BeeWare Documentation

Versión 0.1.dev155+g9b30742

Russell Keith-Magee

12 de mayo de 2024

Índice general

Escribe Python. Ejecútalo en cualquier sitio.

¡Bienvenido a BeeWare! En este tutorial, vamos a construir una interfaz gráfica de usuario utilizando Python, y desplegarla como una aplicación de escritorio, como una aplicación móvil, y como una aplicación web de una sola página. También vamos a ver cómo se pueden utilizar las herramientas de BeeWare para hacer algunas de las tareas comunes que tendrás que hacer como desarrollador de aplicaciones, tales como la prueba de su aplicación.

¡Es una traducción automática!

Esta versión del tutorial se ha generado mediante traducción automática. Sabemos que no es lo ideal, pero pensamos que una mala traducción era mejor que ninguna.

Si quieres ayudarnos a mejorar la traducción, ¡ponte en contacto con nosotros! Tenemos un canal `#translations` en Discord; preséntate allí y te añadiremos al equipo de traducción.

CAPÍTULO 1

¿Qué es BeeWare?

BeeWare no es un único producto, o herramienta, o librería - es una colección de herramientas y librerías, cada una de las cuales trabaja conjuntamente para ayudarte a escribir aplicaciones Python multiplataforma con una GUI nativa. Incluye:

- Toga, un conjunto de herramientas de widgets multiplataforma;
- Briefcase, una herramienta para empaquetar proyectos de Python como artefactos distribuibles que se pueden enviar a los usuarios finales;
- Bibliotecas (como Rubicon ObjC) para acceder a bibliotecas nativas de la plataforma;
- Recopilaciones precompiladas de Python que pueden utilizarse en plataformas en las que no están disponibles los instaladores oficiales de Python.

En este tutorial, utilizaremos todas estas herramientas, pero como usuario, sólo necesitarás interactuar con las dos primeras (Toga y Briefcase). Sin embargo, cada una de las herramientas también se puede utilizar individualmente - por ejemplo, puede utilizar Briefcase para desplegar una aplicación sin necesidad de utilizar Toga como un conjunto de herramientas GUI.

La suite BeeWare está disponible en macOS, Windows, Linux (usando GTK); en plataformas móviles como Android e iOS; y para la Web.

CAPÍTULO 2

¡Vamos!

¿Listo para probar BeeWare por ti mismo? *¡Construyamos una aplicación multiplataforma en Python!*

2.1 Tutorial 0 - ¡Preparémonos!

Antes de crear nuestra primera aplicación BeeWare, tenemos que asegurarnos de que tenemos todos los requisitos previos para ejecutar BeeWare.

2.1.1 Instalar Python

Lo primero que necesitaremos es un intérprete de Python que funcione.

macOS

Linux

Windows

Si utilizas macOS, con Xcode o las herramientas de desarrollo de línea de comandos se incluye una versión reciente de Python. Para comprobar si ya la tienes, ejecuta el siguiente comando:

```
$ python3 --version
```

Si Python está instalado, verás su número de versión. Si no, se te pedirá que instales las herramientas de desarrollo de la línea de comandos.

Si estás en Windows, puedes obtener el instalador oficial desde the Python website. Puedes usar cualquier versión estable de Python desde la 3.8 en adelante. Te aconsejamos que evites las versiones alphas, betas y release candidates a menos que *realmente* sepas lo que estás haciendo.

Si estás en Linux, instalarás Python utilizando el gestor de paquetes del sistema (`apt` en Debian/Ubuntu/Mint, `dnf` en Fedora, o `pacman` en Arch).

Debe asegurarse de que el sistema Python es Python 3.8 o más reciente; si no lo es (por ejemplo, Ubuntu 18.04 viene con Python 3.6), tendrá que actualizar su distribución de Linux a algo más reciente.

La compatibilidad con Raspberry Pi es limitada por el momento.

Si estás en Windows, puedes obtener el instalador oficial desde the Python website. Puedes usar cualquier versión estable de Python desde la 3.8 en adelante. Te aconsejamos que evites las versiones alphas, betas y release candidates a menos que *realmente* sepas lo que estás haciendo.

Distribuciones alternativas de Python

Hay muchas formas diferentes de instalar Python. Puedes instalar Python a través de homebrew. Puedes usar pyenv para gestionar múltiples instalaciones de Python en la misma máquina. Los usuarios de Windows pueden instalar Python desde la Windows App Store. Los usuarios con experiencia en ciencia de datos pueden utilizar Anaconda o Miniconda.

Si usas macOS o Windows, no importa *cómo* has instalado Python - sólo importa que puedas ejecutar `python3` desde el símbolo del sistema/terminal de tu sistema operativo, y obtener un intérprete de Python que funcione.

Si usas Linux, deberías usar el Python de sistema proporcionado por tu sistema operativo. Podrás completar *la mayor parte* de este tutorial usando un Python que no sea de sistema, pero no podrás empaquetar tu aplicación para distribuirla a otros.

2.1.2 Instalar dependencias

A continuación, instale las dependencias adicionales necesarias para su sistema operativo:

macOS

Linux

Windows

Construir aplicaciones BeeWare en macOS requiere:

- **Git**, un sistema de control de versiones. Esto se incluye con Xcode o las herramientas de línea de comandos para desarrolladores, que instaló anteriormente.

Para soportar el desarrollo local, necesitarás instalar algunos paquetes del sistema. La lista de paquetes necesarios varía en función de tu distribución:

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
```

Construir aplicaciones BeeWare en Windows requiere:

- **Git**, un sistema de control de versiones. Puede descargar Git desde git-scm.org.

Después de instalar estas herramientas, debe asegurarse de reiniciar cualquier sesión de terminal. Windows sólo expondrá las herramientas recién instaladas a los terminales iniciados *después* de que se haya completado la instalación.

2.1.3 Crear un entorno virtual

Ahora vamos a crear un entorno virtual - una «caja de arena» que podemos utilizar para aislar nuestro trabajo en este tutorial de nuestra instalación principal de Python. Si instalamos paquetes en el entorno virtual, nuestra instalación principal de Python (y cualquier otro proyecto Python en nuestro ordenador) no se verá afectado. Si hacemos un completo desastre de nuestro entorno virtual, podremos simplemente borrarlo y empezar de nuevo, sin afectar a ningún otro proyecto Python en nuestro ordenador, y sin necesidad de reinstalar 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
```

Errores al ejecutar scripts PowerShell

Si está utilizando PowerShell y recibe el error:

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

Tu cuenta de Windows no tiene permisos para ejecutar scripts. Para solucionarlo:

1. Ejecute Windows PowerShell como Administrador.
2. Ejecuta `set-executionpolicy RemoteSigned`
3. Seleccione `Y` para cambiar la política de ejecución.

Una vez hecho esto, puede volver a ejecutar `beeware-venv\Scripts\activate.ps1` en su sesión PowerShell original (o en una nueva sesión en el mismo directorio).

Si esto ha funcionado, tu prompt debería haber cambiado - debería tener un prefijo `(beeware-venv)`. Esto te permite saber que estás actualmente en tu entorno virtual BeeWare. Siempre que estés trabajando en este tutorial, debes asegurarte de que tu entorno virtual está activado. Si no lo está, vuelve a ejecutar el último comando (el comando `activate`) para reactivar tu entorno.

Entornos virtuales alternativos

Si estás usando Anaconda o miniconda, puede que estés más familiarizado con el uso de entornos conda. También puedes haber oído hablar de `virtualenv`, un predecesor del módulo `venv` de Python. Al igual que con las instalaciones de Python - si estás en macOS o Windows, no importa *cómo* creas tu entorno virtual, siempre y cuando tengas uno. Si estás en Linux, deberías usar `venv` y el sistema Python.

2.1.4 Siguientes pasos

Ya hemos configurado nuestro entorno. Estamos listos para *crear nuestra primera aplicación BeeWare*.

2.2 Tutorial 1 - Tu primera aplicación

Estamos listos para crear nuestra primera aplicación.

2.2.1 Instalar las herramientas BeeWare

En primer lugar, tenemos que instalar **Briefcase**. Briefcase es una herramienta de BeeWare que se puede utilizar para empaquetar su aplicación para la distribución a los usuarios finales - pero también se puede utilizar para arrancar un nuevo proyecto. Asegúrate de que estás en el directorio `beeware-tutorial` que creaste en *Tutorial 0*, con el entorno virtual `beeware-venv` activado, y ejecuta:

macOS

Linux

Windows

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

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

Posibles errores durante la instalación

Si aparecen errores durante la instalación, es casi seguro que se deba a que no se han instalado algunos de los requisitos del sistema. Asegúrese de haber *instalado todos los requisitos previos de la plataforma*.

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

Posibles errores durante la instalación

Es importante que utilices `python -m pip`, en lugar de `pip`. Briefcase necesita asegurarse de que tiene una versión actualizada de `pip` y `setuptools`, y una invocación de `pip` no puede auto-actualizarse. Si quieres saber más, Brett Cannon tiene una entrada de blog detallada sobre el problema.

Una de las herramientas de BeeWare es **Briefcase**. Briefcase se puede utilizar para empaquetar su aplicación para su distribución a los usuarios finales - pero también se puede utilizar para arrancar un nuevo proyecto.

2.2.2 Crear un nuevo proyecto

¡Empecemos nuestro primer proyecto BeeWare! Vamos a utilizar el comando `new` de Briefcase para crear una aplicación llamada **Hello World**. Ejecute lo siguiente desde su símbolo del sistema:

macOS

Linux

Windows

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

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

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

Briefcase nos pedirá algunos detalles de nuestra nueva aplicación. Para los propósitos de este tutorial, utilice lo siguiente:

- **Nombre Formal** - Acepta el valor por defecto: `Hola Mundo`.
- **Nombre de la aplicación** - Acepta el valor por defecto: `helloworld`.
- **Paquete** - Si tiene su propio dominio, introdúzcalo en orden inverso. (Por ejemplo, si posee el dominio «cupcakes.com», introduzca `com.cupcakes` como paquete). Si no posee su propio dominio, acepte el paquete predeterminado (`com.ejemplo`).
- **Nombre del proyecto** - Acepte el valor por defecto: `Hola Mundo`.
- **Descripción** - Acepte el valor por defecto (o, si quiere ser realmente creativo, invente su propia descripción)
- **Autor** - Escriba aquí su propio nombre.
- **Correo electrónico del autor** - Introduzca su propia dirección de correo electrónico. Se utilizará en el archivo de configuración, en el texto de ayuda y en cualquier lugar donde se requiera un correo electrónico al enviar la aplicación a una tienda de aplicaciones.
- **URL** - La URL de la página de destino de su aplicación. De nuevo, si es dueño de su propio dominio, introduzca una URL en ese dominio (incluyendo el `https://`). Si no, acepta la URL por defecto (`https://example.com/helloworld`). No es necesario que esta URL exista realmente (por ahora); sólo se utilizará si publicas tu aplicación en una tienda de aplicaciones.
- **Licencia** - Acepta la licencia por defecto (BSD). Sin embargo, esto no afectará en nada al funcionamiento del tutorial, así que si tienes sentimientos particularmente fuertes sobre la elección de la licencia, siéntete libre de elegir otra licencia.
- **GUI framework** - Acepte la opción por defecto, Toga (el propio conjunto de herramientas GUI de BeeWare).

Briefcase generará entonces un esqueleto de proyecto para que lo utilices. Si has seguido este tutorial hasta ahora, y has aceptado los valores por defecto tal y como se describen, tu sistema de archivos debería parecerse 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
```

Este esqueleto es en realidad una aplicación en pleno funcionamiento sin añadir nada más. La carpeta `src` contiene todo el código de la aplicación, la carpeta `tests` contiene un conjunto de pruebas iniciales, y el archivo `pyproject.toml` describe cómo empaquetar la aplicación para su distribución. Si abres `pyproject.toml` en un editor, verás los detalles de configuración que acabas de proporcionar a Briefcase.

Ahora que tenemos una aplicación stub, podemos utilizar Briefcase para ejecutar la aplicación.

2.2.3 Ejecutar la aplicación en modo desarrollador

Entra en el directorio del proyecto `helloworld` y dile a briefcase que inicie el proyecto en modo desarrollador (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
```

(continúe en la próxima página)

(proviene de la página anterior)

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

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

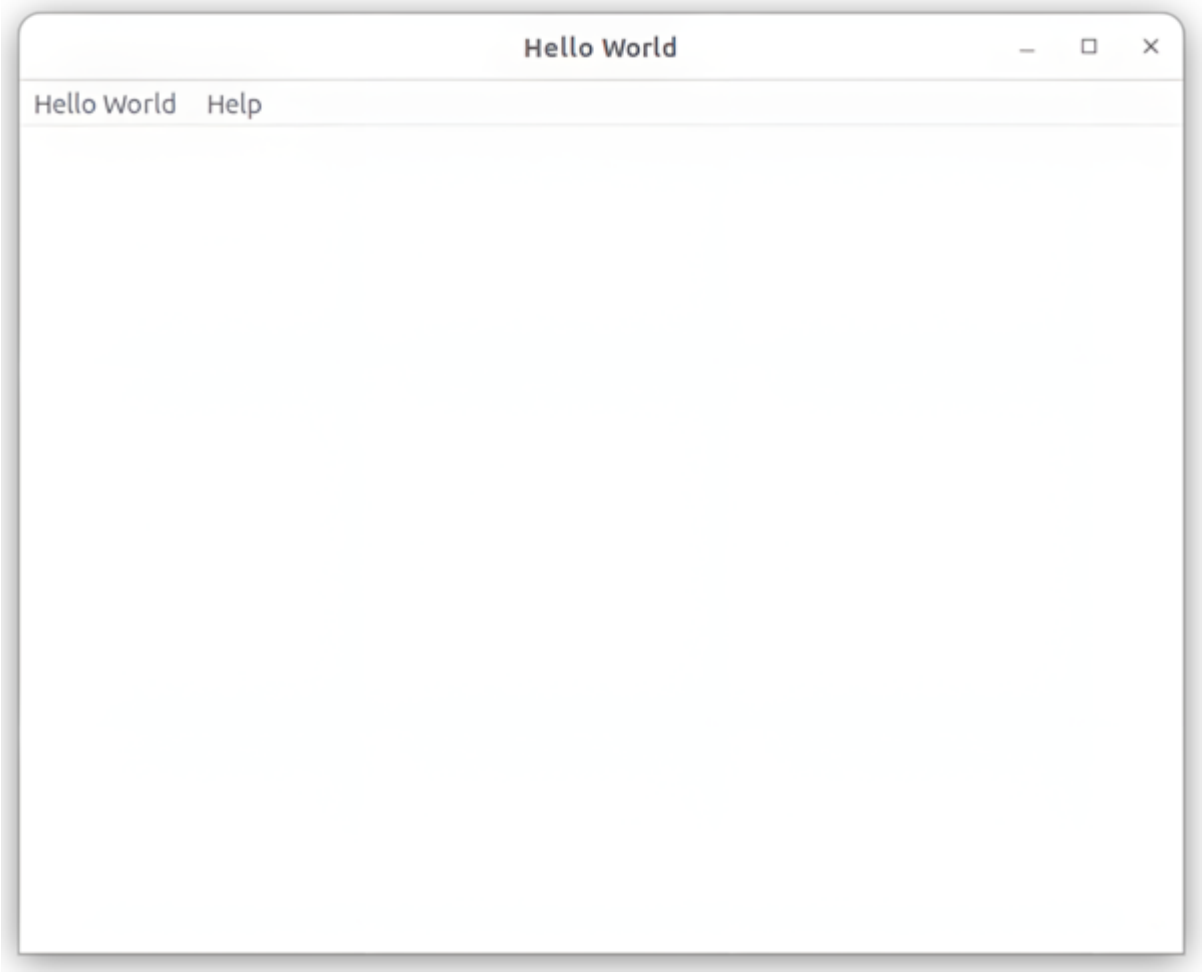
```
(beeware-venv) C:\...>cd helloworld
(beeware-venv) C:\...>briefcase dev

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

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

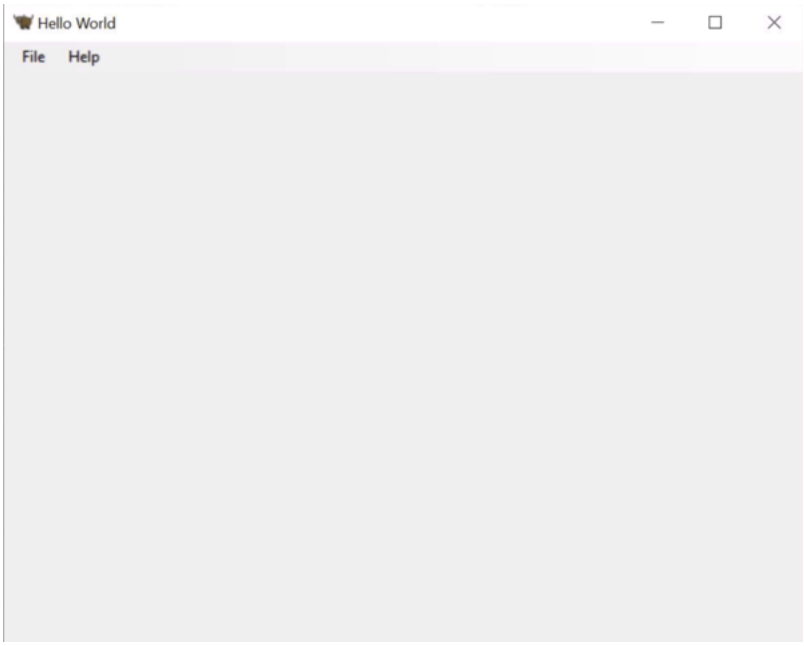
Esto debería abrir una ventana GUI:

macOS

Linux

Windows





Pulsa el botón de cerrar (o selecciona Salir en el menú de la aplicación), ¡y ya está! Enhorabuena - ¡acabas de escribir una aplicación nativa e independiente en Python!

2.2.4 Siguientes pasos

Ahora tenemos una aplicación que funciona, ejecutándose en modo desarrollador. Ahora podemos añadir algo de lógica propia para hacer que nuestra aplicación haga algo un poco más interesante. En *Tutorial 2*, pondremos una interfaz de usuario más útil en nuestra aplicación.

2.3 Tutorial 2 - Hacerlo interesante

En *Tutorial 1*, generamos un proyecto stub que era capaz de ejecutarse, pero no escribimos ningún código nosotros mismos. Echemos un vistazo a lo que se generó para nosotros.

2.3.1 Qué se generó

En el directorio `src/helloworld`, deberías ver 3 archivos: `__init__.py`, `__main__.py` y `app.py`.

`__init__.py` marca el directorio `helloworld` como un módulo importable de Python. Es un archivo vacío; el mero hecho de que exista le dice al intérprete de Python que el directorio `helloworld` define un módulo.

`__main__.py` marca el módulo `helloworld` como un tipo especial de módulo - un módulo ejecutable. Si intentas ejecutar el módulo `helloworld` usando `python -m helloworld`, el archivo `__main__.py` es donde Python empezará a ejecutarse. El contenido de `__main__.py` es relativamente simple:

```
from helloworld.app import main

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

Es decir, importa el método `main` de la aplicación `helloworld`; y si se está ejecutando como punto de entrada, llama al método main(), e inicia el bucle principal de la aplicación. El bucle principal es la forma en que una aplicación GUI escucha la entrada del usuario (como clics de ratón y pulsaciones de teclado).

El archivo más interesante es `app.py` - contiene la lógica que crea la ventana de nuestra aplicación:

```
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()
```

Vamos a ir a través de esta línea por línea:

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

En primer lugar, importamos el conjunto de herramientas de widgets `toga`, así como algunas clases de utilidades y constantes relacionadas con el estilo. Nuestro código aún no las utiliza, pero lo haremos en breve.

A continuación, definimos una clase:

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

Cada aplicación Toga tiene una única instancia `toga.App`, que representa la entidad en ejecución que es la aplicación. La app puede acabar gestionando múltiples ventanas; pero para aplicaciones sencillas, habrá una única ventana principal.

A continuación, definimos un método `startup()`:

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

Lo primero que hace el método de inicio es definir una caja principal. El esquema de diseño de Toga se comporta de forma similar a HTML. Construyes una aplicación construyendo una colección de cajas, cada una de las cuales contiene otras cajas, o widgets reales. Luego aplicas estilos a estas cajas para definir cómo consumirán el espacio disponible en la ventana.

En esta aplicación, definimos una sola caja, pero no ponemos nada en ella.

A continuación, definimos una ventana en la que podemos poner esta caja vacía:

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

Esto crea una instancia de `toga.MainWindow`, que tendrá un título que coincida con el nombre de la aplicación. Una ventana principal es un tipo especial de ventana en Toga - es una ventana que está estrechamente vinculada al ciclo de vida de la aplicación. Cuando la Ventana Principal se cierra, la aplicación sale. La Ventana Principal es también la ventana que tiene el menú de la aplicación (si estás en una plataforma como Windows donde las barras de menú son parte de la ventana)

A continuación añadimos nuestra caja vacía como contenido de la ventana principal, e indicamos a la aplicación que muestre nuestra ventana:

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

Por último, definimos un método `main()`. Esto es lo que crea la instancia de nuestra aplicación:

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

Este método `main()` es el que es importado e invocado por `__main__.py`. Crea y devuelve una instancia de nuestra aplicación `HelloWorld`.

Esa es la aplicación Toga más simple posible. Pongamos algo de nuestro propio contenido en la aplicación, y hagamos que la aplicación haga algo interesante.

2.3.2 Añadir contenido propio

Modifica tu clase `HelloWorld` dentro de `src/helloworld/app.py` para que tenga este aspecto:

```
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: No elimine las importaciones en la parte superior del archivo, o el `main()` en la parte inferior. Solo necesitas actualizar la clase `HelloWorld`.

Veamos en detalle lo que ha cambiado.

Seguimos creando una caja principal; sin embargo, ahora estamos aplicando un estilo:

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

El sistema de diseño integrado de Toga se llama «Pack». Se comporta de forma muy parecida a CSS. Defines objetos en una jerarquía - en HTML, los objetos son , , y otros elementos DOM; en Toga, son widgets y cajas. A continuación, puedes asignar estilos a los elementos individuales. En este caso, estamos indicando que se trata de una caja `COLUMN` - es decir, es una caja que consumirá todo el ancho disponible, y ampliará su altura a medida que se añada contenido, pero intentará ser lo más corta posible.

A continuación, definimos un par de widgets:

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

(continúe en la próxima página)

(proviene de la página anterior)

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

Aquí definimos un Label y un TextInput. Ambos widgets tienen estilos asociados; la etiqueta tendrá 5px de relleno a su izquierda y derecha, y ningún relleno en la parte superior e inferior. El TextInput está marcado como flexible - es decir, absorberá todo el espacio disponible en su eje de diseño.

El TextInput se asigna como una variable de instancia de la clase. Esto nos da fácil acceso a la instancia del widget - algo que usaremos en un momento.

A continuación, definimos una caja para alojar estos dos widgets:

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

La `caja_de_nombre` es una caja igual que la caja principal; sin embargo, esta vez, es una caja `ROW`. Eso significa que el contenido se añadirá horizontalmente, e intentará que su anchura sea lo más estrecha posible. La caja también tiene algo de relleno - 5px en todos los lados.

Ahora definimos un botón:

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

El botón también tiene 5px de relleno en todos los lados. También definimos un *handler* - un método a invocar cuando se pulsa el botón.

A continuación, añadimos el cuadro de nombre y el botón al cuadro principal:

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

Esto completa nuestro diseño; el resto del método de inicio es como antes - definiendo una MainWindow, y asignando la caja principal como contenido de la ventana:

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

Lo último que tenemos que hacer es definir el manejador del botón. Un manejador puede ser cualquier método, generador o co-rutina asíncrona; acepta el widget que generó el evento como argumento, y será invocado cada vez que se pulse el botón:

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

El cuerpo del método es una simple sentencia print - sin embargo, interrogará el valor actual de la entrada name, y usará ese contenido como el texto que se imprime.

Ahora que hemos realizado estos cambios podemos ver cómo quedan iniciando de nuevo la aplicación. Como antes, usaremos el modo desarrollador:

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

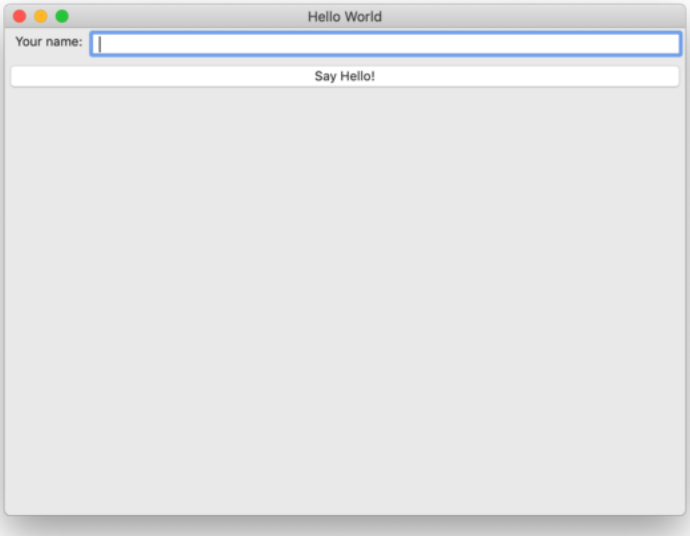
Notarás que esta vez, *no* instala dependencias. Briefcase puede detectar que la aplicación ha sido ejecutada anteriormente, y para ahorrar tiempo, sólo ejecutará la aplicación. Si añades nuevas dependencias a tu aplicación, puedes asegurarte de que se instalan pasando una opción `-r` cuando ejecutes `briefcase dev`.

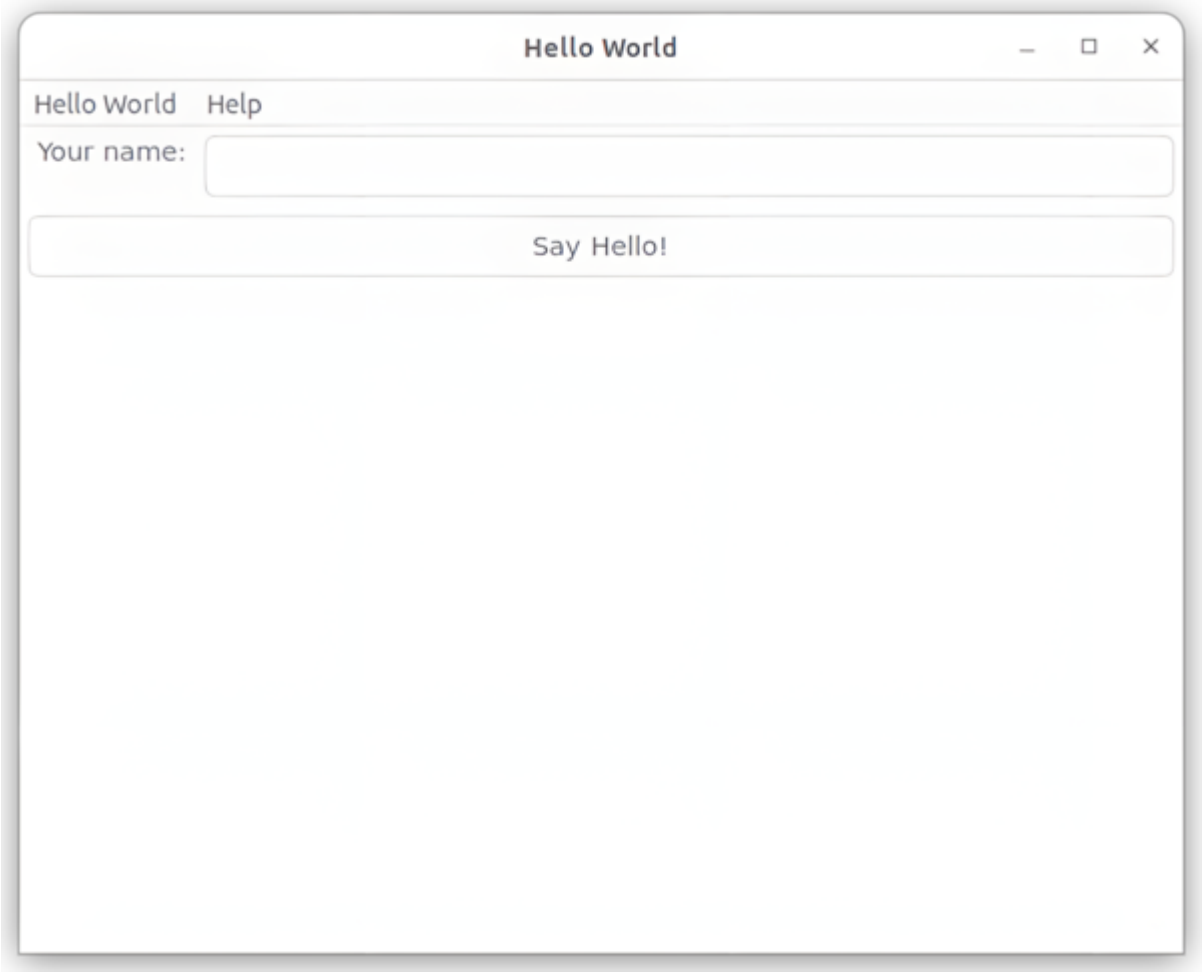
Esto debería abrir una ventana GUI:

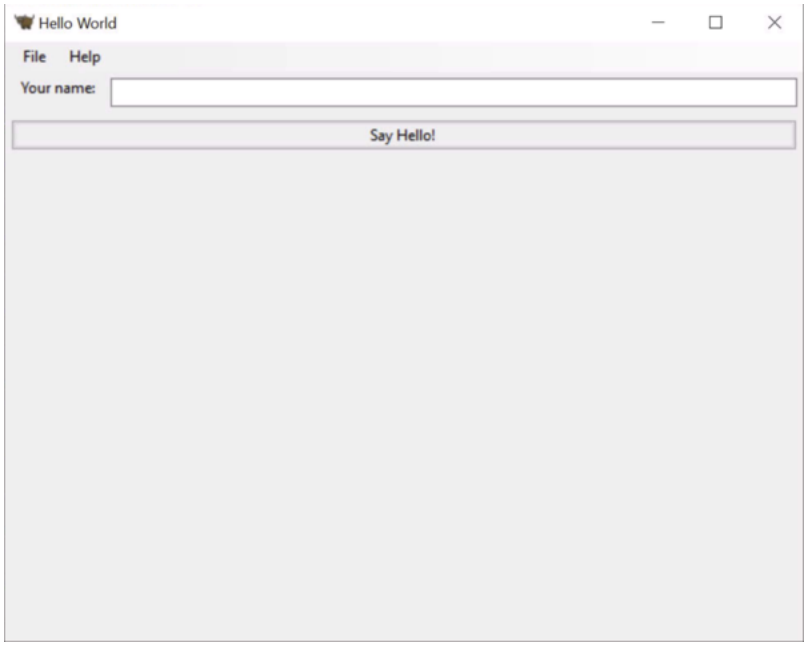
macOS

Linux

Windows







Si introduce un nombre en el cuadro de texto y pulsa el botón GUI, debería ver aparecer la salida en la consola donde inició la aplicación.

2.3.3 Siguientes pasos

Ahora tenemos una aplicación que hace algo un poco más interesante. Pero sólo se ejecuta en nuestro propio ordenador. Vamos a empaquetar esta aplicación para su distribución. En *Tutorial 3*, vamos a empaquetar nuestra aplicación como un instalador independiente que podríamos enviar a un amigo, un cliente, o subir a una App Store.

2.4 Tutorial 3 - Empaquetado para distribución

Hasta ahora, hemos estado ejecutando nuestra aplicación en «modo desarrollador». Esto hace que sea fácil para nosotros para ejecutar nuestra aplicación a nivel local - pero lo que realmente queremos es ser capaz de dar a nuestra aplicación a los demás.

Sin embargo, no queremos tener que enseñar a nuestros usuarios cómo instalar Python, crear un entorno virtual, clonar un repositorio git y ejecutar Briefcase en modo desarrollador. Preferimos simplemente darles un instalador, y que la aplicación simplemente funcione.

Briefcase puede utilizarse para empaquetar su aplicación y distribuirla de este modo.

2.4.1 Creación del andamiaje de aplicaciones

Dado que esta es la primera vez que empaquetamos nuestra aplicación, necesitamos crear algunos archivos de configuración y otros andamios para apoyar el proceso de empaquetado. Desde el directorio `helloworld`, ejecuta:

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
```

Probablemente acabas de ver pasar páginas de contenido en tu terminal… ¿qué acaba de pasar? Briefcase ha hecho lo

siguiente:

1. Se **generó una plantilla de aplicación**. Hay un montón de archivos y configuraciones necesarias para construir un instalador nativo, por encima y más allá del código de su aplicación real. Este andamiaje extra es casi el mismo para cada aplicación en la misma plataforma, excepto por el nombre de la aplicación real que se está construyendo - por lo que Briefcase proporciona una plantilla de aplicación para cada plataforma que soporta. Este paso despliega la plantilla, sustituyendo el nombre de tu aplicación, el ID del paquete y otras propiedades de tu archivo de configuración según sea necesario para soportar la plataforma en la que estás construyendo.

   Si no está satisfecho con la plantilla proporcionada por Briefcase, puede crear la suya propia. Sin embargo, probablemente no quieras hacerlo hasta que tengas un poco más de experiencia utilizando la plantilla por defecto de Briefcase.

2. Descargó e instaló un paquete de soporte**. El enfoque de empaquetado adoptado por Briefcase es mejor descrito como «la cosa más simple que podría funcionar» - envía un intérprete de Python completo y aislado como parte de cada aplicación que construye. Esto es ligeramente ineficiente en cuanto a espacio - si tiene 5 aplicaciones empaquetadas con Briefcase, tendrá 5 copias del intérprete de Python. Sin embargo, este enfoque garantiza que cada aplicación es completamente independiente, utilizando una versión específica de Python que se sabe que funciona con la aplicación.

   De nuevo, Briefcase proporciona un paquete de soporte por defecto para cada plataforma; si lo desea, puede proporcionar su propio paquete de soporte, y hacer que ese paquete se incluya como parte del proceso de compilación. Es posible que desee hacer esto si tiene opciones particulares en el intérprete de Python que necesita tener habilitadas, o si desea eliminar módulos de la biblioteca estándar que no necesita en tiempo de ejecución.

   Briefcase mantiene una caché local de paquetes de soporte, por lo que una vez que haya descargado un paquete de soporte específico, esa copia en caché se utilizará en futuras compilaciones.

3. Se **instalan los requisitos de la aplicación**. Tu aplicación puede especificar cualquier módulo de terceros que se requiera en tiempo de ejecución. Estos serán instalados usando `pip` en el instalador de tu aplicación.

4. Se **instala el código de tu aplicación**. Tu aplicación tendrá su propio código y recursos (por ejemplo, imágenes que se necesitan en tiempo de ejecución); estos archivos se copian en el instalador.

5. Por último, añade cualquier recurso adicional que necesite el propio instalador. Esto incluye cosas como iconos que deben adjuntarse a la aplicación final e imágenes de pantalla de bienvenida.

Una vez completado esto, si miras en el directorio del proyecto, deberías ver un directorio correspondiente a tu plataforma (`macOS`, `linux`, o `windows`) que contiene archivos adicionales. Esta es la configuración de empaquetado específica de la plataforma para tu aplicación.

2.4.2 Construir su aplicación

Ahora puede compilar su aplicación. Este paso realiza cualquier compilación binaria que sea necesaria para que su aplicación sea ejecutable en su plataforma de destino.

macOS

Linux

Windows

```
(beeware-venv) $ briefcase build

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

(continúe en la próxima página)

(proviene de la página anterior)

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

En macOS, el comando `build` no necesita *compilar* nada, pero sí necesita firmar el contenido del binario para que pueda ser ejecutado. Esta firma es una firma *ad hoc* - sólo funcionará en *tu* máquina; si quieres distribuir la aplicación a otros, necesitarás proporcionar 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 vez completado este paso, la carpeta `build` contendrá una carpeta `helloworld-0.0.1` que contiene una réplica de un sistema de archivos Linux `/usr`. Esta réplica del sistema de archivos contendrá una carpeta `bin` con un binario `helloworld`, además de las carpetas `lib` y `share` necesarias para soportar el binario.

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

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

En Windows, el comando `build` no necesita *compilar* nada, pero sí escribir algunos metadatos para que la aplicación sepa su nombre, versión, etc.

Activación del antivirus

Dado que estos metadatos se escriben directamente en el binario precompilado que se despliega desde la plantilla durante el comando `create`, esto puede activar el software antivirus que se ejecuta en su máquina e impedir que se escriban los metadatos. En ese caso, indique al antivirus que permita la ejecución de la herramienta (llamada `rcedit-x64.exe`)

y vuelva a ejecutar el comando anterior.

2.4.3 Ejecutar la aplicación

Ahora puede utilizar Briefcase para ejecutar su aplicación:

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:
- /usr/lib/python3.10
```

(continúe en la próxima página)

(proviene de la página anterior)

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

Esto iniciará la ejecución de su aplicación nativa, utilizando la salida del comando `build`.

Es posible que notes algunas pequeñas diferencias en el aspecto de tu aplicación cuando se está ejecutando. Por ejemplo, los iconos y el nombre mostrados por el sistema operativo pueden ser ligeramente diferentes a los que veías cuando se ejecutaba en modo desarrollador. Esto también se debe a que estás utilizando la aplicación empaquetada, no sólo ejecutando código Python. Desde la perspectiva del sistema operativo, ahora estás ejecutando «una aplicación», no «un programa Python», y esto se refleja en cómo aparece la aplicación.

2.4.4 Creación del instalador

Ahora puedes empaquetar tu aplicación para su distribución, utilizando el comando `package`. El comando `package` realiza cualquier compilación necesaria para convertir el proyecto en un producto final y distribuible. Dependiendo de la plataforma, esto puede implicar compilar un instalador, realizar la firma de código, o hacer otras tareas previas a la distribución.

macOS

Linux

Windows

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

[helloworld] Signing app...
```

(continúe en la próxima página)

(proviene de la página anterior)

```
*****************************************************************************
** 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 carpeta `dist` contendrá un archivo llamado `Hello World-0.0.1.dmg`. Si localizas este archivo en el Finder y haces doble clic en su icono, montarás el DMG, lo que te proporcionará una copia de la aplicación Hello World y un enlace a tu carpeta Aplicaciones para facilitar la instalación. Arrastra el archivo de la aplicación a Aplicaciones, y habrás instalado tu aplicación. Envía el archivo DMG a un amigo y podrá hacer lo mismo.

En este ejemplo, hemos utilizado la opción `--adhoc-sign` - es decir, estamos firmando nuestra aplicación con credenciales *ad hoc* - credenciales temporales que sólo funcionarán en tu máquina. Hemos hecho esto para mantener el tutorial simple. Configurar identidades de firma de código es un poco complicado, y sólo son *necesarias* si pretendes distribuir tu aplicación a otros. Si estuviéramos publicando una aplicación real para que otros la usaran, necesitaríamos especificar credenciales reales.

Cuando esté listo para publicar una aplicación real, consulte la guía práctica de Briefcase sobre Cómo configurar una identidad de firma de código de macOS

La salida del paso paquete será ligeramente diferente dependiendo de su distribución de Linux. Si estás en una distribución derivada de Debian, verás:

```
(beeware-venv) $ briefcase package

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

Targeting glibc 2.35
Targeting Python3.10

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

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

(continúe en la próxima página)

(proviene de la página anterior)

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

La carpeta `dist` contendrá el archivo `.deb` generado.

Si estás en una distribución basada en RHEL, lo verás:

```
(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 carpeta `dist` contendrá el archivo `.rpm` generado.

Si estás en una distribución basada en Arch, lo verás:

```
(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 carpeta `dist` contendrá el archivo `.pkg.tar.zst` generado.

Actualmente no es posible empaquetar otras distribuciones de Linux.

Si desea crear un paquete para una distribución de Linux distinta de la que está utilizando, Briefcase también puede ayudarle, pero tendrá que instalar Docker.

Existen instaladores oficiales de Docker Engine para diversas distribuciones de Unix. Siga las instrucciones para su plataforma; sin embargo, asegúrese de no instalar Docker en modo «sin raíz».

Una vez que hayas instalado Docker, deberías ser capaz de iniciar un contenedor Linux - por ejemplo:

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

te mostrará un prompt Unix (algo como `root@84444e31cff9:/#`) dentro de un contenedor Docker Ubuntu 22.04. Escribe Ctrl-D para salir de Docker y volver a tu shell local.

Una vez que tengas Docker instalado, puedes usar Briefcase para construir un paquete para cualquier distribución de Linux que soporte Briefcase pasando una imagen Docker como argumento. Por ejemplo, para construir un paquete DEB para Ubuntu 22.04 (Jammy), independientemente del sistema operativo en el que se encuentre, puede ejecutar:

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

Esto descargará la imagen Docker para el sistema operativo seleccionado, creará un contenedor capaz de ejecutar las compilaciones de Briefcase y compilará el paquete de la aplicación dentro de la imagen. Una vez completado, la carpeta `dist` contendrá el paquete para la distribución Linux de destino.

```
(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
```

En este ejemplo, hemos utilizado la opción `--adhoc-sign` - es decir, estamos firmando nuestra aplicación con credenciales *ad hoc* - credenciales temporales que sólo funcionarán en tu máquina. Hemos hecho esto para mantener el tutorial simple. Configurar identidades de firma de código es un poco complicado, y sólo son *necesarias* si pretendes distribuir tu aplicación a otros. Si estuviéramos publicando una aplicación real para que otros la usaran, necesitaríamos especificar credenciales reales.

Cuando esté listo para publicar una aplicación real, consulte la guía práctica de Briefcase sobre Cómo configurar una identidad de firma de código de macOS

Una vez completado este paso, la carpeta `dist` contendrá un archivo llamado `Hello_World-0.0.1.msi`. Si haces doble clic en este instalador para ejecutarlo, deberías seguir el proceso de instalación de Windows que ya conoces. Una

vez finalizada la instalación, aparecerá una entrada «Hello World» en el menú de inicio.

2.4.5 Siguientes pasos

Ya tenemos nuestra aplicación empaquetada para su distribución en plataformas de escritorio. Pero, ¿qué ocurre cuando necesitamos actualizar el código de nuestra aplicación? ¿Cómo introducimos esas actualizaciones en nuestra aplicación empaquetada? Visita *Tutorial 4* para averiguarlo…

2.5 Tutorial 4 - Actualización de la aplicación

En el último tutorial, empaquetamos nuestra aplicación como una aplicación nativa. Si estás tratando con una aplicación del mundo real, eso no va a ser el final de la historia - es probable que hagas algunas pruebas, descubras problemas, y necesites hacer algunos cambios. Incluso si tu aplicación es perfecta, con el tiempo querrás publicar la versión 2 de tu aplicación con mejoras.

¿Cómo se actualiza la aplicación instalada cuando se realizan cambios en el código?

2.5.1 Actualización del código de la aplicación

Actualmente, nuestra aplicación imprime en la consola cuando se pulsa el botón. Sin embargo, las aplicaciones GUI no deberían usar la consola para la salida. Necesitan usar diálogos para comunicarse con los usuarios.

Vamos a añadir un cuadro de diálogo para decir hola, en lugar de escribir en la consola. Modifica el callback `say_hello` para que se vea así:

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

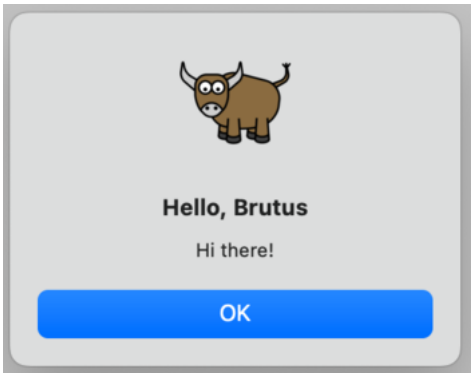
Esto indica a Toga que abra un cuadro de diálogo modal cuando se pulse el botón.

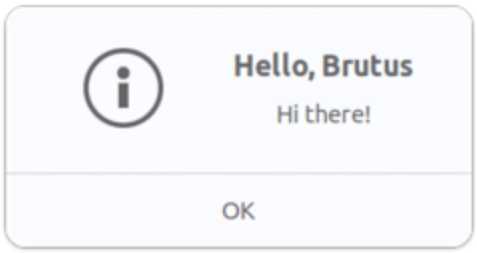
Si ejecutas `briefcase dev`, introduces un nombre y pulsas el botón, verás el nuevo cuadro de diálogo:

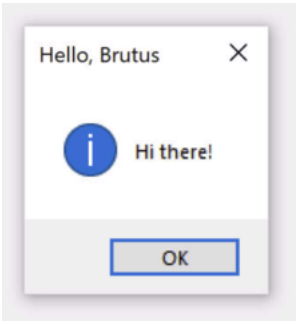
macOS

Linux

Windows







Sin embargo, si ejecutas `briefcase run`, el cuadro de diálogo no aparecerá.

¿Por qué? Bueno, `briefcase dev` opera ejecutando tu código en su lugar - intenta producir un entorno de ejecución lo más realista posible para tu código, pero no proporciona ni utiliza ninguna de las infraestructuras de la plataforma para empaquetar tu código como una aplicación. Parte del proceso de empaquetar tu aplicación implica copiar tu código *dentro* del paquete de la aplicación - y por el momento, tu aplicación todavía tiene el código antiguo en ella.

Así que - necesitamos decirle a briefcase que actualice tu aplicación, copiando la nueva versión del código. Podríamos *hacerlo* borrando el antiguo directorio de la plataforma y empezando desde cero. Sin embargo, Briefcase proporciona una manera más fácil - usted puede actualizar el código de su aplicación empaquetada existente:

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...
```

(continúe en la próxima página)

(proviene de la página anterior)

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

[helloworld] Application updated.
```

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

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

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

[helloworld] Application updated.
```

Si Briefcase no puede encontrar la plantilla de andamiaje, invocará automáticamente `create` para generar un andamiaje nuevo.

Ahora que hemos actualizado el código del instalador, podemos ejecutar `briefcase build` para volver a compilar la aplicación, `briefcase run` para ejecutar la aplicación actualizada, y `briefcase package` para volver a empaquetar la aplicación para su distribución.

(usuarios de macOS, recordad que como se indica en *Tutorial 3*, para el tutorial recomendamos ejecutar `briefcase package` con la bandera `--adhoc-sign` para evitar la complejidad de configurar una identidad de firma de código y mantener el tutorial lo más simple posible)

2.5.2 Actualizar y ejecutar en un solo paso

Si estás iterando rápidamente cambios en el código, es probable que quieras hacer un cambio en el código, actualizar la aplicación, y volver a ejecutar inmediatamente tu aplicación. Para la mayoría de los propósitos, el modo desarrollador (`briefcase dev`) será la forma más fácil de hacer este tipo de iteración rápida; sin embargo, si estás probando algo sobre cómo se ejecuta tu aplicación como un binario nativo, o buscando un error que sólo se manifiesta cuando tu aplicación está empaquetada, puede que necesites usar repetidas llamadas a `briefcase run`. Para simplificar el proceso de actualización y ejecución de la aplicación empaquetada, Briefcase tiene un atajo para soportar este patrón de uso - la opción `-u` (o `--update`) en el comando `run`.

Intentemos hacer otro cambio. Usted puede haber notado que si no escribe un nombre en el cuadro de entrada de texto, el cuadro de diálogo dirá «Hola, «. Vamos a modificar la función `say_hello` de nuevo para manejar este caso extremo.

En la parte superior del fichero, entre las importaciones y la definición de la `clase HelloWorld`, añade un método de utilidad para generar un saludo apropiado dependiendo del valor del nombre que se haya proporcionado:

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

A continuación, modifica la llamada de retorno `say_hello` para utilizar este nuevo método de utilidad:

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

Ejecuta tu aplicación en modo desarrollo (con `briefcase dev`) para confirmar que la nueva lógica funciona; después actualiza, compila y ejecuta la aplicación 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...
```

(continúe en la próxima página)

(proviene de la página anterior)

```
Installing src/helloworld... done

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

[helloworld] Application updated.

[helloworld] Starting app...
```

El comando package también acepta el argumento `-u`, de modo que si realizas un cambio en el código de tu aplicación y quieres volver a empaquetarlo inmediatamente, puedes ejecutar `briefcase package -u`.

2.5.3 Siguientes pasos

Ahora tenemos nuestra aplicación empaquetada para su distribución en plataformas de escritorio, y hemos podido actualizar el código de nuestra aplicación.

Pero, ¿qué pasa con los móviles? En *Tutorial 5*, convertiremos nuestra aplicación en una aplicación móvil, y la desplegaremos en un simulador de dispositivos, y en un teléfono.

2.6 Tutorial 5 - Móviles

Hasta ahora, hemos estado ejecutando y probando nuestra aplicación en el escritorio. Sin embargo, BeeWare también es compatible con plataformas móviles, ¡y la aplicación que hemos escrito también puede desplegarse en tu dispositivo móvil!

iOS Las aplicaciones de iOS solo pueden compilarse en macOS.

¡Creemos nuestra aplicación para iOS!

Android Las aplicaciones Android pueden compilarse en macOS, Windows o Linux.

¡Creemos nuestra aplicación para Android!

2.6.1 Tutorial 5 - Para móviles: iOS

Para compilar aplicaciones de iOS necesitaremos Xcode, que está disponible de forma gratuita en la App Store de macOS.

Una vez que tengamos Xcode instalado, podemos tomar nuestra aplicación e implementarla como una aplicación de iOS.

El proceso de implementación de una aplicación en iOS es muy similar al proceso de implementación como aplicación de escritorio. Primero, ejecuta el comando `crear`, pero esta vez especificamos que queremos crear una aplicación para 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
...
```

(continúe en la próxima página)

(proviene de la página anterior)

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

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

[helloworld] Installing requirements...
...

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

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

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

Una vez que esto se complete, tendremos un directorio `build/helloworld/ios/xcode` que contiene un proyecto Xcode, así como las bibliotecas de soporte y el código de aplicación necesario para la aplicación.

Luego puedes usar Briefcase para compilar tu aplicación 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
```

Ya estamos listos para ejecutar nuestra aplicación, utilizando `briefcase run iOS`. Se te pedirá que selecciones un dispositivo para el que compilar; si tienes instalados simuladores para varias versiones del SDK de iOS, puede que también se te pregunte a qué versión de iOS quieres apuntar. Las opciones que se le muestren pueden diferir de las que se muestran en esta salida; como mínimo, es probable que la lista de dispositivos sea diferente. Para nuestros propósitos, no importa qué simulador elija.

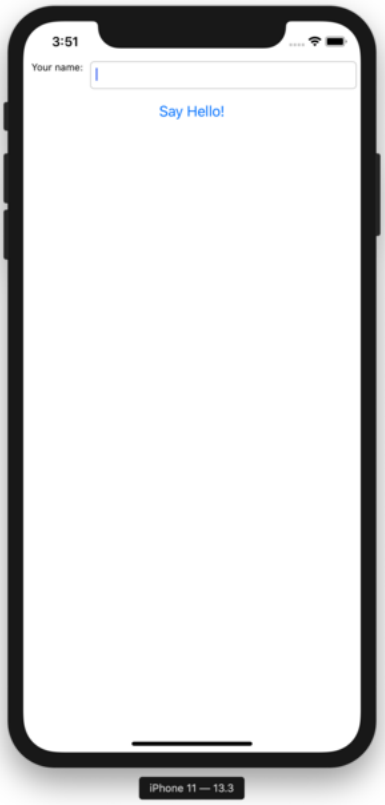
```
(beeware-venv) $ briefcase run iOS

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
```

(continúe en la próxima página)

(proviene de la página anterior)

```
   10) iPhone SE (3rd generation)

>  10

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

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

or:

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

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

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

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

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

Esto iniciará el simulador de iOS, instalará tu aplicación y la iniciará. Deberías ver cómo se inicia el simulador y, finalmente, cómo se abre tu aplicación iOS:



Si sabes de antemano a qué simulador iOS quieres dirigirte, puedes decirle a Briefcase que utilice ese simulador proporcionando una opción `-d` (o `--device`). Utilizando el nombre del dispositivo que seleccionaste cuando creaste tu aplicación, ejecuta:

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

Si tienes varias versiones de iOS disponibles, Briefcase elegirá la versión de iOS más alta; si quieres elegir una versión de iOS en particular, dile que utilice esa versión específica:

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

O bien, puede nombrar un UDID de dispositivo específico:

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

Siguientes pasos

¡Ya tenemos una aplicación en nuestro teléfono! ¿Hay algún otro lugar donde podamos desplegar una aplicación BeeWare? Visita *Tutorial 6* para averiguarlo…

2.6.2 Tutorial 5 - Movilidad: Android

Ahora, vamos a tomar nuestra aplicación, y desplegarla como una aplicación Android.

El proceso de despliegue de una aplicación en Android es muy similar al proceso de despliegue como aplicación de escritorio. Briefcase se encarga de instalar las dependencias para Android, incluyendo el SDK de Android, el emulador de Android y un compilador de Java.

Crear una aplicación Android y compilarla

Primero, ejecuta el comando `create`. Esto descarga una plantilla de aplicación Android y le añade tu código 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
...
```

(continúe en la próxima página)

(proviene de la página anterior)

```
[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
```

Cuando ejecutas `briefcase create android` por primera vez, Briefcase descarga un JDK de Java, y el SDK de Android. El tamaño de los archivos y el tiempo de descarga pueden ser considerables; esto puede llevar un tiempo (10 minutos o más, dependiendo de la velocidad de tu conexión a Internet). Una vez finalizada la descarga, se le pedirá que acepte la licencia del SDK de Android de Google.

Una vez completado esto, tendremos un directorio `build\helloworld\android\gradle` en nuestro proyecto, que contendrá un proyecto Android con una configuración de construcción Gradle. Este proyecto contendrá el código de la aplicación, y un paquete de soporte que contiene el intérprete de Python.

A continuación, podemos utilizar el comando `build` de Briefcase para compilar esto en un archivo de aplicación 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 puede parecer atascado

Durante el paso `briefcase build android`, Gradle (la herramienta de compilación de la plataforma Android) impri-

mirá `CONFIGURING: 100%`, y parecerá que no está haciendo nada. No te preocupes, no está atascado - está descargando más componentes del SDK de Android. Dependiendo de la velocidad de su conexión a Internet, esto puede tardar otros 10 minutos (o más). Este retraso sólo debería ocurrir la primera vez que ejecutes `build`; las herramientas se almacenan en caché, y en tu próxima compilación, se utilizarán las versiones almacenadas en caché.

Ejecutar la aplicación en un dispositivo virtual

Ahora estamos listos para ejecutar nuestra aplicación. Puedes utilizar el comando `run` de Briefcase para ejecutar la aplicación en un dispositivo Android. Empecemos ejecutando en un emulador de Android.

Para ejecutar tu aplicación, ejecuta `briefcase run android`. Al hacerlo, se te pedirá una lista de dispositivos en los que podrías ejecutar la aplicación. El último elemento será siempre una opción para crear un nuevo emulador de 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

>
```

Ahora podemos elegir el dispositivo deseado. Selecciona la opción «Crear un nuevo emulador de Android», y acepta la opción por defecto para el nombre del dispositivo (`beePhone`).

Briefcase `run` arrancará automáticamente el dispositivo virtual. Cuando el dispositivo esté arrancando, verás el logo de Android:

Una vez que el dispositivo haya terminado de arrancar, Briefcase instalará tu aplicación en el dispositivo. Verás brevemente una pantalla de inicio:

La aplicación se iniciará. Verás una pantalla de bienvenida mientras se inicia la aplicación:

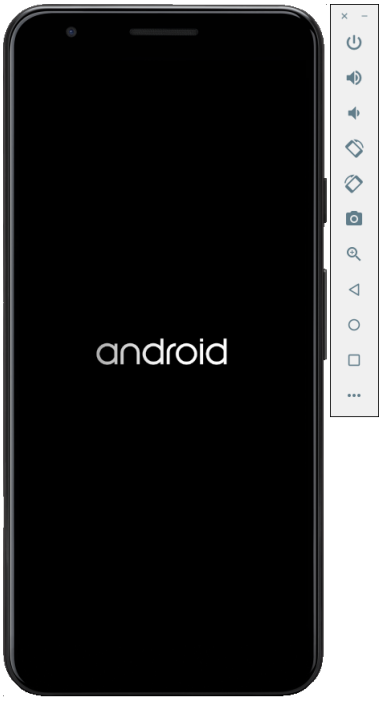


Figura 1: Arranque del dispositivo virtual Android

Figura 2: Dispositivo virtual Android totalmente iniciado, en la pantalla del lanzador

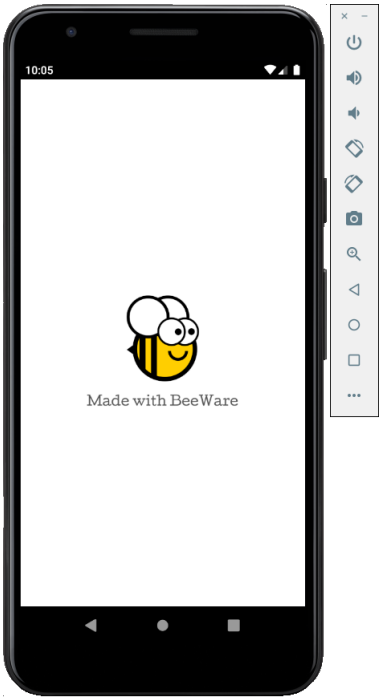


Figura 3: Pantalla de inicio de la aplicación

¡El emulador no arranca!

El emulador de Android es un software complejo que depende de una serie de características del hardware y del sistema operativo, características que pueden no estar disponibles o habilitadas en máquinas más antiguas. Si tiene dificultades para iniciar el emulador de Android, consulte la sección Requerimientos y recomendaciones de la documentación para desarrolladores de Android.

La primera vez que se inicia la aplicación, tiene que desempaquetarse en el dispositivo. Esto puede tardar unos segundos. Una vez desempaquetada, verás la versión para Android de nuestra aplicación de escritorio:

Si no ves que tu aplicación se inicia, es posible que tengas que comprobar el terminal donde ejecutaste `briefcase run` y buscar cualquier mensaje de error.

En el futuro, si desea ejecutar en este dispositivo sin utilizar el menú, puede proporcionar el nombre del emulador a Briefcase, utilizando `briefcase run android -d @beePhone` para ejecutar en el dispositivo virtual directamente.

Ejecutar la aplicación en un dispositivo físico

Si tienes un teléfono o tableta Android físicos, puedes conectarlos a tu ordenador con un cable USB y, a continuación, utilizar el Maletín para apuntar a tu dispositivo físico.

Android requiere que prepares tu dispositivo antes de que pueda ser utilizado para el desarrollo. Tendrás que realizar 2 cambios en las opciones de tu dispositivo:

- Activar opciones de desarrollador
- Activar la depuración USB

Los detalles sobre cómo realizar estos cambios se pueden encontrar en la documentación para desarrolladores de Android.

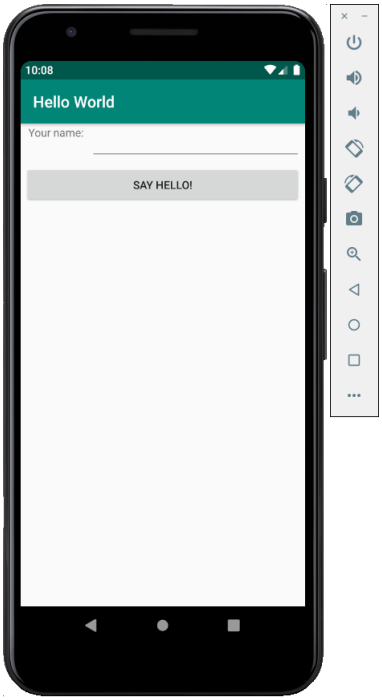


Figura 4: Lanzamiento completo de la aplicación de demostración

Una vez completados estos pasos, tu dispositivo debería aparecer en la lista de dispositivos disponibles cuando ejecutes `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

>
```

Aquí podemos ver un nuevo dispositivo físico con su número de serie en la lista de despliegue - en este caso, un Pixel 3a. En el futuro, si desea ejecutar en este dispositivo sin utilizar el menú, puede proporcionar el número de serie del teléfono a Briefcase (en este caso, `briefcase run android -d 94ZZY0LNE8`). Esto se ejecutará en el dispositivo directamente, sin preguntar.

¡Mi dispositivo no aparece!

Si tu dispositivo no aparece en esta lista, significa que no has activado la depuración USB (o que el dispositivo no está conectado).

Si su dispositivo aparece, pero aparece como «Dispositivo desconocido (no autorizado para el desarrollo)», el modo de desarrollador no se ha habilitado correctamente. Vuelve a ejecutar los pasos para habilitar las opciones de desarrollador, y vuelve a ejecutar `briefcase run android`.

Siguientes pasos

¡Ya tenemos una aplicación en nuestro teléfono! ¿Hay algún otro lugar donde podamos desplegar una aplicación BeeWare? Visita *Tutorial 6* para averiguarlo…

2.7 Tutorial 6 - ¡Ponlo en la web!

Además de ser compatible con plataformas móviles, el conjunto de herramientas de widgets Toga también es compatible con la web Utilizando la misma API que usaste para desplegar tus aplicaciones de escritorio y móviles, puedes desplegar tu aplicación como una aplicación web de una sola página.

Prueba de concepto

El backend Toga Web es el menos maduro de todos los backends Toga. Es lo suficientemente maduro como para mostrar algunas características, pero es probable que tenga errores, y le faltarán muchos de los widgets que están disponibles en otras plataformas. En este momento, el despliegue Web debe considerarse una «Prueba de Concepto» - suficiente para demostrar lo que se puede hacer, pero no lo suficiente como para confiar en él para un desarrollo serio.

Si tiene problemas con este paso del tutorial, puede pasar a la página siguiente.

2.7.1 Despliegue como aplicación web

El proceso de despliegue de una aplicación web de una sola página sigue el mismo patrón familiar: se crea la aplicación, se compila y se ejecuta. Sin embargo, Briefcase puede ser un poco inteligente; si intentas ejecutar una aplicación, y Briefcase determina que no ha sido creada o construida para la plataforma a la que se dirige, realizará los pasos de creación y construcción por ti. Dado que esta es la primera vez que ejecutamos la aplicación para la web, podemos realizar los tres pasos 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
...
```

(continúe en la próxima página)

(proviene de la página anterior)

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

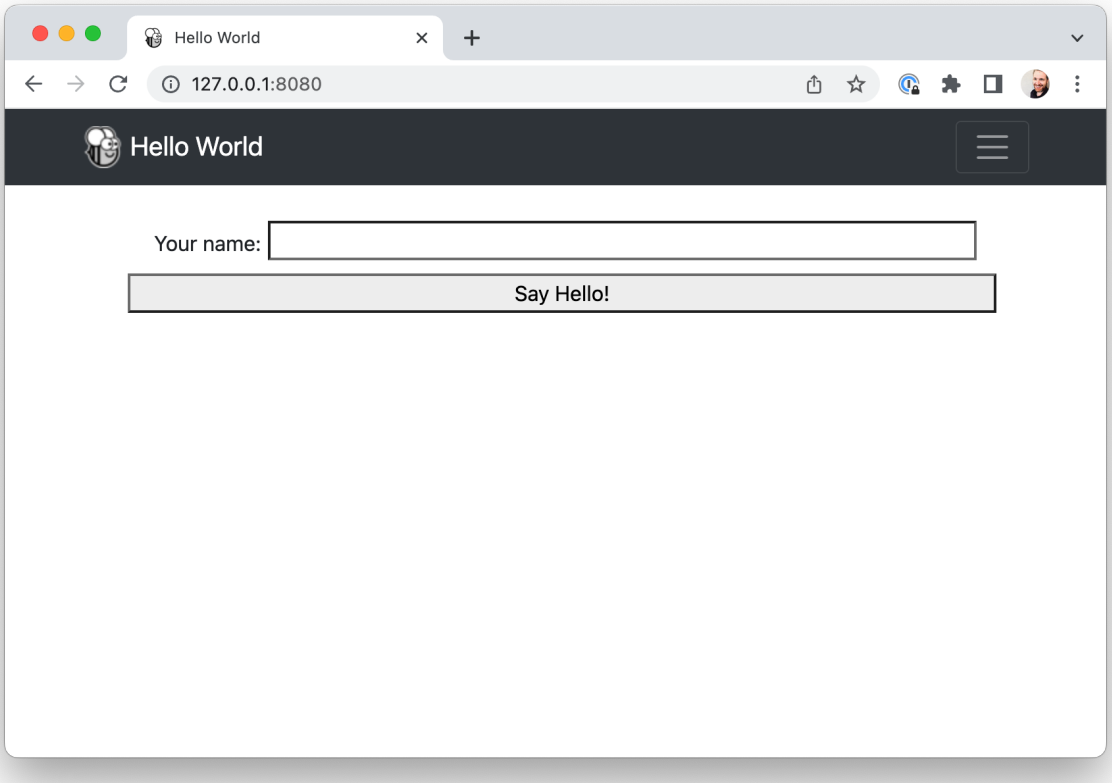
```
(beeware-venv) C:\...>briefcase run web

[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
```

(continúe en la próxima página)

(proviene de la página anterior)

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

Esto abrirá un navegador web, apuntando a http://127.0.0.1:8080:



Si introduce su nombre y pulsa el botón, aparecerá un cuadro de diálogo.

2.7.2 ¿Como funciona esto?

Esta aplicación web es un sitio web estático - una única página fuente HTML, con algunos CSS y otros recursos. Briefcase ha iniciado un servidor web local para servir esta página y que tu navegador pueda verla. Si quisieras poner esta página web en producción, podrías copiar el contenido de la carpeta `www` en cualquier servidor web que pueda servir contenido estático.

Pero cuando pulsas el botón, estás ejecutando código Python… ¿cómo funciona eso? Toga utiliza PyScript para proporcionar un intérprete de Python en el navegador. Briefcase empaqueta el código de tu aplicación como ruedas que PyScript puede cargar en el navegador. Cuando se carga la página, el código de la aplicación se ejecuta en el navegador, construyendo la interfaz de usuario utilizando el DOM del navegador. Cuando pulsas un botón, ese botón ejecuta el código de gestión de eventos en el navegador.

2.7.3 Siguientes pasos

Aunque ya hemos desplegado esta aplicación en ordenadores de sobremesa, dispositivos móviles y la web, la aplicación es bastante sencilla y no incluye bibliotecas de terceros. ¿Podemos incluir librerías del Python Package Index (PyPI) en nuestra aplicación? Visita *Tutorial 7* para averiguarlo…

2.8 Tutorial 7 - Poner en marcha esta (tercera)fiesta

Hasta ahora, la aplicación que hemos construido sólo ha utilizado nuestro propio código, además del código proporcionado por BeeWare. Sin embargo, en una aplicación del mundo real, es probable que desees utilizar una biblioteca de terceros, descargada del Python Package Index (PyPI).

Vamos a modificar nuestra aplicación para incluir una biblioteca de terceros.

2.8.1 Acceso a una API

Una tarea común que una aplicación necesitará realizar es hacer una petición a una API web para recuperar datos, y mostrar esos datos al usuario. Esta es una aplicación de juguete, así que no tenemos una API *real* con la que trabajar, así que usaremos la {JSON} Placeholder API como fuente de datos.

El {JSON} Placeholder API tiene una serie de «falsos» puntos finales de la API que puede utilizar como datos de prueba. Una de esas API es el punto final `/posts/`, que devuelve entradas de blog falsas. Si abre `https://jsonplaceholder.typicode.com/posts/42` en su navegador, obtendrá una carga útil JSON que describe una única entrada: algo de contenido Lorum ipsum para una entrada de blog con ID 42.

La biblioteca estándar de Python contiene todas las herramientas necesarias para acceder a una API. Sin embargo, las API incorporadas son de muy bajo nivel. Son buenas implementaciones del protocolo HTTP, pero requieren que el usuario gestione muchos detalles de bajo nivel, como la redirección de URL, las sesiones, la autenticación y la codificación de la carga útil. Como «usuario normal de navegador» probablemente estés acostumbrado a dar por sentado estos detalles, ya que un navegador gestiona estos detalles por ti.

Como resultado, la gente ha desarrollado bibliotecas de terceros que envuelven las APIs integradas y proporcionan una API más simple que se acerca más a la experiencia cotidiana del navegador. Vamos a utilizar una de esas bibliotecas para acceder a la API {JSON} Placeholder - una biblioteca llamada httpx.

Vamos a añadir una llamada a la API `httpx` a nuestra aplicación. Añade un import al principio de `app.py` para importar `httpx`:

```
import httpx
```

Luego modifica el callback `say_hello()` para que se vea así:

```
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"],
    )
```

Esto cambiará la llamada de retorno `say_hello()` para que cuando sea invocada, lo haga:

- realice una solicitud GET en la API de marcador de posición JSON para obtener el puesto 42;
- decodificar la respuesta como JSON;
- extraer el cuerpo del mensaje; y
- incluir el cuerpo de ese mensaje como texto del diálogo.

Vamos a ejecutar nuestra aplicación actualizada en el modo de desarrollador de Briefcase para comprobar que nuestro cambio ha funcionado.

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>
```

(continúe en la próxima página)

(proviene de la página anterior)

```
    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'
```

¿Qué ha pasado? Hemos añadido `httpx` a nuestro *código*, pero no lo hemos añadido a nuestro entorno virtual de desarrollo. Podemos solucionarlo instalando `httpx` con `pip`, y volviendo a ejecutar `briefcase dev`:

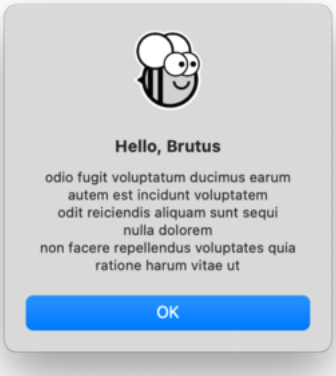
macOS

Linux

Windows

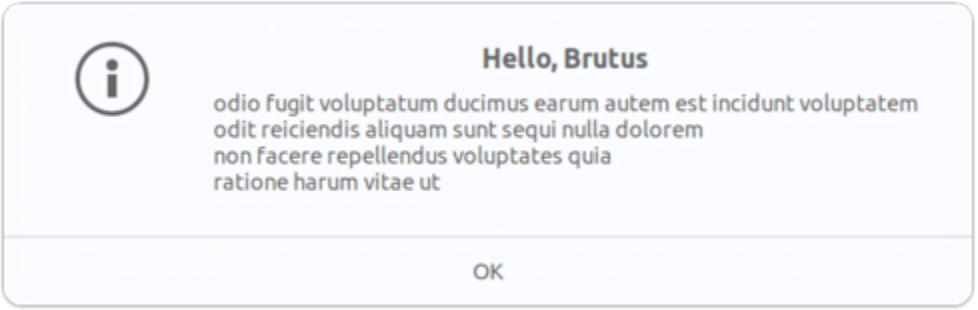
```
(beeware-venv) $ python -m pip install httpx
(beeware-venv) $ briefcase dev
```

Cuando introduzcas un nombre y pulses el botón, deberías ver un cuadro de diálogo parecido a:



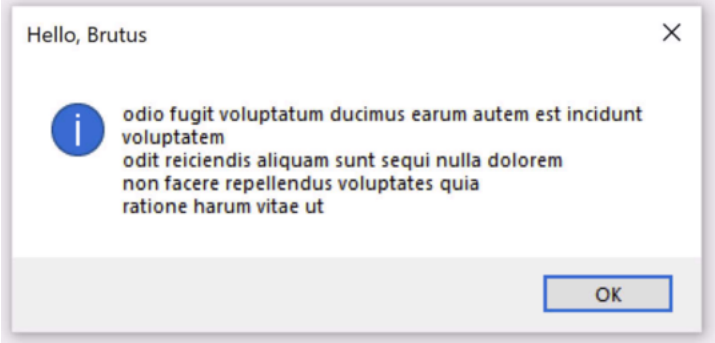

```
(beeware-venv) $ python -m pip install httpx
(beeware-venv) $ briefcase dev
```

Cuando introduzcas un nombre y pulses el botón, deberías ver un cuadro de diálogo parecido a:




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

Cuando introduzcas un nombre y pulses el botón, deberías ver un cuadro de diálogo parecido a:



Ya tenemos una aplicación que funciona, que utiliza una biblioteca de terceros y que se ejecuta en modo de desarrollo

2.8.2 Ejecutar la aplicación actualizada

Vamos a empaquetar este código de aplicación actualizado como una aplicación independiente. Como hemos hecho cambios en el código, tenemos que seguir los mismos pasos que en *Tutorial 4*:

macOS

Linux

Windows

Actualice el código en la aplicación empaquetada:

```
(beeware-venv) $ briefcase update

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

[helloworld] Application updated.
```

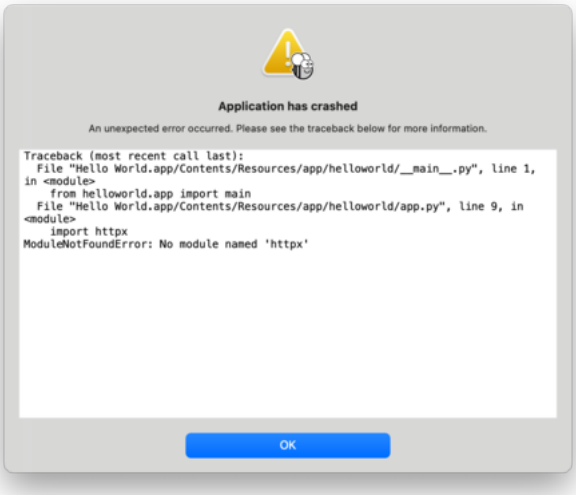
Reconstruye la aplicación:

```
(beeware-venv) $ briefcase build

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

Y, por último, ejecuta la aplicación:

```
(beeware-venv) $ briefcase run

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

Sin embargo, cuando la aplicación se ejecute, verás un error en la consola y un cuadro de diálogo de bloqueo:



Actualice el código en la aplicación empaquetada:

```
(beeware-venv) $ briefcase update

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

[helloworld] Application updated.
```

Reconstruye la aplicación:

```
(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
```

Y, por último, ejecuta la aplicación:

```
(beeware-venv) $ briefcase run

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

Sin embargo, cuando la aplicación se ejecute, verás un error en la consola:

```
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.
```

Actualice el código en la aplicación empaquetada:

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

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

[helloworld] Application updated.
```

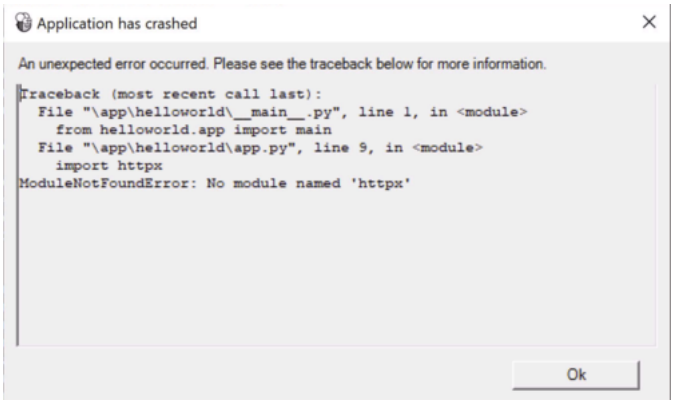
Reconstruye la aplicación:

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

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

Y, por último, ejecuta la aplicación:

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

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

Sin embargo, cuando la aplicación se ejecute, verás un error en la consola y un cuadro de diálogo de bloqueo:



Una vez más, la aplicación no ha podido iniciarse porque `httpx` ha sido instalado - pero ¿por qué? ¿No hemos instalado ya `httpx`?

Sí, pero sólo en el entorno de desarrollo. Tu entorno de desarrollo es totalmente local a tu máquina - y sólo se activa cuando lo activas explícitamente. Aunque Briefcase tiene un modo de desarrollo, la principal razón por la que utilizarías Briefcase es para empaquetar tu código y poder dárselo a otra persona.

La única forma de garantizar que alguien más tenga un entorno Python que contenga todo lo que necesita es construir un entorno Python completamente aislado. Esto significa que hay una instalación de Python completamente aislada, y un conjunto de dependencias completamente aislado. Esto es lo que Briefcase construye cuando ejecutas `briefcase build` - un entorno Python aislado. Esto también explica por qué `httpx` no está instalado - ha sido instalado en tu entorno de *desarrollo*, pero no en la aplicación empaquetada.

Por lo tanto, tenemos que decirle a Briefcase que nuestra aplicación tiene una dependencia externa.

2.8.3 Actualización de las dependencias

En el directorio raíz de tu aplicación, hay un archivo llamado `pyproject.toml`. Este archivo contiene todos los detalles de configuración de la aplicación que proporcionaste cuando ejecutaste `briefcase new`.

`pyproject.toml` está dividido en secciones; una de las secciones describe la configuración de tu aplicación:

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

(continúe en la próxima página)

(proviene de la página anterior)

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

La opción `requires` describe las dependencias de nuestra aplicación. Es una lista de cadenas, especificando librerías (y, opcionalmente, versiones) de librerías que quieres que se incluyan con tu app.

Modifique la configuración `requires` para que diga:

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

Al añadir esta opción, le estamos diciendo a Briefcase «cuando compiles mi aplicación, ejecuta `pip install httpx` en el paquete de la aplicación». Cualquier cosa que sea una entrada legal para `pip install` se puede utilizar aquí - por lo que podría especificar:

- Una versión específica de la biblioteca (por ejemplo, `"httpx==0.19.0"`);
- Un rango de versiones de la biblioteca (por ejemplo, `"httpx>=0.19"`);
- Una ruta a un repositorio git (por ejemplo, `"git+https://github.com/encode/httpx"`); o bien
- Una ruta de archivo local (No obstante, ten cuidado: si entregas tu código a otra persona, es probable que esta ruta no exista en su máquina)

Más abajo en `pyproject.toml`, verás otras secciones que dependen del sistema operativo, como `[tool.briefcase.app.helloworld.macOS]` y `[tool.briefcase.app.helloworld.windows]`. Estas secciones también tienen una configuración `requires`. Estas opciones te permiten definir dependencias adicionales específicas de la plataforma; así, por ejemplo, si necesitas una biblioteca específica de la plataforma para manejar algún aspecto de tu aplicación, puedes especificar esa biblioteca en la sección `requires` específica de la plataforma, y esa opción sólo se utilizará para esa plataforma. Notarás que las librerías `toga` están todas especificadas en la sección `requires` específica de la plataforma - esto es porque las librerías necesarias para mostrar una interfaz de usuario son específicas de la plataforma.

En nuestro caso, queremos que `httpx` se instale en todas las plataformas, por lo que utilizamos el parámetro `requires` a nivel de aplicación. Las dependencias a nivel de aplicación siempre se instalarán; las dependencias específicas de la plataforma se instalan *además* de las dependencias a nivel de aplicación.

Algunos paquetes binarios pueden no estar disponibles

En plataformas de escritorio (macOS, Windows, Linux), se puede añadir cualquier `pip`-instalable a sus requisitos. En plataformas móviles y web, las opciones son ligeramente limitadas <https://briefcase.readthedocs.io/en/latest/background/faq.html#can-i-use-third-party-python-packages-in-my-app>`__.

En resumen; cualquier paquete *puro* de Python (es decir, paquetes que *no* contienen un módulo binario) puede ser utilizado sin dificultad. Sin embargo, si su dependencia contiene un componente binario, debe ser compilado; en este momento, la mayoría de los paquetes de Python no proporcionan soporte de compilación para plataformas que no sean de escritorio.

BeeWare puede proporcionar binarios para algunos módulos binarios populares (incluyendo `numpy`, `pandas`, y `cryptography`). Es *normalmente* posible compilar paquetes para plataformas móviles, pero no es fácil de configurar - fuera del alcance de un tutorial introductorio como éste.

Ahora que hemos informado a Briefcase sobre nuestros requisitos adicionales, podemos intentar empaquetar nuestra aplicación de nuevo. Asegúrate de que has guardado los cambios en `pyproject.toml`, y luego actualiza tu aplicación de nuevo - esta vez, introduciendo la bandera `-r`. Esto le dice a Briefcase que actualice los requisitos en la aplicación empaquetada:

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

[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 vez que haya actualizado, puede ejecutar `briefcase build` y `briefcase run` - y usted debe ver su aplicación empaquetada, con el nuevo comportamiento de diálogo.

Nota: La opción `-r` para actualizar los requisitos también es respetada por los comandos `build` y `run`, así que si quieres actualizar, construir y ejecutar en un solo paso, puedes usar `briefcase run -u -r`.

2.8.4 Siguientes pasos

Ya tenemos una aplicación que utiliza una biblioteca de terceros Sin embargo, te habrás dado cuenta de que cuando pulsas el botón, la aplicación deja de responder. ¿Podemos hacer algo para solucionarlo? Visita *Tutorial 8* para averiguarlo…

2.9 Tutorial 8 - Suavizar

A menos que tengas una conexión a Internet *realmente* rápida, puede que notes que cuando pulsas el botón, la GUI de tu aplicación se bloquea un poco. Esto se debe a que la petición web que hemos realizado es *síncrona*. Cuando nuestra aplicación realiza la petición web, espera a que la API devuelva una respuesta antes de continuar. Mientras espera, *no* permite a la aplicación redibujar - y como resultado, la aplicación se bloquea.

2.9.1 Bucles de eventos GUI

Para entender por qué ocurre esto, tenemos que profundizar en los detalles del funcionamiento de una aplicación GUI. Los detalles varían en función de la plataforma, pero los conceptos de alto nivel son los mismos, independientemente de la plataforma o el entorno GUI que utilices.

Una aplicación GUI es, fundamentalmente, un único bucle parecido a:

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

Este bucle se llama *Bucle de Evento*. (Estos no son nombres de métodos reales - es una ilustración de lo que está pasando en «pseudo-código»).

Cuando haces clic en un botón, arrastras una barra de desplazamiento o tecleas una tecla, estás generando un «evento». Ese «evento» se coloca en una cola, y la aplicación procesará la cola de eventos la próxima vez que tenga la oportunidad de hacerlo. El código de usuario que se activa en respuesta al evento se denomina *manejador de eventos*. Estos manejadores de eventos son invocados como parte de la llamada `process_events()`.

Una vez que una aplicación ha procesado todos los eventos disponibles, `redibujará()` la GUI. Esto tiene en cuenta cualquier cambio que los eventos hayan causado en la pantalla de la aplicación, así como cualquier otra cosa que esté ocurriendo en el sistema operativo - por ejemplo, las ventanas de otra aplicación pueden oscurecer o revelar parte de la ventana de nuestra aplicación, y el redibujado de nuestra aplicación tendrá que reflejar la parte de la ventana que es visible en ese momento.

El detalle importante a tener en cuenta: mientras una aplicación está procesando un evento, *no puede redibujar*, y *no puede procesar otros eventos*.

Esto significa que cualquier lógica de usuario contenida en un manejador de eventos necesita completarse rápidamente. Cualquier retraso en la finalización del manejador de eventos será observado por el usuario como una ralentización (o detención) en las actualizaciones de la interfaz gráfica de usuario. Si este retraso es lo suficientemente largo, tu sistema operativo puede reportarlo como un problema - los iconos «beachball» de macOS y «spinner» de Windows son el sistema operativo diciéndote que tu aplicación está tardando demasiado en un manejador de eventos.

Operaciones sencillas como «actualizar una etiqueta» o «volver a calcular el total de las entradas» son fáciles de completar rápidamente. Sin embargo, hay muchas operaciones que no pueden completarse rápidamente. Si realizas un cálculo matemático complejo, o indexas todos los archivos de un sistema de ficheros, o realizas una petición de red de gran tamaño, no puedes «hacerlo rápido»: las operaciones son intrínsecamente lentas.

Entonces, ¿cómo realizar operaciones de larga duración en una aplicación GUI?

2.9.2 Programación asíncrona

Lo que necesitamos es una manera de decirle a una aplicación en medio de un manejador de eventos de larga duración que está bien liberar temporalmente el control de nuevo al bucle de eventos, siempre y cuando podamos reanudar donde lo dejamos. Depende de la aplicación determinar cuándo puede ocurrir esta liberación; pero si la aplicación libera el control al bucle de eventos regularmente, podemos tener un manejador de eventos de larga duración *y* mantener una interfaz de usuario responsiva.

Podemos hacerlo utilizando *programación asíncrona*. La programación asíncrona es una forma de describir un programa que permite al intérprete ejecutar varias funciones al mismo tiempo, compartiendo recursos entre todas las funciones que se ejecutan simultáneamente.

Las funciones asíncronas (conocidas como *corrutinas*) deben declararse explícitamente como asíncronas. También necesitan declarar internamente cuándo existe la oportunidad de cambiar el contexto a otra co-rutina.

En Python, la programación asíncrona se implementa utilizando las palabras clave `async` y `await`, y el módulo asyncio de la biblioteca estándar. La palabra clave `async` nos permite declarar que una función es una co-rutina asíncrona. La palabra clave `await` proporciona una forma de declarar cuando existe la oportunidad de cambiar el contexto a otra co-rutina. El módulo asyncio proporciona algunas otras herramientas útiles y primitivas para la codificación asíncrona.

2.9.3 Hacer que el tutorial sea asíncrono

Para hacer que nuestro tutorial sea asíncrono, modifica el manejador de eventos `say_hello()` para que se vea así:

```
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"],
    )
```

Sólo hay 4 cambios en este código con respecto a la versión anterior:

1. El método se define como `async def`, en lugar de sólo `def`. Esto indica a Python que el método es una co-rutina asíncrona.
2. El cliente que se crea es un `AsyncClient()` asíncrono, en lugar de un `Client()` síncrono. Esto indica a `httpx` que debe operar en modo asíncrono, en lugar de síncrono.
3. El gestor de contexto utilizado para crear el cliente está marcado como `async`. Esto le indica a Python que existe la oportunidad de liberar el control a medida que se entra y se sale del gestor de contexto.
4. La llamada `get` se hace con una palabra clave `await`. Esto indica a la aplicación que, mientras esperamos la respuesta de la red, puede ceder el control al bucle de eventos.

Toga te permite utilizar métodos regulares o co-rutinas asíncronas como manejadores; Toga gestiona todo entre bastidores para asegurarse de que el manejador es invocado o esperado según sea necesario.

Si guardas estos cambios y vuelves a ejecutar la aplicación (ya sea con `briefcase dev` en modo desarrollo, o actualizando y volviendo a ejecutar la aplicación empaquetada), no habrá ningún cambio obvio en la aplicación. Sin embargo, al hacer clic en el botón para activar el cuadro de diálogo, puede notar una serie de mejoras sutiles:

- El botón vuelve a un estado «no pulsado», en lugar de quedar atrapado en un estado «pulsado».
- El icono «bola de playa»/»spinner» no aparece
- Si mueves o cambias el tamaño de la ventana de la aplicación mientras esperas a que aparezca el cuadro de diálogo, la ventana volverá a dibujarse.
- Si intentas abrir el menú de una aplicación, el menú aparecerá inmediatamente.

2.9.4 Siguientes pasos

Ahora tenemos una aplicación que es ágil y sensible, incluso cuando está esperando en una API lenta. Pero, ¿cómo podemos asegurarnos de que la aplicación sigue funcionando a medida que continuamos desarrollándola? ¿Cómo probamos nuestra aplicación? Visita *Tutorial 9* para descubrirlo…

2.10 Tutorial 9 - Tiempo de Ejecución del conjunto de pruebas

La mayor parte del desarrollo de software no consiste en escribir código nuevo, sino en modificar el existente. Garantizar que el código existente sigue funcionando de la forma que esperamos es una parte clave del proceso de desarrollo de software. Una forma de asegurar el comportamiento de nuestra aplicación es con un *conjunto de pruebas*.

2.10.1 Ejecución del conjunto de pruebas

Resulta que nuestro proyecto ya tiene un conjunto de pruebas! Cuando originalmente generamos nuestro proyecto, se generaron dos directorios de nivel superior: `src` y `tests`. La carpeta `src` contiene el código de nuestra aplicación; la carpeta `tests` contiene nuestro conjunto de pruebas. Dentro de la carpeta `tests` hay un archivo llamado `test_app.py` con el siguiente contenido:

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

Esto es un Pytest *caso de prueba* - un bloque de código que puede ser ejecutado para verificar algún comportamiento de tu aplicación. En este caso, la prueba es un marcador de posición, y no verifica nada específico sobre nuestra aplicación, pero es una prueba que podemos realizar.

Podemos ejecutar este conjunto de pruebas utilizando la opción `--test` de `briefcase dev`. Como es la primera vez que ejecutamos pruebas, también necesitamos pasar la opción `-r` para asegurarnos de que también se instalan los requisitos de las pruebas:

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
```

(continúe en la próxima página)

(proviene de la página anterior)

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

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

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

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

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

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

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

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

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

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

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

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

¡Un éxito! Acabamos de ejecutar una única prueba que verifica que las matemáticas de Python funcionan como esperábamos (¡Qué alivio!).

Vamos a reemplazar esta prueba de marcador de posición con una prueba para verificar que nuestro método `greeting()` se comporta como esperamos. Sustituye el contenido de `test_app.py` por el siguiente:

```
from helloworld.app import greeting


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

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


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

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

Esto define dos nuevos casos de pruebas, verificando los dos comportamientos que esperamos ver: la salida cuando se proporciona un nombre, y la salida cuando el nombre está vacío.

Ahora podemos volver a ejecutar el conjunto de pruebas. Esta vez, no necesitamos proporcionar la opción `-r`, ya que los requisitos de la prueba ya se han instalado; sólo tenemos que utilizar la opción `--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
```

(continúe en la próxima página)

(proviene de la página anterior)

```
[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 ===============================
```

Excelente Nuestro método de utilidad `greeting()` está funcionando como se esperaba.

2.10.2 Desarrollo basado en pruebas

Ahora que tenemos un conjunto de pruebas, podemos utilizarlo para impulsar el desarrollo de nuevas funciones. Vamos a modificar nuestra aplicación para que tenga un saludo especial para un usuario en particular. Podemos empezar por añadir un caso de prueba para el nuevo comportamiento que nos gustaría ver en la parte inferior de `test_app.py`:

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

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

A continuación, ejecute el conjunto de pruebas con esta nueva prueba:

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!
```

(continúe en la próxima página)

(proviene de la página anterior)

```
E         + Hello, Brutus

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

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

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

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

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

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

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

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

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

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

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

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

=================================== FAILURES ===================================
_________________________________ test_brutus _________________________________
```

(continúe en la próxima página)

(proviene de la página anterior)

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

Esta vez, vemos un fallo en el test - y la salida explica el origen del fallo: el test está esperando la salida «¡BeeWare los IDEs de Python!», pero nuestra implementación de `greeting()` está devolviendo «Hola, Brutus». Modifiquemos la implementación de `greeting()` en `src/helloworld/app.py` para que tenga el nuevo comportamiento:

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

Si ejecutamos las pruebas de nuevo, ahora veremos que nuestras pruebas pasan:

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

(continúe en la próxima página)

(proviene de la página anterior)

```
...
collecting ... collected 3 items

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

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

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

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

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

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

2.10.3 Pruebas en tiempo real

Hasta ahora, hemos estado ejecutando las pruebas en modo de desarrollo. Esto es especialmente útil cuando estás desarrollando nuevas características, ya que puedes iterar rápidamente en la adición de pruebas, y la adición de código para hacer que esas pruebas pasen. Sin embargo, en algún momento, usted querrá verificar que su código también se ejecuta correctamente cuando dentro del entorno de aplicación paquete.

Las opciones `--test` y `-r` también se pueden pasar al comando `run`. Si utilizas `briefcase run --test -r`, se ejecutará el mismo conjunto de pruebas, pero se ejecutará dentro del paquete de aplicaciones empaquetado en lugar de en tu entorno de desarrollo:

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

(continúe en la próxima página)

(proviene de la página anterior)

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

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

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

[helloworld] Test suite passed!
```

```
(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...
================================================================================
```

(continúe en la próxima página)

(proviene de la página anterior)

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

```
(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 ===============================
```

Al igual que con `briefcase dev --test`, la opción `-r` sólo es necesaria la primera vez que se ejecuta el conjunto de pruebas para asegurarse de que las dependencias de prueba están presentes. En las siguientes ejecuciones, puede omitir esta opción.

También puedes utilizar la opción `--test` en backends móviles: - así `briefcase run iOS --test` y `briefcase run android --test` funcionarán, ejecutando el conjunto de pruebas en el dispositivo móvil que selecciones.

2.10.4 Siguientes pasos

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 Tutorial 10 – Crea tu propia aplicación

Hasta ahora, nuestra aplicación ha utilizado un ícono predeterminado de «abeja gris». ¿Cómo actualizamos la aplicación para usar nuestro propio ícono?

2.11.1 Añadir un icono

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...
```

(continúe en la próxima página)

(proviene de la página anterior)

```
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␣
```

(continúe en la próxima página)

(proviene de la página anterior)

```
↪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
```

(continúe en la próxima página)

(proviene de la página anterior)

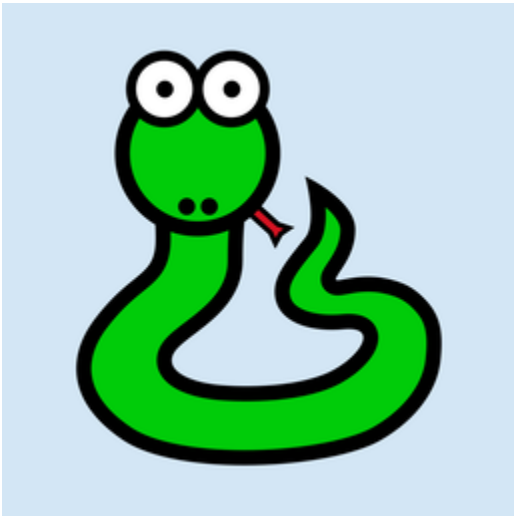
```
  [helloworld] Removing unneeded app content...
  Removing unneeded app bundle content... done

  [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.
```

(continúe en la próxima página)

(proviene de la página anterior)

```
[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...
```

(continúe en la próxima página)

(proviene de la página anterior)

```
[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 Siguientes pasos

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.