur RHEL, vous verrez :

```
(beeware-venv) $ briefcase package

[helloworld] Finalizing application configuration...
Targeting fedora:40 (Vendor base rhel)
Determining glibc version... done
Targeting glibc 2.39
Targeting Python3.12

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

Write RPM spec file... done

Building source archive... done

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

[helloworld] Packaged dist/helloworld-0.0.1-1.fc40.x86_64.rpm
```

Le dossier `dist` contiendra le fichier `.rpm` qui a été généré.

Si vous êtes sur une distribution basée sur Arch, vous verrez :

```
(beeware-venv) $ briefcase package

[helloworld] Finalizing application configuration...
Targeting arch:20240101 (Vendor base arch)
Determining glibc version... done
Targeting glibc 2.38
Targeting Python3.12
```

(suite sur la page suivante)

(suite de la page précédente)

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

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

Le dossier `dist` contiendra le fichier `.pkg.tar.zst` qui a été généré.

Les autres distributions Linux ne sont actuellement pas prises en charge pour l'empaquetage.

Si vous souhaitez créer un paquet pour une distribution Linux autre que celle que vous utilisez, Briefcase peut également vous aider - mais vous devrez installer Docker.

Des installateurs officiels pour Docker Engine sont disponibles pour une série de distributions Unix. Suivez les instructions correspondant à votre plate-forme, mais assurez-vous de ne pas installer Docker en mode « sans racine ».

Une fois que vous avez installé Docker, vous devriez être en mesure de démarrer un conteneur Linux - par exemple :

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

vous montrera une invite Unix (quelque chose comme `root@84444e31cff9:/#`) à l'intérieur d'un conteneur Docker Ubuntu 22.04. Tapez Ctrl-D pour quitter Docker et revenir à votre shell local.

Une fois que vous avez installé Docker, vous pouvez utiliser Briefcase pour construire un paquet pour n'importe quelle distribution Linux que Briefcase supporte en passant une image Docker comme argument. Par exemple, pour construire un paquet DEB pour Ubuntu 22.04 (Jammy), quel que soit le système d'exploitation sur lequel vous êtes, vous pouvez exécuter :

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

Ceci téléchargera l'image Docker pour le système d'exploitation choisi, créera un conteneur capable d'exécuter les builds de Briefcase, et construira le paquetage de l'application à l'intérieur de l'image. Une fois terminé, le dossier `dist` contiendra le paquet pour la distribution Linux cible.

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

************************************************************************
** WARNING: No signing identity provided                              **
************************************************************************

    Briefcase will not sign the app. To provide a signing identity,
    use the `--identity` option; or, to explicitly disable signing,
    use `--adhoc-sign`.

************************************************************************

[helloworld] Building MSI...
Compiling application manifest...
Compiling... done

Compiling application installer...
helloworld.wxs
helloworld-manifest.wxs
Compiling... done
```

(suite sur la page suivante)

(suite de la page précédente)

```
Linking application installer...
Linking... done

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

Dans cet exemple, nous avons utilisé l'option `--adhoc-sign` - c'est-à-dire que nous signons notre application avec des informations d'identification *ad hoc* - des informations d'identification temporaires qui ne fonctionneront que sur votre machine. Nous avons fait cela pour que le tutoriel reste simple. La mise en place d'identités de signature de code est un peu fastidieuse, et elles ne sont *nécessaires* que si vous avez l'intention de distribuer votre application à d'autres personnes. Si nous publiions une application réelle pour que d'autres puissent l'utiliser, nous devrions spécifier de vraies informations d'identification.

Lorsque vous êtes prêt à publier une application réelle, consultez le guide Briefcase How-To sur Setting up a macOS code signing identity

Une fois cette étape terminée, le dossier *dist* ` contiendra un fichier nommé *Hello_World-0.0.1.msi* `. Si vous double-cliquez sur ce programme d'installation pour le lancer, vous devriez passer par un processus d'installation Windows familier. Une fois l'installation terminée, il y aura une entrée « Hello World » dans votre menu de démarrage.

2.4.5 Étapes suivantes

Notre application est désormais prête à être distribuée sur les plates-formes de bureau. Mais que se passe-t-il lorsque nous devons mettre à jour le code de notre application ? Comment intégrer ces mises à jour dans notre application packagée ? Consultez *Tutoriel 4* pour le découvrir…

2.5 Tutoriel 4 - Mise à jour de votre application

Dans le dernier tutoriel, nous avons packagé notre application en tant qu'application native. Si vous avez affaire à une application réelle, l'histoire ne s'arrêtera pas là : vous ferez probablement des tests, découvrirez des problèmes et devrez apporter des modifications. Même si votre application est parfaite, vous finirez par vouloir publier la version 2 de votre application avec des améliorations.

Alors, comment mettre à jour l'application installée lorsque vous modifiez le code ?

2.5.1 Mise à jour du code de l'application

Notre application imprime actuellement sur la console lorsque vous appuyez sur le bouton. Cependant, les applications d'interface graphique ne devraient pas vraiment utiliser la console pour la sortie. Elles doivent utiliser des boîtes de dialogue pour communiquer avec les utilisateurs.

Ajoutons une boîte de dialogue pour dire bonjour, au lieu d'écrire dans la console. Modifiez le callback `say_hello` pour qu'il ressemble à ceci : :

```
async def say_hello(self, widget):
    await self.main_window.dialog(
        toga.InfoDialog(
            f"Hello, {self.name_input.value}",
            "Hi there!",
        )
    )
```

We need to make the method `async` so that when we display the dialog, the rest of the application continues to run. Don't worry about this detail too much right now - we'll give a more detailed explanation in *Tutorial 8*.

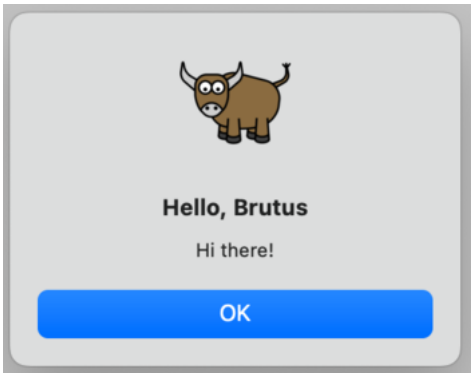
Toga ouvre ainsi une boîte de dialogue modale lorsque le bouton est enfoncé.

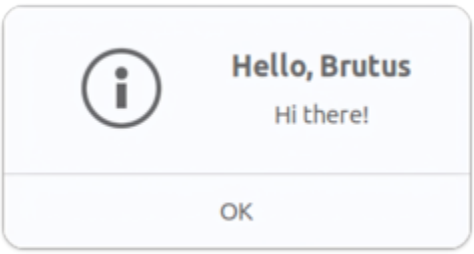
Si vous lancez `briefcase dev`, entrez un nom et appuyez sur le bouton, vous verrez la nouvelle boîte de dialogue :

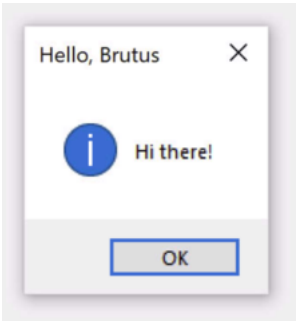
macOS

Linux

Windows







Cependant, si vous exécutez `briefcase run`, la boîte de dialogue n'apparaîtra pas.

Pourquoi ? Eh bien, `briefcase dev` fonctionne en exécutant votre code en place - il essaie de produire un environnement d'exécution aussi réaliste que possible pour votre code, mais il ne fournit ni n'utilise aucune des infrastructures de la plate-forme pour envelopper votre code en tant qu'application. Une partie du processus d'emballage de votre application implique de copier votre code *dans* le paquet d'application - et pour le moment, votre application contient toujours l'ancien code.

Nous devons donc demander à Briefcase de mettre à jour votre application, en copiant la nouvelle version du code. Nous pourrions le faire en supprimant l'ancien répertoire de la plateforme et en repartant de zéro. Cependant, Briefcase offre un moyen plus simple - vous pouvez mettre à jour le code de votre application groupée existante :

macOS

Linux

Windows

```
(beeware-venv) $ briefcase update

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

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

[helloworld] Application updated.
```

```
(beeware-venv) $ briefcase update

[helloworld] Finalizing application configuration...
Targeting ubuntu:jammy (Vendor base debian)
Determining glibc version... done
Targeting glibc 2.35
Targeting Python3.10

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

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


[helloworld] Application updated.
```

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

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

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

[helloworld] Application updated.
```

Si Briefcase ne trouve pas le modèle échafaudé, il invoquera automatiquement `create` pour générer un nouvel échafaudage.

Maintenant que nous avons mis à jour le code de l'installateur, nous pouvons lancer `briefcase build` pour recompiler l'application, `briefcase run` pour exécuter l'application mise à jour, et `briefcase package` pour reconditionner l'application en vue de sa distribution.

(Utilisateurs de macOS, rappelez-vous que comme indiqué dans *Tutoriel 3*, pour le tutoriel nous recommandons d'exécuter `briefcase package` avec le drapeau `--adhoc-sign` pour éviter la complexité de la mise en place d'une identité de signature de code et garder le tutoriel aussi simple que possible)

2.5.2 Mise à jour et exécution en une seule étape

Si vous effectuez rapidement des changements de code, vous voudrez probablement faire un changement de code, mettre à jour l'application et la réexécuter immédiatement. Dans la plupart des cas, le mode développeur (`briefcase dev`) sera le moyen le plus facile de faire ce genre d'itération rapide ; cependant, si vous testez quelque chose sur la façon dont votre application fonctionne en tant que binaire natif, ou si vous chassez un bogue qui ne se manifeste que lorsque votre

application est sous forme de paquetage, vous pouvez avoir besoin d'utiliser des appels répétés à `briefcase run`. Pour simplifier le processus de mise à jour et d'exécution de l'application packagée, Briefcase dispose d'un raccourci pour supporter ce schéma d'utilisation - l'option *-u* (ou *–update*) de la commande `run`.

Essayons d'apporter une autre modification. Vous avez peut-être remarqué que si vous ne tapez pas de nom dans la zone de saisie, la boîte de dialogue dira « Hello, « . Modifions à nouveau la fonction `say_hello` pour gérer ce cas particulier.

En haut du fichier, entre les imports et la définition de `class HelloWorld`, ajoutez une méthode utilitaire pour générer un message d'accueil approprié en fonction de la valeur du nom qui a été fourni : :

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

Ensuite, modifiez le callback `say_hello` pour utiliser cette nouvelle méthode utilitaire : :

```
async def say_hello(self, widget):
    await self.main_window.dialog(
        toga.InfoDialog(
            greeting(self.name_input.value),
            "Hi there!",
        )
    )
```

Exécutez votre application en mode développement (avec `briefcase dev`) pour confirmer que la nouvelle logique fonctionne ; puis mettez à jour, compilez et exécutez l'application avec une seule commande :

macOS

Linux

Windows

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

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

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

[helloworld] Application updated.

[helloworld] Building application...
...

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

[helloworld] Starting app...
```

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

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

(suite sur la page suivante)

(suite de la page précédente)

```
Targeting ubuntu:jammy (Vendor base debian)
Determining glibc version... done
Targeting glibc 2.35
Targeting Python3.10

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

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

[helloworld] Application updated.

[helloworld] Building application...
...

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

[helloworld] Starting app...
```

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

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

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

[helloworld] Application updated.

[helloworld] Starting app...
```

La commande package accepte aussi l'argument `-u`, donc si vous faites un changement dans le code de votre application et que vous voulez repackager immédiatement, vous pouvez lancer `briefcase package -u`.

2.5.3 Étapes suivantes

Notre application est désormais prête à être distribuée sur les plates-formes de bureau et nous avons pu mettre à jour le code de notre application.

Mais qu'en est-il de l'application mobile ? Dans *Tutoriel 5*, nous allons convertir notre application en une application mobile, et la déployer sur un simulateur d'appareil, et sur un téléphone.

2.6 Tutoriel 5 - Rendre votre application mobile

Jusqu'à présent, nous avons exécuté et testé notre application sur un ordinateur de bureau. Cependant, BeeWare prend également en charge les plates-formes mobiles - et l'application que nous avons écrite peut également être déployée sur votre appareil mobile !

iOS Les applications iOS ne peuvent être compilées que sur macOS.

Tutoriel 5 - Le mobile : iOS

Android Les applications Android peuvent être compilées sur macOS, Windows ou Linux.

Tutoriel 5 - Le mobile : Android

2.6.1 Tutoriel 5 - Le mobile : iOS

To compile iOS applications we'll need Xcode, which is available for free from the macOS App Store. Once Xcode is installed, launch it from Applications and accept the Xcode License Agreement. Next, Xcode will show which components are built-in, and which components you may download. Select the checkbox next to iOS (followed by the current version number), and then click « Download and Install » to install it. Xcode will install the system components, followed by the iOS Simulator.

Une fois Xcode installé, nous pouvons prendre notre application et la déployer en tant qu'application iOS.

Le processus de déploiement d'une application sur iOS est très similaire au processus de déploiement d'une application de bureau. Tout d'abord, vous exécutez la commande `create` - mais cette fois, nous spécifions que nous voulons créer une application iOS :

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

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-iOS-Xcode-template.git, branch␣
↪v0.3.18
...

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

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

[helloworld] Installing requirements...
...

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

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

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

Une fois cette opération terminée, nous aurons un répertoire `build/helloworld/ios/xcode` contenant un projet Xcode, ainsi que les bibliothèques de support et le code nécessaire à l'application.

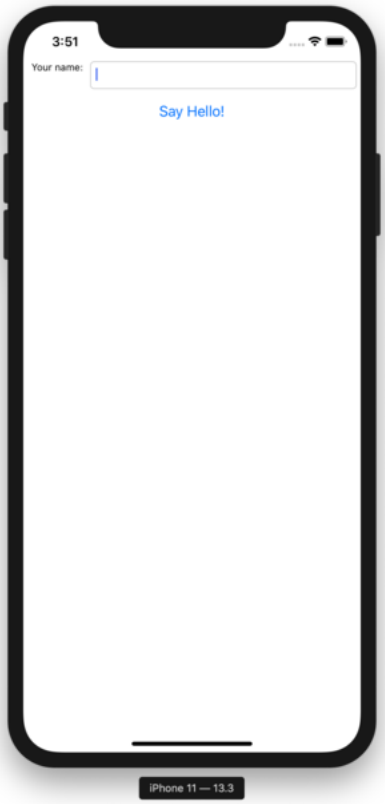
Vous pouvez ensuite utiliser Briefcase pour compiler votre application en utilisant `briefcase build iOS` :

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

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

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

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

Nous sommes maintenant prêts à exécuter notre application, en utilisant `briefcase run iOS`. Il vous sera demandé de sélectionner un appareil pour lequel compiler ; si vous avez installé des simulateurs pour plusieurs versions du SDK iOS, il vous sera peut-être demandé quelle version d'iOS vous voulez cibler. Les options affichées peuvent différer de celles présentées dans cette sortie - au moins, la liste des appareils sera probablement différente. En ce qui nous concerne, le simulateur choisi n'a pas d'importance.

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

Select simulator device:

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

>  10

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

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

or:

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

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

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

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

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

Ceci lancera le simulateur iOS, installera votre application et la démarrera. Vous devriez voir le simulateur démarrer, puis ouvrir votre application iOS :



While the app is running, you'll see a series of Simulator log output messages in the console. Typing Ctrl+C into the terminal will halt the messages in the console, but it will not close the simulator. This is so you can test new changes without restarting the simulator.

Si vous savez à l'avance quel simulateur iOS vous voulez cibler, vous pouvez dire à Briefcase d'utiliser ce simulateur en fournissant une option `-d` (ou `--device`). En utilisant le nom de l'appareil que vous avez sélectionné lorsque vous avez créé votre application, exécutez :

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

Si vous avez plusieurs versions iOS disponibles, Briefcase choisira la version iOS la plus élevée ; si vous voulez choisir une version iOS particulière, vous lui direz d'utiliser cette version spécifique :

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

Vous pouvez également nommer un appareil spécifique UDID :

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

Étapes suivantes

Nous avons maintenant une application sur notre téléphone ! Y a-t-il un autre endroit où nous pouvons déployer une application BeeWare ? Consultez *Tutoriel 6* pour le savoir…

2.6.2 Tutoriel 5 - Le mobile : Android

Nous allons maintenant prendre notre application et la déployer en tant qu'application Android.

Le processus de déploiement d'une application sur Android est très similaire au processus de déploiement d'une application de bureau. Briefcase prend en charge l'installation des dépendances pour Android, y compris le SDK Android, l'émulateur Android et un compilateur Java.

Créer une application Android et la compiler

Tout d'abord, lancez la commande `create`. Celle-ci télécharge un modèle d'application Android et y ajoute votre code Python.

macOS

Linux

Windows

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

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-android-gradle-template.git,␣
↪branch v0.3.18
...

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

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

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

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

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

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

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

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-android-gradle-template.git,␣
↪branch v0.3.18
...

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

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

(suite sur la page suivante)

(suite de la page précédente)

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

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

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

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

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

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-android-gradle-template.git,␣
↪branch v0.3.18
...

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

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

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

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

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

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

Lorsque vous lancez `briefcase create android` pour la première fois, Briefcase télécharge un JDK Java et le SDK Android. La taille des fichiers et le temps de téléchargement peuvent être considérables ; cela peut prendre un certain temps (10 minutes ou plus, selon la vitesse de votre connexion Internet). Une fois le téléchargement terminé, vous serez invité à accepter la licence Android SDK de Google.

Une fois cette opération terminée, nous aurons un répertoire `buildhelloworld\android\gradle` dans notre projet, qui contiendra un projet Android avec une configuration de construction Gradle. Ce projet contiendra le code de votre application, ainsi qu'un package de support contenant l'interpréteur Python.

Nous pouvons ensuite utiliser la commande `build` de Briefcase pour compiler ce fichier dans un fichier d'application APK Android.

macOS

Linux

Windows

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

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

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

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

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

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

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

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

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

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

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

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

Gradle peut sembler bloqué

Pendant l'étape `briefcase build android`, Gradle (l'outil de construction de la plateforme Android) affiche `CONFIGURING : 100%`, et semble ne rien faire. Ne vous inquiétez pas, il n'est pas bloqué - il est en train de télécharger plus de composants du SDK Android. Selon la vitesse de votre connexion Internet, cela peut prendre encore 10 minutes (ou plus). Ce décalage ne devrait se produire que la première fois que vous lancez `build`; les outils

sont mis en cache, et lors de votre prochain build, les versions mises en cache seront utilisées.

Exécuter l'application sur un appareil virtuel

Nous sommes maintenant prêts à exécuter notre application. Vous pouvez utiliser la commande `run` de Briefcase pour exécuter l'application sur un appareil Android. Commençons par l'exécuter sur un émulateur Android.

Pour lancer votre application, exécutez `briefcase run android`. Vous obtiendrez alors une liste d'appareils sur lesquels vous pouvez faire fonctionner l'application. Le dernier élément sera toujours une option pour créer un nouvel émulateur Android.

macOS

Linux

Windows

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

Select device:

  1) Create a new Android emulator

>
```

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

Select device:

  1) Create a new Android emulator

>
```

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

Select device:

  1) Create a new Android emulator

>
```

Nous pouvons maintenant choisir l'appareil que nous souhaitons. Sélectionnez l'option « Créer un nouvel émulateur Android », et acceptez le choix par défaut du nom de l'appareil (`beePhone`).

Briefcase `run` démarrera automatiquement l'appareil virtuel. Lorsque l'appareil démarre, vous verrez le logo Android :

Une fois que l'appareil a fini de démarrer, Briefcase installera votre application sur l'appareil. Vous verrez brièvement un écran de lancement :

L'application démarre alors. Vous verrez un écran de démarrage pendant que l'application démarre :

L'émulateur n'a pas démarré !

L'émulateur Android est un logiciel complexe qui s'appuie sur un certain nombre de caractéristiques du matériel et du système d'exploitation - des caractéristiques qui peuvent ne pas être disponibles ou activées sur des machines plus

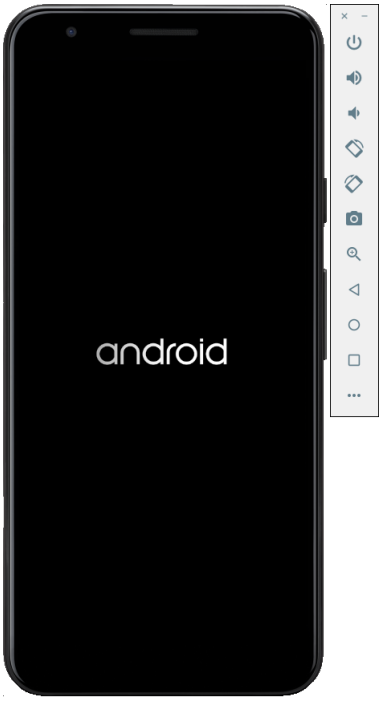


FIG. 1 – Démarrage d'un dispositif virtuel Android

FIG. 2 – Appareil virtuel Android entièrement démarré, sur l'écran du lanceur

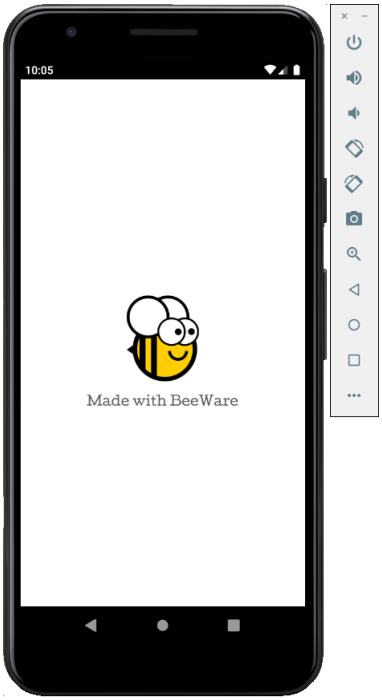


Fig. 3 – Écran d'accueil de l'application

anciennes. Si vous rencontrez des difficultés au démarrage de l'émulateur Android, consultez la section « Exigences et recommandations <https://developer.android.com/studio/run/emulator#requirements>`__ » de la documentation destinée aux développeurs Android.

La première fois que l'application démarre, elle doit se décompresser sur l'appareil. Cela peut prendre quelques secondes. Une fois qu'elle est décompressée, vous verrez la version Android de notre application de bureau :

Si vous ne voyez pas votre application se lancer, vous devrez peut-être vérifier le terminal où vous avez lancé `briefcase run` et rechercher les messages d'erreur.

While the app is running, you'll see a lot of messages being streamed in the console. This is a stream of the application's logs from the emulator. Typing Ctrl+C into the terminal will halt the streamed information in the console, but it will not close the emulator. This is so you can test new changes without restarting the emulator.

A l'avenir, si vous souhaitez utiliser cet appareil sans utiliser le menu, vous pouvez fournir le nom de l'émulateur à Briefcase, en utilisant `briefcase run android -d @beePhone` pour utiliser directement l'appareil virtuel.

Exécuter l'application sur un appareil physique

Si vous avez un téléphone ou une tablette Android physique, vous pouvez le connecter à votre ordinateur à l'aide d'un câble USB, puis utiliser la mallette pour cibler votre appareil physique.

Android exige que vous prépariez votre appareil avant de pouvoir l'utiliser pour le développement. Vous devrez apporter deux modifications aux options de votre appareil :

- Activer les options pour les développeurs
- Activer le débogage USB

Les détails sur la manière d'effectuer ces changements peuvent être trouvés dans la documentation du développeur Android <https://developer.android.com/studio/debug/dev-options#enable>`__.

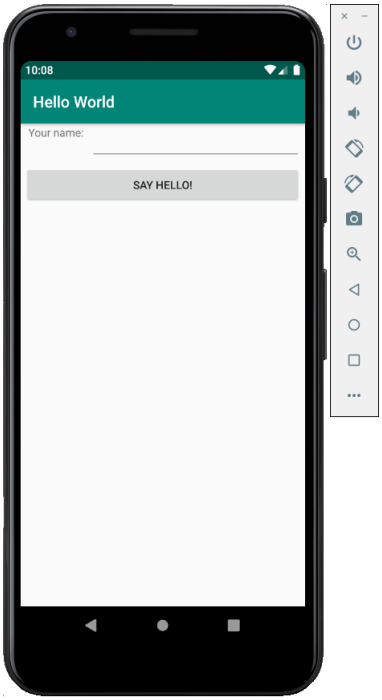


FIG. 4 – Lancement de l'application de démonstration

Une fois ces étapes terminées, votre appareil devrait apparaître dans la liste des appareils disponibles lorsque vous lancez `briefcase run android`.

macOS

Linux

Windows

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

Select device:

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

>
```

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

Select device:

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

>
```

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

Select device:

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

>
```

Ici, nous pouvons voir un nouvel appareil physique avec son numéro de série sur la liste de déploiement - dans ce cas, un Pixel 3a. À l'avenir, si vous souhaitez exécuter sur cet appareil sans utiliser le menu, vous pouvez fournir le numéro de série du téléphone à Briefcase (dans ce cas, `briefcase run android -d 94ZZY0LNE8`). Cela lancera l'application directement sur l'appareil, sans l'inviter à le faire.

Mon appareil n'apparaît pas !

Si votre appareil n'apparaît pas du tout dans cette liste, c'est que vous n'avez pas activé le débogage USB (ou que l'appareil n'est pas branché !).

Si votre appareil apparaît, mais qu'il est listé comme « Unknown device (not authorized for development) », le mode développeur n'a pas été correctement activé. Réexécutez les étapes pour activer les options de développement, et réexécutez `briefcase run android`.

Étapes suivantes

Nous avons maintenant une application sur notre téléphone ! Y a-t-il un autre endroit où nous pouvons déployer une application BeeWare ? Consultez *Tutoriel 6* pour le savoir…

2.7 Tutoriel 6 - Mettez-le sur le web !

En plus de prendre en charge les plateformes mobiles, la boîte à outils Toga widget prend également en charge le web ! En utilisant la même API que celle utilisée pour déployer vos applications de bureau et mobiles, vous pouvez déployer votre application en tant qu'application web à page unique.

Preuve de concept

Le backend de Toga Web est le moins mature de tous les backends de Toga. Il est suffisamment mature pour présenter quelques fonctionnalités, mais il est probable qu'il soit bogué et qu'il manque de nombreux widgets disponibles sur d'autres plates-formes. À ce stade, le déploiement sur le Web doit être considéré comme une « preuve de concept » - suffisante pour démontrer ce qui peut être fait, mais pas assez pour être utilisée pour un développement sérieux.

Si vous avez des problèmes avec cette étape du tutoriel, vous pouvez passer à la page suivante.

2.7.1 Déploiement en tant qu'application web

Le processus de déploiement en tant qu'application web à page unique suit le même schéma familier - vous créez l'application, vous la construisez, puis vous l'exécutez. Cependant, Briefcase peut être un peu plus intelligent ; si vous essayez d'exécuter une application et que Briefcase détermine qu'elle n'a pas été créée ou construite pour la plateforme ciblée, il effectuera les étapes de création et de construction pour vous. Puisque c'est la première fois que nous lançons l'application pour le web, nous pouvons exécuter les trois étapes en une seule commande :

macOS

Linux

Windows

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

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-web-static-template.git, branch␣
↪v0.3.18
...

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

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

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

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

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

[helloworld] Created build/helloworld/web/static

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

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

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

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

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

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-web-static-template.git, branch␣
↪v0.3.18
...

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

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

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

(suite sur la page suivante)

(suite de la page précédente)

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

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

[helloworld] Created build/helloworld/web/static

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

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

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

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

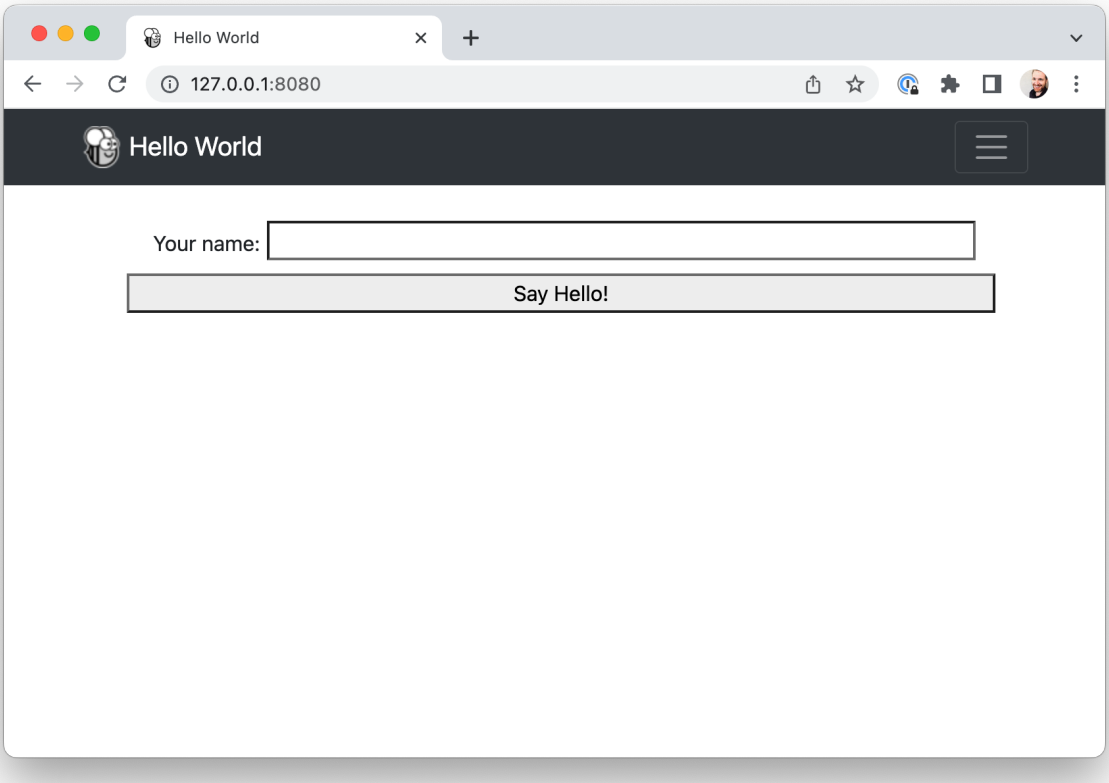
```
(beeware-venv) C:\...>briefcase run web

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-web-static-template.git, branch␣
↪v0.3.18
...

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

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

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

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

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

[helloworld] Created build\helloworld\web\static

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

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

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

(suite sur la page suivante)

(suite de la page précédente)

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

Cela ouvrira un navigateur web, pointant sur http://127.0.0.1:8080 :



Si vous entrez votre nom et cliquez sur le bouton, une boîte de dialogue apparaît.

2.7.2 Comment ça marche ?

Cette application web est un site web statique - une seule page source HTML, avec quelques feuilles de style CSS et d'autres ressources. Briefcase a démarré un serveur web local pour servir cette page afin que votre navigateur puisse la voir. Si vous voulez mettre cette page web en production, vous pouvez copier le contenu du dossier `www` sur n'importe quel serveur web qui peut servir du contenu statique.

Mais lorsque vous appuyez sur le bouton, vous exécutez du code Python… Comment cela fonctionne-t-il ? Toga utilise PyScript pour fournir un interprète Python dans le navigateur. Briefcase présente le code de votre application sous forme de roues que PyScript peut charger dans le navigateur. Lorsque la page est chargée, le code de l'application s'exécute dans le navigateur, construisant l'interface utilisateur en utilisant le DOM du navigateur. Lorsque vous cliquez sur un bouton, ce bouton exécute le code de gestion des événements dans le navigateur.

2.7.3 Étapes suivantes

Bien que nous ayons maintenant déployé cette application sur les ordinateurs de bureau, les téléphones portables et le web, l'application est assez simple et n'implique pas de bibliothèques tierces. Pouvons-nous inclure des bibliothèques du Python Package Index (PyPI) dans notre application ? Consultez *Tutoriel 7* pour le savoir…

2.8 Tutoriel n° 7 - Démarrer cette (troisième) partie

Jusqu'à présent, l'application que nous avons construite n'a utilisé que notre propre code, ainsi que le code fourni par BeeWare. Cependant, dans une application réelle, vous voudrez probablement utiliser une bibliothèque tierce, téléchargée à partir du Python Package Index (PyPI).

Modifions notre application pour y inclure une bibliothèque tierce.

2.8.1 Adding a package

Let's modify our application to say a little bit more than just « Hi, there ! ».

To generate some more interesting text for the dialog, we're going to use a library called Faker . Faker is a Python package that generates fake content, including names and text blocks. The names and words in the text block are generated from an arbitrary list of words provided by Faker. We're going to use Faker to construct a fake message, as if someone is responding to the user.

Ajoutons un appel API `httpx` à notre application. Ajoutez un import au début de `app.py` pour importer `httpx` : :

```
import faker
```

Ensuite, modifiez le callback `say_hello()` pour qu'il ressemble à ceci : :

```
async def say_hello(self, widget):
    fake = faker.Faker()
    await self.main_window.dialog(
        toga.InfoDialog(
            greeting(self.name_input.value),
            f"A message from {fake.name()}: {fake.text()}",
        )
    )
```

Exécutons notre application mise à jour dans le mode développeur de Briefcase pour vérifier que notre changement a fonctionné.

macOS

Linux

Windows

Android

iOS

```
(beeware-venv) $ briefcase dev
Traceback (most recent call last):
File ".../venv/bin/briefcase", line 5, in <module>
    from briefcase.__main__ import main
File ".../venv/lib/python3.13/site-packages/briefcase/__main__.py", line 3, in <module>
    from .cmdline import parse_cmdline
File ".../venv/lib/python3.13/site-packages/briefcase/cmdline.py", line 6, in <module>
```

(suite sur la page suivante)

(suite de la page précédente)

```
    from briefcase.commands import DevCommand, NewCommand, UpgradeCommand
File ".../venv/lib/python3.13/site-packages/briefcase/commands/__init__.py", line 1, in
↪<module>
    from .build import BuildCommand  # noqa
File ".../venv/lib/python3.13/site-packages/briefcase/commands/build.py", line 5, in
↪<module>
    from .base import BaseCommand, full_options
File ".../venv/lib/python3.13/site-packages/briefcase/commands/base.py", line 14, in
↪<module>
    import faker
ModuleNotFoundError: No module named 'faker'
```

```
(beeware-venv) $ briefcase dev
Traceback (most recent call last):
File ".../venv/bin/briefcase", line 5, in <module>
    from briefcase.__main__ import main
File ".../venv/lib/python3.13/site-packages/briefcase/__main__.py", line 3, in <module>
    from .cmdline import parse_cmdline
File ".../venv/lib/python3.13/site-packages/briefcase/cmdline.py", line 6, in <module>
    from briefcase.commands import DevCommand, NewCommand, UpgradeCommand
File ".../venv/lib/python3.13/site-packages/briefcase/commands/__init__.py", line 1, in
↪<module>
    from .build import BuildCommand  # noqa
File ".../venv/lib/python3.13/site-packages/briefcase/commands/build.py", line 5, in
↪<module>
    from .base import BaseCommand, full_options
File ".../venv/lib/python3.13/site-packages/briefcase/commands/base.py", line 14, in
↪<module>
    import faker
ModuleNotFoundError: No module named 'faker'
```

```
(beeware-venv) C:\...>briefcase dev
Traceback (most recent call last):
File "...\venv\bin\briefcase", line 5, in <module>
    from briefcase.__main__ import main
File "...\venv\lib\python3.13\site-packages\briefcase\__main__.py", line 3, in <module>
    from .cmdline import parse_cmdline
File "...\venv\lib\python3.13\site-packages\briefcase\cmdline.py", line 6, in <module>
    from briefcase.commands import DevCommand, NewCommand, UpgradeCommand
File "...\venv\lib\python3.13\site-packages\briefcase\commands\__init__.py", line 1, in
↪<module>
    from .build import BuildCommand  # noqa
File "...\venv\lib\python3.13\site-packages\briefcase\commands\build.py", line 5, in
↪<module>
    from .base import BaseCommand, full_options
File "...\venv\lib\python3.13\site-packages\briefcase\commands\base.py", line 14, in
↪<module>
    import faker
ModuleNotFoundError: No module named 'faker'
```

You can't run an Android app in developer mode - use the instructions for your chosen desktop platform.

You can't run an iOS app in developer mode - use the instructions for your chosen desktop platform.

Qu'est-ce qui s'est passé ? Nous avons ajouté `httpx` à notre *code*, mais nous ne l'avons pas ajouté à notre environnement virtuel de développement. Nous pouvons corriger cela en installant `httpx` avec `pip`, puis en relançant `briefcase dev` :

macOS

Linux

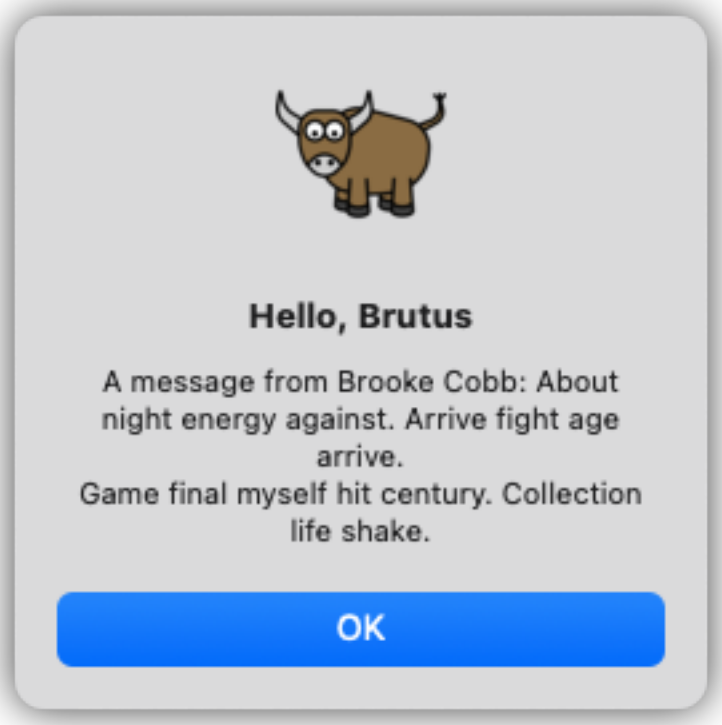
Windows

Android

iOS

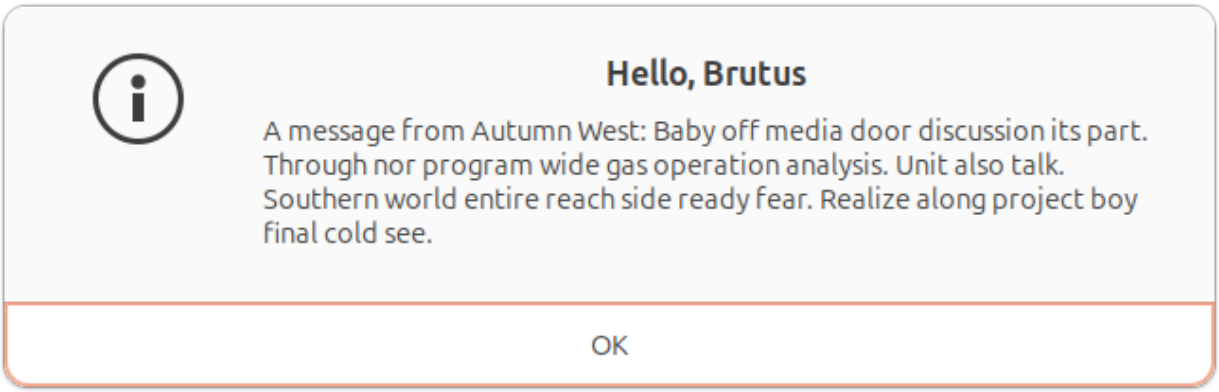
```
(beeware-venv) $ python -m pip install faker
(beeware-venv) $ briefcase dev
```

Lorsque vous entrez un nom et que vous appuyez sur le bouton, une boîte de dialogue doit s'afficher :



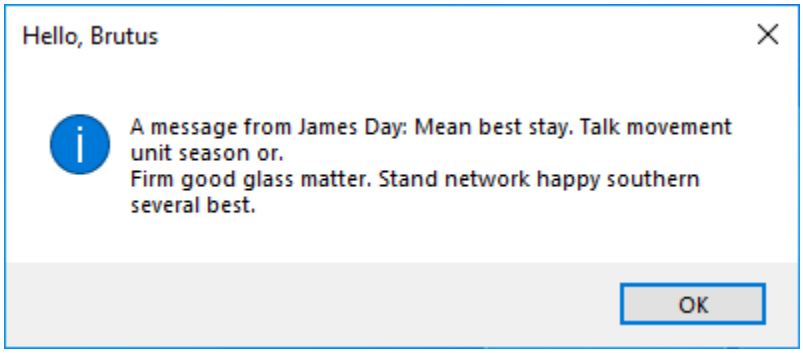

```
(beeware-venv) $ python -m pip install faker
(beeware-venv) $ briefcase dev
```

Lorsque vous entrez un nom et que vous appuyez sur le bouton, une boîte de dialogue doit s'afficher :




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

Lorsque vous entrez un nom et que vous appuyez sur le bouton, une boîte de dialogue doit s'afficher :



You can't run an Android app in developer mode - use the instructions for your chosen desktop platform.

You can't run an iOS app in developer mode - use the instructions for your chosen desktop platform.

Nous avons maintenant une application fonctionnelle, utilisant une bibliothèque tierce, fonctionnant en mode développement !

2.8.2 Exécution de l'application mise à jour

Nous allons faire en sorte que ce code d'application mis à jour soit empaqueté en tant qu'application autonome. Puisque nous avons modifié le code, nous devons suivre les mêmes étapes que dans *Tutoriel 4* :

macOS

Linux

Windows

Android

iOS

Mettre à jour le code dans l'application packagée :

```
(beeware-venv) $ briefcase update

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

(suite sur la page suivante)

(suite de la page précédente)

```
...

[helloworld] Application updated.
```

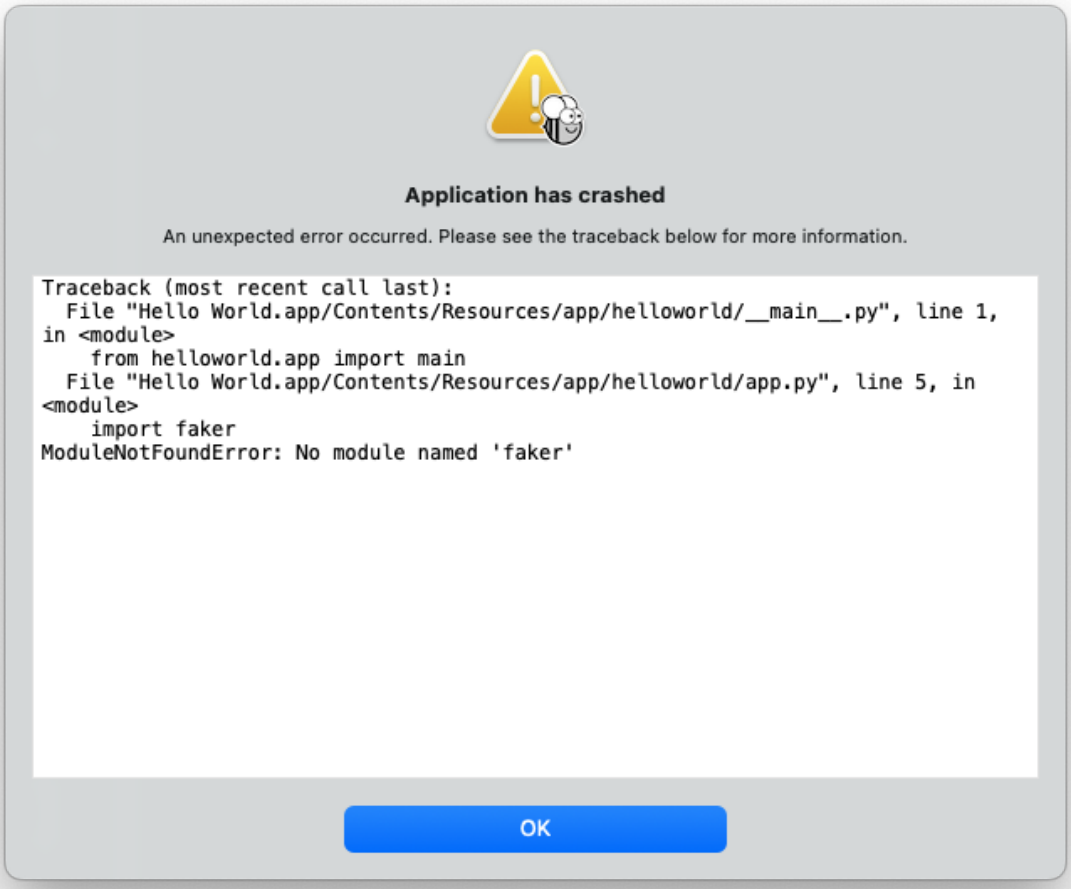
Reconstruire l'application :

```
(beeware-venv) $ briefcase build

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

Enfin, lancez l'application :

```
(beeware-venv) $ briefcase run

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

Cependant, lorsque l'application s'exécute, vous verrez une erreur dans la console, ainsi qu'une boîte de dialogue de plantage :



Mettre à jour le code dans l'application packagée :

```
(beeware-venv) $ briefcase update

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

[helloworld] Application updated.
```

Reconstruire l'application :

```
(beeware-venv) $ briefcase build

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

[helloworld] Building application...
...

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

(suite sur la page suivante)

(suite de la page précédente)

```
↪helloworld
```

Enfin, lancez l'application :

```
(beeware-venv) $ briefcase run

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

Cependant, lorsque l'application s'exécute, une erreur apparaît dans la console :

```
Traceback (most recent call last):
  File "/usr/lib/python3.13/runpy.py", line 194, in _run_module_as_main
    return _run_code(code, main_globals, None,
  File "/usr/lib/python3.13/runpy.py", line 87, in _run_code
    exec(code, run_globals)
  File "/home/brutus/beeware-tutorial/helloworld/build/linux/ubuntu/jammy/helloworld-0.0.
↪1/usr/app/hello_world/__main__.py", line 1, in <module>
    from helloworld.app import main
  File "/home/brutus/beeware-tutorial/helloworld/build/linux/ubuntu/jammy/helloworld-0.0.
↪1/usr/app/hello_world/app.py", line 8, in <module>
    import faker
ModuleNotFoundError: No module named 'faker'

Unable to start app helloworld.
```

Mettre à jour le code dans l'application packagée :

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

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

[helloworld] Application updated.
```

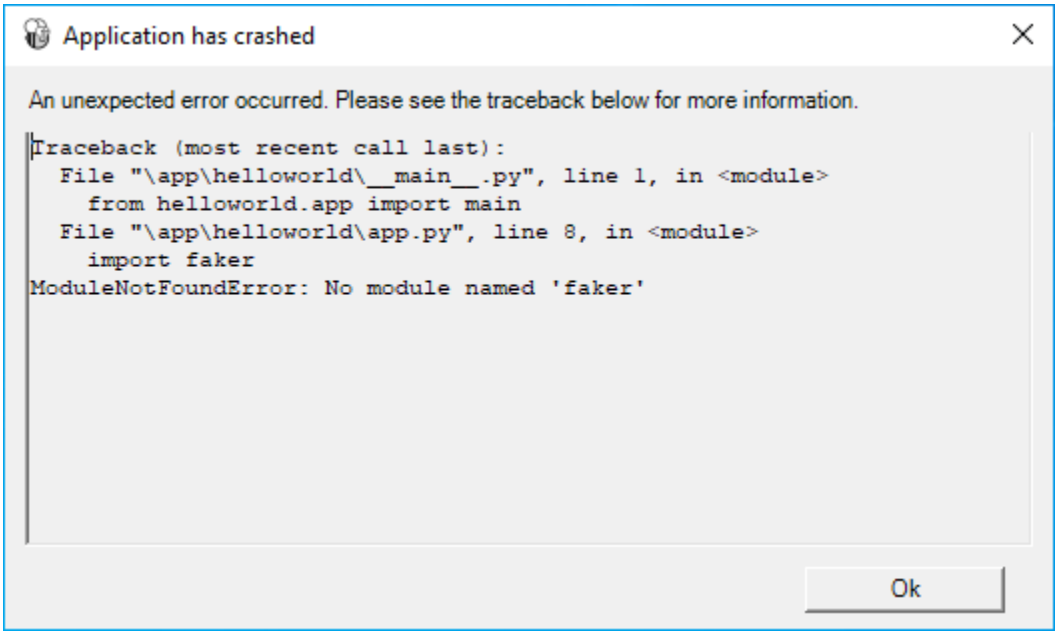
Reconstruire l'application :

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

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

Enfin, lancez l'application :

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

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

Cependant, lorsque l'application s'exécute, vous verrez une erreur dans la console, ainsi qu'une boîte de dialogue de plantage :



Mettre à jour le code dans l'application packagée :

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

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

[helloworld] Application updated.
```

Reconstruire l'application :

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

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

And finally, run the app (selecting a simulator when prompted) :

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

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

Cependant, lorsque l'application s'exécute, une erreur apparaît dans la console :

```
--------- beginning of crash
E/AndroidRuntime: FATAL EXCEPTION: main
E/AndroidRuntime: Process: com.example.helloworld, PID: 8289
E/AndroidRuntime: java.lang.RuntimeException: Unable to start activity ComponentInfo{com.
↪example.helloworld/org.beeware.android.MainActivity}: com.chaquo.python.PyException:␣
↪ModuleNotFoundError: No module named 'faker'
E/AndroidRuntime:   at android.app.ActivityThread.performLaunchActivity(ActivityThread.
```

(suite sur la page suivante)

(suite de la page précédente)

```
↪java:3635)
E/AndroidRuntime:   at android.app.ActivityThread.handleLaunchActivity(ActivityThread.
↪java:3792)
E/AndroidRuntime:   at android.app.servertransaction.LaunchActivityItem.
↪execute(LaunchActivityItem.java:103)
E/AndroidRuntime:   at android.app.servertransaction.TransactionExecutor.
↪executeCallbacks(TransactionExecutor.java:135)
E/AndroidRuntime:   at android.app.servertransaction.TransactionExecutor.
↪execute(TransactionExecutor.java:95)
E/AndroidRuntime:   at android.app.ActivityThread$H.handleMessage(ActivityThread.
↪java:2210)
E/AndroidRuntime:   at android.os.Handler.dispatchMessage(Handler.java:106)
E/AndroidRuntime:   at android.os.Looper.loopOnce(Looper.java:201)
E/AndroidRuntime:   at android.os.Looper.loop(Looper.java:288)
E/AndroidRuntime:   at android.app.ActivityThread.main(ActivityThread.java:7839)
E/AndroidRuntime:   at java.lang.reflect.Method.invoke(Native Method)
E/AndroidRuntime:   at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.
↪run(RuntimeInit.java:548)
E/AndroidRuntime:   at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:1003)
E/AndroidRuntime: Caused by: com.chaquo.python.PyException: ModuleNotFoundError: No␣
↪module named 'faker'
E/AndroidRuntime:   at <python>.helloworld.app.<module>(app.py:8)
E/AndroidRuntime:   at <python>.java.chaquopy.import_override(import.pxi:60)
E/AndroidRuntime:   at <python>.__main__.<module>(__main__.py:1)
E/AndroidRuntime:   at <python>.runpy._run_code(<frozen runpy>:88)
E/AndroidRuntime:   at <python>.runpy._run_module_code(<frozen runpy>:98)
E/AndroidRuntime:   at <python>.runpy.run_module(<frozen runpy>:226)
E/AndroidRuntime:   at <python>.chaquopy_java.call(chaquopy_java.pyx:352)
E/AndroidRuntime:   at <python>.chaquopy_java.Java_com_chaquo_python_PyObject_
↪callAttrThrowsNative(chaquopy_java.pyx:324)
E/AndroidRuntime:   at com.chaquo.python.PyObject.callAttrThrowsNative(Native Method)
E/AndroidRuntime:   at com.chaquo.python.PyObject.callAttrThrows(PyObject.java:232)
E/AndroidRuntime:   at com.chaquo.python.PyObject.callAttr(PyObject.java:221)
E/AndroidRuntime:   at org.beeware.android.MainActivity.onCreate(MainActivity.java:85)
E/AndroidRuntime:   at android.app.Activity.performCreate(Activity.java:8051)
E/AndroidRuntime:   at android.app.Activity.performCreate(Activity.java:8031)
E/AndroidRuntime:   at android.app.Instrumentation.callActivityOnCreate(Instrumentation.
↪java:1329)
E/AndroidRuntime:   at android.app.ActivityThread.performLaunchActivity(ActivityThread.
↪java:3608)
E/AndroidRuntime:   ... 12 more
I/Process : Sending signal. PID: 8289 SIG: 9
```

Mettre à jour le code dans l'application packagée :

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

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

[helloworld] Application updated.
```

Reconstruire l'application :

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

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

And finally, run the app (selecting a simulator when prompted) :

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

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

Cependant, lorsque l'application s'exécute, une erreur apparaît dans la console :

```
Application has crashed!
========================
Traceback (most recent call last):
  File "/Users/rkm/Library/Developer/CoreSimulator/Devices/FD7EA28A-6D72-4064-9D8A-
↪53CC8308BB6F/data/Containers/Bundle/Application/D9DD590B-DA32-4EE1-8F78-78658379CAB7/
↪Hello World.app/app/helloworld/__main__.py", line 1, in <module>
    from helloworld.app import main
  File "/Users/rkm/Library/Developer/CoreSimulator/Devices/FD7EA28A-6D72-4064-9D8A-
↪53CC8308BB6F/data/Containers/Bundle/Application/D9DD590B-DA32-4EE1-8F78-78658379CAB7/
↪Hello World.app/app/helloworld/app.py", line 8, in <module>
    import faker
ModuleNotFoundError: No module named 'faker'
```

Une fois de plus, l'application n'a pas pu démarrer parce que `httpx` a été installé - mais pourquoi ? N'avons-nous pas déjà installé *httpx* ?

Nous l'avons fait, mais uniquement dans l'environnement de développement. Votre environnement de développement est entièrement local à votre machine - et n'est activé que lorsque vous l'activez explicitement. Bien que Briefcase dispose d'un mode de développement, la principale raison pour laquelle vous utilisez Briefcase est d'empaqueter votre code afin de le donner à quelqu'un d'autre.

La seule façon de garantir que quelqu'un d'autre disposera d'un environnement Python contenant tout ce dont il a besoin est de construire un environnement Python complètement isolé. Cela signifie qu'il y a une installation Python complètement isolée, et un ensemble de dépendances complètement isolé. C'est ce que Briefcase construit quand vous lancez `briefcase build` - un environnement Python isolé. Cela explique aussi pourquoi `httpx` n'est pas installé - il a été installé dans votre environnement de *développement*, mais pas dans l'application packagée.

Nous devons donc indiquer à Briefcase que notre application a une dépendance externe.

2.8.3 Mise à jour des dépendances

Dans le répertoire racine de votre application, il y a un fichier nommé `pyproject.toml`. Ce fichier contient tous les détails de configuration de l'application que vous avez fournis lorsque vous avez lancé `briefcase new`.

`pyproject.toml` est divisé en sections ; l'une d'entre elles décrit les paramètres de votre application : :

```
[tool.briefcase.app.helloworld]
formal_name = "Hello World"
description = "A Tutorial app"
```

(suite sur la page suivante)

(suite de la page précédente)

```
long_description = """More details about the app should go here.
"""
sources = ["src/helloworld"]
requires = []
```

L'option `requires` décrit les dépendances de notre application. C'est une liste de chaînes de caractères, spécifiant les bibliothèques (et, optionnellement, les versions) des bibliothèques que vous voulez inclure dans votre application.

Modifiez le paramètre `requires` de façon à ce qu'il se lise : :

```
requires = [
    "faker",
]
```

En ajoutant ce paramètre, nous disons à Briefcase « lorsque vous compilez mon application, lancez `pip install httpx` dans le bundle de l'application ». Tout ce qui serait une entrée légale pour *pip install* ` peut être utilisé ici - ainsi, vous pourriez spécifier :

- Une version spécifique de la bibliothèque (par exemple, `"httpx==0.19.0"`);
- Une gamme de versions de la bibliothèque (par exemple, `"httpx>=0.19"`);
- Un chemin vers un dépôt git (par exemple, `"git+https://github.com/encode/httpx"`); ou
- Un chemin d'accès à un fichier local (Cependant, attention : si vous donnez votre code à quelqu'un d'autre, ce chemin d'accès n'existera probablement pas sur sa machine !)

Plus loin dans `pyproject.toml`, vous remarquerez d'autres sections qui dépendent du système d'exploitation, comme `[tool.briefcase.app.helloworld.macOS]` et `[tool.briefcase.app.helloworld.windows]`. Ces sections ont *également* un paramètre `requires`. Ces paramètres vous permettent de définir des dépendances supplémentaires spécifiques à une plate-forme - ainsi, par exemple, si vous avez besoin d'une bibliothèque spécifique à une plate-forme pour gérer un aspect de votre application, vous pouvez spécifier cette bibliothèque dans la section `requires'' spécifique à la plate-forme, et ce paramètre ne sera utilisé que pour cette plate-forme. Vous remarquerez que les bibliothèques ``toga` sont toutes spécifiées dans la section `requires` spécifique à la plate-forme - c'est parce que les bibliothèques nécessaires pour afficher une interface utilisateur sont spécifiques à la plate-forme.

Dans notre cas, nous voulons que `httpx` soit installé sur toutes les plateformes, donc nous utilisons le paramètre `requires` au niveau de l'application. Les dépendances au niveau de l'application seront toujours installées ; les dépendances spécifiques à la plate-forme sont installées *en plus* de celles au niveau de l'application.

Maintenant que nous avons informé Briefcase de nos exigences supplémentaires, nous pouvons essayer d'empaqueter à nouveau notre application. Assurez-vous que vous avez sauvegardé vos changements dans `pyproject.toml`, puis mettez à jour votre application à nouveau - cette fois-ci, en passant le drapeau `-r`. Cela indique à Briefcase de mettre à jour les exigences dans l'application packagée :

macOS

Linux

Windows

Android

iOS

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

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

(suite sur la page suivante)

(suite de la page précédente)

```
[helloworld] Updating requirements...
Collecting faker
  Using cached faker-37.3.0-py3-none-any.whl.metadata (15 kB)
...
Installing collected packages: tzdata, travertino, std-nslog, rubicon-objc, fonttools,␣
↪toga-core, faker, toga-cocoa
Successfully installed faker-37.3.0 fonttools-4.58.1 rubicon-objc-0.5.1 std-nslog-1.0.3␣
↪toga-cocoa-0.5.1 toga-core-0.5.1 travertino-0.5.1 tzdata-2025.2

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

[helloworld] Application updated.
```

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

[helloworld] Finalizing application configuration...
Targeting ubuntu:jammy (Vendor base debian)
Determining glibc version... done
Targeting glibc 2.35
Targeting Python3.13

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

[helloworld] Updating requirements...
Collecting faker
  Using cached faker-37.3.0-py3-none-any.whl.metadata (15 kB)
...
Installing collected packages: tzdata, travertino, std-nslog, rubicon-objc, fonttools,␣
↪toga-core, faker, toga-cocoa
Successfully installed faker-37.3.0 fonttools-4.58.1 rubicon-objc-0.5.1 std-nslog-1.0.3␣
↪toga-cocoa-0.5.1 toga-core-0.5.1 travertino-0.5.1 tzdata-2025.2

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

[helloworld] Application updated.
```

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

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

[helloworld] Updating requirements...
Collecting faker
  Using cached faker-37.3.0-py3-none-any.whl.metadata (15 kB)
...
Installing collected packages: tzdata, travertino, std-nslog, rubicon-objc, fonttools,␣
↪toga-core, faker, toga-cocoa
Successfully installed faker-37.3.0 fonttools-4.58.1 rubicon-objc-0.5.1 std-nslog-1.0.3␣
```

(suite sur la page suivante)

(suite de la page précédente)

```
↪toga-cocoa-0.5.1 toga-core-0.5.1 travertino-0.5.1 tzdata-2025.2

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

[helloworld] Application updated.
```

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

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

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

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

[helloworld] Application updated.
```

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

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

[helloworld] Updating requirements...
Looking in indexes: https://pypi.org/simple, https://pypi.anaconda.org/beeware/simple
Collecting faker
  Using cached faker-37.4.0-py3-none-any.whl.metadata (15 kB)
...

Installing app requirements for iPhone simulator... done

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

[helloworld] Application updated.
```

Une fois la mise à jour effectuée, vous pouvez lancer `briefcase build` et `briefcase run` - et vous devriez voir votre application packagée, avec le nouveau comportement du dialogue.

Note

L'option `-r` pour la mise à jour des exigences est également honorée par les commandes `build` et `run`, donc si vous voulez mettre à jour, compiler et exécuter en une seule étape, vous pouvez utiliser `briefcase run -u -r`.

2.8.4 Third-Party Python Packages for Mobile and Web

Faker is just one example of a third-party Python package - a collection of code that isn't part what Python provides out of the box. These third-party packages are most commonly distributed using the Python Package Index (PyPI), and installed into your local virtual environment. We've been using `pip` in this tutorial, but there are other options.

Sur les plateformes de bureau (macOS, Windows, Linux), tout `pip`-installable peut être ajouté à vos exigences. Sur les plateformes mobiles et web, vos options sont légèrement limitées.

In short ; any *pure Python* package (i.e. any package created from a project written *only* in Python) can be used without difficulty. Some packages, though, are created from projects that contain both Python and other languages (e.g. C, C++, Rust, etc). Code written in those languages needs to be compiled to platform-specific binary modules before it can be used, and those pre-compiled binary modules are only available on specific platforms. Mobile and web platforms have very different requirements than « standard » desktop platforms. At this time, most Python packages don't provide pre-compiled binaries for mobile and web platforms.

On PyPI, packages are often provided in a pre-built distribution format called *wheels*. To check whether a package is pure Python, look at the PyPI downloads page for the project. If the wheels provided have a `-py3-none-any.whl` suffix (e.g., Faker), then they are pure Python wheels. However, if the wheels have version and platform-specific extensions (e.g., Pillow, which has wheels with suffixes like `-cp313-cp313-macosx_11_0_arm64.whl` and `-cp39-cp39-win_amd64.whl`), then the wheel *contains a binary component*. That package cannot be installed on mobile or web platforms unless a wheel compatible with those platforms has been provided.

At this time, *most* binary packages on PyPI don't provide mobile- or web-compatible wheels. To fill this gap, BeeWare provides binaries for some popular binary modules (including `numpy`, `pandas`, and `cryptography`). These wheels are *not* distributed on PyPI, but Briefcase will install those wheels if they're available.

BeeWare peut fournir des binaires pour certains modules binaires populaires (y compris `numpy`, `pandas`, et `cryptographie`). Il est *habituellement* possible de compiler des paquets pour les plateformes mobiles, mais ce n'est pas facile à mettre en place – ce qui sort du cadre d'un tutoriel d'introduction comme celui-ci.

2.8.5 Étapes suivantes

We've now got an app that uses a third-party library ! In *Tutorial 8* we'll learn how to ensure our app remains responsive as we add more complex application logic.

2.9 Tutoriel 8 - Le rendre lisse

So far, our application has been relatively simple - displaying GUI widgets, calling a simple third-party library, and displaying output in a dialog. All these operations happen very quickly, and our application remains responsive.

However, in a real world application, we'll need to perform complex tasks or calculations that may take a while to complete - and as those tasks are performed, we want our application to remain responsive. Let's make a change to our application that might take a little time to complete, and see the changes that need to be made to accommodate that behavior.

2.9.1 Accessing an API

A common time-consuming task an app will need to perform is to make a request on a web API to retrieve data, and display that data to the user. Web APIs sometimes take a second or two to respond, so if we're calling an API like that, we need to ensure our application doesn't become unresponsive while we wait for the web API to return an answer.

This is a toy app, so we don't have a *real* API to work with, so we'll use the {JSON} Placeholder API as a source of data. The {JSON} Placeholder API has a number of « fake » API endpoints you can use as test data. One of those APIs is the `/posts/` endpoint, which returns fake blog posts. If you open `https://jsonplaceholder.typicode.com/posts/42` in your browser, you'll get a JSON payload describing a single post - some Lorum ipsum content for a blog post with ID 42.

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

As a result, people have developed third-party libraries that wrap the built-in APIs and provide a simpler API that is a closer match for the everyday browser experience. We're going to use one of those libraries to access the {JSON} Placeholder API - a library called httpx. Briefcase uses `httpx` internally, so it's already in your local environment - you don't need to install it separately to use it here.

Let's add a `httpx` API call to our app. Modify the `requires` setting in our `pyproject.toml` to include the new requirement :

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

Add an import to the top of the `app.py` to import `httpx` :

```
import httpx
```

Pour rendre notre tutoriel asynchrone, modifiez le gestionnaire d'événement `say_hello()` pour qu'il ressemble à ceci : :

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

    payload = response.json()

    await self.main_window.dialog(
        toga.InfoDialog(
            greeting(self.name_input.value),
            f"A message from {fake.name()}: {payload['body']}",
        )
    )
```

This will change the `say_hello()` callback so that when it is invoked, it will :
— make a GET request on the JSON placeholder API to obtain post 42 ;
— decode the response as JSON ;
— extract the body of the post ; and
— include the body of that post as the text of the « message » dialog, in place of the text generated by Faker.
Lets run our updated app in Briefcase developer mode to check that our change has worked. As we've added a new requirement, we need to tell developer mode to reinstall requirements, by using the `-r` argument :

macOS

Linux

Windows

Android

iOS

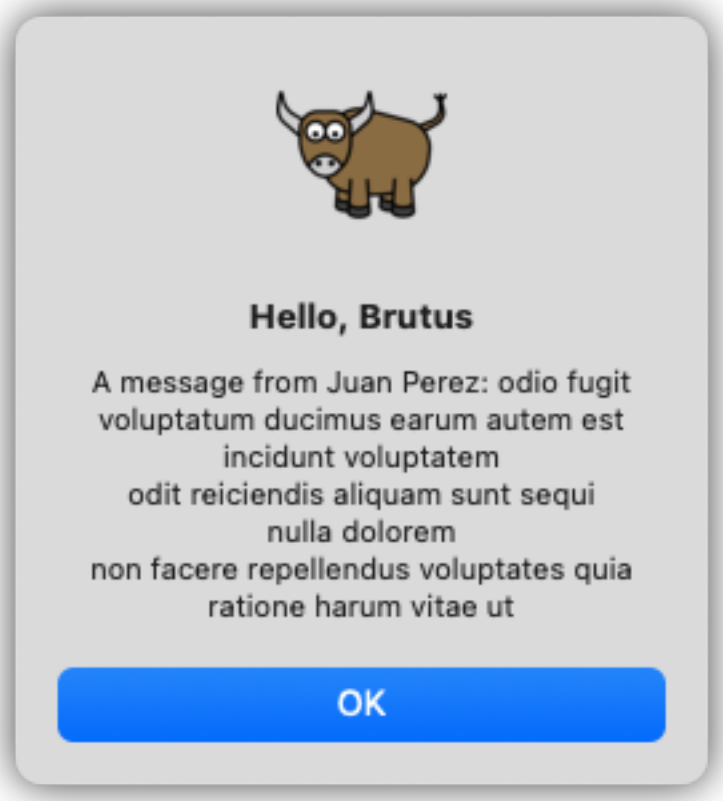
```
(beeware-venv) $ briefcase dev -r

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

(suite sur la page suivante)

(suite de la page précédente)

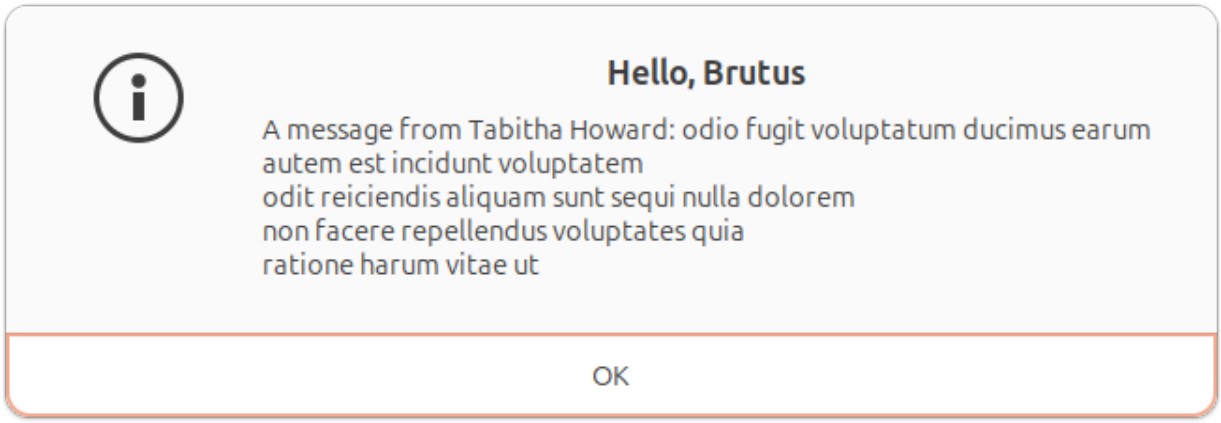
```
[helloworld] Starting in dev mode...
================================================================================
```

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



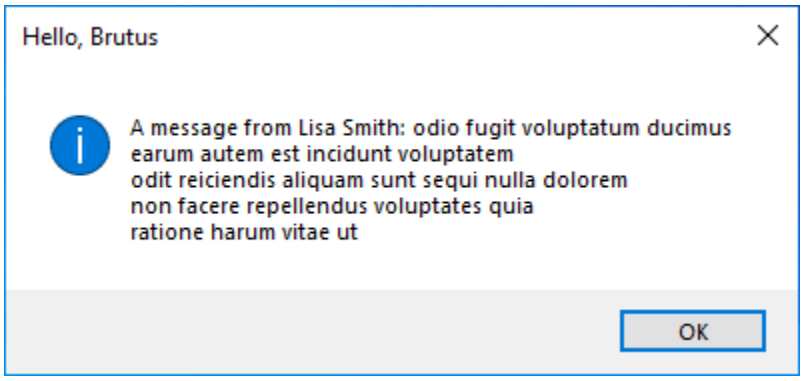

```
(beeware-venv) $ briefcase dev -r

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

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




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

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

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



You can't run an Android app in developer mode - use the instructions for your chosen desktop platform.

You can't run an iOS app in developer mode - use the instructions for your chosen desktop platform.

Unless you've got a *really* fast internet connection, you may notice that when you press the button, the GUI for your app locks up for a little bit. The operating system may even manifest this with a « beachball » or « spinner » cursor to indicate that the app is being unresponsive.

A moins que vous ne disposiez d'une connexion internet *très* rapide, vous remarquerez peut-être que lorsque vous appuyez sur le bouton, l'interface graphique de votre application se bloque pendant un petit moment. C'est parce que la requête web que nous avons faite est *synchrone*. Lorsque notre application effectue la requête web, elle attend que l'API renvoie une réponse avant de continuer. Pendant cette attente, l'API ne permet pas à l'application de se redessiner, ce qui a pour effet de bloquer l'application.

2.9.2 Boucles d'événements de l'interface graphique

Pour comprendre pourquoi cela se produit, nous devons entrer dans les détails du fonctionnement d'une application GUI. Les spécificités varient en fonction de la plate-forme, mais les concepts de haut niveau sont les mêmes, quelle que soit la plate-forme ou l'environnement d'interface graphique que vous utilisez.

Une application GUI est, fondamentalement, une boucle unique qui ressemble à quelque chose comme : :

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

Cette boucle est appelée *boucle d'événements*. (Il ne s'agit pas de noms de méthodes réels, mais d'une illustration de ce qui se passe dans le « pseudo-code »).

Lorsque vous cliquez sur un bouton, faites glisser une barre de défilement ou tapez une touche, vous générez un « événement ». Cet « événement » est placé dans une file d'attente, et l'application traitera la file d'événements lorsqu'elle en aura l'occasion. Le code utilisateur déclenché en réponse à l'événement est appelé « gestionnaire d'événement ». Ces gestionnaires d'événements sont invoqués dans le cadre de l'appel `process_events()`.

Une fois qu'une application a traité tous les événements disponibles, elle va `redraw()` l'interface graphique. Cela prend en compte tous les changements que les événements ont causés à l'affichage de l'application, ainsi que tout ce qui se passe dans le système d'exploitation - par exemple, les fenêtres d'une autre application peuvent masquer ou révéler une partie de la fenêtre de notre application, et le redessin de notre application devra refléter la partie de la fenêtre qui est actuellement visible.

Détail important : pendant qu'une application traite un événement, *elle ne peut pas redessiner*, et *elle ne peut pas traiter d'autres événements*.

Cela signifie que toute logique utilisateur contenue dans un gestionnaire d'événements doit être exécutée rapidement. Tout retard dans l'exécution du gestionnaire d'événements sera observé par l'utilisateur sous la forme d'un ralentissement (ou d'un arrêt) des mises à jour de l'interface graphique. Si ce délai est suffisamment long, votre système d'exploitation peut signaler qu'il s'agit d'un problème - les icônes macOS « beachball » et Windows « spinner » indiquent que votre application prend trop de temps dans un gestionnaire d'événements.

Des opérations simples comme « mettre à jour une étiquette » ou « recalculer le total des entrées » sont faciles à réaliser rapidement. Cependant, de nombreuses opérations ne peuvent pas être effectuées rapidement. Si vous effectuez un calcul mathématique complexe, si vous indexez tous les fichiers d'un système de fichiers ou si vous effectuez une requête réseau importante, vous ne pouvez pas « faire vite » - les opérations sont intrinsèquement lentes.

Alors, comment effectuer des opérations à long terme dans une application GUI ?

2.9.3 Programmation asynchrone

Ce dont nous avons besoin, c'est d'un moyen de dire à une application au milieu d'un gestionnaire d'événements de longue durée qu'il est acceptable de relâcher temporairement le contrôle dans la boucle d'événements, tant que nous pouvons reprendre là où nous nous sommes arrêtés. C'est à l'application de déterminer quand cette libération peut avoir lieu ; mais si l'application libère le contrôle dans la boucle d'événements régulièrement, nous pouvons avoir un gestionnaire d'événements de longue durée *et* maintenir une interface utilisateur réactive.

Nous pouvons le faire en utilisant la *programmation asynchrone*. La programmation asynchrone est une façon de décrire un programme qui permet à l'interpréteur d'exécuter plusieurs fonctions en même temps, en partageant les ressources entre toutes les fonctions qui s'exécutent simultanément.

Les fonctions asynchrones (appelées *co-routines*) doivent être explicitement déclarées comme étant asynchrones. Elles doivent également déclarer en interne lorsqu'il est possible de changer de contexte et de passer à une autre co-routine.

En Python, la programmation asynchrone est implémentée à l'aide des mots-clés `async` et `await`, et du module `asyncio <https://docs.python.org/3/library/asyncio.html>`__ dans la bibliothèque standard. Le mot-clé ``async` nous permet de déclarer qu'une fonction est une co-routine asynchrone. Le mot-clé `await` permet de déclarer qu'il existe une opportunité de changer de contexte vers une autre co-routine. Le module asyncio fournit d'autres outils et primitives utiles pour le codage asynchrone.

2.9.4 Rendre le didacticiel asynchrone

Pour rendre notre tutoriel asynchrone, modifiez le gestionnaire d'événement `say_hello()` pour qu'il ressemble à ceci : :

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

    payload = response.json()

    await self.main_window.dialog(
        toga.InfoDialog(
            greeting(self.name_input.value),
            f"A message from {fake.name()}: {payload['body']}",
        )
    )
```

Il n'y a que 4 changements dans ce code par rapport à la version précédente :

1. Le client créé est un `AsyncClient()` asynchrone, plutôt qu'un `Client()` synchrone. Cela indique à `httpx` qu'il doit fonctionner en mode asynchrone, plutôt qu'en mode synchrone.
2. Le gestionnaire de contexte utilisé pour créer le client est marqué comme `async`. Cela indique à Python qu'il y a une opportunité de relâcher le contrôle lorsque le gestionnaire de contexte est entré et sorti.
3. The `get` call is made with an `await` keyword. This instructs the app that while we are waiting for the response from the network, the app can release control to the event loop. We've seen this keyword before - we also use `await` when displaying the dialog box. The reason for that usage is the same as it is for the HTTP request - we need to tell the app that while the dialog is displayed, and we're waiting for the user to push a button, it's OK to release control back to the event loop.

It's also important to note that the handler itself is defined as `async def`, rather than just `def`. This tells Python that the method is an asynchronous coroutine. We made this change back in Tutorial 3 when we added the dialog box. You can only use `await` statements inside a method that is declared as `async def`.

Toga vous permet d'utiliser des méthodes normales ou des co-programmes asynchrones en tant que gestionnaires ; Toga gère tout en coulisses pour s'assurer que le gestionnaire est invoqué ou attendu selon les besoins.

Si vous sauvegardez ces changements et relancez l'application (soit avec `briefcase dev` en mode développement, soit en mettant à jour et en relançant l'application packagée), il n'y aura pas de changements évidents dans l'application. Cependant, lorsque vous cliquez sur le bouton pour déclencher le dialogue, vous pouvez remarquer un certain nombre d'améliorations subtiles :

— Le bouton revient à l'état « décliqué » au lieu d'être bloqué à l'état « cliqué ».
— L'icône « beachball »/ »spinner » n'apparaît pas
— Si vous déplacez ou redimensionnez la fenêtre de l'application en attendant que la boîte de dialogue s'affiche, la fenêtre se redessinera.
— Si vous essayez d'ouvrir un menu d'application, le menu s'affiche immédiatement.

We can now run the full app. However, as we've added an extra requirement (`httpx`) we also need to update our app's requirements ; we can do this by passing `-r` to `briefcase run`. This will update our app's requirements, then re-build the app, then launch the app :

macOS

Linux

Windows

Android

iOS

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

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

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

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

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

You should see you app running, and remaining responsive when you press the button and network content is retrieved.

2.9.5 Étapes suivantes

This has been a taste for what you can do with the tools provided by the BeeWare project. Over the course of this tutorial, you have :

- Created a new GUI app project ;
- Run that app in development mode ;
- Built the app as a standalone binary for a desktop operating system ;
- Packaged that project for distribution to others ;
- Run the app on a mobile simulator and/or device ;
- Run the app as a web app ;
- Added a third-party dependency to your app ; and
- Modified the app so that it remains responsive.

So - where to from here ?

- If you'd like to go further, there are some additional *topic tutorials* that go into detail on specific aspects of application development.
- If you'd like to know more about how to build complex user interfaces with Toga, you can dive into Toga's documentation. Toga also has it's own tutorial demonstrating how to use various features of the widget toolkit.
- If you'd like to know more about the capabilities of Briefcase, you can dive into Briefcase's documentation.

2.10 Going further

Want to go deeper on specific topics ? Here are some additional tutorials that explore common aspects of application development. Each tutorial is standalone, and can be completed in any order ; but they all assume you've completed the core tutorial.

Customizing icons Customize your application's appearance by replacing the default « gray bee » icon.

Customizing icons Application Testing How do you ensure that your application works, and stays working ? By adding a test suite to your project !

Tutoriel 9 - Temps de test Camera access Use the camera on your mobile or desktop device to take and view a picture from within your application.

Using the camera

2.10.1 Customizing icons

Jusqu'à présent, notre application utilise l'icône par défaut « abeille grise ». Comment mettre à jour l'application pour qu'elle utilise notre propre icône ?

Ajouter une icône

Every platform uses a different format for application icons - and some platforms need *multiple* icons in different sizes and shapes. To account for this, Briefcase provides a shorthand way to configure an icon once, and then have that definition expand in to all the different icons needed for each individual platform.

Edit your `pyproject.toml`, adding a new `icon` configuration item in the `[tool.briefcase.app.helloworld]` configuration section, just above the `sources` definition :

```
icon = "icons/helloworld"
```

This icon definition doesn't specify any file extension. The value will be used as a prefix ; each platform will add additional items to this prefix to build the files needed for each platform. Some platforms require *multiple* icon files ; this prefix will be combined with file size and variant modifiers to generate the list of icon files that are used.

We can now run `briefcase update` again - but this time, we pass in the `--update-resources` flag, telling Briefcase that we want to install new application resources (i.e., the icons) :

macOS

Linux

Windows

Android

iOS

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

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

[helloworld] Updating application resources...
Unable to find icons/helloworld.icns for application icon; using default

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

[helloworld] Application updated.
```

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

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

[helloworld] Updating application resources...
Unable to find icons/helloworld-16.png for 16px application icon; using default
Unable to find icons/helloworld-32.png for 32px application icon; using default
Unable to find icons/helloworld-64.png for 64px application icon; using default
Unable to find icons/helloworld-128.png for 128px application icon; using default
Unable to find icons/helloworld-256.png for 256px application icon; using default
Unable to find icons/helloworld-512.png for 512px application icon; using default

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

[helloworld] Application updated.
```

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

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

[helloworld] Updating application resources...
Unable to find icons/helloworld.ico for application icon; using default

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

[helloworld] Application updated.
```

```
(beeware-venv) $ briefcase update android --update-resources

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

[helloworld] Updating application resources...
Unable to find icons/helloworld-round-48.png for 48px round application icon; using␣
↪default
Unable to find icons/helloworld-round-72.png for 72px round application icon; using␣
↪default
Unable to find icons/helloworld-round-96.png for 96px round application icon; using␣
↪default
Unable to find icons/helloworld-round-144.png for 144px round application icon; using␣
↪default
Unable to find icons/helloworld-round-192.png for 192px round application icon; using␣
↪default
Unable to find icons/helloworld-square-48.png for 48px square application icon; using␣
↪default
Unable to find icons/helloworld-square-72.png for 72px square application icon; using␣
↪default
Unable to find icons/helloworld-square-96.png for 96px square application icon; using␣
↪default
Unable to find icons/helloworld-square-144.png for 144px square application icon; using␣
↪default
Unable to find icons/helloworld-square-192.png for 192px square application icon; using␣
↪default
Unable to find icons/helloworld-square-320.png for 320px square application icon; using␣
↪default
Unable to find icons/helloworld-square-480.png for 480px square application icon; using␣
↪default
Unable to find icons/helloworld-square-640.png for 640px square application icon; using␣
↪default
Unable to find icons/helloworld-square-960.png for 960px square application icon; using␣
↪default
Unable to find icons/helloworld-square-1280.png for 1280px square application icon;␣
↪using default
Unable to find icons/helloworld-adaptive-108.png for 108px adaptive application icon;␣
↪using default
Unable to find icons/helloworld-adaptive-162.png for 162px adaptive application icon;␣
```

(suite sur la page suivante)

(suite de la page précédente)

```
↪using default
Unable to find icons/helloworld-adaptive-216.png for 216px adaptive application icon;␣
↪using default
Unable to find icons/helloworld-adaptive-324.png for 324px adaptive application icon;␣
↪using default
Unable to find icons/helloworld-adaptive-432.png for 432px adaptive application icon;␣
↪using default

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

[helloworld] Application updated.
```

```
(beeware-venv) $ briefcase iOS --update-resources

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

  [helloworld] Updating application resources...
  Unable to find icons/helloworld-20.png for 20px application icon; using default
  Unable to find icons/helloworld-29.png for 29px application icon; using default
  Unable to find icons/helloworld-40.png for 40px application icon; using default
  Unable to find icons/helloworld-58.png for 58px application icon; using default
  Unable to find icons/helloworld-60.png for 60px application icon; using default
  Unable to find icons/helloworld-76.png for 76px application icon; using default
  Unable to find icons/helloworld-80.png for 80px application icon; using default
  Unable to find icons/helloworld-87.png for 87px application icon; using default
  Unable to find icons/helloworld-120.png for 120px application icon; using default
  Unable to find icons/helloworld-152.png for 152px application icon; using default
  Unable to find icons/helloworld-167.png for 167px application icon; using default
  Unable to find icons/helloworld-180.png for 180px application icon; using default
  Unable to find icons/helloworld-640.png for 640px application icon; using default
  Unable to find icons/helloworld-1024.png for 1024px application icon; using default
  Unable to find icons/helloworld-1280.png for 1280px application icon; using default
  Unable to find icons/helloworld-1920.png for 1920px application icon; using default

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

  [helloworld] Application updated.
```

This reports the specific icon file (or files) that Briefcase is expecting. However, as we haven't provided the actual icon files, the install fails, and Briefcase falls back to a default value (the same « gray bee » icon).

Let's provide some actual icons. Download `this icons.zip bundle`, and unzip it into the root of your project directory. After unzipping, your project directory should look something like :

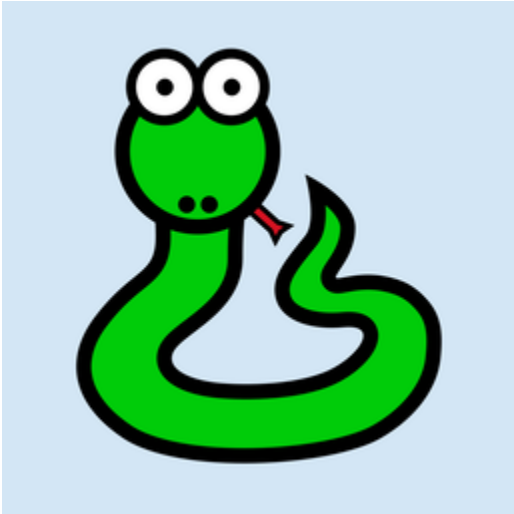
```
beeware-tutorial/
├── beeware-venv/
│   └── ...
└── helloworld/
    ├── ...
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ├── icons/
    │   ├── helloworld.icns
    │   ├── helloworld.ico
    │   ├── helloworld.png
    │   ├── helloworld-16.png
    │   └── ...
    ├── src/
    │   └── ...
    └── pyproject.toml
```

There's a *lot* of icons in this folder, but most of them should look the same : a green snake on a light blue background :



The only exception will be the icons with `-adaptive-` in their name ; these will have a transparent background. This represents all the different icon sizes and shapes you need to support an app on every platform that Briefcase supports.

Now that we have icons, we can update the application again. However, `briefcase update` will only copy the updated resources into the build directory ; we also want to rebuild the app to make sure the new icon is included, then start the app. We can shortcut this process by passing `--update-resources` to our call to `run` - this will update the app, update the app's resources, and then start the app :

macOS

Linux

Windows

Android

iOS

```
(beeware-venv) $ briefcase run --update-resources

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

[helloworld] Updating application resources...
Installing icons/helloworld.icns as application icon... done

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

(suite sur la page suivante)

(suite de la page précédente)

```
Removing unneeded app bundle content... done

[helloworld] Application updated.

[helloworld] Ad-hoc signing app...
      100.0% • 00:01

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

[helloworld] Starting app...
```

```
(beeware-venv) $ briefcase run --update-resources

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

[helloworld] Updating application resources...
Installing icons/helloworld-16.png as 16px application icon... done
Installing icons/helloworld-32.png as 32px application icon... done
Installing icons/helloworld-64.png as 64px application icon... done
Installing icons/helloworld-128.png as 128px application icon... done
Installing icons/helloworld-256.png as 256px application icon... done
Installing icons/helloworld-512.png as 512px application icon... done

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

[helloworld] Application updated.

[helloworld] Building application...
Build bootstrap binary...
...

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

[helloworld] Starting app...
```

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

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

[helloworld] Updating application resources...
Installing icons/helloworld.ico as application icon... done

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

[helloworld] Application updated.
```

(suite sur la page suivante)

(suite de la page précédente)

```
[helloworld] Building App...
Removing any digital signatures from stub app... done
Setting stub app details... done

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

[helloworld] Starting app...
```

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

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

[helloworld] Updating application resources...
Installing icons/helloworld-round-48.png as 48px round application icon... done
Installing icons/helloworld-round-72.png as 72px round application icon... done
Installing icons/helloworld-round-96.png as 96px round application icon... done
Installing icons/helloworld-round-144.png as 144px round application icon... done
Installing icons/helloworld-round-192.png as 192px round application icon... done
Installing icons/helloworld-square-48.png as 48px square application icon... done
Installing icons/helloworld-square-72.png as 72px square application icon... done
Installing icons/helloworld-square-96.png as 96px square application icon... done
Installing icons/helloworld-square-144.png as 144px square application icon... done
Installing icons/helloworld-square-192.png as 192px square application icon... done
Installing icons/helloworld-square-320.png as 320px square application icon... done
Installing icons/helloworld-square-480.png as 480px square application icon... done
Installing icons/helloworld-square-640.png as 640px square application icon... done
Installing icons/helloworld-square-960.png as 960px square application icon... done
Installing icons/helloworld-square-1280.png as 1280px square application icon... done
Installing icons/helloworld-adaptive-108.png as 108px adaptive application icon... done
Installing icons/helloworld-adaptive-162.png as 162px adaptive application icon... done
Installing icons/helloworld-adaptive-216.png as 216px adaptive application icon... done
Installing icons/helloworld-adaptive-324.png as 324px adaptive application icon... done
Installing icons/helloworld-adaptive-432.png as 432px adaptive application icon... done

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

[helloworld] Application updated.

[helloworld] Starting app...
```

Note

If you're using a recent version of Android, you may notice that the app icon has been changed to a green snake, but the background of the icon is *white*, rather than light blue. We'll fix this in the next step.

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

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

(suite sur la page suivante)

(suite de la page précédente)

```
Installing src/helloworld... done

[helloworld] Updating application resources...
Installing icons/helloworld-20.png as 20px application icon... done
Installing icons/helloworld-29.png as 29px application icon... done
Installing icons/helloworld-40.png as 40px application icon... done
Installing icons/helloworld-58.png as 58px application icon... done
Installing icons/helloworld-60.png as 60px application icon... done
Installing icons/helloworld-76.png as 76px application icon... done
Installing icons/helloworld-80.png as 80px application icon... done
Installing icons/helloworld-87.png as 87px application icon... done
Installing icons/helloworld-120.png as 120px application icon... done
Installing icons/helloworld-152.png as 152px application icon... done
Installing icons/helloworld-167.png as 167px application icon... done
Installing icons/helloworld-180.png as 180px application icon... done
Installing icons/helloworld-640.png as 640px application icon... done
Installing icons/helloworld-1024.png as 1024px application icon... done
Installing icons/helloworld-1280.png as 1280px application icon... done
Installing icons/helloworld-1920.png as 1920px application icon... done

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

[helloworld] Application updated.

[helloworld] Starting app...
```

Note

If you get a stack trace referencing `faker` or `httpx` when you run the app, it's possible you missed running your app during step 7 or 8 of the tutorial. Re-run the app, adding the `-r` argument to update the app requirements.

When you run the app on iOS or Android, in addition to the icon change, you should also notice that the splash screen incorporates the new icon. However, the light blue background of the icon looks a little out of place against the white background of the splash screen. We can fix this by customizing the background color of the splash screen. Add the following definition to your `pyproject.toml`, just after the `icon` definition :

```
splash_background_color = "#D3E6F5"
```

Unfortunately, Briefcase isn't able to apply this change to an already generated project, as it requires making modifications to one of the files that was generated during the original call to `briefcase create`. To apply this change, we have to re-create the app by re-running `briefcase create`. When we do this, we'll be prompted to confirm that we want to overwrite the existing project :

macOS

Linux

Windows

Android

iOS

```
(beeware-venv) $ briefcase create

Application 'helloworld' already exists; overwrite [y/N]? y

[helloworld] Removing old application bundle...

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

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

```
(beeware-venv) $ briefcase create

Application 'helloworld' already exists; overwrite [y/N]? y

[helloworld] Removing old application bundle...

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

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

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

Application 'helloworld' already exists; overwrite [y/N]? y

[helloworld] Removing old application bundle...

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

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

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

Application 'helloworld' already exists; overwrite [y/N]? y

[helloworld] Removing old application bundle...

[helloworld] Generating application template...
...
[helloworld] Created build/helloworld/android/gradle
```

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

Application 'helloworld' already exists; overwrite [y/N]? y

[helloworld] Removing old application bundle...

[helloworld] Generating application template...
...
[helloworld] Created build/helloworld/ios/xcode
```

You can then re-build and re-run the app using `briefcase run`. You won't notice any changes to the desktop app ; but the Android or iOS apps should now have a light blue splash screen background.

You'll need to re-create the app like this whenever you make a change to your `pyproject.toml` that doesn't relate to source code or dependencies. Any change to descriptions, version numbers, colors, or permissions will require a re-create step. Because of this, while you are developing your project, you shouldn't make any manual changes to the contents of the `build` folder, and you shouldn't add the `build` folder to your version control system. The `build` folder should be considered entirely ephemeral - an output of the build system that can be recreated as needed to reflect the current configuration of your project.

2.10.2 Tutoriel 9 - Temps de test

La plupart des développements de logiciels n'impliquent pas l'écriture d'un nouveau code, mais la modification d'un code existant. S'assurer que le code existant continue à fonctionner comme nous l'attendons est une partie essentielle du processus de développement logiciel. Une façon de s'assurer du comportement de notre application est d'utiliser une *suite de tests*.

Exécution de la suite de tests

Il s'avère que notre projet possède déjà une suite de tests ! Lorsque nous avons généré notre projet à l'origine, deux répertoires de premier niveau ont été générés : `src` et `tests`. Le dossier `src` contient le code de notre application ; le dossier `tests` contient notre suite de tests. Dans le dossier `tests` se trouve un fichier nommé `test_app.py` avec le contenu suivant : :

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

Ceci est un Pytest *test case* - un bloc de code qui peut être exécuté pour vérifier un certain comportement de votre application. Dans ce cas, le test est un placeholder, et ne teste rien de notre application - mais c'est un test que nous pouvons effectuer.

Nous pouvons lancer cette suite de tests en utilisant l'option `--test` de `briefcase dev`. Comme c'est la première fois que nous lançons des tests, nous devons également passer l'option `-r` pour nous assurer que les exigences de test sont également installées :

macOS

Linux

Windows

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

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

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

(suite sur la page suivante)

(suite de la page précédente)

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

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

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

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

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

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

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

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

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

[helloworld] Running test suite in dev environment...
================================================================================
============================= test session starts ==============================
platform win32 -- Python 3.11.0
pytest==7.2.0
py==1.11.0
pluggy==1.0.0
cachedir: C:\Users\brutus\AppData\Local\Temp\.pytest_cache
rootdir: C:\Users\brutus
plugins: anyio-3.6.2
collecting ... collected 1 item

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

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

Succès ! Nous venons d'exécuter un seul test qui vérifie que les mathématiques Python fonctionnent de la manière attendue (Quel soulagement !).

Remplaçons ce placeholder test par un test pour vérifier que notre méthode `greeting()` se comporte comme nous

l'attendons. Remplacez le contenu de `test_app.py` par ce qui suit : :

```
from helloworld.app import greeting


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

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


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

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

Ceci définit deux nouveaux tests, vérifiant les deux comportements que nous attendons : la sortie lorsqu'un nom est fourni, et la sortie lorsque le nom est vide.

Nous pouvons maintenant réexécuter la suite de tests. Cette fois, nous n'avons pas besoin de fournir l'option `-r`, puisque les pré-requis pour les tests ont déjà été installés ; nous avons seulement besoin d'utiliser l'option `--test` :

macOS

Linux

Windows

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

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

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

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

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

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

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

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

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

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

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

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

Excellent ! Notre méthode utilitaire `greeting()` fonctionne comme prévu.

Développement piloté par les tests

Maintenant que nous disposons d'une suite de tests, nous pouvons l'utiliser pour développer de nouvelles fonctionnalités. Modifions notre application pour avoir un message d'accueil spécial pour un utilisateur particulier. Nous pouvons commencer par ajouter un scénario de test pour le nouveau comportement que nous aimerions voir au bas de `test_app.py` : :

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

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

Ensuite, exécutez la suite de tests avec ce nouveau test :

macOS

Linux

Windows

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

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

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

=================================== FAILURES ===================================
_____________________________________ test_brutus ______________________________________

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

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

(suite sur la page suivante)

(suite de la page précédente)

```
E         - BeeWare the IDEs of Python!
E         + Hello, Brutus

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

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

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

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

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

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

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

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

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

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

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

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

=================================== FAILURES ===================================
```

(suite sur la page suivante)

(suite de la page précédente)

```
_____________________________________ test_brutus _____________________________________

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

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

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

Cette fois, nous voyons un échec du test - et la sortie explique la source de l'échec : le test attend la sortie « BeeWare the IDEs of Python ! », mais notre implémentation de `greeting()` retourne « Hello, Brutus ». Modifions l'implémentation de `greeting()` dans `src/helloworld/app.py` pour avoir le nouveau comportement : :

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

Si nous exécutons à nouveau les tests, nous constatons qu'ils sont réussis :

macOS

Linux

Windows

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

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

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

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

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

[helloworld] Running test suite in dev environment...
================================================================================
```

(suite sur la page suivante)

(suite de la page précédente)

```
============================= test session starts ==============================
...
collecting ... collected 3 items

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

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

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

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

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

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

Tests d'exécution

Jusqu'à présent, nous avons exécuté les tests en mode développement. C'est particulièrement utile lorsque vous développez de nouvelles fonctionnalités, car vous pouvez rapidement itérer sur l'ajout de tests et l'ajout de code pour faire passer ces tests. Cependant, à un moment donné, vous voudrez vérifier que votre code s'exécute correctement dans l'environnement de l'application groupée.

Les options `--test` et `-r` peuvent également être passées à la commande `run`. Si vous utilisez `briefcase run --test -r`, la même suite de tests s'exécutera, mais elle s'exécutera dans le paquetage de l'application plutôt que dans votre environnement de développement :

macOS

Linux

Windows

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

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

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

[helloworld] Starting test suite...
================================================================================
Configuring isolated Python...
```

(suite sur la page suivante)

(suite de la page précédente)

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

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

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

[helloworld] Test suite passed!
```

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

[helloworld] Finalizing application configuration...
Targeting ubuntu:jammy (Vendor base debian)
Determining glibc version... done
Targeting glibc 2.35
Targeting Python3.10

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

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

[helloworld] Starting test suite...
================================================================================
============================= test session starts ==============================
...
```

(suite sur la page suivante)

(suite de la page précédente)

```
collecting ... collected 3 items

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

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

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

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

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

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

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

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

Comme pour `briefcase dev --test`, l'option `-r` n'est nécessaire que la première fois que vous exécutez la suite de tests pour vous assurer que les dépendances des tests sont présentes. Lors des exécutions suivantes, vous pouvez omettre cette option.

Vous pouvez également utiliser l'option *–test* ` sur les backends mobiles : - ainsi *briefcase run iOS –test* ` et `briefcase run android --test` fonctionneront tous les deux, lançant la suite de tests sur l'appareil mobile que vous avez sélectionné.

2.10.3 Using the camera

Almost every modern computing device has a camera of some sort. In this tutorial, we'll write new application that is able to request access to this camera, take a photograph, and then display that photograph in the new application that uses your device's camera.

This tutorial won't work on all platforms!

Unfortunately, at present, this tutorial will only work on macOS and Android.

Although iPhones all have cameras, the iOS *Simulator* doesn't have a working camera. Windows and Linux devices also have cameras, but Toga doesn't currently have the ability to access the camera on these platforms.

The code presented here will *run* on Windows or Linux ; but it will raise an error when you try to take a photograph.

The code will work if it is run on an actual iOS device, but will fail to take a photograph if deployed to the iOS simulator.

Start a new project

For this tutorial, we're not going to build onto the application from the core tutorial - we're going to start a fresh project. You can use the same virtual environment you used in the first project ; but we need to re-run the `briefcase new` wizard.

Change back to the directory that contains the `helloworld` project folder, and start a new project named « Hello Camera » :

```
(beeware-venv) $ cd ..
(beeware-venv) $ briefcase new
...
[hellocamera] Generated new application 'Hello Camera'

To run your application, type:

    $ cd hellocamera
    $ briefcase dev

(beeware-venv) $ cd hellocamera
```

Add code to take a photo

The wizard has generated a new empty Toga project. We can now add the code to take and display a photograph. Edit the `app.py` for the new application so that it has the following content :

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


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

        self.photo = toga.ImageView(height=300, margin=5)
        camera_button = toga.Button(
            "Take photo",
```

(suite sur la page suivante)

(suite de la page précédente)

```
            on_press=self.take_photo,
            margin=5,
        )

        main_box.add(self.photo)
        main_box.add(camera_button)

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

    async def take_photo(self, widget, **kwargs):
        try:
            if not self.camera.has_permission:
                await self.camera.request_permission()

            image = await self.camera.take_photo()
            if image:
                self.photo.image = image
        except NotImplementedError:
            await self.main_window.dialog(
                toga.InfoDialog(
                    "Oh no!",
                    "The Camera API is not implemented on this platform",
                )
            )
        except PermissionError:
            await self.main_window.dialog(
                toga.InfoDialog(
                    "Oh no!",
                    "You have not granted permission to take photos",
                )
            )


def main():
    return HelloCamera()
```

This code has two changes over the default app that is generated by Briefcase. These additions are highlighted above :

1. The first highlighted code block (in the `startup()` method) adds the two widgets needed to control the camera : an `ImageView` to display a photo ; and a `Button` to trigger the camera.
2. The second highlighted code block (the `take_photo()` method) defines the event handler when the button is pressed. This handler first confirms if the application has permission to take a photo ; if permission doesn't exist, it is requested. Then, a photo is taken. The request for permission and the request to take a photo are both asynchronous requests, so they require the use of `await` ; while the app is waiting for the user to confirm permissions or take the photo, the app's event loop can continue in the background.

If the camera successfully takes a photo, it will return an `Image` object that can be assigned as the content of the `ImageView`. If the photo request was canceled by the user, the `self.camera.take_photo()` call will return `None`, and the result can be ignored. If the user doesn't grant permission to use the camera, or the camera isn't implemented on the current platform, an error will be raised, and a dialog will be shown to the user.

Adding device permissions

Part of this code we've just added asks for permission to use the camera. This is a common feature of modern app platforms - you can't access hardware features without explicitly asking the user's permission first.

However, this request comes in two parts. The first is in the code we've just seen; but before the app can ask for permissions, it needs to declare the permissions it is going to ask for.

The permissions required by each platform are slightly different, but Briefcase has a cross-platform representation for many common hardware permissions. In the `[tool.briefcase.app.hellocamera]` configuration section of your app's `pyproject.toml` file, add the following (just above the `sources` declaration) :

```
permission.camera = "App will take mugshots."
```

This declares that your app needs to access the camera, and provides a short description why the camera is required. This description is needed on some platforms (most notably macOS and iOS) and will be displayed to the user as a additional information when the permission dialog is presented.

We can now generate and run the app :

macOS

Android

```
(beeware-venv)$ briefcase create
(beeware-venv)$ briefcase build
(beeware-venv)$ briefcase run
```

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

When the app runs, you'll be presented with a button. Press the button, and the platform's default camera dialog will be displayed. Take a photo; the camera dialog will disappear, and the photo will be displayed on in the app, just above the button. You could then take another photo; this will replace the first photo.

Adding more permissions

Permissions are declared in the files that are generated during the original call to `briefcase create`. Unfortunately, Briefcase can't update these files once they've been initially generated; so if you want to add a new permission to your app, or modify existing permissions, you'll need to re-create the app. You can do this by re-running `briefcase create`; this will warn you that the existing app will be overwritten, and then regenerate the application with the new permissions.

2.11 Contributing to this tutorial

This tutorial is written using Sphinx and reStructuredText. This guide will help you contribute fixes or new content to this tutorial.

Translations of this tutorial are managed using Weblate. If you'd like to contribute to the translation effort, join the `#translations` channel on Discord and introduce yourself !

2.11.1 Set up your development environment

To build the BeeWare tutorial you **must** have a Python 3.12 interpreter installed and available on your path (i.e., `python3` must start a Python 3.12 interpreter).

macOS

Linux

Windows

```
$ python3 --version
$ pip3 --version
```

```
$ python3 --version
$ pip3 --version
```

```
C:\...>python3 --version
C:\...>pip3 --version
```

Install Enchant

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 Apple Silicon machine (M-series), 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 / Debian

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

Fedora

```
$ sudo dnf install enchant
```

Arch / Manjaro

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

OpenSUSE Tumbleweed

```
$ sudo zypper install enchant
```

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

Create a virtual environment

The recommended way of setting up your development environment for BeeWare is to install a virtual environment, install the required dependencies and start coding. To set up a virtual environment, run :

macOS

Linux

Windows

```
$ python3 -m venv venv
$ source venv/bin/activate
```

```
$ python3 -m venv venv
$ source venv/bin/activate
```

```
C:\...>python3 -m venv venv
C:\...>venv\Scripts\activate
```

Your prompt should now have a `(venv)` prefix in front of it.

Clone the BeeWare repository

For updates to BeeWare documentation :

Next, go to the BeeWare page on GitHub, fork the repository into your own account, and then clone a copy of that repository onto your computer by clicking on « Clone or Download ». If you have the GitHub desktop application installed on your computer, you can select « Open in Desktop » ; otherwise, copy the URL provided, and use it to clone using the command line :

macOS

Linux

Windows

Fork the BeeWare repository, and then :

```
(venv) $ git clone https://github.com/<your username>/beeware.git
```

(substituting your GitHub username)

Fork the BeeWare repository, and then :

```
(venv) $ git clone https://github.com/<your username>/beeware.git
```

(substituting your GitHub username)

Fork the BeeWare repository, and then :

```
(venv) C:\...>git clone https://github.com/<your username>/beeware.git
```

(substituting your GitHub username)

Install BeeWare tutorial docs dependencies

Now that you have the source code, you can install BeeWare docs requirements into your development environment. Since we're installing from source, we can't rely on pip to resolve the dependencies to source packages, so we have to manually install each package :

macOS

Linux

Windows

```
(venv) $ cd beeware
(venv) $ python -m pip install -e ".[dev]"
```

```
(venv) $ cd beeware
(venv) $ python -m pip install -e .[dev]
```

```
(venv) C:\...>cd beeware
(venv) C:\...>python -m pip install -e .[dev]
```

Install pre-commit

BeeWare 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
```

When you commit any change, pre-commit will run automatically. 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"
black....................................................................Failed
- hook id: black
- files were modified by this hook

reformatted some/interesting_file.py

All done!
1 file reformatted.

flake8...................................................................Passed
check toml.....................................(no files to check)Skipped
check yaml.....................................(no files to check)Skipped
check for case conflicts.................................................Passed
```

(suite sur la page suivante)

(suite de la page précédente)

```
check docstring is first.....................................................Passed
fix end of files.............................................................Passed
trim trailing whitespace.....................................................Passed
isort........................................................................Passed
pyupgrade....................................................................Passed
docformatter.................................................................Passed
```

```
(venv) $ git add some/interesting_file.py
(venv) $ git commit -m "Minor change"
black........................................................................Failed
- hook id: black
- files were modified by this hook

reformatted some/interesting_file.py

All done!
1 file reformatted.

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

```
(venv) C:\...>git add some/interesting_file.py
(venv) C:\...>git commit -m "Minor change"
black........................................................................Failed
- hook id: black
- files were modified by this hook

reformatted some\interesting_file.py

All done!
1 file reformatted.

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

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"
black....................................................................Passed
flake8...................................................................Passed
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
[bugfix e3e0f73] Minor change
1 file changed, 4 insertions(+), 2 deletions(-)
```

```
(venv) $ git add some/interesting_file.py
(venv) $ git commit -m "Minor change"
black....................................................................Passed
flake8...................................................................Passed
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
[bugfix e3e0f73] Minor change
1 file changed, 4 insertions(+), 2 deletions(-)
```

```
(venv) C:\...>git add some\interesting_file.py
(venv) C:\...>git commit -m "Minor change"
black....................................................................Passed
flake8...................................................................Passed
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
```

Now you are ready to start hacking on BeeWare docs !

2.11.2 Building BeeWare's documentation

Build documentation locally

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, BeeWare 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.

Live preview mode will only monitor the `docs` directory for changes. If you're updating the inline documentation associated with BeeWare source code, you'll need to use the `docs-live-src` target to build docs :

macOS

Linux

Windows

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

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

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

This behaves the same as `docs-live`, but will also monitor any changes to the `core/src` folder, reflecting any changes to inline documentation. However, the rebuild process takes much longer, so you may not want to use this target unless you're actively editing inline documentation.

Documentation linting

The build process will identify reStructuredText problems, but BeeWare performs some additional « lint » checks. To run the lint checks :

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 :

— 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. When adding to this list, remember :

— We prefer US spelling, with some liberties for programming-specific colloquialism (e.g., « apps ») and verbing of nouns (e.g., « scrollable »)
— Any reference to a product name should use the product's preferred capitalization. (e.g., « macOS », « GTK », « pytest », « Pygame », « PyScript »).
— If a term is being used « as code », then it should be quoted as a literal rather than being added to the 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.

2.11.3 What to work on ?

If you're looking for specific areas to improve, there are tickets tagged « documentation » in BeeWare's issue tracker.

However, you don't need to be constrained by these tickets. If you can identify an error in the tutorial, or an improvement that can be made, start writing ! Anything that improves the experience of the end user is a welcome change.

2.11.4 Submitting a pull request

Before you submit a pull request, there's a few bits of housekeeping to do.

Submit from a feature branch, not your `main` branch

Before you start working on your change, make sure you've created a branch. By default, when you clone your repository fork, you'll be checked out on your `main` branch. This is a direct copy of BeeWare's `main` branch. To contribute to BeeWare itself, not the docs, please review the repo README.

While you *can* submit a pull request from your `main` branch, it's preferable if you *don't* do this. If you submit a pull request that is *almost* right, the core team member who reviews your pull request may be able to make the necessary changes, rather than giving feedback asking for a minor change. However, if you submit your pull request from your `main` branch, reviewers are prevented from making modifications.

Instead, you should make your changes on a *feature branch*. A feature branch has a simple name to identify the change that you've made.

To create a feature branch, run :

macOS

Linux

Windows

```
(venv) $ git checkout -b fix-layout-bug
```

```
(venv) $ git checkout -b fix-layout-bug
```

```
(venv) C:\...>git checkout -b fix-layout-bug
```

Commit your changes to this branch, then push to GitHub and create a pull request.