BeeWare Documentation

Release 0.1.dev147+gc9c3811

Russell Keith-Magee

06 mag 2024

Indice

Scrivere Python. Esegui ovunque.

Benvenuti a BeeWare! In questa esercitazione costruiremo un'interfaccia grafica utilizzando Python e la distribuiremo come applicazione desktop, applicazione mobile e applicazione web a pagina singola. Vedremo anche come utilizzare gli strumenti di BeeWare per eseguire alcune delle attività più comuni che dovrete svolgere come sviluppatori di applicazioni, come ad esempio testare la vostra applicazione.

Questa è una traduzione automatica!

Questa versione del tutorial è stata generata da una traduzione automatica. Sappiamo che non è l'ideale, ma abbiamo pensato che una cattiva traduzione fosse meglio di nessuna traduzione.

Se volete contribuire a migliorare la traduzione, contattateci! Abbiamo un canale `#translations` in Discord. Presentati lì e ti aggiungeremo al team di traduzione.

CAPITOLO 1

Che cos'è BeeWare?

BeeWare non è un singolo prodotto, o strumento, o libreria: è una raccolta di strumenti e librerie, ognuno dei quali lavora insieme per aiutarvi a scrivere applicazioni Python multipiattaforma con un'interfaccia grafica nativa. Include:

- Toga, un toolkit di widget multipiattaforma;
- Briefcase, uno strumento per impacchettare i progetti Python come artefatti distribuibili che possono essere spediti agli utenti finali;
- Librerie (come Rubicon ObjC) per accedere alle librerie native della piattaforma;
- Costruzioni precompilate di Python che possono essere utilizzate su piattaforme in cui non sono disponibili gli installatori ufficiali di Python.

In questo tutorial utilizzeremo tutti questi strumenti, ma come utente dovrete interagire solo con i primi due (Toga e Briefcase). Tuttavia, ciascuno di questi strumenti può essere utilizzato anche singolarmente: ad esempio, si può usare Briefcase per distribuire un'applicazione senza usare Toga come toolkit per l'interfaccia grafica.

La suite BeeWare è disponibile su macOS, Windows, Linux (utilizzando GTK); su piattaforme mobili come Android e iOS; e per il Web.

CAPITOLO 2

Andiamo!

Siete pronti a provare BeeWare? *Costruiamo un'applicazione multipiattaforma in Python!*

2.1 Esercitazione 0 - Configuriamoci!

Prima di creare la nostra prima applicazione BeeWare, dobbiamo assicurarci di avere tutti i prerequisiti per l'esecuzione di BeeWare.

2.1.1 Installare Python

La prima cosa di cui abbiamo bisogno è un interprete Python funzionante.

macOS

Linux

Windows

Se si utilizza macOS, una versione recente di Python è inclusa in Xcode o negli strumenti di sviluppo a riga di comando. Per verificare se ne disponete già, eseguite il seguente comando:

```
$ python3 --version
```

Se Python è installato, verrà visualizzato il suo numero di versione. In caso contrario, verrà richiesto di installare gli strumenti di sviluppo a riga di comando.

Se si utilizza Windows, è possibile ottenere il programma di installazione ufficiale dal sito web di Python <https://www.python.org/downloads>`_. È possibile utilizzare qualsiasi versione stabile di Python dalla 3.8 in poi. Si consiglia di evitare le alpha, le beta e le release candidate a meno che non si sappia veramente cosa si sta facendo.

Se siete su Linux, installerete Python usando il gestore di pacchetti del sistema (`apt` su Debian/Ubuntu/Mint, `dnf` su Fedora, o `pacman` su Arch).

Dovete assicurarvi che il sistema Python sia Python 3.8 o più recente; se non lo è (ad esempio, Ubuntu 18.04 viene fornito con Python 3.6), dovrete aggiornare la vostra distribuzione Linux con qualcosa di più recente.

Il supporto per Raspberry Pi è al momento limitato.

Se si utilizza Windows, è possibile ottenere il programma di installazione ufficiale dal sito web di Python <https://www.python.org/downloads>`_. È possibile utilizzare qualsiasi versione stabile di Python dalla 3.8 in poi. Si consiglia di evitare le alpha, le beta e le release candidate a meno che non si sappia veramente cosa si sta facendo.

Distribuzioni Python alternative

Esistono molti modi diversi per installare Python. Si può installare Python tramite homebrew. Si può usare pyenv per gestire più installazioni di Python sulla stessa macchina. Gli utenti di Windows possono installare Python dal Windows App Store. Gli utenti con un background di scienza dei dati potrebbero voler usare Anaconda o Miniconda.

Se siete su macOS o Windows, non importa *come* avete installato Python: importa solo che possiate eseguire `python3` dal prompt dei comandi/terminale del vostro sistema operativo e ottenere un interprete Python funzionante.

Se utilizzate Linux, dovreste usare il Python di sistema fornito dal vostro sistema operativo. Sarete in grado di completare *la maggior parte* di questo tutorial utilizzando un Python non di sistema, ma non sarete in grado di pacchettizzare la vostra applicazione per distribuirla ad altri.

2.1.2 Installare le dipendenze

Successivamente, installate le dipendenze aggiuntive necessarie per il vostro sistema operativo:

macOS

Linux

Windows

La creazione di applicazioni BeeWare su macOS richiede:

- **Git**, un sistema di controllo delle versioni. È incluso in Xcode o negli strumenti per sviluppatori a riga di comando, installati in precedenza.

Per supportare lo sviluppo locale, è necessario installare alcuni pacchetti di sistema. L'elenco dei pacchetti necessari varia a seconda della distribuzione:

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 creazione di applicazioni BeeWare su Windows richiede:

- **Git**, un sistema di controllo delle versioni. È possibile scaricare Git da git-scm.org.

Dopo l'installazione di questi strumenti, è necessario assicurarsi di riavviare tutte le sessioni di terminale. Windows esporrà i terminali degli strumenti appena installati solo dopo il completamento dell'installazione.

2.1.3 Impostare un ambiente virtuale

Ora creeremo un ambiente virtuale, una «sandbox» che potremo usare per isolare il nostro lavoro su questo tutorial dalla nostra installazione principale di Python. Se installiamo dei pacchetti nell'ambiente virtuale, la nostra installazione principale di Python (e qualsiasi altro progetto Python sul nostro computer) non ne risentirà. Se facciamo un pasticcio completo nel nostro ambiente virtuale, potremo semplicemente cancellarlo e ricominciare da capo, senza influenzare nessun altro progetto Python sul nostro computer e senza la necessità di reinstallare 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 -m venv beeware-venv
C:\...>beeware-venv\Scripts\activate
```

Errori nell'esecuzione degli script PowerShell

Se si utilizza PowerShell e si riceve l'errore:

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

L'account di Windows non ha i permessi per eseguire gli script. Per risolvere il problema:

1. Eseguire Windows PowerShell come amministratore.
2. Eseguire `set-executionpolicy RemoteSigned`
3. Selezionare `Y` per modificare il criterio di esecuzione.

Una volta fatto questo, si può rieseguire `beeware-venv\Scripts\activate.ps1` nella sessione PowerShell originale (o in una nuova sessione nella stessa directory).

Se questo ha funzionato, il prompt dovrebbe essere cambiato: dovrebbe avere il prefisso `(beeware-venv)`. Questo permette di sapere che ci si trova nell'ambiente virtuale BeeWare. Ogni volta che si lavora a questo tutorial, bisogna assicurarsi che l'ambiente virtuale sia attivato. In caso contrario, eseguire nuovamente l'ultimo comando (il comando `activate`) per riattivare l'ambiente.

Ambienti virtuali alternativi

Se si usa Anaconda o miniconda, forse si ha più familiarità con l'uso degli ambienti conda. Potreste anche aver sentito parlare di `virtualenv`, un predecessore del modulo `venv` integrato in Python. Come per le installazioni di Python, se siete su macOS o Windows, non importa *come* create il vostro ambiente virtuale, purché ne abbiate uno. Se siete su Linux, dovreste usare `venv` e il Python di sistema.

2.1.4 Prossimi passi

Ora abbiamo configurato il nostro ambiente. Siamo pronti a *creare la nostra prima applicazione BeeWare*.

2.2 Esercitazione 1 - La vostra prima applicazione

Siamo pronti a creare la nostra prima applicazione.

2.2.1 Installare gli strumenti BeeWare

Per prima cosa, è necessario installare **Briefcase**. Briefcase è uno strumento di BeeWare che può essere usato per confezionare l'applicazione da distribuire agli utenti finali, ma può anche essere usato per avviare un nuovo progetto. Assicuratevi di essere nella cartella `beeware-tutorial` creata in *Tutorial 0*, con l'ambiente virtuale `beeware-venv` attivato ed eseguite:

macOS

Linux

Windows

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

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

Possibili errori durante l'installazione

Se si riscontrano errori durante l'installazione, è quasi certamente perché alcuni dei requisiti di sistema non sono stati installati. Assicurarsi di aver *installato tutti i pre-requisiti della piattaforma*.

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

Possibili errori durante l'installazione

È importante utilizzare `python -m pip`, piuttosto che un semplice `pip`. Briefcase deve assicurarsi di avere una versione aggiornata di `pip` e `setuptools`, e un'invocazione nuda di `pip` non può autoaggiornarsi. Se volete saperne di più, Brett Cannon ha un post dettagliato sul blog sul problema.

Uno degli strumenti di BeeWare è **Briefcase**. Briefcase può essere utilizzato per confezionare l'applicazione da distribuire agli utenti finali, ma anche per avviare un nuovo progetto.

2.2.2 Bootstrap di un nuovo progetto

Avviamo il nostro primo progetto BeeWare! Useremo il comando `new` di Briefcase per creare un'applicazione chiamata **Hello World**. Eseguite il seguente comando dal prompt dei comandi:

macOS

Linux

Windows

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

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

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

Briefcase ci chiederà alcuni dettagli della nostra nuova applicazione. Ai fini di questa esercitazione, utilizzate quanto segue:

- **Nome formale** - Accettare il valore predefinito: `Hello World`.
- **Nome applicazione** - Accettare il valore predefinito: `helloworld`.
- **Bundle** - Se si possiede un proprio dominio, inserirlo in ordine inverso. (Ad esempio, se si possiede il dominio «cupcakes.com», inserire `com.cupcakes` come bundle). Se non si possiede un dominio proprio, accettare il bundle predefinito (`com.example`).
- **Nome del progetto** - Accettare il valore predefinito: `Hello World`.
- **Descrizione** - Accettare il valore predefinito (o, se si vuole essere molto creativi, creare una descrizione personalizzata)
- **Autore** - Inserire qui il proprio nome.
- **Email dell'autore** - Inserire il proprio indirizzo e-mail. Verrà utilizzato nel file di configurazione, nel testo della guida e ovunque sia richiesto un indirizzo e-mail quando si invia l'applicazione a un app store.
- **URL** - L'URL della pagina di destinazione dell'applicazione. Anche in questo caso, se si possiede un dominio proprio, inserire un URL di quel dominio (compreso `https://`). Altrimenti, accettare l'URL predefinito (`https://example.com/helloworld`). Questo URL non deve esistere (per ora); sarà usato solo se si pubblica l'applicazione su un app store.
- **Licenza** - Accettare la licenza predefinita (BSD). Questo non influisce in alcun modo sul funzionamento del tutorial, quindi se si hanno sentimenti particolarmente forti riguardo alla scelta della licenza, si può scegliere liberamente un'altra licenza.
- **Quadro GUI** - Accettare l'opzione predefinita, Toga (il toolkit GUI di BeeWare).

Briefcase genererà quindi uno scheletro di progetto da utilizzare. Se avete seguito questo tutorial fino a questo punto e avete accettato le impostazioni predefinite come descritto, il vostro file system dovrebbe avere un aspetto simile a:

```
beeware-tutorial/
    beeware-venv/
        ...
    helloworld/
        CHANGELOG
        LICENSE
        README.rst
        pyproject.toml
        src/
            helloworld/
                resources/
                    helloworld.icns
                    helloworld.ico
                    helloworld.png
                __init__.py
                __main__.py
                app.py
        tests/
            __init__.py
            helloworld.py
            test_app.py
```

Questo scheletro è in realtà un'applicazione completamente funzionante, senza aggiungere altro. La cartella `src` contiene tutto il codice dell'applicazione, la cartella `tests` contiene una suite di test iniziale e il file `pyproject.toml` descrive come confezionare l'applicazione per la distribuzione. Se si apre `pyproject.toml` in un editor, si vedranno i dettagli di configurazione appena forniti a Briefcase.

Ora che abbiamo un'applicazione stub, possiamo usare Briefcase per eseguire l'applicazione.

2.2.3 Eseguire l'applicazione in modalità sviluppatore

Spostatevi nella cartella del progetto `helloworld` e dite a briefcase di avviare il progetto in modalità sviluppatore (o `dev`):

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
```

(continues on next page)

(continua dalla pagina precedente)

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

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

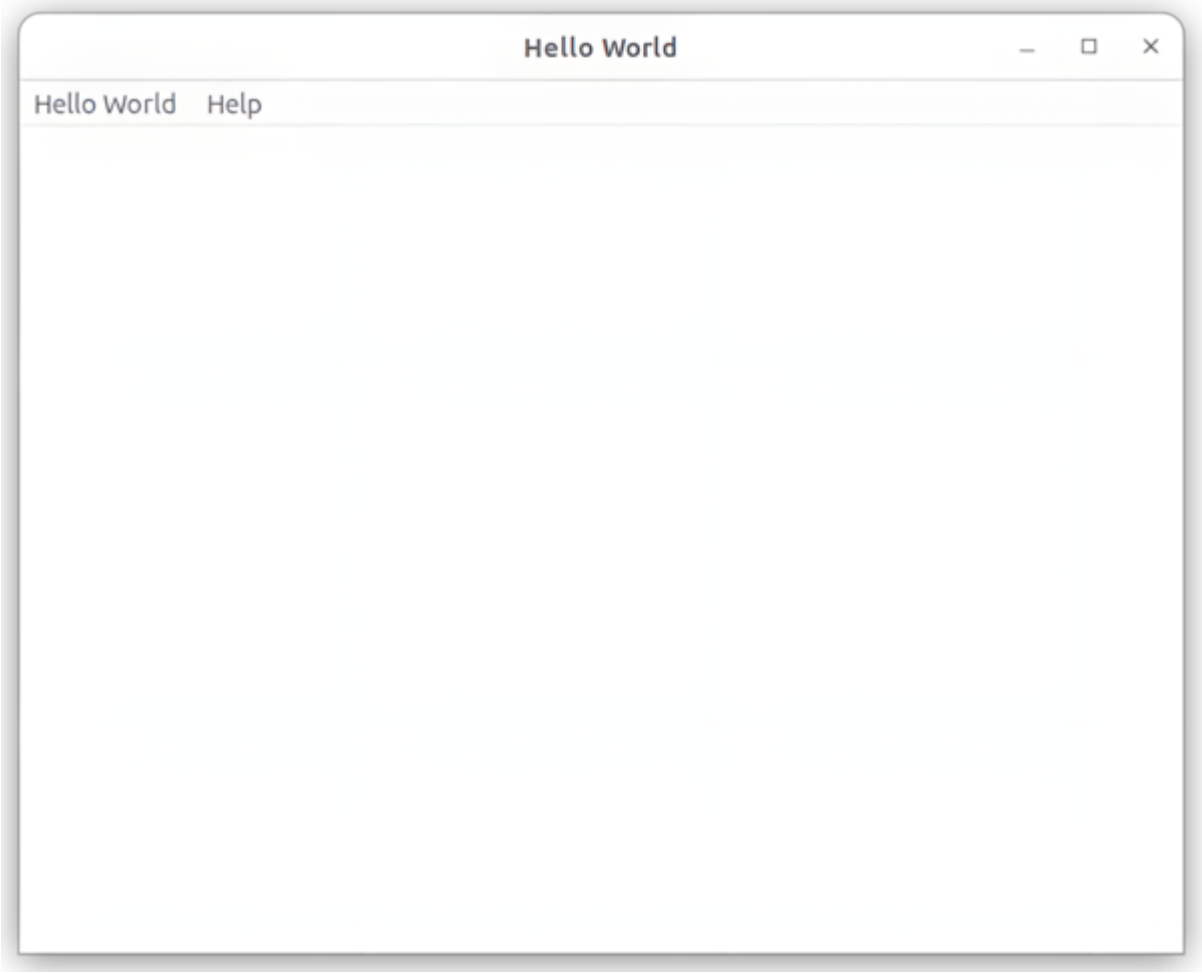
```
(beeware-venv) C:\...>cd helloworld
(beeware-venv) C:\...>briefcase dev

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

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

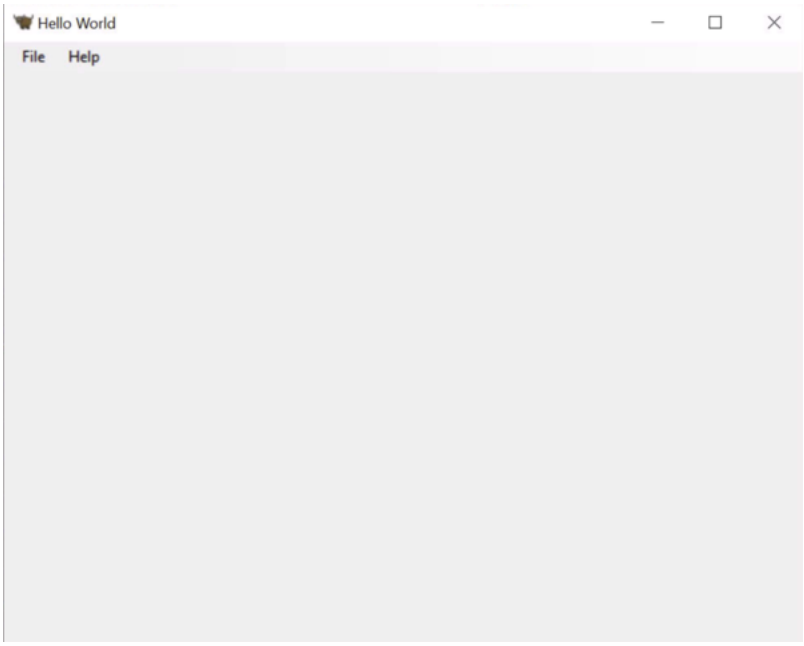
Si dovrebbe aprire una finestra dell'interfaccia grafica:

macOS

Linux

Windows





Premete il pulsante di chiusura (o selezionate Quit dal menu dell'applicazione) e il gioco è fatto! Congratulazioni: avete appena scritto un'applicazione nativa standalone in Python!

2.2.4 Prossimi passi

Ora abbiamo un'applicazione funzionante, eseguita in modalità sviluppatore. Ora possiamo aggiungere un po" di logica per far fare alla nostra applicazione qualcosa di più interessante. In *Tutorial 2*, realizzeremo un'interfaccia utente più utile per la nostra applicazione.

2.3 Esercitazione 2 - Rendere interessante

In *Tutorial 1*, abbiamo generato un progetto stub in grado di funzionare, ma non abbiamo scritto alcun codice. Diamo un'occhiata a ciò che è stato generato per noi.

2.3.1 Cosa è stato generato

Nella cartella `src/helloworld`, si dovrebbero vedere 3 file: `__init__.py`, `__main__.py` e `app.py`.

`__init__.py` segna la cartella `helloworld` come un modulo Python importabile. È un file vuoto; il solo fatto che esista indica all'interprete Python che la cartella `helloworld` definisce un modulo.

`__main__.py` segna il modulo `helloworld` come un tipo speciale di modulo, un modulo eseguibile. Se si cerca di eseguire il modulo `helloworld` usando `python -m helloworld`, il file `__main__.py` è il punto in cui Python inizierà l'esecuzione. Il contenuto di `__main__.py` è relativamente semplice:

```
from helloworld.app import main

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

Cioè, importa il metodo `main` dall'applicazione `helloworld` e, se viene eseguito come punto di ingresso, chiama il metodo main() e avvia il ciclo principale dell'applicazione. Il ciclo principale è il modo in cui un'applicazione GUI ascolta gli input dell'utente (come i clic del mouse e la pressione della tastiera).

Il file più interessante è `app.py`: contiene la logica che crea la finestra della nostra applicazione:

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

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

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

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

Esaminiamo questa riga per riga:

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

Per prima cosa, importiamo il toolkit di widget `toga` e alcune classi e costanti di utilità legate allo stile. Il nostro codice non le usa ancora, ma le useremo a breve.

Quindi, definiamo una classe:

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

Ogni applicazione Toga ha una singola istanza `toga.App`, che rappresenta l'entità in esecuzione che è l'applicazione. L'applicazione può finire per gestire più finestre, ma per le applicazioni semplici ci sarà una sola finestra principale.

Quindi, definiamo un metodo `startup()`:

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

La prima cosa che il metodo di avvio fa è definire un riquadro principale. Lo schema di layout di Toga si comporta in modo simile all'HTML. Si costruisce un'applicazione costruendo un insieme di riquadri, ognuno dei quali contiene altri riquadri, o widget veri e propri. Si applicano poi degli stili a questi riquadri per definire il modo in cui consumeranno lo spazio disponibile della finestra.

In questa applicazione, definiamo una singola casella, ma non inseriamo nulla al suo interno.

Quindi, definiamo una finestra in cui inserire questa casella vuota:

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

Questo crea un'istanza di `toga.MainWindow`, che avrà un titolo corrispondente al nome dell'applicazione. Una finestra principale è un tipo speciale di finestra in Toga: è una finestra strettamente legata al ciclo di vita dell'applicazione. Quando la finestra principale viene chiusa, l'applicazione esce. La finestra principale è anche la finestra che contiene il menu dell'applicazione (se si utilizza una piattaforma come Windows in cui le barre dei menu fanno parte della finestra)

Aggiungiamo quindi la nostra casella vuota come contenuto della finestra principale e istruiamo l'applicazione a mostrare la nostra finestra:

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

Infine, definiamo un metodo `main()`. Questo è ciò che crea l'istanza della nostra applicazione:

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

Questo metodo `main()` è quello che viene importato e invocato da `__main__.py`. Crea e restituisce un'istanza della nostra applicazione `HelloWorld`.

Questa è l'applicazione Toga più semplice possibile. Inseriamo nell'applicazione alcuni contenuti personali e facciamo in modo che l'applicazione faccia qualcosa di interessante.

2.3.2 Aggiunta di contenuti propri

Modificate la classe `HelloWorld` all'interno di `src/helloworld/app.py` in modo che abbia questo aspetto:

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

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

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

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

        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}")
```

Nota: Non rimuovete le importazioni all'inizio del file o il `main()` in fondo. È necessario aggiornare solo la classe `HelloWorld`.

Vediamo nel dettaglio cosa è cambiato.

Stiamo ancora creando un riquadro principale, ma ora stiamo applicando uno stile:

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

Il sistema di layout integrato di Toga si chiama «Pack». Si comporta in modo molto simile ai CSS. Si definiscono gli oggetti in una gerarchia: in HTML, gli oggetti sono , e altri elementi DOM; in Toga, sono widget e box. Si possono poi assegnare stili ai singoli elementi. In questo caso, stiamo indicando che si tratta di un riquadro `COLUMN`, cioè un riquadro che consumerà tutta la larghezza disponibile e si espanderà in altezza man mano che si aggiungono contenuti, ma cercherà di essere il più corto possibile.

Successivamente, definiamo un paio di widget:

```
name_label = toga.Label(
    "Your name: ",
```

(continues on next page)

(continua dalla pagina precedente)

```
    style=Pack(padding=(0, 5))
)
self.name_input = toga.TextInput(style=Pack(flex=1))
```

Qui definiamo una Label e un TextInput. A entrambi i widget sono associati degli stili; l'etichetta avrà un padding di 5px a sinistra e a destra e nessun padding in alto e in basso. Il TextInput è contrassegnato come flessibile, cioè assorbirà tutto lo spazio disponibile nel suo asse di layout.

Il TextInput è assegnato come variabile di istanza della classe. Questo ci consente di accedere facilmente all'istanza del widget, che utilizzeremo tra poco.

Quindi, definiamo un riquadro per contenere questi due widget:

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

Il `nome_box` è un box come quello principale, ma questa volta è un box `ROW`. Ciò significa che il contenuto sarà aggiunto orizzontalmente e cercherà di avere una larghezza il più possibile ridotta. Il riquadro ha anche un padding di 5px su tutti i lati.

Ora definiamo un pulsante:

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

Il pulsante ha anche 5px di padding su tutti i lati. Definiamo anche un *handler*, un metodo da invocare quando il pulsante viene premuto.

Quindi, aggiungiamo la casella del nome e il pulsante alla casella principale:

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

Questo completa il nostro layout; il resto del metodo di avvio è come in precedenza: definire una MainWindow e assegnare il riquadro principale come contenuto della finestra:

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

L'ultima cosa da fare è definire il gestore del pulsante. Un gestore può essere qualsiasi metodo, generatore o co-routine asincrona; accetta come argomento il widget che ha generato l'evento e sarà invocato ogni volta che il pulsante viene premuto:

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

Il corpo del metodo è una semplice istruzione di stampa, che però interroga il valore corrente dell'input name e utilizza il suo contenuto come testo stampato.

Ora che abbiamo apportato queste modifiche, possiamo vederne l'aspetto avviando nuovamente l'applicazione. Come prima, useremo la modalità sviluppatore:

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...
========================================================================
```

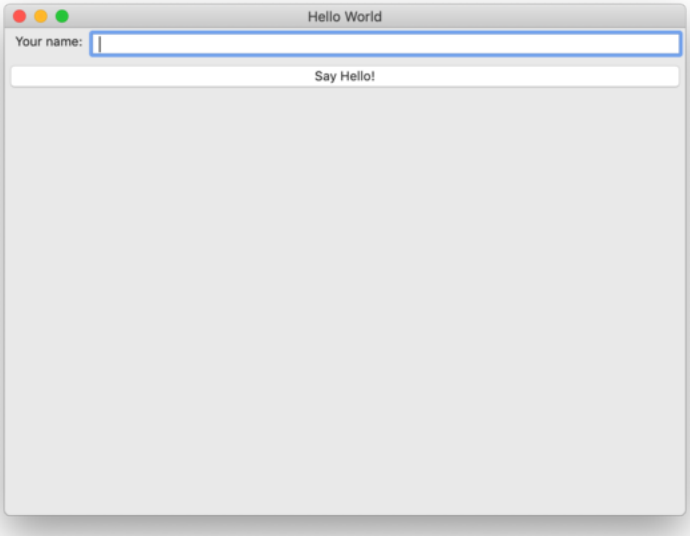
Si noterà che questa volta *non* installa le dipendenze. Briefcase è in grado di rilevare che l'applicazione è già stata eseguita in precedenza e, per risparmiare tempo, eseguirà solo l'applicazione. Se si aggiungono nuove dipendenze alla propria applicazione, ci si può assicurare che vengano installate passando l'opzione `-r` quando si esegue `briefcase dev`.

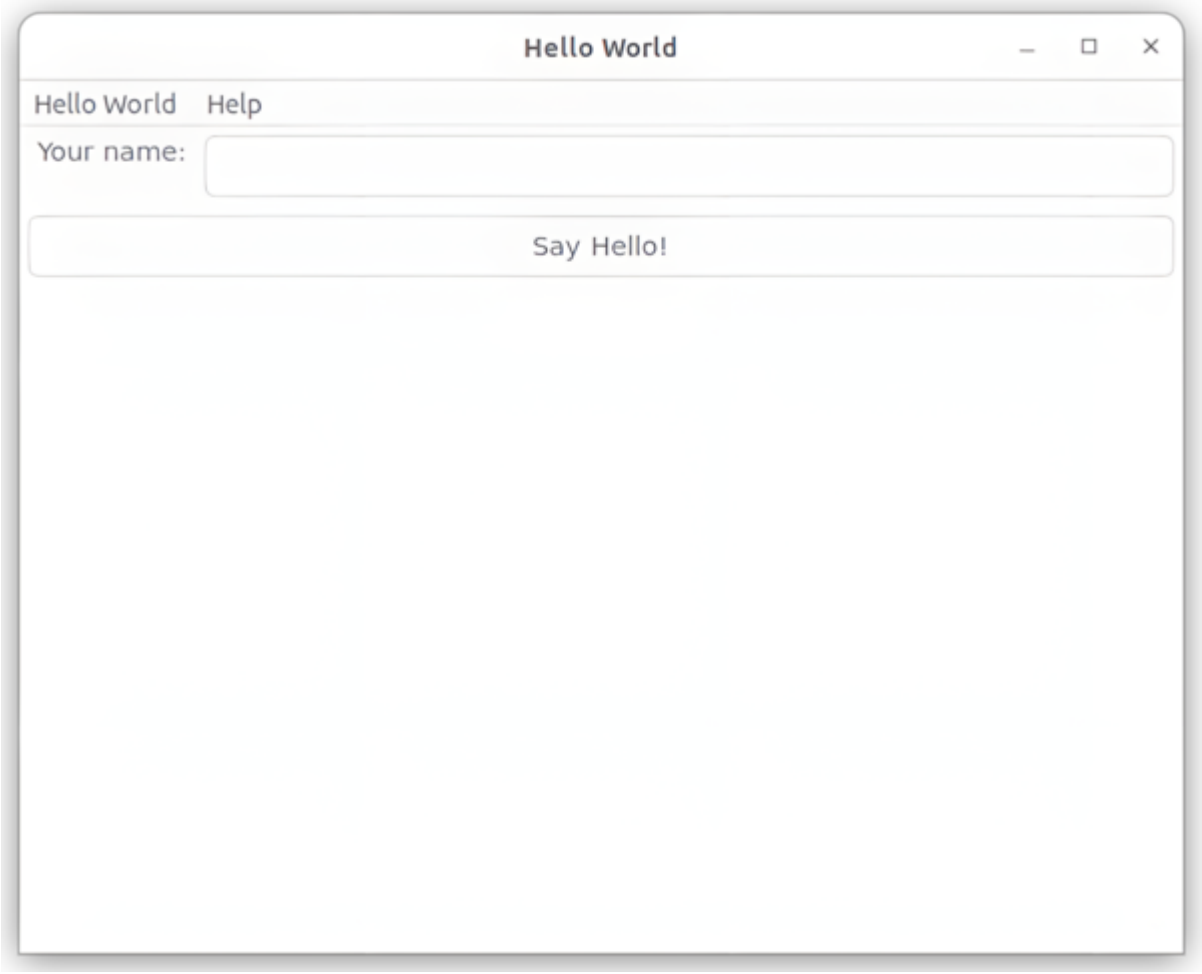
Si dovrebbe aprire una finestra dell'interfaccia grafica:

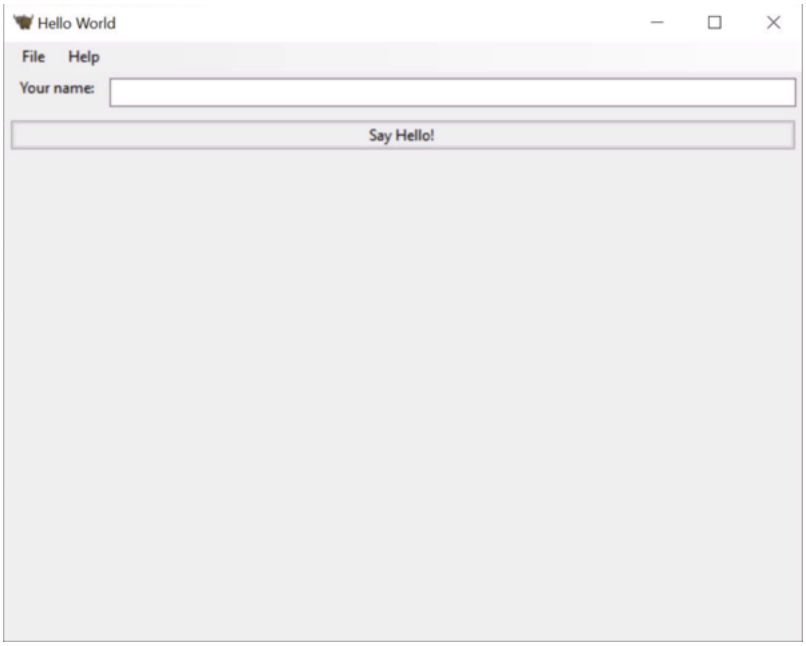
macOS

Linux

Windows







Se si inserisce un nome nella casella di testo e si preme il pulsante GUI, si dovrebbe vedere l'output nella console in cui è stata avviata l'applicazione.

2.3.3 Prossimi passi

Ora abbiamo un'applicazione che fa qualcosa di più interessante. Ma funziona solo sul nostro computer. Impacchettiamo questa applicazione per la distribuzione. In *Tutorial 3*, impacchetteremo la nostra applicazione come un programma di installazione autonomo da inviare a un amico, a un cliente o da caricare su un App Store.

2.4 Esercitazione 3 - Confezionamento per la distribuzione

Finora abbiamo eseguito la nostra applicazione in «modalità sviluppatore». Questo ci consente di eseguire facilmente la nostra applicazione a livello locale, ma ciò che vogliamo veramente è poterla fornire ad altri.

Tuttavia, non vogliamo insegnare ai nostri utenti come installare Python, creare un ambiente virtuale, clonare un repository git ed eseguire Briefcase in modalità sviluppatore. Preferiamo dare loro un programma di installazione e far sì che l'applicazione funzioni e basta.

Briefcase può essere utilizzato per impacchettare l'applicazione per la distribuzione in questo modo.

2.4.1 Creare lo scaffold dell'applicazione

Poiché è la prima volta che impacchettiamo la nostra applicazione, dobbiamo creare alcuni file di configurazione e altre impalcature per supportare il processo di impacchettamento. Dalla cartella `helloworld`, eseguire:

macOS

Linux

Windows

```
(beeware-venv) $ briefcase create

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

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

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

[helloworld] Installing requirements...
...

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

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

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

```
(beeware-venv) $ briefcase create

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

Targeting glibc 2.35
Targeting Python3.10

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

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

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

[helloworld] Installing requirements...
...

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

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

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

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

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

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

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

[helloworld] Installing requirements...
...

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

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

Probabilmente avete appena visto passare delle pagine di contenuto nel vostro terminale… cosa è successo? Briefcase

ha fatto quanto segue:

1. Ha **generato un modello di applicazione**. Per costruire un programma di installazione nativo sono necessari molti file e configurazioni, oltre al codice dell'applicazione vera e propria. Questa impalcatura aggiuntiva è quasi la stessa per ogni applicazione sulla stessa piattaforma, tranne che per il nome dell'applicazione vera e propria che si sta costruendo, quindi Briefcase fornisce un modello di applicazione per ogni piattaforma che supporta. Questo passo esegue il modello, sostituendo il nome dell'applicazione, l'ID del bundle e altre proprietà del file di configurazione, come richiesto per supportare la piattaforma su cui si sta costruendo.

   Se non si è soddisfatti del modello fornito da Briefcase, è possibile crearne uno proprio. Tuttavia, probabilmente non è il caso di farlo prima di aver acquisito un po" di esperienza nell'uso del modello predefinito di Briefcase.

2. Ha **scaricato e installato un pacchetto di supporto**. L'approccio alla pacchettizzazione adottato da Briefcase è meglio descritto come «la cosa più semplice che possa funzionare»: spedisce un interprete Python completo e isolato come parte di ogni applicazione che costruisce. Questo è leggermente inefficiente dal punto di vista dello spazio: se si hanno 5 applicazioni pacchettizzate con Briefcase, si avranno 5 copie dell'interprete Python. Tuttavia, questo approccio garantisce che ogni applicazione sia completamente indipendente, utilizzando una versione specifica di Python che è nota per funzionare con l'applicazione.

   Anche in questo caso, Briefcase fornisce un pacchetto di supporto predefinito per ogni piattaforma; se lo si desidera, si può fornire il proprio pacchetto di supporto e farlo includere come parte del processo di compilazione. Questo può essere utile se si hanno particolari opzioni nell'interprete Python che devono essere abilitate, o se si vogliono togliere dalla libreria standard i moduli che non servono in fase di esecuzione.

   Briefcase mantiene una cache locale dei pacchetti di supporto, per cui una volta scaricato un pacchetto di supporto specifico, la copia in cache verrà utilizzata nelle build future.

3. Si tratta di **requisiti dell'applicazione installata**. L'applicazione può specificare qualsiasi modulo di terze parti richiesto in fase di esecuzione. Questi saranno installati usando `pip` nel programma di installazione dell'applicazione.

4. Ha **installato il codice dell'applicazione**. L'applicazione avrà il suo codice e le sue risorse (ad esempio, le immagini necessarie per l'esecuzione); questi file vengono copiati nel programma di installazione.

5. Infine, aggiunge qualsiasi risorsa aggiuntiva necessaria al programma di installazione stesso. Questo include cose come le icone che devono essere allegate all'applicazione finale e le immagini della schermata iniziale.

Una volta completata questa operazione, se si guarda nella cartella del progetto, si dovrebbe vedere una cartella corrispondente alla propria piattaforma (`macOS`, `linux` o `windows`) che contiene file aggiuntivi. Questa è la configurazione di pacchettizzazione specifica della piattaforma per l'applicazione.

2.4.2 Creazione dell'applicazione

È ora possibile compilare l'applicazione. Questo passaggio esegue la compilazione binaria necessaria affinché l'applicazione sia eseguibile sulla piattaforma di destinazione.

macOS

Linux

Windows

```
(beeware-venv) $ briefcase build

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

(continues on next page)

(continua dalla pagina precedente)

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

Su macOS, il comando `build` non ha bisogno di *compilare* nulla, ma deve firmare il contenuto del binario in modo che possa essere eseguito. Questa firma è una firma *ad hoc*: funzionerà solo sulla *vostra* macchina; se volete distribuire l'applicazione ad altri, dovrete fornire una firma completa.

```
(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
Update file permissions...
...
Updating file permissions... done
Stripping binary... done

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

Una volta completato questo passaggio, la cartella `build` conterrà una cartella `helloworld-0.0.1` che contiene un mirror di un file system Linux `/usr`. Questo file system mirror conterrà una cartella `bin` con un binario di `helloworld`, oltre alle cartelle `lib` e `share` necessarie per supportare il binario.

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

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

Su Windows, il comando `build` non ha bisogno di *compilare* nulla, ma deve scrivere alcuni metadati in modo che l'applicazione conosca il suo nome, la sua versione e così via.

Attivazione dell'antivirus

Poiché questi metadati vengono scritti direttamente nel binario precompilato che viene lanciato dal modello durante il comando `create`, è possibile che il software antivirus in esecuzione sul computer impedisca la scrittura dei metadati. In

questo caso, istruire l'antivirus per consentire l'esecuzione dello strumento (chiamato `rcedit-x64.exe`) e rieseguire il comando precedente.

2.4.3 Esecuzione dell'applicazione

Ora è possibile utilizzare Briefcase per eseguire l'applicazione:

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

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:
```

(continues on next page)

(continua dalla pagina precedente)

```
- /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
--------------------------------------------------------------------------------
```

In questo modo si avvia l'esecuzione dell'applicazione nativa, utilizzando l'output del comando `build`.

È possibile notare alcune piccole differenze nell'aspetto dell'applicazione quando è in esecuzione. Ad esempio, le icone e il nome visualizzati dal sistema operativo potrebbero essere leggermente diversi da quelli visualizzati durante l'esecuzione in modalità sviluppatore. Ciò è dovuto anche al fatto che si sta utilizzando l'applicazione pacchettizzata e non solo il codice Python in esecuzione. Dal punto di vista del sistema operativo, ora si sta eseguendo «un'applicazione», non «un programma Python», e questo si riflette sull'aspetto dell'applicazione.

2.4.4 Creazione del programma di installazione

È ora possibile pacchettizzare l'applicazione per la distribuzione, utilizzando il comando `package`. Il comando package esegue qualsiasi compilazione necessaria per convertire il progetto scaffolded in un prodotto finale distribuibile. A seconda della piattaforma, ciò può comportare la compilazione di un programma di installazione, l'esecuzione della firma del codice o altre operazioni preliminari alla distribuzione.

macOS

Linux

Windows

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

[helloworld] Signing app...
```

(continues on next page)

(continua dalla pagina precedente)

```
********************************************************************************
** 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
```

La cartella `dist` conterrà un file chiamato `Hello World-0.0.1.dmg`. Se individuate questo file nel Finder e fate doppio clic sulla sua icona, monterete il DMG, ottenendo una copia dell'applicazione Hello World e un collegamento alla cartella Applicazioni per facilitare l'installazione. Trascinate il file dell'app in Applicazioni e avrete installato la vostra applicazione. Inviate il file DMG a un amico, che dovrebbe essere in grado di fare lo stesso.

In questo esempio, abbiamo usato l'opzione `--adhoc-sign`, cioè stiamo firmando la nostra applicazione con credenziali *ad hoc*, credenziali temporanee che funzioneranno solo sul vostro computer. Abbiamo fatto questo per mantenere il tutorial semplice. L'impostazione delle identità di firma del codice è un po" complicata e sono *richieste* solo se si intende distribuire la propria applicazione ad altri. Se stessimo pubblicando un'applicazione reale da utilizzare, dovremmo specificare delle credenziali reali.

Quando si è pronti a pubblicare un'applicazione reale, consultare la guida Briefcase How-To su Impostazione di un'identità di firma del codice macOS

L'output del passaggio del pacchetto sarà leggermente diverso a seconda della vostra distribuzione Linux. Se si utilizza una distribuzione derivata da Debian, si vedrà:

```
(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'.
```

(continues on next page)

(continua dalla pagina precedente)

```
Building Debian package... done

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

La cartella `dist` conterrà il file `.deb` generato.

Se si utilizza una distribuzione basata su RHEL, si vedrà:

```
(beeware-venv) $ briefcase package

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

Targeting glibc 2.35
Targeting Python3.10

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

Write RPM spec file... done

Building source archive... done

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

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

La cartella `dist` conterrà il file `.rpm` generato.

Se si utilizza una distribuzione basata su Arch, si vedrà:

```
(beeware-venv) $ briefcase package

[helloworld] Finalizing application configuration...
Targeting arch:rolling (Vendor base arch)
Determining glibc version... done
Targeting glibc 2.37
Targeting Python3.10

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

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

La cartella `dist` conterrà il file `.pkg.tar.zst` che è stato generato.

Altre distribuzioni Linux non sono attualmente supportate per il packaging.

Se volete creare un pacchetto per una distribuzione Linux diversa da quella che state usando, Briefcase può aiutarvi, ma dovrete installare Docker.

Gli installatori ufficiali di Docker Engine sono disponibili per diverse distribuzioni Unix. Seguite le istruzioni per la vostra piattaforma; tuttavia, assicuratevi di non installare Docker in modalità «rootless».

Una volta installato Docker, dovreste essere in grado di avviare un contenitore Linux, ad esempio:

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

mostrerà un prompt Unix (qualcosa come `root@84444e31cff9:/#`) all'interno di un contenitore Docker Ubuntu 22.04. Digitare Ctrl-D per uscire da Docker e tornare alla shell locale.

Una volta installato Docker, si può usare Briefcase per creare un pacchetto per qualsiasi distribuzione Linux supportata da Briefcase, passando un'immagine Docker come argomento. Per esempio, per creare un pacchetto DEB per Ubuntu 22.04 (Jammy), indipendentemente dal sistema operativo in uso, si può eseguire:

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

Questo scaricherà l'immagine Docker per il sistema operativo selezionato, creerà un contenitore in grado di eseguire le build di Briefcase e costruirà il pacchetto dell'applicazione all'interno dell'immagine. Una volta completato, la cartella `dist` conterrà il pacchetto per la distribuzione Linux di destinazione.

```
(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

Linking application installer...
Linking... done

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

In questo esempio, abbiamo usato l'opzione `--adhoc-sign`, cioè stiamo firmando la nostra applicazione con credenziali *ad hoc*, credenziali temporanee che funzioneranno solo sul vostro computer. Abbiamo fatto questo per mantenere il tutorial semplice. L'impostazione delle identità di firma del codice è un po" complicata e sono *richieste* solo se si intende distribuire la propria applicazione ad altri. Se stessimo pubblicando un'applicazione reale da utilizzare, dovremmo specificare delle credenziali reali.

Quando si è pronti a pubblicare un'applicazione reale, consultare la guida Briefcase How-To su Impostazione di un'identità di firma del codice macOS

Una volta completato questo passaggio, la cartella `dist` conterrà un file chiamato `Hello_World-0.0.1.msi`. Se si fa doppio clic su questo programma di installazione per eseguirlo, si dovrebbe seguire il consueto processo di installazione di Windows. Una volta completata l'installazione, nel menu di avvio sarà presente una voce «Hello World».

2.4.5 Prossimi passi

Ora la nostra applicazione è stata confezionata per essere distribuita su piattaforme desktop. Ma cosa succede quando dobbiamo aggiornare il codice della nostra applicazione? Come facciamo a inserire gli aggiornamenti nella nostra applicazione pacchettizzata? Consultate *Tutorial 4* per scoprirlo…

2.5 Esercitazione 4 - Aggiornamento dell'applicazione

Nell'ultima esercitazione abbiamo confezionato la nostra applicazione come applicazione nativa. Se avete a che fare con un'applicazione reale, la storia non finisce qui: probabilmente farete dei test, scoprirete dei problemi e dovrete apportare delle modifiche. Anche se la vostra applicazione è perfetta, alla fine vorrete pubblicare la versione 2 della vostra applicazione con dei miglioramenti.

Come si aggiorna l'applicazione installata quando si apportano modifiche al codice?

2.5.1 Aggiornamento del codice dell'applicazione

La nostra applicazione attualmente stampa sulla console quando si preme il pulsante. Tuttavia, le applicazioni GUI non dovrebbero utilizzare la console per l'output. Devono utilizzare le finestre di dialogo per comunicare con gli utenti.

Aggiungiamo una finestra di dialogo per dire ciao, invece di scrivere nella console. Modificare il callback `say_hello` in modo che assomigli a questo:

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

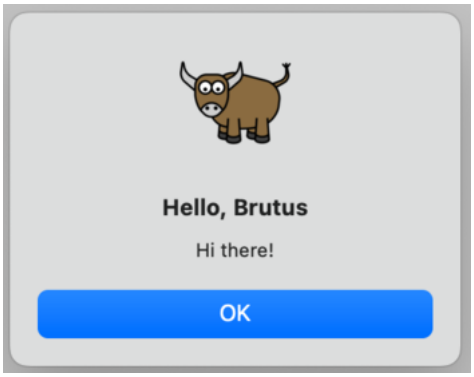
Questo indica a Toga di aprire una finestra di dialogo modale quando viene premuto il pulsante.

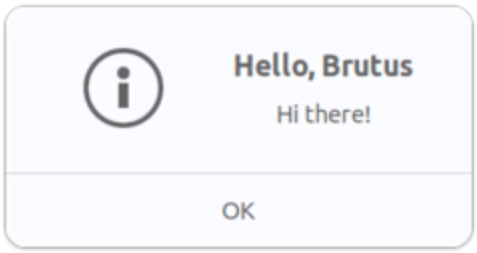
Se si esegue `briefcase dev`, si inserisce un nome e si preme il pulsante , si vedrà la nuova finestra di dialogo:

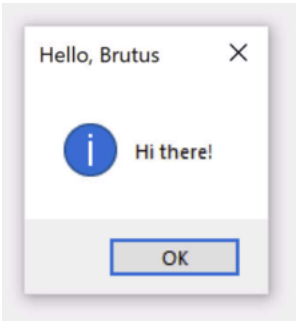
macOS

Linux

Windows







Tuttavia, se si esegue `briefcase run`, la finestra di dialogo non appare.

Perché? Beh, `briefcase dev` opera eseguendo il vostro codice sul posto - cerca di produrre un ambiente di runtime il più realistico possibile per il vostro codice, ma non fornisce o utilizza alcuna infrastruttura della piattaforma per avvolgere il vostro codice come un'applicazione. Parte del processo di impacchettamento dell'applicazione comporta la copia del codice *nel* bundle dell'applicazione e, al momento, l'applicazione contiene ancora il vecchio codice.

Quindi, dobbiamo dire a briefcase di aggiornare l'applicazione, copiando la nuova versione del codice. Potremmo farlo cancellando la vecchia cartella della piattaforma e ricominciando da zero. Tuttavia, Briefcase offre un modo più semplice: è possibile aggiornare il codice dell'applicazione esistente:

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

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

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

(continues on next page)

(continua dalla pagina precedente)

```
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.
```

Se Briefcase non riesce a trovare il modello impalcabile, invocherà automaticamente `create` per generare un nuovo scaffold.

Ora che abbiamo aggiornato il codice dell'installatore, possiamo eseguire `briefcase build` per ricompilare l'applicazione, `briefcase run` per eseguire l'applicazione aggiornata e `briefcase package` per riconfezionare l'applicazione per la distribuzione.

(Per gli utenti di macOS, ricordate che, come indicato in *Tutorial 3*, per il tutorial si consiglia di eseguire il pacchetto `briefcase` con il flag `--adhoc-sign` per evitare la complessità di impostare un'identità di firma del codice e mantenere il tutorial il più semplice possibile)

2.5.2 Aggiornamento ed esecuzione in un unico passaggio

Se state iterando rapidamente le modifiche al codice, è probabile che vogliate apportare una modifica al codice, aggiornare l'applicazione e rieseguirla immediatamente. Per la maggior parte degli scopi, la modalità sviluppatore (`briefcase dev`) è il modo più semplice per eseguire questo tipo di iterazione rapida; tuttavia, se si sta testando qualcosa sul modo in cui l'applicazione viene eseguita come binario nativo, o si sta cercando un bug che si manifesta solo quando l'applicazione è in forma pacchettizzata, potrebbe essere necessario utilizzare ripetute chiamate a `briefcase run`. Per semplificare il processo di aggiornamento ed esecuzione dell'applicazione in bundle, Briefcase ha una scorciatoia per supportare questo modello di utilizzo: l'opzione `-u` (o `--update`) del comando `run`.

Proviamo a fare un'altra modifica. Si sarà notato che se non si digita un nome nella casella di testo, la finestra di dialogo dirà «Ciao». Modifichiamo di nuovo la funzione `say_hello` per gestire questo caso limite.

All'inizio del file, tra le importazioni e la definizione della `classe HelloWorld`, aggiungiamo un metodo di utilità per generare un saluto appropriato a seconda del valore del nome fornito:

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

Quindi, modificare il callback `say_hello` per utilizzare questo nuovo metodo di utilità:

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

Eseguire l'applicazione in modalità di sviluppo (con `briefcase dev`) per verificare che la nuova logica funzioni; quindi aggiornare, compilare ed eseguire l'applicazione con un solo comando:

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...
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...
```

(continues on next page)

(continua dalla pagina precedente)

```
Installing src/helloworld... done

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

[helloworld] Application updated.

[helloworld] Starting app...
```

Il comando package accetta anche l'argomento `-u`, per cui se si apporta una modifica al codice dell'applicazione e si desidera eseguire immediatamente il repackaging, si può eseguire `briefcase package -u`.

2.5.3 Prossimi passi

Ora abbiamo la nostra applicazione confezionata per la distribuzione su piattaforme desktop e siamo stati in grado di aggiornare il codice della nostra applicazione.

Ma che dire della telefonia mobile? In *Tutorial 5*, convertiremo la nostra applicazione in un'applicazione mobile e la distribuiremo su un simulatore di dispositivo e su un telefono.

2.6 Esercitazione 5 - Il mobile

Finora abbiamo eseguito e testato la nostra applicazione sul desktop. Tuttavia, BeeWare supporta anche le piattaforme mobili e l'applicazione che abbiamo scritto può essere distribuita anche sul vostro dispositivo mobile!

iOS le applicazioni iOS possono essere compilate solo su macOS.

Costruiamo la nostra applicazione per iOS!

Android Le applicazioni Android possono essere compilate su macOS, Windows o Linux.

Costruiamo la nostra applicazione per Android!

2.6.1 Esercitazione 5 - Il mobile: iOS

Per compilare le applicazioni iOS è necessario Xcode, disponibile gratuitamente sull'App Store di macOS <https://apps.apple.com/au/app/xcode/id497799835?mt=12>`__.

Una volta installato Xcode, possiamo prendere la nostra applicazione e distribuirla come applicazione iOS.

Il processo di distribuzione di un'applicazione su iOS è molto simile a quello per la distribuzione come applicazione desktop. Per prima cosa, si esegue il comando `create`, ma questa volta si specifica che si vuole creare un'applicazione iOS:

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

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

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

(continues on next page)

(continua dalla pagina precedente)

```
...

[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
```

Una volta completata questa operazione, avremo una cartella `build/helloworld/ios/xcode` contenente un progetto Xcode, oltre alle librerie di supporto e al codice dell'applicazione necessario per l'applicazione.

Si può quindi usare Briefcase per compilare l'applicazione usando `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
```

Ora siamo pronti a eseguire la nostra applicazione, usando `briefcase run iOS`. Verrà richiesto di selezionare un dispositivo per il quale compilare; se sono stati installati simulatori per più versioni dell'SDK di iOS, potrebbe anche essere richiesto quale versione di iOS si desidera utilizzare. Le opzioni visualizzate potrebbero essere diverse da quelle mostrate in questo output; come minimo, l'elenco dei dispositivi sarà diverso. Per i nostri scopi, non importa quale simulatore si sceglie.

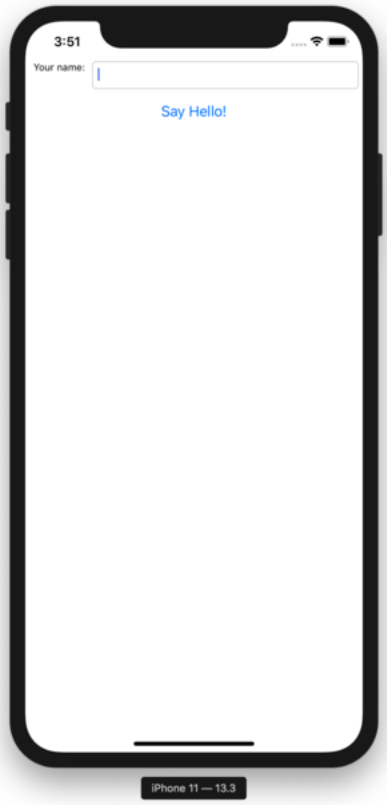
```
(beeware-venv) $ briefcase run iOS

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)
```

(continues on next page)

(continua dalla pagina precedente)

```
>  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)...
================================================================================
...
```

Questo avvierà il simulatore iOS, installerà la vostra applicazione e la avvierà. Si dovrebbe vedere il simulatore avviarsi e alla fine aprire l'applicazione iOS:



Se si sa in anticipo a quale simulatore iOS si vuole puntare, si può dire a Briefcase di usare quel simulatore fornendo un'opzione `-d` (o `--device`). Utilizzando il nome del dispositivo selezionato al momento della creazione dell'applicazione, eseguite:

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

Se sono disponibili più versioni di iOS, Briefcase sceglierà la versione di iOS più alta; se si vuole scegliere una versione di iOS in particolare, si dice di usare quella versione specifica:

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

In alternativa, è possibile assegnare un nome UDID a un dispositivo specifico:

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

Prossimi passi

Ora abbiamo un'applicazione sul nostro telefono! C'è un altro posto dove possiamo distribuire un'applicazione BeeWare? Consultate *Tutorial 6* per scoprirlo…

2.6.2 Esercitazione 5 - Il mobile: Android

Ora prenderemo la nostra applicazione e la distribuiremo come applicazione Android.

Il processo di distribuzione di un'applicazione su Android è molto simile a quello di un'applicazione desktop. Briefcase gestisce l'installazione delle dipendenze per Android, tra cui l'SDK Android, l'emulatore Android e un compilatore Java.

Creare un'applicazione Android e compilarla

Per prima cosa, eseguire il comando `create`. Questo scarica un modello di applicazione Android e vi aggiunge il codice 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 v0.3.14
...

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

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

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

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

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

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

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

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

(continues on next page)

(continua dalla pagina precedente)

```
...

[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) C:\...>briefcase create android

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

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

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

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

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

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

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

Quando si esegue `briefcase create android` per la prima volta, Briefcase scarica un JDK Java e l'SDK Android. Le dimensioni dei file e i tempi di download possono essere considerevoli; il download può richiedere un po" di tempo (10 minuti o più, a seconda della velocità della connessione a Internet). Al termine del download, verrà richiesto di accettare la licenza Android SDK di Google.

Una volta completata questa operazione, nel nostro progetto avremo una cartella `buildhelloworld\android\gradle`, che conterrà un progetto Android con una configurazione di compilazione Gradle. Questo progetto conterrà il codice dell'applicazione e un pacchetto di supporto contenente l'interprete Python.

Possiamo quindi usare il comando `build` di Briefcase per compilare questo file in un'applicazione Android APK.

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 può sembrare bloccato

Durante il passaggio `briefcase build android`, Gradle (lo strumento di creazione della piattaforma Android) stampa `CONFIGURING: 100%` e sembra non fare nulla. Non preoccupatevi, non è bloccato: sta scaricando altri componenti dell'SDK Android. A seconda della velocità della connessione a Internet, potrebbero essere necessari altri 10 minuti (o più). Questo ritardo dovrebbe verificarsi solo la prima volta che si esegue `build`; gli strumenti vengono memorizzati nella cache e nella compilazione successiva verranno utilizzate le versioni memorizzate nella cache.

Eseguire l'applicazione su un dispositivo virtuale

Ora siamo pronti a eseguire la nostra applicazione. È possibile utilizzare il comando `run` di Briefcase per eseguire l'applicazione su un dispositivo Android. Cominciamo con l'esecuzione su un emulatore Android.

Per eseguire la vostra applicazione, eseguite `briefcase run android`. In questo modo, verrà richiesto un elenco di dispositivi su cui è possibile eseguire l'applicazione. L'ultima voce sarà sempre un'opzione per creare un nuovo emulatore 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

>
```

Ora possiamo scegliere il dispositivo desiderato. Selezionate l'opzione «Crea un nuovo emulatore Android» e accettate la scelta predefinita per il nome del dispositivo (`beePhone`).

Briefcase `run` avvierà automaticamente il dispositivo virtuale. Quando il dispositivo si avvia, si vedrà il logo di Android:

Una volta terminato l'avvio del dispositivo, Briefcase installerà l'applicazione sul dispositivo. Verrà visualizzata brevemente una schermata di avvio:

L'applicazione si avvia. Durante l'avvio dell'applicazione verrà visualizzata una schermata iniziale:

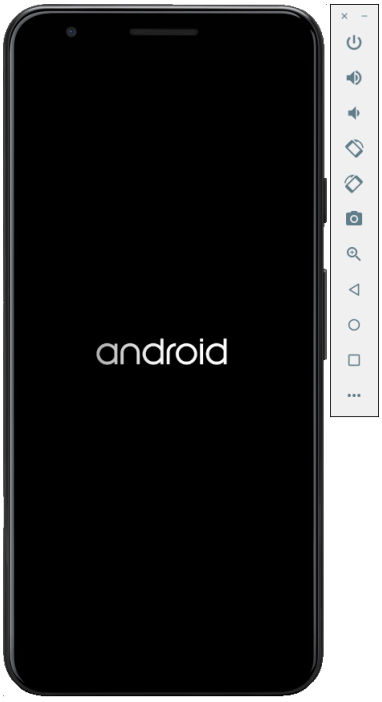


Fig. 1: Avvio del dispositivo virtuale Android

Fig. 2: Dispositivo virtuale Android completamente avviato, sulla schermata del launcher

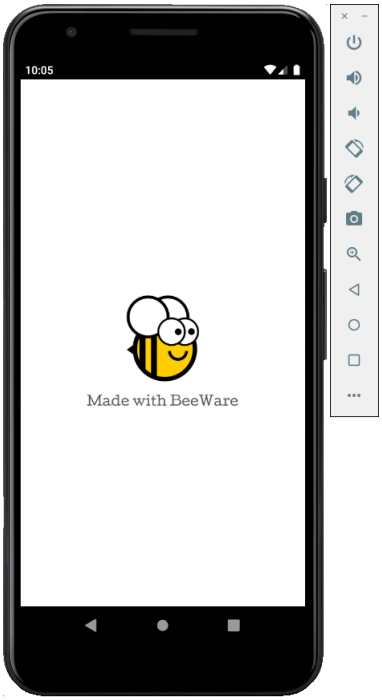


Fig. 3: Schermata iniziale dell'app

L'emulatore non si è avviato!

L'emulatore Android è un software complesso che si basa su una serie di caratteristiche dell'hardware e del sistema operativo, caratteristiche che potrebbero non essere disponibili o abilitate su macchine più vecchie. In caso di difficoltà nell'avvio dell'emulatore Android, consultare la sezione Requisiti e raccomandazioni della documentazione per sviluppatori Android.

Al primo avvio, l'applicazione deve scompattarsi sul dispositivo. Questa operazione può richiedere alcuni secondi. Una volta scompattata, verrà visualizzata la versione Android dell'applicazione desktop:

Se non si riesce a vedere l'avvio dell'applicazione, potrebbe essere necessario controllare il terminale in cui è stato eseguito `briefcase run` e cercare eventuali messaggi di errore.

In futuro, se si vuole eseguire su questo dispositivo senza usare il menu, si può fornire il nome dell'emulatore a Briefcase, usando `briefcase run android -d @beePhone` per eseguire direttamente sul dispositivo virtuale.

Eseguire l'applicazione su un dispositivo fisico

Se si dispone di un telefono o di un tablet Android fisico, è possibile collegarlo al computer con un cavo USB e quindi utilizzare la valigetta per puntare al dispositivo fisico.

Android richiede la preparazione del dispositivo prima di poterlo utilizzare per lo sviluppo. È necessario apportare due modifiche alle opzioni del dispositivo:

- Abilitare le opzioni per gli sviluppatori
- Abilitare il debug USB

I dettagli su come apportare queste modifiche sono disponibili nella documentazione per sviluppatori Android <https://developer.android.com/studio/debug/dev-options#enable>`__.

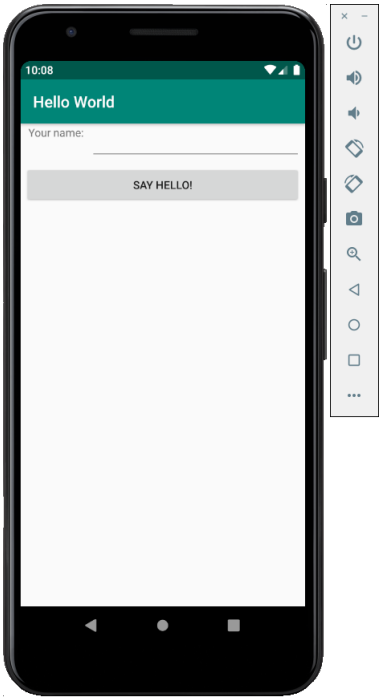


Fig. 4: Applicazione demo completamente lanciata

Una volta completati questi passaggi, il dispositivo dovrebbe apparire nell'elenco dei dispositivi disponibili quando si esegue `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

>
```

Qui possiamo vedere un nuovo dispositivo fisico con il suo numero di serie nell'elenco di distribuzione - in questo caso, un Pixel 3a. In futuro, se si vuole eseguire su questo dispositivo senza usare il menu, si può fornire il numero di serie del telefono a Briefcase (in questo caso, `briefcase run android -d 94ZZY0LNE8`). In questo modo si eseguirà direttamente sul dispositivo, senza richiedere nulla.

Il mio dispositivo non appare!

Se il dispositivo non compare in questo elenco, significa che non è stato attivato il debug USB (oppure che il dispositivo non è collegato!).

Se il dispositivo appare, ma è elencato come «Dispositivo sconosciuto (non autorizzato per lo sviluppo)», la modalità sviluppatore non è stata abilitata correttamente. Eseguire nuovamente «i passaggi per abilitare le opzioni per gli sviluppatori <https://developer.android.com/studio/debug/dev-options#enable>`__ e rieseguire `briefcase run android`.

Prossimi passi

Ora abbiamo un'applicazione sul nostro telefono! C'è un altro posto dove possiamo distribuire un'applicazione BeeWare? Consultate *Tutorial 6* per scoprirlo…

2.7 Tutorial 6 - Mettetelo sul web!

Oltre a supportare le piattaforme mobili, il toolkit di widget Toga supporta anche il Web! Utilizzando la stessa API usata per distribuire le applicazioni desktop e mobili, è possibile distribuire l'applicazione come applicazione web a pagina singola.

Prova di concetto

Il backend di Toga Web è il meno maturo di tutti i backend di Toga. È abbastanza maturo per mostrare alcune funzionalità, ma è probabile che sia pieno di bug e che manchino molti dei widget disponibili su altre piattaforme. In questo momento, l'implementazione del Web dovrebbe essere considerata una «prova di concetto»: sufficiente per dimostrare ciò che si può fare, ma non abbastanza per fare affidamento su uno sviluppo serio.

Se avete problemi con questa fase del tutorial, potete passare alla pagina successiva.

2.7.1 Distribuzione come applicazione web

Il processo di distribuzione come applicazione web a pagina singola segue lo stesso schema familiare: si crea l'applicazione, la si costruisce e la si esegue. Tuttavia, Briefcase può essere un po" intelligente; se si tenta di eseguire un'applicazione e Briefcase determina che non è stata creata o costruita per la piattaforma a cui è destinata, eseguirà le fasi di creazione e costruzione per voi. Poiché questa è la prima volta che eseguiamo l'applicazione per il Web, possiamo eseguire tutti e tre i passaggi con un solo comando:

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␣
↪v0.3.14
...

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

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

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

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

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

[helloworld] Created build/helloworld/web/static

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

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

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

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

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

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

(continues on next page)

(continua dalla pagina precedente)

```
[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)...
================================================================================
```

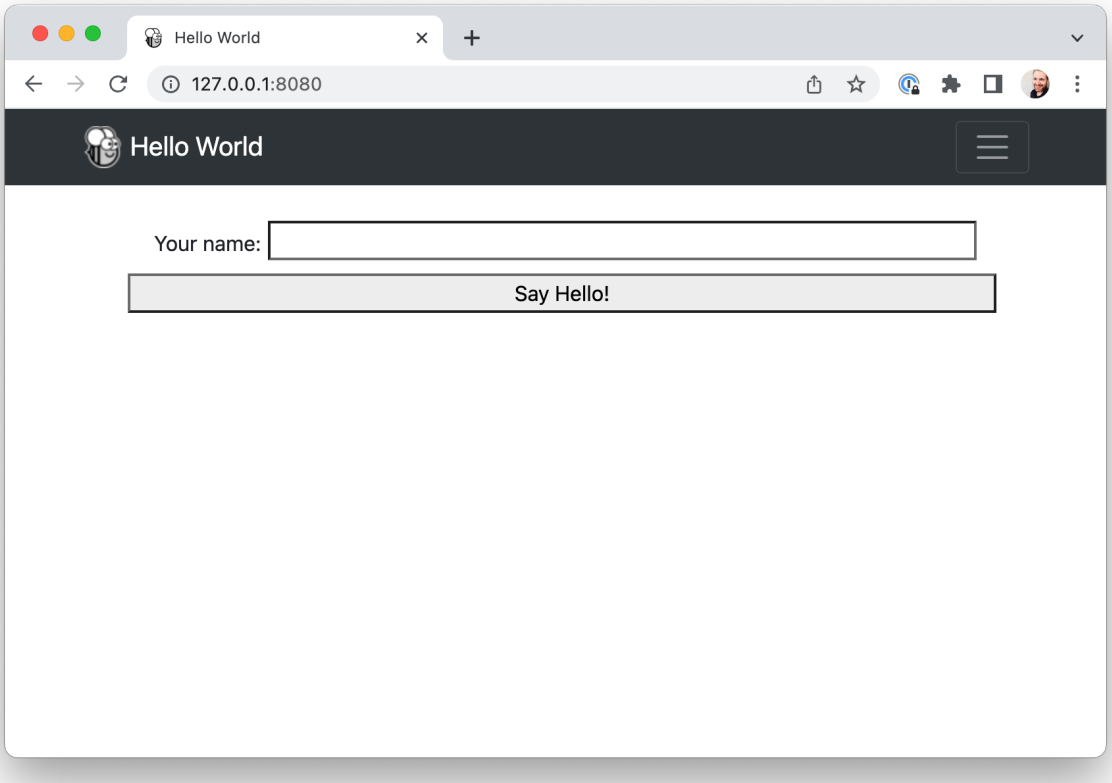
```
(beeware-venv) C:\...>briefcase run web

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

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

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

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

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

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

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

(continues on next page)

(continua dalla pagina precedente)

```
[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)...
================================================================================
```

Si aprirà un browser web che punta a http://127.0.0.1:8080:



Se si inserisce il proprio nome e si fa clic sul pulsante, viene visualizzata una finestra di dialogo.

2.7.2 Come funziona?

Questa applicazione web è un sito web statico - una singola pagina sorgente HTML, con alcuni CSS e altre risorse. Briefcase ha avviato un server web locale per servire questa pagina, in modo che il browser possa visualizzarla. Se si volesse mettere in produzione questa pagina web, si potrebbe copiare il contenuto della cartella `www` su qualsiasi server web in grado di servire contenuti statici.

Ma quando si preme il pulsante, si esegue codice Python… Come funziona? Toga utilizza PyScript per fornire un interprete Python nel browser. Briefcase impacchetta il codice dell'applicazione come ruote che PyScript può caricare nel browser. Quando la pagina viene caricata, il codice dell'applicazione viene eseguito nel browser, costruendo l'interfaccia utente utilizzando il DOM del browser. Quando si fa clic su un pulsante, questo esegue il codice di gestione degli eventi nel browser.

2.7.3 Prossimi passi

Anche se abbiamo distribuito questa applicazione su desktop, mobile e web, l'applicazione è piuttosto semplice e non coinvolge librerie di terze parti. Possiamo includere nella nostra applicazione le librerie del Python Package Index (PyPI)? Consultate *Tutorial 7* per scoprirlo…

2.8 Esercitazione 7 - Iniziare questa (terza) festa

Finora, l'applicazione che abbiamo costruito ha utilizzato solo il nostro codice e quello fornito da BeeWare. Tuttavia, in un'applicazione reale, è probabile che si voglia utilizzare una libreria di terze parti, scaricata dal Python Package Index (PyPI).

Modifichiamo la nostra applicazione per includere una libreria di terze parti.

2.8.1 Accesso a un'API

Un compito comune che un'applicazione deve svolgere è quello di fare una richiesta a un'API web per recuperare dati e mostrarli all'utente. Questa è un'applicazione giocattolo, quindi non abbiamo un'API *reale* con cui lavorare, quindi useremo l'API segnaposto {JSON} come fonte di dati.

L'API {JSON} Placeholder ha una serie di endpoint API «falsi» che si possono usare come dati di prova. Una di queste API è l'endpoint `/posts/`, che restituisce finti post del blog. Se si apre `https://jsonplaceholder.typicode.com/posts/42` nel browser, si otterrà un payload JSON che descrive un singolo post - un contenuto Lorum ipsum per un post di un blog con ID 42.

La libreria standard di Python contiene tutti gli strumenti necessari per accedere a un'API. Tuttavia, le API integrate sono di livello molto basso. Sono buone implementazioni del protocollo HTTP, ma richiedono all'utente di gestire molti dettagli di basso livello, come il reindirizzamento degli URL, le sessioni, l'autenticazione e la codifica del payload. Come «normale utente di browser», probabilmente siete abituati a dare per scontati questi dettagli, che il browser gestisce per voi.

Di conseguenza, sono state sviluppate librerie di terze parti che avvolgono le API integrate e forniscono un'API più semplice e più vicina all'esperienza quotidiana del browser. Utilizzeremo una di queste librerie per accedere all'API {JSON} Placeholder: una libreria chiamata httpx.

Aggiungiamo una chiamata API `httpx` alla nostra applicazione. Aggiungiamo un'importazione all'inizio di `app.py` per importare `httpx`:

```
import httpx
```

Modificare quindi il callback `say_hello()` in modo che assomigli a questo:

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

    payload = response.json()

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

Questo modificherà il callback `say_hello()` in modo che quando viene invocato, lo farà:

- effettuare una richiesta GET all'API JSON dei segnaposto per ottenere il post 42;
- decodificare la risposta come JSON;
- estrarre il corpo del messaggio; e
- includere il corpo di quel post come testo della finestra di dialogo.

Eseguiamo la nostra applicazione aggiornata in modalità sviluppatore di Briefcase per verificare che la nostra modifica abbia funzionato.

macOS

Linux

Windows

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

```
(beeware-venv) $ briefcase dev
Traceback (most recent call last):
File ".../venv/bin/briefcase", line 5, in <module>
    from briefcase.__main__ import main
File ".../venv/lib/python3.9/site-packages/briefcase/__main__.py", line 3, in <module>
    from .cmdline import parse_cmdline
File ".../venv/lib/python3.9/site-packages/briefcase/cmdline.py", line 6, in <module>
```

(continues on next page)

(continua dalla pagina precedente)

```
    from briefcase.commands import DevCommand, NewCommand, UpgradeCommand
File ".../venv/lib/python3.9/site-packages/briefcase/commands/__init__.py", line 1, in
↪<module>
    from .build import BuildCommand  # noqa
File ".../venv/lib/python3.9/site-packages/briefcase/commands/build.py", line 5, in
↪<module>
    from .base import BaseCommand, full_options
File ".../venv/lib/python3.9/site-packages/briefcase/commands/base.py", line 14, in
↪<module>
    import httpx
ModuleNotFoundError: No module named 'httpx'
```

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

Cosa è successo? Abbiamo aggiunto `httpx` al nostro *codice*, ma non l'abbiamo aggiunto al nostro ambiente virtuale di sviluppo. Possiamo risolvere il problema installando `httpx` con `pip` e poi eseguendo nuovamente `briefcase dev`:

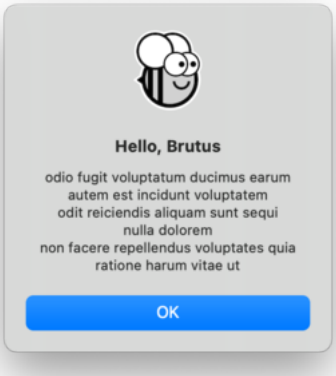
macOS

Linux

Windows

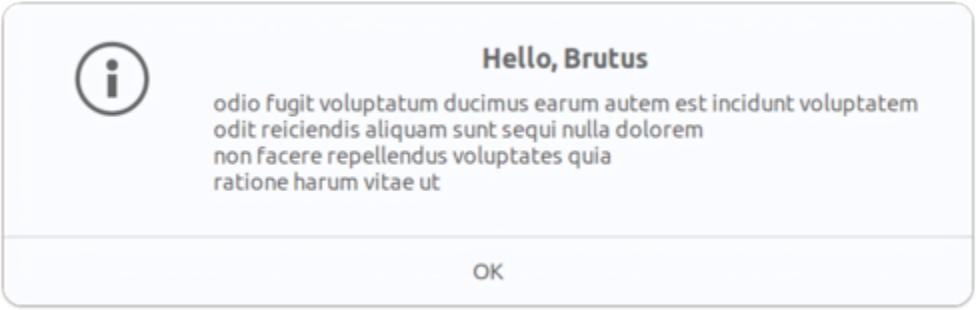
```
(beeware-venv) $ python -m pip install httpx
(beeware-venv) $ briefcase dev
```

Quando si inserisce un nome e si preme il pulsante, dovrebbe apparire una finestra di dialogo simile a questa:



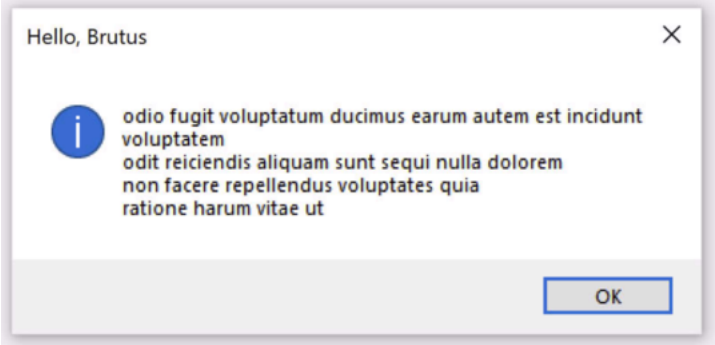

```
(beeware-venv) $ python -m pip install httpx
(beeware-venv) $ briefcase dev
```

Quando si inserisce un nome e si preme il pulsante, dovrebbe apparire una finestra di dialogo simile a questa:




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

Quando si inserisce un nome e si preme il pulsante, dovrebbe apparire una finestra di dialogo simile a questa:



Ora abbiamo un'applicazione funzionante, che utilizza una libreria di terze parti, in modalità di sviluppo!

2.8.2 Esecuzione dell'applicazione aggiornata

Facciamo in modo che il codice aggiornato dell'applicazione venga confezionato come applicazione autonoma. Poiché abbiamo apportato modifiche al codice, dobbiamo seguire gli stessi passi di *Tutorial 4*:

macOS

Linux

Windows

Aggiornare il codice dell'applicazione confezionata:

```
(beeware-venv) $ briefcase update

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

[helloworld] Application updated.
```

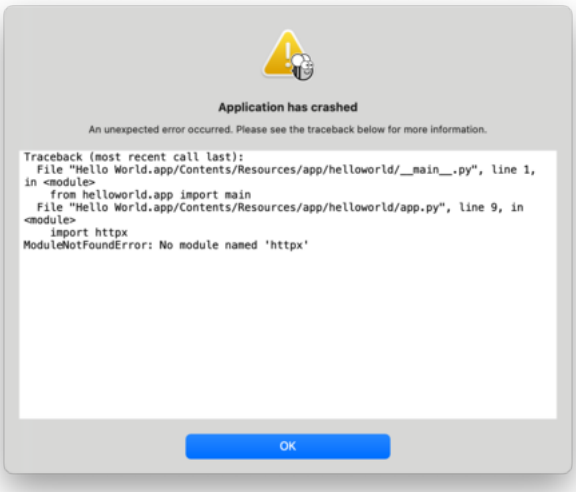
Ricostruire l'applicazione:

```
(beeware-venv) $ briefcase build

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

Infine, eseguire l'applicazione:

```
(beeware-venv) $ briefcase run

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

Tuttavia, quando l'applicazione viene eseguita, viene visualizzato un errore nella console e una finestra di dialogo di arresto anomalo:



Aggiornare il codice dell'applicazione confezionata:

```
(beeware-venv) $ briefcase update

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

[helloworld] Application updated.
```

Ricostruire l'applicazione:

```
(beeware-venv) $ briefcase build

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

[helloworld] Building application...
...

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

Infine, eseguire l'applicazione:

```
(beeware-venv) $ briefcase run

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

Tuttavia, quando l'applicazione viene eseguita, viene visualizzato un errore nella console:

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

Unable to start app helloworld.
```

Aggiornare il codice dell'applicazione confezionata:

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

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

[helloworld] Application updated.
```

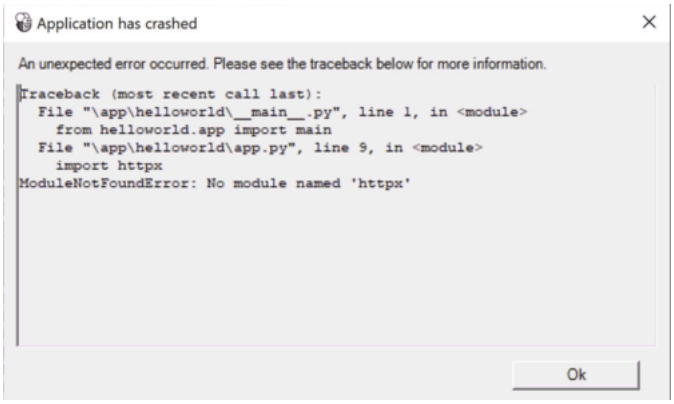
Ricostruire l'applicazione:

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

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

Infine, eseguire l'applicazione:

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

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

Tuttavia, quando l'applicazione viene eseguita, viene visualizzato un errore nella console e una finestra di dialogo di arresto anomalo:



Ancora una volta, l'applicazione non è riuscita ad avviarsi perché `httpx` è stato installato - ma perché? Non abbiamo già installato `httpx`?

È così, ma solo nell'ambiente di sviluppo. L'ambiente di sviluppo è interamente locale alla vostra macchina e viene attivato solo quando lo attivate esplicitamente. Anche se Briefcase ha una modalità di sviluppo, il motivo principale per cui si usa Briefcase è per impacchettare il codice in modo da poterlo dare a qualcun altro.

L'unico modo per garantire che qualcun altro abbia un ambiente Python che contiene tutto ciò di cui ha bisogno è costruire un ambiente Python completamente isolato. Questo significa che c'è un'installazione di Python completamente isolata e un insieme di dipendenze completamente isolato. Questo è ciò che Briefcase costruisce quando si lancia `briefcase build` - un ambiente Python isolato. Questo spiega anche perché `httpx` non è installato: è stato installato nell'ambiente di *sviluppo*, ma non nell'applicazione confezionata.

Quindi, dobbiamo dire a Briefcase che la nostra applicazione ha una dipendenza esterna.

2.8.3 Aggiornamento delle dipendenze

Nella directory principale dell'applicazione, c'è un file chiamato `pyproject.toml`. Questo file contiene tutti i dettagli di configurazione dell'applicazione forniti al momento dell'esecuzione di `briefcase new`.

il file `pyproject.toml` è suddiviso in sezioni; una delle sezioni descrive le impostazioni per l'applicazione:

```
[tool.briefcase.app.helloworld]
formal_name = "Hello World"
description = "A Tutorial app"
long_description = """More details about the app should go here.
"""
```

(continues on next page)

(continua dalla pagina precedente)

```
sources = ["src/helloworld"]
requires = []
```

L'opzione `requires` descrive le dipendenze della nostra applicazione. Si tratta di un elenco di stringhe, che specificano le librerie (e, opzionalmente, le versioni) delle librerie che si vogliono includere nella propria applicazione.

Modificare l'impostazione `requires` in modo che si legga:

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

Aggiungendo questa impostazione, si dice a Briefcase «quando costruisci la mia applicazione, esegui `pip install httpx` nel bundle dell'applicazione». Qualsiasi cosa che sia un input legale per `pip install` può essere usato qui, quindi si può specificare:

- Una versione specifica della libreria (ad esempio, "`httpx==0.19.0`");
- Un intervallo di versioni della libreria (ad esempio, "`httpx>=0.19`");
- Un percorso a un repository git (ad esempio, "`git+https://github.com/encode/httpx`"); oppure
- Un percorso di file locale (tuttavia, attenzione: se si dà il codice a qualcun altro, questo percorso probabilmente non esisterà sulla sua macchina)

Più avanti in `pyproject.toml`, si noteranno altre sezioni che dipendono dal sistema operativo, come `[tool.briefcase.app.helloworld.macOS]` e `[tool.briefcase.app.helloworld.windows]`. Queste sezioni hanno anche un'impostazione `requires`. Queste impostazioni consentono di definire dipendenze aggiuntive specifiche per la piattaforma; quindi, ad esempio, se si ha bisogno di una libreria specifica per la piattaforma per gestire alcuni aspetti dell'applicazione, si può specificare tale libreria nella sezione `requires` specifica per la piattaforma, e tale impostazione sarà utilizzata solo per quella piattaforma. Si noterà che le librerie `toga` sono tutte specificate nella sezione `requires` specifica della piattaforma, perché le librerie necessarie per visualizzare l'interfaccia utente sono specifiche della piattaforma.

Nel nostro caso, vogliamo che `httpx` sia installato su tutte le piattaforme, quindi usiamo l'impostazione `requires` a livello di applicazione. Le dipendenze a livello di applicazione saranno sempre installate; le dipendenze specifiche della piattaforma sono installate *in aggiunta* a quelle a livello di applicazione.

Alcuni pacchetti binari potrebbero non essere disponibili

Sulle piattaforme desktop (macOS, Windows, Linux), qualsiasi `pip` installabile può essere aggiunto ai requisiti. Sulle piattaforme mobili e web, le opzioni sono leggermente limitate <https://briefcase.readthedocs.io/en/latest/background/faq.html#can-i-use-third-party-python-packages-in-my-app>`__.

In breve, qualsiasi pacchetto *puro* Python (cioè i pacchetti che *non* contengono un modulo binario) può essere usato senza problemi. Tuttavia, se la vostra dipendenza contiene un componente binario, deve essere compilata; al momento, la maggior parte dei pacchetti Python non fornisce supporto alla compilazione per piattaforme non desktop.

BeeWare può fornire binari per alcuni moduli binari popolari (tra cui `numpy`, `pandas` e `cryptography`). È *di solito* possibile compilare pacchetti per le piattaforme mobili, ma non è facile da configurare, il che esula dallo scopo di un tutorial introduttivo come questo.

Ora che abbiamo comunicato a Briefcase i nostri requisiti aggiuntivi, possiamo riprovare a pacchettizzare la nostra applicazione. Assicurarsi di aver salvato le modifiche a `pyproject.toml` e poi aggiornare di nuovo l'applicazione, questa volta inserendo il flag `-r`. Questo indica a Briefcase di aggiornare i requisiti nell'applicazione pacchettizzata:

macOS

Linux

Windows

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

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

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

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

[helloworld] Application updated.
```

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

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

Targeting glibc 2.35
Targeting Python3.10

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

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

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

[helloworld] Application updated.
```

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

(continues on next page)

(continua dalla pagina precedente)

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

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

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

[helloworld] Application updated.
```

Una volta effettuato l'aggiornamento, si può eseguire `briefcase build` e `briefcase run` e si dovrebbe vedere l'applicazione confezionata, con il nuovo comportamento della finestra di dialogo.

Nota: L'opzione `-r` per l'aggiornamento dei requisiti viene rispettata anche dai comandi `build` e `run`, quindi se si vuole aggiornare, compilare ed eseguire in un unico passaggio, si può usare `briefcase run -u -r`.

2.8.4 Prossimi passi

Ora abbiamo un'applicazione che utilizza una libreria di terze parti! Tuttavia, avrete notato che quando si preme il pulsante, l'app diventa poco reattiva. Possiamo fare qualcosa per risolvere questo problema? Consultate *Tutorial 8* per scoprirlo…

2.9 Esercitazione 8 - Renderlo liscio

A meno che non si disponga di una connessione Internet *molto* veloce, si potrebbe notare che quando si preme il pulsante, l'interfaccia grafica dell'applicazione si blocca per un po". Questo perché la richiesta web che abbiamo fatto è *sincrona*. Quando l'applicazione effettua la richiesta web, attende che l'API restituisca una risposta prima di continuare. Mentre aspetta, non permette all'applicazione di ridisegnare e di conseguenza l'applicazione si blocca.

2.9.1 Loop di eventi della GUI

Per capire perché questo accade, dobbiamo entrare nei dettagli del funzionamento di un'applicazione GUI. Le specifiche variano a seconda della piattaforma, ma i concetti di alto livello sono gli stessi, indipendentemente dalla piattaforma o dall'ambiente GUI utilizzato.

Un'applicazione GUI è, fondamentalmente, un singolo ciclo che assomiglia a:

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

Questo ciclo è chiamato *Event Loop*. (Non si tratta di nomi di metodi reali, ma di un'illustrazione di ciò che avviene in «pseudo-codice»).

Quando si fa clic su un pulsante, si trascina una barra di scorrimento o si digita un tasto, si genera un «evento». Questo «evento» viene inserito in una coda e l'applicazione elaborerà la coda di eventi quando ne avrà l'opportunità. Il codice utente che viene attivato in risposta all'evento è chiamato *gestore di eventi*. Questi gestori di eventi vengono invocati come parte della chiamata `process_events()`.

Una volta che un'applicazione ha elaborato tutti gli eventi disponibili, essa `ridisegna()` la GUI. Questa operazione tiene conto di tutti i cambiamenti che gli eventi hanno causato alla visualizzazione dell'applicazione, nonché di qualsiasi altra cosa stia accadendo nel sistema operativo: ad esempio, le finestre di un'altra applicazione possono oscurare o rivelare parte della finestra della nostra applicazione, e il ridisegno della nostra applicazione dovrà riflettere la porzione di finestra attualmente visibile.

Il dettaglio importante da notare: mentre un'applicazione sta elaborando un evento, *non può ridisegnare* e *non può elaborare altri eventi*.

Ciò significa che qualsiasi logica utente contenuta in un gestore di eventi deve essere completata rapidamente. Qualsiasi ritardo nel completamento del gestore di eventi sarà osservato dall'utente come un rallentamento (o un arresto) degli aggiornamenti della GUI. Se il ritardo è sufficientemente lungo, il sistema operativo può segnalarlo come un problema: le icone «beachball» di macOS e «spinner» di Windows indicano che l'applicazione sta impiegando troppo tempo in un gestore di eventi.

Operazioni semplici come «aggiornare un'etichetta» o «ricalcolare il totale degli input» sono facili da completare rapidamente. Tuttavia, ci sono molte operazioni che non possono essere completate rapidamente. Se si sta eseguendo un calcolo matematico complesso, o l'indicizzazione di tutti i file di un file system, o l'esecuzione di una richiesta di rete di grandi dimensioni, non è possibile «farlo rapidamente»: le operazioni sono intrinsecamente lente.

Quindi, come si eseguono operazioni di lunga durata in un'applicazione GUI?

2.9.2 Programmazione asincrona

Abbiamo bisogno di un modo per dire a un'applicazione, nel mezzo di un gestore di eventi di lunga durata, che va bene rilasciare temporaneamente il controllo al ciclo di eventi, a patto che si possa riprendere da dove si era interrotto. Spetta all'applicazione determinare quando questo rilascio può avvenire; ma se l'applicazione rilascia regolarmente il controllo al ciclo di eventi, possiamo avere un gestore di eventi di lunga durata *e* mantenere un'interfaccia utente reattiva.

È possibile farlo utilizzando la *programmazione asincrona*. La programmazione asincrona è un modo per descrivere un programma che consente all'interprete di eseguire più funzioni contemporaneamente, condividendo le risorse tra tutte le funzioni in esecuzione simultanea.

Le funzioni asincrone (note come *co-routine*) devono essere dichiarate esplicitamente come asincrone. Inoltre, devono dichiarare internamente quando esiste la possibilità di cambiare contesto a un'altra co-routine.

In Python, la programmazione asincrona è implementata utilizzando le parole chiave `async` e `await` e il modulo asyncio della libreria standard. La parola chiave `async` ci permette di dichiarare che una funzione è una co-routine asincrona. La parola chiave `await` fornisce un modo per dichiarare quando esiste l'opportunità di cambiare contesto a un'altra co-routine. Il modulo asyncio fornisce altri strumenti e primitive utili per la codifica asincrona.

2.9.3 Rendere l'esercitazione asincrona

Per rendere il nostro tutorial asincrono, modificare il gestore dell'evento `say_hello()` in modo che assomigli a questo:

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

    payload = response.json()

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

In questo codice ci sono solo 4 modifiche rispetto alla versione precedente:

1. Il metodo è definito come `async def`, invece che semplicemente `def`. Questo indica a Python che il metodo è una co-routine asincrona.
2. Il client creato è un asincrono `AsyncClient()`, invece di un sincrono `Client()`. Questo indica a `httpx` che deve operare in modalità asincrona, anziché sincrona.
3. Il gestore di contesto usato per creare il client è contrassegnato come `async`. Questo indica a Python che c'è l'opportunità di rilasciare il controllo quando il gestore di contesto viene inserito e abbandonato.
4. La chiamata `get` viene effettuata con la parola chiave `await`. Questo indica all'applicazione che, mentre si attende la risposta dalla rete, l'applicazione può rilasciare il controllo al ciclo degli eventi.

Toga consente di utilizzare metodi regolari o co-routine asincrone come gestori; Toga gestisce tutto dietro le quinte per assicurarsi che il gestore sia invocato o atteso come richiesto.

Se si salvano queste modifiche e si esegue nuovamente l'applicazione (con `briefcase dev` in modalità di sviluppo, oppure aggiornando ed eseguendo nuovamente l'applicazione confezionata), non ci saranno cambiamenti evidenti nell'applicazione. Tuttavia, quando si fa clic sul pulsante per attivare la finestra di dialogo, si possono notare alcuni sottili miglioramenti:

- Il pulsante torna a uno stato «non cliccato», anziché essere bloccato in uno stato «cliccato».
- L'icona «beachball»/»spinner» non appare
- Se si sposta/ridimensiona la finestra dell'applicazione mentre si attende la visualizzazione della finestra di dialogo, la finestra verrà ridisegnata.
- Se si tenta di aprire il menu di un'applicazione, il menu viene visualizzato immediatamente.

2.9.4 Prossimi passi

Ora abbiamo un'applicazione che è efficiente e reattiva, anche quando è in attesa di un'API lenta. Ma come possiamo assicurarci che l'applicazione continui a funzionare mentre continuiamo a svilupparla? Come possiamo testare la nostra applicazione? Consultate *Tutorial 9* per scoprirlo…

2.10 Esercitazione 9 - Tempi di verifica

La maggior parte dello sviluppo del software non comporta la scrittura di nuovo codice, ma la modifica di quello esistente. Assicurarsi che il codice esistente continui a funzionare nel modo in cui ci aspettiamo è una parte fondamentale del processo di sviluppo del software. Un modo per garantire il comportamento della nostra applicazione è una *serie di test*.

2.10.1 Esecuzione della suite di test

Si scopre che il nostro progetto ha già una suite di test! Quando abbiamo generato il nostro progetto, sono state generate due directory di primo livello: `src` e `tests`. La cartella `src` contiene il codice della nostra applicazione; la cartella `tests` contiene la nostra suite di test. All'interno della cartella `tests` c'è un file chiamato `test_app.py` con il seguente contenuto:

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

Questo è un Pytest *caso di test* - un blocco di codice che può essere eseguito per verificare un comportamento dell'applicazione. In questo caso, il test è un segnaposto e non verifica nulla della nostra applicazione, ma è un test che possiamo eseguire.

Possiamo eseguire questa suite di test usando l'opzione `--test` di `briefcase dev`. Poiché è la prima volta che si eseguono dei test, è necessario inserire anche l'opzione `-r` per assicurarsi che anche i requisiti dei test siano installati:

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

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

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

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

[helloworld] Installing requirements...
```

(continues on next page)

(continua dalla pagina precedente)

```
...
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 ===============================
```

Successo! Abbiamo appena eseguito un singolo test che verifica che la matematica di Python funziona nel modo previsto (che sollievo!).

Sostituiamo questo test segnaposto con un test per verificare che il nostro metodo `greeting()` si comporti come ci aspettiamo. Sostituire il contenuto di `test_app.py` con il seguente:

```
from helloworld.app import greeting


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

(continues on next page)

(continua dalla pagina precedente)

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


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

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

Questo definisce due nuovi test, che verificano i due comportamenti che ci aspettiamo di vedere: l'output quando viene fornito un nome e l'output quando il nome è vuoto.

Ora possiamo eseguire nuovamente la suite di test. Questa volta non è necessario fornire l'opzione `-r`, poiché i requisiti per i test sono già stati installati; è sufficiente usare l'opzione `--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
```

(continues on next page)

(continua dalla pagina precedente)

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

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

Eccellente! Il nostro metodo di utilità `greeting()` funziona come previsto.

2.10.2 Sviluppo guidato dai test

Ora che abbiamo una suite di test, possiamo usarla per guidare lo sviluppo di nuove funzionalità. Modifichiamo la nostra applicazione per avere un saluto speciale per un utente in particolare. Possiamo iniziare aggiungendo un caso di test per il nuovo comportamento che vorremmo vedere in fondo a `test_app.py`:

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

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

Quindi, eseguire la suite di test con questo nuovo 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!'
E         - BeeWare the IDEs of Python!
E         + Hello, Brutus

tests/test_app.py:19: AssertionError
=========================== short test summary info ============================
```

(continues on next page)

(continua dalla pagina precedente)

```
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 ===================================
_____________________________________ test_brutus ______________________________________

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

(continues on next page)

(continua dalla pagina precedente)

```
>       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 ========================
```

Questa volta, vediamo un fallimento del test e l'output spiega la fonte del fallimento: il test si aspetta l'output «BeeWare the IDEs of Python!», ma la nostra implementazione di `greeting()` restituisce «Hello, Brutus». Modifichiamo l'implementazione di `greeting()` in `src/helloworld/app.py` per avere il nuovo comportamento:

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

Se eseguiamo nuovamente i test, vedremo che i nostri test passano:

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

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

(continues on next page)

(continua dalla pagina precedente)

```
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 ===============================
```

2.10.3 Test di runtime

Finora abbiamo eseguito i test in modalità di sviluppo. Questo è particolarmente utile quando si sviluppano nuove funzionalità, in quanto si può iterare rapidamente sull'aggiunta di test e sull'aggiunta di codice per far sì che tali test passino. Tuttavia, a un certo punto, si vorrà verificare che il codice venga eseguito correttamente anche nell'ambiente dell'applicazione bundle.

Le opzioni `--test` e `-r` possono essere passate anche al comando `run`. Se si usa `briefcase run --test -r`, verrà eseguita la stessa suite di test, ma all'interno del bundle dell'applicazione confezionata e non nell'ambiente di sviluppo:

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...
Pre-initializing Python runtime...
PythonHome: /Users/brutus/beeware-tutorial/helloworld/macOS/app/Hello World/Hello World.
↪app/Contents/Resources/support/python-stdlib
PYTHONPATH:
```

(continues on next page)

(continua dalla pagina precedente)

```
- /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 ==============================
...
collecting ... collected 3 items

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

(continues on next page)

(continua dalla pagina precedente)

```
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 ===============================
```

Come per `briefcase dev --test`, l'opzione `-r` è necessaria solo la prima volta che si esegue la suite di test, per assicurarsi che le dipendenze del test siano presenti. Nelle esecuzioni successive, si può omettere questa opzione.

Si può anche usare l'opzione `--test` sui backend mobili: - quindi `briefcase run iOS --test` e `briefcase run android --test` funzioneranno entrambi, eseguendo la suite di test sul dispositivo mobile selezionato.

2.10.4 Prossimi passi

We've now got a a test suite for our application. But it still looks like a tutorial app. Is there anything we can do about that? Turn to *Tutorial 10* to find out...

2.11 Esercitazione 10 - Create la vostra applicazione

Finora la nostra applicazione ha utilizzato l'icona predefinita «ape grigia». Come possiamo aggiornare l'app per utilizzare la nostra icona?

2.11.1 Aggiunta di un'icona

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...
```

(continues on next page)

(continua dalla pagina precedente)

```
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␣
```

(continues on next page)

(continua dalla pagina precedente)

```
↪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;␣
↪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
```

(continues on next page)

(continua dalla pagina precedente)

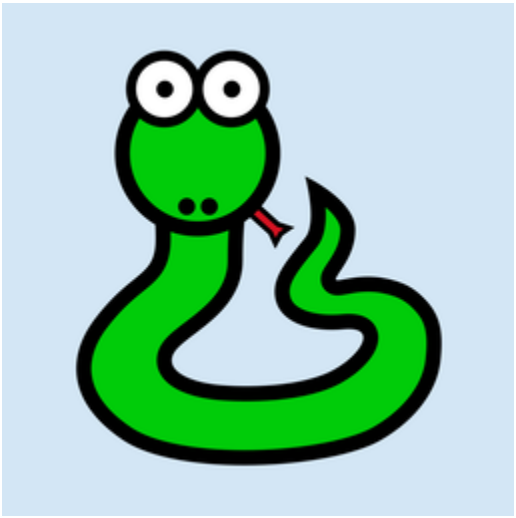
```
  [helloworld] Removing unneeded app content...
  Removing unneeded app bundle content... done

  [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 unpack it into the root of your project directory. After unpacking, your project directory should look something like:

```
beeware-tutorial/
    beeware-venv/
        ...
    helloworld/
        ...
        pyproject.toml
        icons/
            helloworld.icns
            helloworld.ico
            helloworld.png
            helloworld-16.png
            ...
        src/
        ...
```

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...
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.

[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.
```

(continues on next page)

(continua dalla pagina precedente)

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

Nota: 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...
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...
```

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...
```

(continues on next page)

(continua dalla pagina precedente)

```
[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.

2.11.2 Prossimi passi

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

Some places to go from here:

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