BeeWare Documentation

Release 0.1.dev155+g6381907

Russell Keith-Magee

13.05.2024

Inhaltsverzeichnis

Einmal schreiben. Überall ausführen.

Willkommen bei BeeWare! In diesem Tutorial werden wir eine grafische Benutzeroberfläche mit Python erstellen und sie als Desktop-, Mobil- und Web-Anwendung bereitstellen. Außerdem sehen wir uns an, wie Sie die BeeWare-Tools nutzen können, um übliche Aufgaben als App-Entwickler zu erledigen, wie zum Beispiel das Testen Ihrer App.

Dies ist eine maschinelle Übersetzung!

Diese Version des Tutorials wurde durch maschinelle Übersetzung erstellt. Wir wissen, dass dies nicht ideal ist, aber wir waren der Meinung, dass eine schlechte Übersetzung besser ist als gar keine Übersetzung.

Wenn du helfen möchtest, die Übersetzung zu verbessern, melde dich! Wir haben einen `#translations`-Kanal in Discord; stellen Sie sich dort vor und wir werden Sie dem Übersetzungsteam hinzufügen.

KAPITEL 1

Was ist BeeWare?

BeeWare ist kein einzelnes Produkt, Werkzeug oder eine Bibliothek - es ist eine Sammlung von Werkzeugen und Bibliotheken, die alle zusammenarbeiten, um Ihnen zu helfen, plattformübergreifende Python-Anwendungen mit einer nativen GUI zu schreiben. Es umfasst:

- Toga, ein plattformübergreifendes Widget-Toolkit;
- Briefcase, ein Werkzeug zum Verpacken von Python-Projekten als verteilbare Artefakte, die an Endbenutzer ausgeliefert werden können;
- Bibliotheken (wie Rubicon ObjC) für den Zugriff auf plattformspezifische Bibliotheken;
- Vorkompilierte Python-Builds, die auf Plattformen verwendet werden können, für die keine offiziellen Python-Installationsprogramme verfügbar sind.

In diesem Tutorium werden wir alle diese Tools verwenden, aber als Benutzer müssen Sie nur mit den ersten beiden (Toga und Briefcase) interagieren. Jedes der Tools kann jedoch auch einzeln verwendet werden - zum Beispiel können Sie Briefcase verwenden, um eine App bereitzustellen, ohne Toga als GUI-Toolkit zu verwenden.

Die BeeWare-Suite ist für macOS, Windows, Linux (unter Verwendung von GTK), für mobile Plattformen wie Android und iOS und für das Web verfügbar.

KAPITEL 2

Los geht's!

Sind Sie bereit, BeeWare selbst auszuprobieren? *Lassen Sie uns eine plattformübergreifende Anwendung in Python erstellen!*

2.1 Tutorial 0 - Erstinstallation

Bevor wir unsere erste BeeWare-App erstellen, müssen wir sicherstellen, dass wir alle Voraussetzungen für den Betrieb von BeeWare erfüllen.

2.1.1 Python installieren

Als erstes brauchen wir einen funktionierenden Python-Interpreter.

macOS

Linux

Windows

Wenn Sie mit macOS arbeiten, ist eine aktuelle Version von Python in Xcode oder den Kommandozeilen-Entwicklerwerkzeugen enthalten. Um zu prüfen, ob Sie Python bereits haben, führen Sie den folgenden Befehl aus:

```
$ python3 --version
```

Wenn Python installiert ist, wird die Versionsnummer angezeigt. Andernfalls werden Sie aufgefordert, die Entwicklerwerkzeuge für die Kommandozeile zu installieren.

Wenn Sie unter Windows arbeiten, können Sie das offizielle Installationsprogramm von der Python-Website herunterladen. Sie können jede stabile Version von Python ab 3.8 verwenden. Wir raten dazu, Alphas, Betas und Release Candidates zu meiden, es sei denn, Sie wissen *wirklich*, was Sie tun.

Unter Linux installieren Sie Python mit dem Paketmanager des Systems (`apt` unter Debian/Ubuntu/Mint, `dnf` unter Fedora, oder `pacman` unter Arch).

Sie sollten sich vergewissern, dass das System Python 3.8 oder eine neuere Version verwendet. Ist dies nicht der Fall (z. B. wird Ubuntu 18.04 mit Python 3.6 ausgeliefert), müssen Sie Ihre Linux-Distribution auf eine neuere Version aktualisieren.

Raspberry Pi wird derzeit nur begrenzt unterstützt.

Wenn Sie unter Windows arbeiten, können Sie das offizielle Installationsprogramm von der Python-Website herunterladen. Sie können jede stabile Version von Python ab 3.8 verwenden. Wir raten dazu, Alphas, Betas und Release Candidates zu meiden, es sei denn, Sie wissen *wirklich*, was Sie tun.

Alternative Python-Distributionen

Es gibt viele verschiedene Möglichkeiten, Python zu installieren. Sie können Python über homebrew installieren. Sie können pyenv verwenden, um mehrere Python-Installationen auf demselben Rechner zu verwalten. Windows-Benutzer können Python aus dem Windows App Store installieren. Für Benutzer mit einem datenwissenschaftlichen Hintergrund könnten auch Anaconda oder Miniconda interessant sein.

Unter macOS oder Windows ist es egal, *wie* Sie Python installiert haben - wichtig ist nur, dass Sie `python3` über die Eingabeaufforderung/Terminalanwendung Ihres Betriebssystems ausführen können und einen funktionierenden Python-Interpreter erhalten.

Wenn Sie mit Linux arbeiten, sollten Sie das von Ihrem Betriebssystem bereitgestellte Python verwenden. Sie können den *größten Teil* dieses Tutorials mit einem systemfremden Python durchführen, aber Sie werden nicht in der Lage sein, Ihre Anwendung für die Weitergabe an andere zu verpacken.

2.1.2 Abhängigkeiten installieren

Als Nächstes installieren Sie die zusätzlichen Abhängigkeiten, die für Ihr Betriebssystem erforderlich sind:

macOS

Linux

Windows

Die Erstellung von BeeWare-Anwendungen unter macOS erfordert:

- **Git**, ein Versionskontrollsystem. Es ist in Xcode oder den Kommandozeilen-Entwicklungswerkzeugen enthalten, die Sie oben installiert haben.

Um die lokale Entwicklung zu unterstützen, müssen Sie einige Systempakete installieren. Die Liste der erforderlichen Pakete hängt von Ihrer Distribution ab:

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
```

Die Erstellung von BeeWare-Anwendungen unter Windows erfordert:

- **Git**, ein Versionskontrollsystem. Sie können Git von git-scm.org herunterladen.

Nach der Installation dieser Tools sollten Sie sicherstellen, dass Sie alle Terminalsitzungen neu starten. Windows zeigt neu installierte Tools nur auf Terminals an, die *nach* Abschluss der Installation gestartet wurden.

2.1.3 Einrichten einer virtuellen Umgebung

Wir werden nun eine virtuelle Umgebung erstellen - eine „Sandbox“, die wir verwenden können, um unsere Arbeit an diesem Tutorial von unserer Haupt-Python-Installation zu isolieren. Wenn wir Pakete in die virtuelle Umgebung installieren, wird unsere Haupt-Python-Installation (und alle anderen Python-Projekte auf unserem Computer) davon nicht betroffen sein. Wenn wir unsere virtuelle Umgebung komplett durcheinander bringen, können wir sie einfach löschen und neu beginnen, ohne dass andere Python-Projekte auf unserem Computer betroffen sind und ohne dass wir Python neu installieren müssen.

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
```

Fehler bei der Ausführung von PowerShell-Skripten

Wenn Sie PowerShell verwenden und die folgende Fehlermeldung erhalten:

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

Ihr Windows-Konto hat keine Berechtigung zum Ausführen von Skripten. So beheben Sie dies:

1. Öffnen Sie Windows PowerShell als Administrator.
2. Führen Sie `set-executionpolicy RemoteSigned` aus
3. Wählen Sie `Y`, um die Ausführungspolitik zu ändern.

Danach können Sie `beeware-venv\Scripts\activate.ps1` in Ihrer ursprünglichen PowerShell-Sitzung (oder in einer neuen Sitzung im selben Verzeichnis) erneut ausführen.

Wenn dies geklappt hat, sollte Ihre Eingabeaufforderung nun geändert sein - sie sollte ein `(beeware-venv)`-Präfix haben. Dies zeigt Ihnen, dass Sie sich gerade in Ihrer virtuellen BeeWare-Umgebung befinden. Wann immer Sie an diesem Tutorial arbeiten, sollten Sie sicherstellen, dass Ihre virtuelle Umgebung aktiviert ist. Ist dies nicht der Fall, führen Sie den letzten Befehl (den Befehl `activate`) erneut aus, um Ihre Umgebung wieder zu aktivieren.

Alternative virtuelle Umgebungen

Wenn Sie Anaconda oder miniconda benutzen, sind Sie vielleicht mehr mit der Verwendung von conda-Umgebungen vertraut. Vielleicht haben Sie auch schon von `virtualenv` gehört, einem Vorgänger von Pythons eingebautem `venv` Modul. Wie bei Python-Installationen ist es unter macOS oder Windows egal, *wie* Sie Ihre virtuelle Umgebung erstellen, solange Sie eine haben. Wenn Sie unter Linux arbeiten, sollten Sie sich an `venv` und das System-Python halten.

2.1.4 Nächste Schritte

Wir haben jetzt unsere Umgebung eingerichtet. Wir sind bereit, *unsere erste BeeWare-Anwendung zu erstellen*.

2.2 Tutorial 1 - Ihre erste App

Wir sind bereit, unsere erste Anwendung zu erstellen.

2.2.1 Installieren Sie die BeeWare-Tools

Zunächst müssen wir **Briefcase** installieren. Briefcase ist ein BeeWare-Werkzeug, das verwendet werden kann, um Ihre Anwendung für die Verteilung an Endbenutzer zu verpacken - aber es kann auch verwendet werden, um ein neues Projekt zu starten. Vergewissern Sie sich, dass Sie sich in dem Verzeichnis `beeware-tutorial` befinden, das Sie in *Tutorial 0* erstellt haben, und dass die virtuelle Umgebung `beeware-venv` aktiviert ist, und starten Sie:

macOS

Linux

Windows

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

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

Mögliche Fehler bei der Installation

Wenn Sie während der Installation Fehler sehen, liegt das höchstwahrscheinlich daran, dass einige der Systemvoraussetzungen nicht installiert wurden. Stellen Sie sicher, dass Sie *alle Plattformvoraussetzungen* installiert haben.

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

Mögliche Fehler bei der Installation

Es ist wichtig, dass Sie `python -m pip` und nicht `pip` verwenden. Briefcase muss sicherstellen, dass es eine aktuelle Version von `pip` und `setuptools` hat, und ein bloßer Aufruf von `pip` kann sich nicht selbst aktualisieren. Wenn Sie mehr darüber wissen wollen, hat Brett Cannon einen detaillierten Blogbeitrag zu diesem Problem.

Eines der BeeWare-Tools ist **Briefcase**. Briefcase kann verwendet werden, um Ihre Anwendung für die Verteilung an Endbenutzer zu verpacken - es kann aber auch zur Erstellung eines neuen Projekts verwendet werden.

2.2.2 Erstellen Sie ein neues Projekt

Beginnen wir mit unserem ersten BeeWare-Projekt! Wir werden den Briefcase-Befehl `new` verwenden, um eine Anwendung namens **Hello World** zu erstellen. Führen Sie den folgenden Befehl in Ihrer Eingabeaufforderung aus:

macOS

Linux

Windows

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

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

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

Briefcase wird uns nach einigen Details zu unserer neuen Anwendung fragen. Verwenden Sie für dieses Tutorial die folgenden Angaben:

- **Formaler Name** - Akzeptieren Sie den Standardwert: `Hello World`.
- **App Name** - Akzeptieren Sie den Standardwert: `helloworld`.
- **Bundle** - Wenn Sie eine eigene Domain besitzen, geben Sie diese in umgekehrter Reihenfolge an. (Wenn Sie zum Beispiel die Domain „cupcakes.com“ besitzen, geben Sie `com.cupcakes` als Bundle ein). Wenn Sie keine eigene Domain besitzen, akzeptieren Sie das Standard-Bundle (`com.example`).
- **Projektname** - Übernehmen Sie den Standardwert: `Hello World`.
- **Description** - Übernehmen Sie den Standardwert (wenn Sie wirklich kreativ sein wollen, denken Sie sich Ihre eigene Beschreibung aus!)
- **Author** - Geben Sie hier Ihren eigenen Namen ein.
- **Author's email** - Geben Sie Ihre eigene E-Mail-Adresse ein. Diese wird in der Konfigurationsdatei, im Hilfetext und überall dort verwendet, wo bei der Einreichung der App bei einem App-Store eine E-Mail erforderlich ist.
- **URL** - Die URL der Landing Page für Ihre Anwendung. Auch hier gilt: Wenn Sie Ihre eigene Domain besitzen, geben Sie eine URL auf dieser Domain ein (einschließlich `https://`). Ansonsten akzeptieren Sie einfach die Standard-URL (`https://example.com/helloworld`). Diese URL muss (vorerst) nicht tatsächlich existieren; sie wird nur verwendet, wenn Sie Ihre Anwendung in einem App Store veröffentlichen.
- **Lizenz** - Akzeptieren Sie die Standardlizenz (BSD). Dies hat jedoch keinen Einfluss auf die Funktionsweise des Tutorials - wenn Sie also besonders starke Gefühle bezüglich der Lizenzwahl haben, können Sie gerne eine andere Lizenz wählen.
- **GUI-Framework** - Akzeptieren Sie die Standardoption Toga (BeeWare's eigenes GUI-Toolkit).

Briefcase generiert dann ein Projektgerüst, das Sie verwenden können. Wenn Sie diesem Tutorial bis hierher gefolgt sind und die Vorgaben wie beschrieben akzeptiert haben, sollte Ihr Dateisystem etwa so aussehen:

```
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
```

Dieses Gerüst ist bereits eine voll funktionsfähige Anwendung, ohne etwas anderes hinzufügen zu müssen. Der Ordner `rc` enthält den gesamten Code für die Anwendung, der Ordner `tests` enthält eine erste Testreihe und die Datei `pyproject.toml` beschreibt, wie die Anwendung für die Verteilung verpackt werden soll. Wenn Sie die Datei `pyproject.toml` in einem Editor öffnen, sehen Sie die Konfigurationsdetails, die Sie Briefcase gerade mitgeteilt haben.

Nun, da wir eine Rumpfanwendung haben, können wir Briefcase verwenden, um die Anwendung auszuführen.

2.2.3 Führen Sie die Anwendung im Entwicklermodus aus

Wechseln Sie in das `helloworld` Projektverzeichnis und sagen Sie briefcase, daß es das Projekt im Developer (oder `dev`) Modus starten soll:

macOS

Linux

Windows

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

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

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

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

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

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

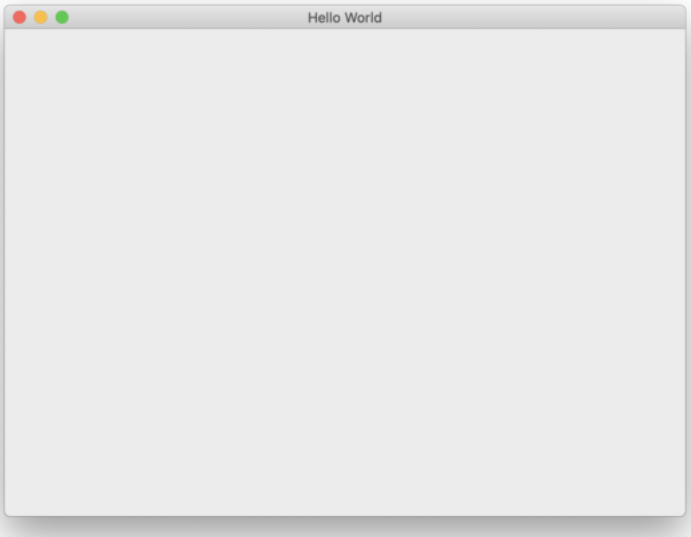
```
(beeware-venv) C:\...>cd helloworld
(beeware-venv) C:\...>briefcase dev

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

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

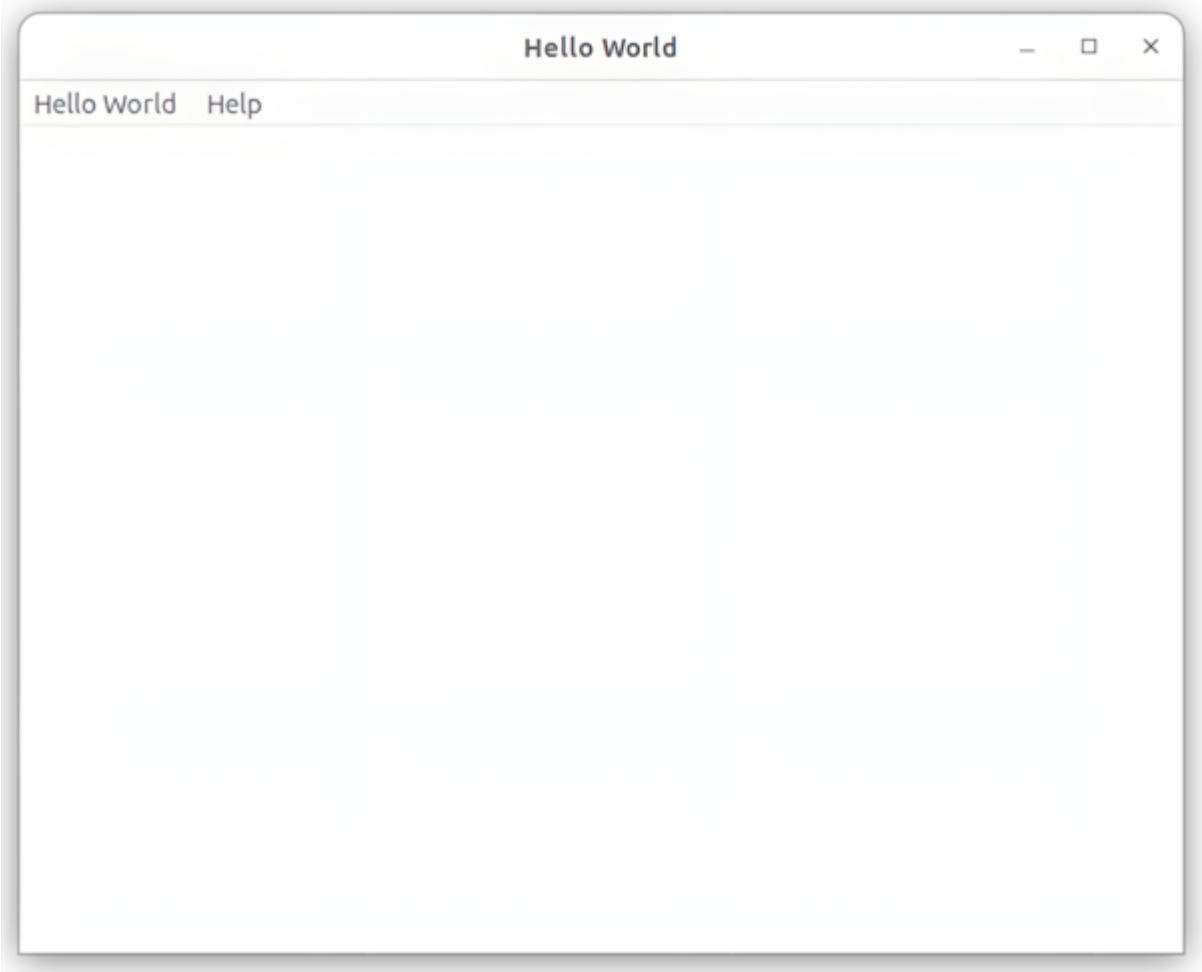
Dies sollte ein GUI-Fenster öffnen:

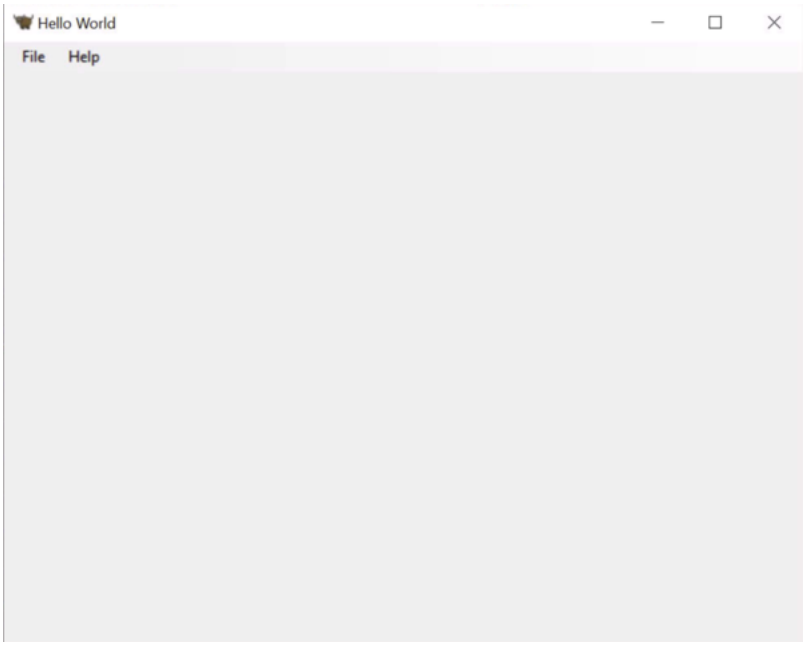
macOS

Linux

Windows







Drücken Sie die Schaltfläche „Schließen“ (oder wählen Sie „Beenden“ aus dem Menü der Anwendung), und Sie sind fertig! Herzlichen Glückwunsch - Sie haben gerade eine eigenständige, native Anwendung in Python geschrieben!

2.2.4 Nächste Schritte

Wir haben jetzt eine funktionierende Anwendung, die im Entwicklermodus läuft. Jetzt können wir unsere eigene Logik hinzufügen, damit unsere Anwendung etwas Interessanteres tut. In *Tutorial 2* werden wir unserer Anwendung eine nützlichere Benutzeroberfläche geben.

2.3 Tutorial 2 - Es wird interessant

In *Tutorial 1* haben wir ein lauffähiges Rumpfprojekt erstellt, aber eigenen Code haben wir nicht geschrieben. Schauen wir uns an, was für uns generiert wurde.

2.3.1 Was generiert wurde

In dem Verzeichnis `src/helloworld` sollten 3 Dateien zu sehen sein: `__init__.py`, `__main__.py` und `app.py`.

`__init__.py` markiert das `helloworld`-Verzeichnis als importierbares Python-Modul. Es ist eine leere Datei; allein die Tatsache, dass sie existiert, sagt dem Python-Interpreter, dass das `helloworld`-Verzeichnis ein Modul definiert.

`__main__.py` markiert das `helloworld` Modul als eine besondere Art von Modul - ein ausführbares Modul. Wenn Sie versuchen, das Modul `helloworld` mit `python -m helloworld` zu starten, ist die Datei `__main__.py` der Ort, an dem Python mit der Ausführung beginnt. Der Inhalt von `__main__.py` ist relativ einfach:

```
from helloworld.app import main

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

Das heißt, es importiert die Methode `main` aus der Anwendung `helloworld`; und wenn es als Einstiegspunkt ausgeführt wird, ruft es die Methode main() auf und startet die Hauptschleife der Anwendung. Die Hauptschleife ist die Art und Weise, wie eine GUI-Anwendung auf Benutzereingaben (wie Mausklicks und Tastatureingaben) wartet.

Die interessantere Datei ist `app.py` - diese enthält die Logik, die unser Anwendungsfenster erstellt:

```
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()
```

Gehen wir diese Zeile für Zeile durch:

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

Als erstes importieren wir das `toga`-Widget-Toolkit, sowie einige stilbezogene Utility-Klassen und Konstanten. Unser Code verwendet diese noch nicht - aber wir werden sie in Kürze nutzen.

Definieren wir eine Klasse:

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

Jede Toga-Anwendung hat eine einzige `toga.App`-Instanz, die laufende Entität, die Anwendung, darstellt. Die Anwendung kann mehrere Fenster verwalten, aber für einfache Anwendungen gibt es ein einziges Hauptfenster.

Als nächstes definieren wir eine `Startup()` Methode:

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

Das erste, was die Startup-Methode macht, ist die Definition eines Hauptfeldes. Das Layout-Schema von Toga verhält sich ähnlich wie HTML. Sie bauen eine Anwendung auf, indem Sie eine Sammlung von Boxen konstruieren, von denen jede andere Boxen oder Widgets enthält. Sie wenden dann Stile auf diese Boxen an, um zu definieren, wie sie den verfügbaren Platz im Fenster nutzen werden.

In dieser Anwendung definieren wir ein einzelnes Feld, in das wir jedoch nichts einfügen.

Als Nächstes definieren wir ein Fenster, in das wir dieses leere Feld einfügen können:

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

Dies erzeugt eine Instanz eines `toga.MainWindow`, das einen Titel hat, der dem Namen der Anwendung entspricht. Ein Hauptfenster ist eine besondere Art von Fenster in Toga - es ist ein Fenster, das eng an den Lebenszyklus der Anwendung gebunden ist. Wenn das Hauptfenster geschlossen wird, wird die Anwendung beendet. Das Hauptfenster ist auch das Fenster, in dem sich das Menü der Anwendung befindet (wenn Sie auf einer Plattform wie Windows arbeiten, wo die Menüleisten Teil des Fensters sind)

Dann fügen wir unsere leere Box als Inhalt des Hauptfensters hinzu und weisen die Anwendung an, unser Fenster anzuzeigen:

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

Als letztes definieren wir eine `main()` Methode. Diese erzeugt die Instanz unserer Anwendung:

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

Diese `main()` Methode ist diejenige, die von `__main__.py` importiert und aufgerufen wird. Sie erzeugt eine Instanz unserer `HelloWorld`-Anwendung und gibt diese zurück.

Das ist die einfachste mögliche Toga-Anwendung. Lassen Sie uns einige unserer eigenen Inhalte in die Anwendung einfügen und die Anwendung etwas Interessantes tun lassen.

2.3.2 Hinzufügen von eigenen Inhalten

Ändern Sie Ihre `HelloWorld` Klasse in `src/helloworld/app.py` so, dass sie wie folgt aussieht:

```
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}")
```

Bemerkung: Entfernen Sie nicht die Importe am Anfang der Datei oder `main()` am Ende. Sie müssen nur die Klasse `HelloWorld` aktualisieren.

Schauen wir uns im Detail an, was sich geändert hat.

Wir erstellen immer noch ein Hauptfeld, aber wir wenden jetzt einen Stil an:

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

Das in Toga eingebaute Layoutsystem heißt „Pack“. Es verhält sich sehr ähnlich wie CSS. Sie definieren Objekte in einer Hierarchie - in HTML sind die Objekte , und andere DOM-Elemente; in Toga sind es Widgets und Boxen. Sie können dann den einzelnen Elementen Stile zuweisen. In diesem Fall geben wir an, dass es sich um einen `COLUMN`-Kasten handelt - das heißt, es handelt sich um einen Kasten, der die gesamte verfügbare Breite beansprucht und sich in der Höhe vergrößert, wenn Inhalt hinzugefügt wird, aber er versucht, so kurz wie möglich zu sein.

Als nächstes definieren wir ein paar Widgets:

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

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

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

Hier definieren wir ein Label und einen TextInput. Beiden Widgets sind Stile zugeordnet; das Label hat links und rechts ein Padding von 5px und oben und unten kein Padding. Der TextInput ist als flexibel gekennzeichnet, d. h. er nimmt den gesamten verfügbaren Platz in seiner Layoutachse ein.

Der TextInput wird als Instanzvariable der Klasse zugewiesen. Dies ermöglicht uns einen einfachen Zugriff auf die Instanz des Widgets - etwas, das wir gleich verwenden werden.

Als nächstes definieren wir eine Box, die diese beiden Widgets enthält:

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

Die `name_box` ist eine Box genau wie die Hauptbox, aber dieses Mal ist es eine `ROW`-Box. Das bedeutet, daß der Inhalt horizontal eingefügt wird, und daß versucht wird, die Breite so schmal wie möglich zu machen. Die Box hat auch etwas Padding - 5px auf allen Seiten.

Jetzt definieren wir eine Schaltfläche:

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

Die Schaltfläche hat auch 5px Polsterung auf allen Seiten. Wir definieren auch einen *Handler* - eine Methode, die aufgerufen wird, wenn die Schaltfläche gedrückt wird.

Dann fügen wir das Namensfeld und die Schaltfläche zum Hauptfeld hinzu:

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

Damit ist unser Layout fertiggestellt; der Rest der Startup-Methode ist wie zuvor - Definition eines MainWindow und Zuweisung des Hauptfeldes als Inhalt des Fensters:

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

Als Letztes müssen wir den Handler für die Schaltfläche definieren. Ein Handler kann eine beliebige Methode, ein Generator oder eine asynchrone Co-Routine sein; er akzeptiert das Widget, das Ereignis erzeugt hat, als Argument und wird immer dann aufgerufen, wenn die Schaltfläche gedrückt wird:

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

Der Hauptteil der Methode ist eine einfache Druckanweisung - sie fragt jedoch den aktuellen Wert der Namenseingabe ab und verwendet diesen Inhalt als den Text, der gedruckt wird.

Nachdem wir nun diese Änderungen vorgenommen haben, können wir sehen, wie sie aussehen, indem wir die Anwendung erneut starten. Wie zuvor werden wir den Entwicklermodus verwenden:

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

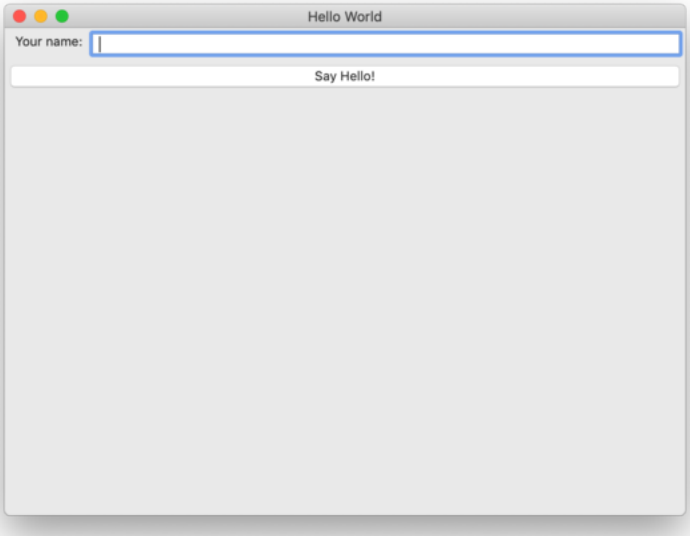
Sie werden feststellen, dass dieses Mal *keine* Abhängigkeiten installiert werden. Briefcase kann erkennen, dass die Anwendung schon einmal ausgeführt wurde, und um Zeit zu sparen, wird nur die Anwendung ausgeführt. Wenn Sie neue Abhängigkeiten zu Ihrer Anwendung hinzufügen, können Sie sicherstellen, dass diese installiert werden, indem Sie die Option `-r` beim Aufruf von `briefcase dev` mitgeben.

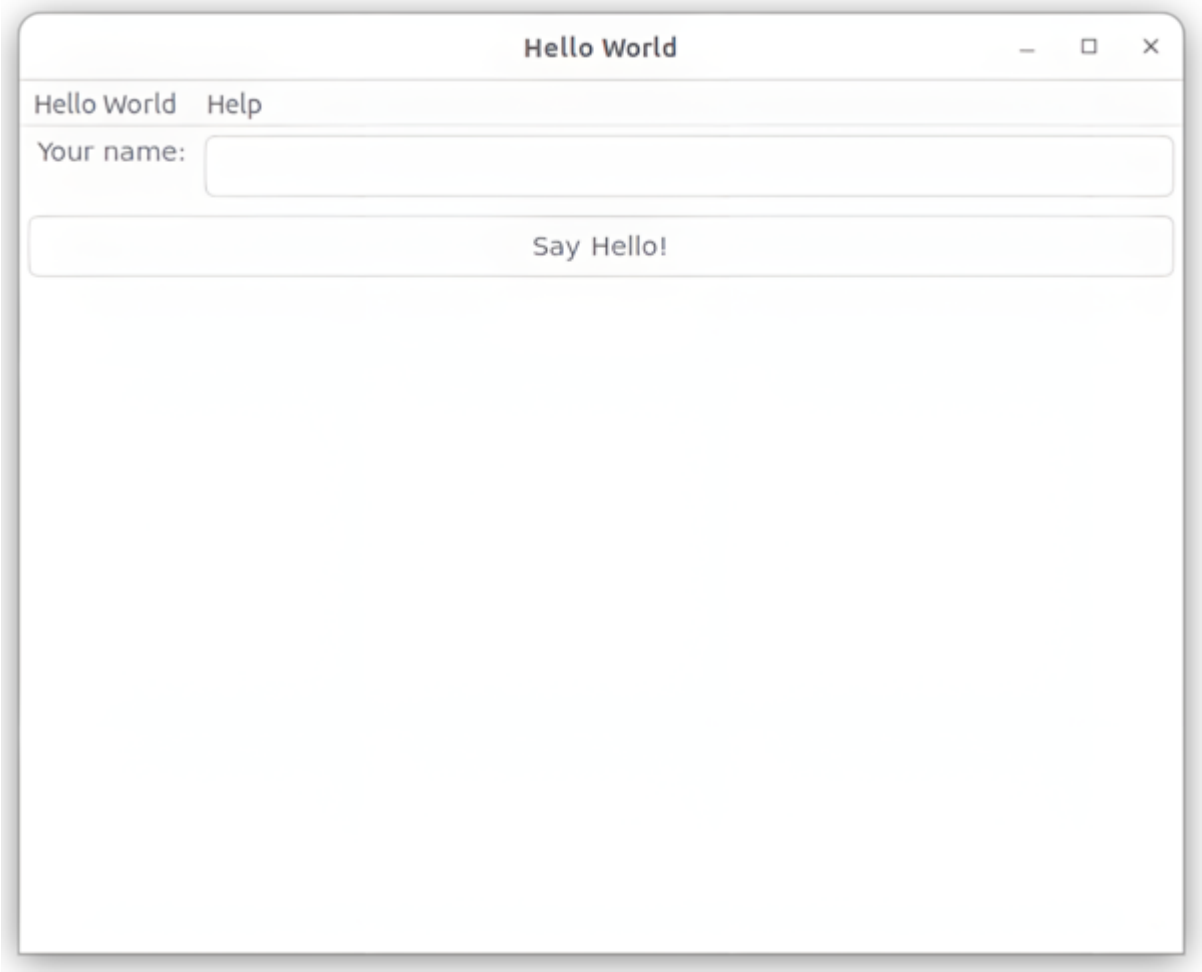
Dies sollte ein GUI-Fenster öffnen:

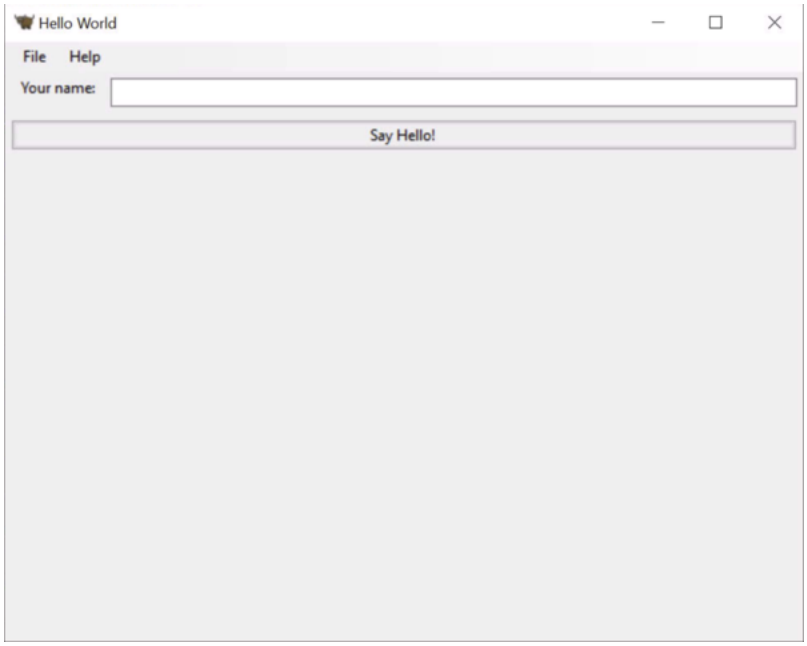
macOS

Linux

Windows







Wenn Sie einen Namen in das Textfeld eingeben und die Schaltfläche GUI drücken, sollte die Ausgabe in der Konsole erscheinen, in der Sie die Anwendung gestartet haben.

2.3.3 Nächste Schritte

Wir haben jetzt eine Anwendung, die etwas Interessanteres macht. Aber sie läuft nur auf unserem eigenen Computer. Lassen Sie uns diese Anwendung für die Verteilung verpacken. In *Tutorial 3* werden wir unsere Anwendung als eigenständiges Installationsprogramm verpacken, das wir an einen Freund oder Kunden schicken oder in einen App Store hochladen können.

2.4 Tutorial 3 - Verpackung für den Vertrieb

Bislang haben wir unsere Anwendung im „Entwicklermodus“ ausgeführt. Das macht es uns leicht, unsere Anwendung lokal auszuführen - aber was wir wirklich wollen, ist, dass wir unsere Anwendung an andere weitergeben können.

Wir möchten unseren Benutzern jedoch nicht beibringen müssen, wie man Python installiert, eine virtuelle Umgebung erstellt, ein Git-Repository klont und Briefcase im Entwicklermodus startet. Wir würden ihnen lieber einfach ein Installationsprogramm geben und die Anwendung einfach funktionieren lassen.

Briefcase kann verwendet werden, um Ihre Anwendung für die Verteilung auf diese Weise zu verpacken.

2.4.1 Erstellen Ihres Anwendungsgerüsts

Da dies das erste Mal ist, dass wir unsere Anwendung paketieren, müssen wir einige Konfigurationsdateien und andere Gerüste erstellen, um den Paketierungsprozess zu unterstützen. Starten Sie im Verzeichnis `helloworld`:

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
```

Wahrscheinlich haben Sie gerade gesehen, wie seitenweise Inhalte an Ihrem Terminal vorbeigezogen sind… was ist

also passiert? Briefcase hat das Folgende getan:

1. Es **generiert eine Anwendungsvorlage**. Um ein natives Installationsprogramm zu erstellen, sind eine Menge Dateien und Konfigurationen erforderlich, die über den Code der eigentlichen Anwendung hinausgehen. Dieses zusätzliche Gerüst ist für jede Anwendung auf der gleichen Plattform fast gleich, mit Ausnahme des Namens der eigentlichen Anwendung, die erstellt wird - daher bietet Briefcase eine Anwendungsvorlage für jede unterstützte Plattform. In diesem Schritt wird die Vorlage ausgerollt, wobei der Name Ihrer Anwendung, die Bundle-ID und andere Eigenschaften Ihrer Konfigurationsdatei entsprechend der Plattform, auf der Sie bauen, ersetzt werden.

   Wenn Sie mit der von Briefcase bereitgestellten Vorlage nicht zufrieden sind, können Sie Ihre eigene Vorlage erstellen. Sie sollten dies jedoch erst dann tun, wenn Sie etwas mehr Erfahrung mit der Standardvorlage von Briefcase haben.

2. Es **lud ein Support-Paket herunter und installierte es**. Der Paketierungsansatz von Briefcase lässt sich am besten als „das Einfachste, was möglich ist“ beschreiben - es liefert einen vollständigen, isolierten Python-Interpreter als Teil jeder Anwendung, die es erstellt. Dies ist etwas ineffizient - wenn Sie 5 Anwendungen mit Briefcase verpackt haben, haben Sie 5 Kopien des Python-Interpreters. Allerdings garantiert dieser Ansatz, dass jede Anwendung völlig unabhängig ist und eine bestimmte Python-Version verwendet, von der bekannt ist, dass sie mit der Anwendung funktioniert.

   Auch hier bietet Briefcase ein Standard-Unterstützungspaket für jede Plattform; wenn Sie möchten, können Sie Ihr eigenes Unterstützungspaket bereitstellen und dieses Paket als Teil des Erstellungsprozesses einbinden lassen. Dies kann sinnvoll sein, wenn Sie bestimmte Optionen im Python-Interpreter aktiviert haben wollen oder wenn Sie Module aus der Standardbibliothek entfernen wollen, die Sie zur Laufzeit nicht benötigen.

   Briefcase verwaltet einen lokalen Zwischenspeicher für Support-Pakete. Wenn Sie also ein bestimmtes Support-Paket heruntergeladen haben, wird diese zwischengespeicherte Kopie bei zukünftigen Builds verwendet.

3. Es **installiert Anwendungsanforderungen**. Ihre Anwendung kann alle Module von Drittanbietern angeben, die zur Laufzeit benötigt werden. Diese werden mittels `pip` in das Installationsprogramm Ihrer Anwendung installiert.

4. Es **Installiert Ihren Anwendungscode**. Ihre Anwendung hat ihren eigenen Code und Ressourcen (z. B. Bilder, die zur Laufzeit benötigt werden); diese Dateien werden in das Installationsprogramm kopiert.

5. Er **installiert die von Ihrer Anwendung benötigten Ressourcen**. Schließlich fügt er alle zusätzlichen Ressourcen hinzu, die das Installationsprogramm selbst benötigt. Dazu gehören Dinge wie Symbole, die an die endgültige Anwendung angehängt werden müssen, und Splash-Screen-Bilder.

Sobald dies abgeschlossen ist, sollten Sie im Projektverzeichnis ein Verzeichnis sehen, das Ihrer Plattform entspricht (`macOS`, `linux` oder `windows`) und zusätzliche Dateien enthält. Dies ist die plattformspezifische Paketierungskonfiguration für Ihre Anwendung.

2.4.2 Erstellung Ihrer Anwendung

Sie können nun Ihre Anwendung kompilieren. In diesem Schritt wird die Binärkompilierung durchgeführt, die erforderlich ist, damit Ihre Anwendung auf Ihrer Zielplattform ausgeführt werden kann.

macOS

Linux

Windows

```
(beeware-venv) $ briefcase build

[helloworld] Adhoc signing app...
...
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

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

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

Unter macOS muss der Befehl `build` nichts *kompilieren*, aber er muss den Inhalt der Binärdatei signieren, damit sie ausgeführt werden kann. Diese Signatur ist eine *ad hoc* Signatur - sie funktioniert nur auf *Ihrem* Rechner; wenn Sie die Anwendung an andere weitergeben wollen, müssen Sie eine vollständige Signatur bereitstellen.

```
(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
```

Sobald dieser Schritt abgeschlossen ist, wird der `build`-Ordner einen `helloworld-0.0.1`-Ordner enthalten, der einen Spiegel des Linux `/usr`-Dateisystems enthält. Dieser Spiegel des Dateisystems enthält einen `bin`-Ordner mit einer `helloworld`-Binärdatei, sowie die `lib`- und `share`-Ordner, die zur Unterstützung der Binärdatei benötigt werden.

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

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

Unter Windows muss der `build`-Befehl nichts *kompilieren*, aber er muss einige Metadaten schreiben, damit die Anwendung ihren Namen, ihre Version und so weiter kennt.

Auslösen von Antivirus

Da diese Metadaten direkt in die vorkompilierte Binärdatei geschrieben werden, die während des `create`-Befehls aus der Vorlage ausgerollt wird, kann dies Antivirensoftware auf Ihrem Rechner auslösen und verhindern, dass die

Metadaten geschrieben werden. In diesem Fall weisen Sie das Antivirusprogramm an, die Ausführung des Tools (mit dem Namen `rcedit-x64.exe`) zu erlauben und führen Sie den obigen Befehl erneut aus.

2.4.3 Ausführen Ihrer Anwendung

Sie können nun Briefcase verwenden, um Ihre Anwendung auszuführen:

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:
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
- /usr/lib/python3.10
- /usr/lib/python3.10/lib-dynload
- /home/brutus/beeware-tutorial/helloworld/build/helloworld/linux/ubuntu/jammy/
↪helloworld-0.0.1/usr/lib/helloworld/app
- /home/brutus/beeware-tutorial/helloworld/build/helloworld/linux/ubuntu/jammy/
↪helloworld-0.0.1/usr/lib/helloworld/app_packages
Configure argc/argv...
Initializing Python runtime...
Running app module: helloworld
--------------------------------------------------------------------------------
```

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

[helloworld] Starting app...

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

Dies startet Ihre native Anwendung unter Verwendung der Ausgabe des `build` Befehls.

Möglicherweise bemerken Sie einige kleine Unterschiede im Aussehen Ihrer Anwendung, wenn sie ausgeführt wird. Beispielsweise können die vom Betriebssystem angezeigten Symbole und der Name etwas anders aussehen als bei der Ausführung im Entwicklermodus. Das liegt auch daran, dass Sie die gepackte Anwendung verwenden und nicht nur den Python-Code ausführen. Aus der Sicht des Betriebssystems führen Sie nun „eine Anwendung“ und nicht „ein Python-Programm“ aus, was sich in der Darstellung der Anwendung widerspiegelt.

2.4.4 Erstellung Ihres Installationsprogramms

Sie können nun Ihre Anwendung mit dem Befehl `package` zur Verteilung verpacken. Der Befehl `package` führt alle Kompilierungen durch, die notwendig sind, um das gerüstete Projekt in ein endgültiges, verteilbares Produkt zu verwandeln. Abhängig von der Plattform kann dies die Kompilierung eines Installationsprogramms, die Durchführung einer Codesignierung oder andere Aufgaben vor der Verteilung beinhalten.

macOS

Linux

Windows

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

[helloworld] Signing app...
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

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

Der Ordner `dist` enthält eine Datei namens `Hello World-0.0.1.dmg`. Wenn Sie diese Datei im Finder finden und auf das Symbol doppelklicken, wird die DMG-Datei gemountet und Sie erhalten eine Kopie der Hello World-Anwendung und einen Link zu Ihrem Programme-Ordner für eine einfache Installation. Ziehen Sie die Anwendungsdatei in den Ordner Programme, und schon ist Ihre Anwendung installiert. Schicken Sie die DMG-Datei an einen Freund oder eine Freundin, der/die das Gleiche tun kann.

In diesem Beispiel haben wir die Option `--adhoc-sign` verwendet, d.h. wir signieren unsere Anwendung mit *ad hoc* Anmeldeinformationen - temporären Anmeldeinformationen, die nur auf Ihrem Rechner funktionieren. Wir haben dies getan, um das Tutorial einfach zu halten. Das Einrichten von Code-Signatur-Identitäten ist ein wenig fummelig, und sie sind nur *erforderlich*, wenn Sie Ihre Anwendung an andere weitergeben wollen. Würden wir eine echte Anwendung veröffentlichen, die andere nutzen können, müssten wir echte Anmeldeinformationen angeben.

Wenn Sie bereit sind, eine echte Anwendung zu veröffentlichen, sehen Sie sich die Briefcase-Anleitung zum Thema „Einrichten einer macOS Code Signing Identity <https://briefcase.readthedocs.io/en/latest/how-to/code-signing/macOS.html>“ an

Die Ausgabe des Paketschritts wird je nach Linux-Distribution leicht unterschiedlich sein. Wenn Sie mit einer von Debian abgeleiteten Distribution arbeiten, werden Sie sehen:

```
(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
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

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

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

Der `dist`-Ordner enthält die `.deb`-Datei, die erzeugt wurde.

Wenn Sie mit einer RHEL-basierten Distribution arbeiten, werden Sie es sehen:

```
(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
```

Der Ordner `dist` enthält die erzeugte `.rpm`-Datei.

Wenn Sie mit einer Arch-basierten Distribution arbeiten, werden Sie es sehen:

```
(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
```

Der Ordner `dist` enthält die Datei `.pkg.tar.zst`, die erzeugt wurde.

Andere Linux-Distributionen werden derzeit nicht für die Paketierung unterstützt.

Wenn Sie ein Paket für eine andere Linux-Distribution als die, die Sie verwenden, erstellen möchten, kann Briefcase ebenfalls helfen - allerdings müssen Sie Docker installieren.

Offizielle Installationsprogramme für die Docker Engine sind für eine Reihe von Unix-Distributionen verfügbar. Folgen Sie den Anweisungen für Ihre Plattform; stellen Sie jedoch sicher, dass Sie Docker nicht im „Rootless"-Modus installieren.

Sobald Sie Docker installiert haben, sollten Sie in der Lage sein, einen Linux-Container zu starten - zum Beispiel:

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

zeigt Ihnen eine Unix-Eingabeaufforderung (etwa `root@84444e31cff9:/#`) innerhalb eines Ubuntu 22.04 Docker-Containers. Geben Sie Strg-D ein, um Docker zu beenden und zu Ihrer lokalen Shell zurückzukehren.

Sobald Sie Docker installiert haben, können Sie Briefcase verwenden, um ein Paket für jede Linux-Distribution zu erstellen, die Briefcase unterstützt, indem Sie ein Docker-Image als Argument übergeben. Um beispielsweise ein DEB-Paket für Ubuntu 22.04 (Jammy) zu erstellen, können Sie, unabhängig vom Betriebssystem, das Sie verwenden, Folgendes ausführen:

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

Dadurch wird das Docker-Image für das von Ihnen ausgewählte Betriebssystem heruntergeladen, ein Container erstellt, der Briefcase-Builds ausführen kann, und das Anwendungspaket innerhalb des Images erstellt. Sobald dies abgeschlossen ist, enthält der Ordner `dist` das Paket für die Linux-Zieldistribution.

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

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

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

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

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

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

Linking application installer...
Linking... done

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

In diesem Beispiel haben wir die Option `--adhoc-sign` verwendet, d.h. wir signieren unsere Anwendung mit *ad hoc* Anmeldeinformationen - temporären Anmeldeinformationen, die nur auf Ihrem Rechner funktionieren. Wir haben dies getan, um das Tutorial einfach zu halten. Das Einrichten von Code-Signatur-Identitäten ist ein wenig fummelig, und sie sind nur *erforderlich*, wenn Sie Ihre Anwendung an andere weitergeben wollen. Würden wir eine echte Anwendung veröffentlichen, die andere nutzen können, müssten wir echte Anmeldeinformationen angeben.

Wenn Sie bereit sind, eine echte Anwendung zu veröffentlichen, sehen Sie sich die Briefcase-Anleitung zum Thema „Einrichten einer macOS Code Signing Identity <https://briefcase.readthedocs.io/en/latest/how-to/code-signing/macOS.html>“ an

Sobald dieser Schritt abgeschlossen ist, enthält der Ordner `dist` eine Datei mit dem Namen `Hello_World-0.0.1.msi`. Wenn Sie auf dieses Installationsprogramm doppelklicken, um es zu starten, sollten Sie den bekannten Windows-Installationsprozess durchlaufen. Sobald die Installation abgeschlossen ist, finden Sie einen „Hello World“-Eintrag in Ihrem Startmenü.

2.4.5 Nächste Schritte

Wir haben unsere Anwendung nun für die Verteilung auf Desktop-Plattformen verpackt. Aber was passiert, wenn wir den Code in unserer Anwendung aktualisieren müssen? Wie bekommen wir diese Aktualisierungen in unsere paketierte Anwendung? Schauen Sie sich *Tutorial 4* an, um das herauszufinden…

2.5 Tutorial 4 - Aktualisieren Ihrer Anwendung

Im letzten Lehrgang haben wir unsere Anwendung als native Anwendung verpackt. Wenn Sie es mit einer realen Anwendung zu tun haben, wird das nicht das Ende der Geschichte sein - Sie werden wahrscheinlich einige Tests durchführen, Probleme entdecken und einige Änderungen vornehmen müssen. Selbst wenn Ihre Anwendung perfekt ist, werden Sie irgendwann die Version 2 Ihrer Anwendung mit Verbesserungen veröffentlichen wollen.

Wie aktualisieren Sie also Ihre installierte Anwendung, wenn Sie Codeänderungen vornehmen?

2.5.1 Aktualisierung des Anwendungscodes

Unsere Anwendung gibt derzeit auf der Konsole aus, wenn Sie die Schaltfläche drücken. Allerdings sollten GUI-Anwendungen die Konsole nicht wirklich für die Ausgabe verwenden. Sie müssen Dialoge verwenden, um mit den Benutzern zu kommunizieren.

Fügen wir ein Dialogfeld hinzu, um Hallo zu sagen, anstatt in die Konsole zu schreiben. Ändern Sie den `say_hello` Callback so, dass er wie folgt aussieht:

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

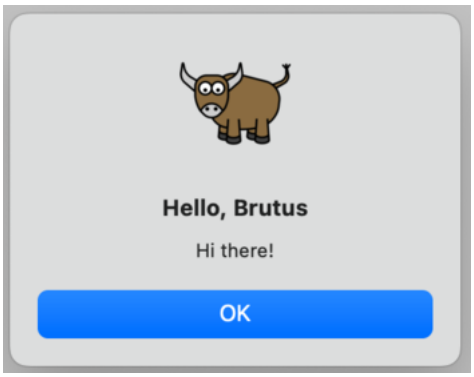
Damit wird Toga angewiesen, ein modales Dialogfeld zu öffnen, wenn die Schaltfläche gedrückt wird.

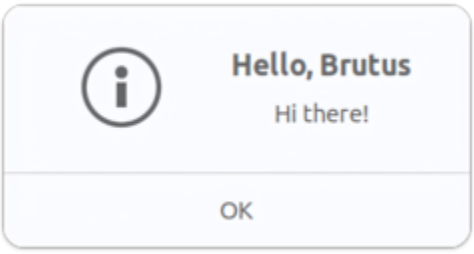
Wenn Sie `briefcase dev` ausführen, einen Namen eingeben und die Taste drücken, sehen Sie das neue Dialogfenster:

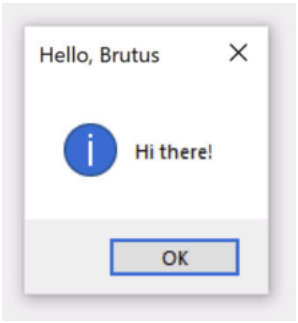
macOS

Linux

Windows







Wenn Sie jedoch `briefcase run` ausführen, wird das Dialogfeld nicht angezeigt.

Warum ist das so? Nun, `briefcase dev` arbeitet damit, dass es Ihren Code an Ort und Stelle ausführt - es versucht, eine möglichst realistische Laufzeitumgebung für Ihren Code zu erzeugen, aber es stellt keine der Plattforminfrastrukturen zur Verfügung, um Ihren Code als Anwendung zu verpacken. Ein Teil des Prozesses der Paketierung Ihrer Anwendung beinhaltet das Kopieren Ihres Codes *in* das Anwendungsbündel - und im Moment hat Ihre Anwendung noch den alten Code in sich.

Wir müssen also briefcase anweisen, Ihre Anwendung zu aktualisieren, indem wir die neue Version des Codes hineinkopieren. Wir *könnten* dies tun, indem wir das alte Plattformverzeichnis löschen und von vorne beginnen. Briefcase bietet jedoch einen einfacheren Weg - Sie können den Code für Ihre bestehende gebündelte Anwendung aktualisieren:

macOS

Linux

Windows

```
(beeware-venv) $ briefcase update

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

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

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
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...
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.
```

Wenn Briefcase die gerüstete Vorlage nicht finden kann, ruft es automatisch `create` auf, um ein neues Gerüst zu erzeugen.

Nun, da wir den Installationscode aktualisiert haben, können wir `briefcase build` ausführen, um die Anwendung neu zu kompilieren, `briefcase run`, um die aktualisierte Anwendung zu starten, und `briefcase package`, um die Anwendung für die Verteilung neu zu verpacken.

(macOS-Benutzer sollten bedenken, dass wir, wie in *Tutorial 3* erwähnt, für das Tutorial empfehlen, `briefcase package` mit dem `--adhoc-sign`-Flag auszuführen, um die Komplexität der Einrichtung einer Code-Signatur-Identität zu vermeiden und das Tutorial so einfach wie möglich zu halten)

2.5.2 Aktualisierung und Ausführung in einem Schritt

Wenn Sie Code-Änderungen schnell iterieren, werden Sie wahrscheinlich eine Code-Änderung vornehmen, die Anwendung aktualisieren und sie sofort wieder ausführen wollen. Für die meisten Zwecke ist der Entwicklermodus (`briefcase dev`) der einfachste Weg, diese Art der schnellen Iteration durchzuführen. Wenn Sie jedoch etwas darüber testen, wie Ihre Anwendung als native Binärdatei läuft, oder einen Fehler suchen, der nur auftritt, wenn Ihre Anwendung in gepackter Form vorliegt, müssen Sie möglicherweise wiederholte Aufrufe von `briefcase run` verwenden. Um den Prozess des Aktualisierens und Ausführens der gebündelten Anwendung zu vereinfachen, hat Briefcase eine Abkürzung, die dieses Verwendungsmuster unterstützt - die `-u` (oder `--update`) Option auf dem `run` Befehl.

Versuchen wir, eine weitere Änderung vorzunehmen. Sie haben vielleicht bemerkt, dass das Dialogfeld „Hallo“ sagt, wenn Sie keinen Namen in das Texteingabefeld eingeben. Ändern wir die Funktion „say_hello“ erneut, um diesen Sonderfall zu behandeln.

Am Anfang der Datei, zwischen den Importen und der `class HelloWorld`-Definition, fügen Sie eine Utility-Methode hinzu, um eine entsprechende Begrüßung in Abhängigkeit vom Wert des angegebenen Namens zu erzeugen:

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

Ändern Sie dann den „say_hello“-Callback, um diese neue Dienstprogramm-Methode zu verwenden:

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

Führen Sie Ihre Anwendung im Entwicklungsmodus aus (mit `briefcase dev`), um zu bestätigen, dass die neue Logik funktioniert; dann aktualisieren, bauen und starten Sie die Anwendung mit einem Befehl:

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
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

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

[helloworld] Application updated.

[helloworld] Building application...
...

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

[helloworld] Starting app...
```

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

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

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

[helloworld] Application updated.

[helloworld] Starting app...
```

Der Befehl package akzeptiert auch das Argument `-u`. Wenn Sie also eine Änderung an Ihrem Anwendungscode vornehmen und sofort neu packen wollen, können Sie `briefcase package -u` ausführen.

2.5.3 Nächste Schritte

Wir haben unsere Anwendung jetzt für die Verteilung auf Desktop-Plattformen verpackt und konnten den Code in unserer Anwendung aktualisieren.

Aber was ist mit mobilen Geräten? In *Tutorial 5* werden wir unsere Anwendung in eine mobile Anwendung umwandeln und sie in einem Gerätesimulator und auf einem Telefon einsetzen.

2.6 Tutorial 5 - Mobiles Arbeiten

Bislang haben wir unsere Anwendung auf dem Desktop ausgeführt und getestet. BeeWare unterstützt jedoch auch mobile Plattformen - und die von uns geschriebene Anwendung kann auch auf Ihrem mobilen Gerät eingesetzt werden!

iOS iOS-Anwendungen können nur unter macOS kompiliert werden.

Lasst uns unsere App für iOS bauen!

Android Android-Anwendungen können unter macOS, Windows oder Linux kompiliert werden.

Lassen Sie uns unsere App für Android erstellen!

2.6.1 Tutorial 5 - Mobiles Arbeiten: iOS

Um iOS-Anwendungen zu kompilieren, benötigen wir Xcode, das kostenlos im macOS App Store <https://apps.apple.com/au/app/xcode/id497799835?mt=12> erhältlich ist.

Sobald wir Xcode installiert haben, können wir unsere Anwendung als iOS-App bereitstellen.

Der Prozess der Bereitstellung einer Anwendung für iOS ist dem Prozess der Bereitstellung als Desktop-Anwendung sehr ähnlich. Zuerst führen Sie den Befehl `create` aus - aber dieses Mal geben wir an, dass wir eine iOS-Anwendung erstellen wollen:

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

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

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

[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
```

Sobald dies abgeschlossen ist, haben wir ein `build/helloworld/ios/xcode` Verzeichnis, das ein Xcode Projekt enthält, sowie die unterstützenden Bibliotheken und den Anwendungscode, der für die Anwendung benötigt wird.

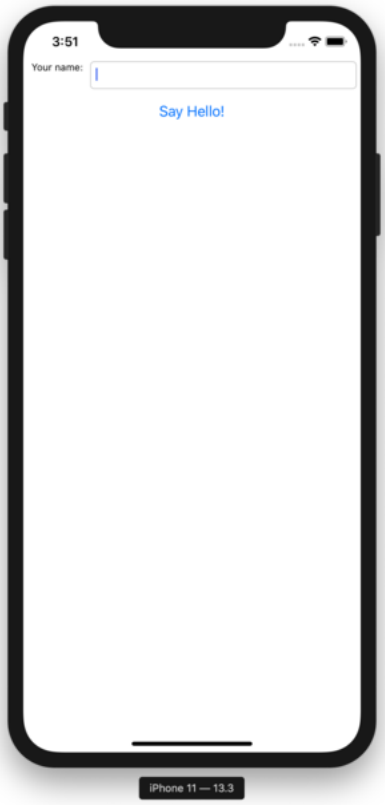
Sie können dann Briefcase verwenden, um Ihre Anwendung mit `briefcase build iOS` zu kompilieren:

```
(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
```

Wir sind nun bereit, unsere Anwendung mit `briefcase run iOS` zu starten. Sie werden aufgefordert, ein Gerät auszuwählen, für das kompiliert werden soll. Wenn Sie Simulatoren für mehrere iOS SDK-Versionen installiert haben, werden Sie möglicherweise auch gefragt, welche iOS-Version Sie verwenden möchten. Die Optionen, die Ihnen angezeigt werden, können sich von den in dieser Ausgabe gezeigten Optionen unterscheiden - zumindest wird die Liste der Geräte wahrscheinlich anders aussehen. Für unsere Zwecke spielt es keine Rolle, welchen Simulator Sie wählen.

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

Select simulator device:

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

>  10

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

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

or:

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

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

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

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

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

Dadurch wird der iOS-Simulator gestartet, Ihre Anwendung installiert und gestartet. Sie sollten sehen, dass der Simulator startet und schließlich Ihre iOS-Anwendung öffnet:



Wenn Sie bereits im Voraus wissen, welchen iOS-Simulator Sie verwenden möchten, können Sie Briefcase mit der Option `-d` (oder `--device`) anweisen, diesen Simulator zu verwenden. Verwenden Sie den Namen des Geräts, das Sie beim Erstellen Ihrer Anwendung ausgewählt haben, und führen Sie es aus:

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

Wenn Sie mehrere iOS-Versionen zur Verfügung haben, wählt Briefcase die höchste iOS-Version aus; wenn Sie eine bestimmte iOS-Version auswählen möchten, weisen Sie das Programm an, diese Version zu verwenden:

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

Oder Sie können ein bestimmtes Gerät UDID nennen:

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

Nächste Schritte

Wir haben jetzt eine Anwendung auf unserem Telefon! Gibt es noch einen anderen Ort, an dem wir eine BeeWare-Anwendung einsetzen können? Schauen Sie sich *Tutorial 6* an, um das herauszufinden…

2.6.2 Tutorial 5 - Mobiles Arbeiten: Android

Jetzt nehmen wir unsere Anwendung und stellen sie als Android-Anwendung bereit.

Der Prozess der Bereitstellung einer Anwendung für Android ist dem Prozess der Bereitstellung als Desktop-Anwendung sehr ähnlich. Briefcase übernimmt die Installation von Abhängigkeiten für Android, einschließlich des Android SDK, des Android-Emulators und eines Java-Compilers.

Eine Android-App erstellen und kompilieren

Zuerst führen Sie den Befehl `create` aus. Dies lädt eine Android-App-Vorlage herunter und fügt Ihren Python-Code hinzu.

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
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
...

[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
```

Wenn Sie `briefcase create android` zum ersten Mal ausführen, lädt Briefcase ein Java JDK und das Android SDK herunter. Die Dateigrößen und Downloadzeiten können beträchtlich sein; dies kann eine Weile dauern (10 Minuten oder länger, abhängig von der Geschwindigkeit Ihrer Internetverbindung). Wenn der Download abgeschlossen ist, werden Sie aufgefordert, die Android-SDK-Lizenz von Google zu akzeptieren.

Sobald dies abgeschlossen ist, haben wir in unserem Projekt ein Verzeichnis `build\helloworld\android\gradle`, das ein Android-Projekt mit einer Gradle-Build-Konfiguration enthält. Dieses Projekt enthält Ihren Anwendungscode und ein Support-Paket, das den Python-Interpreter enthält.

Wir können dann den Befehl `build` von Briefcase verwenden, um diese in eine Android APK App-Datei zu kompilie-

ren.

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 kann stecken bleiben

Während des Schrittes `briefcase build android` gibt Gradle (das Werkzeug zur Erstellung der Android-Plattform) `CONFIGURING: 100%` aus und scheint nichts zu tun. Keine Sorge, es steckt nicht fest - es lädt weitere Android SDK Komponenten herunter. Je nach der Geschwindigkeit Ihrer Internetverbindung kann dies weitere 10 Minuten (oder länger) dauern. Diese Verzögerung sollte nur beim allerersten Mal auftreten, wenn Sie `build` ausführen; die Tools werden zwischengespeichert und beim nächsten Build werden die zwischengespeicherten Versionen verwendet.

Ausführen der Anwendung auf einem virtuellen Gerät

Wir sind nun bereit, unsere Anwendung zu starten. Sie können den Befehl `run` von Briefcase verwenden, um die Anwendung auf einem Android-Gerät auszuführen. Beginnen wir mit der Ausführung auf einem Android-Emulator.

Um Ihre Anwendung auszuführen, starten Sie `briefcase run android`. Daraufhin erhalten Sie eine Liste von Geräten, auf denen Sie die Anwendung ausführen können. Der letzte Punkt ist immer eine Option zum Erstellen eines neuen Android-Emulators.

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

>
```

Nun können wir unser gewünschtes Gerät auswählen. Wählen Sie die Option „Einen neuen Android-Emulator erstellen“ und akzeptieren Sie die Standardauswahl für den Gerätenamen (`beePhone`).

Briefcase `run` wird das virtuelle Gerät automatisch starten. Wenn das Gerät hochfährt, sehen Sie das Android-Logo:

Sobald das Gerät hochgefahren ist, wird Briefcase Ihre Anwendung auf dem Gerät installieren. Sie sehen kurz einen Startbildschirm:

Die Anwendung wird dann gestartet. Während die App startet, wird ein Begrüßungsbildschirm angezeigt:

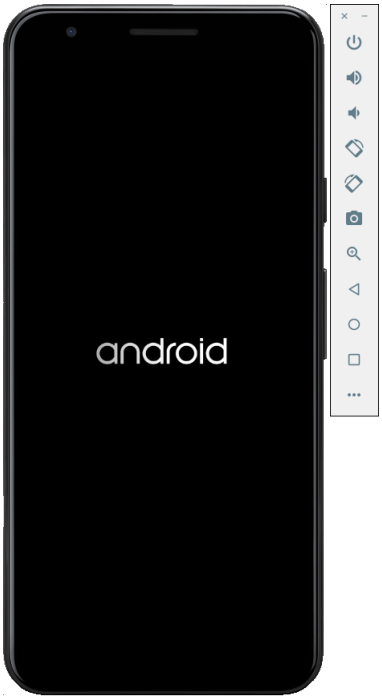


Abb. 1: Booten eines virtuellen Android-Geräts

Abb. 2: Virtuelles Android-Gerät vollständig gestartet, auf dem Startbildschirm

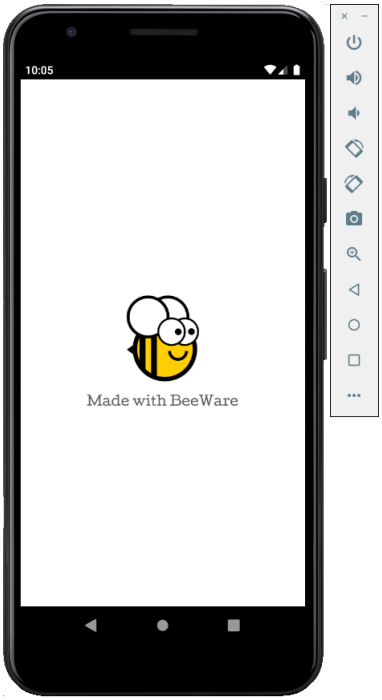


Abb. 3: App-Startbildschirm

Der Emulator ist nicht gestartet!

Der Android-Emulator ist eine komplexe Software, die auf eine Reihe von Hardware- und Betriebssystemfunktionen angewiesen ist - Funktionen, die auf älteren Rechnern möglicherweise nicht verfügbar oder aktiviert sind. Wenn Sie Schwierigkeiten beim Starten des Android-Emulators haben, lesen Sie den Abschnitt „Anforderungen und Empfehlungen <https://developer.android.com/studio/run/emulator#requirements>“ in der Android-Entwicklerdokumentation.

Wenn die App zum ersten Mal gestartet wird, muss sie sich auf dem Gerät entpacken. Das kann ein paar Sekunden dauern. Sobald sie entpackt ist, sehen Sie die Android-Version unserer Desktop-App:

Wenn Sie nicht sehen, dass Ihre Anwendung startet, sollten Sie Ihr Terminal überprüfen, in dem Sie `briefcase run` ausgeführt haben, und nach Fehlermeldungen suchen.

Wenn Sie in Zukunft auf diesem Gerät arbeiten wollen, ohne das Menü zu benutzen, können Sie Briefcase den Namen des Emulators mitteilen, indem Sie `briefcase run android -d @beePhone` verwenden, um direkt auf dem virtuellen Gerät zu arbeiten.

Ausführen der App auf einem physischen Gerät

Wenn Sie ein physisches Android-Telefon oder -Tablet besitzen, können Sie es mit einem USB-Kabel an Ihren Computer anschließen und dann das Briefcase verwenden, um Ihr physisches Gerät anzuvisieren.

Android erfordert, dass Sie Ihr Gerät vorbereiten, bevor Sie es für die Entwicklung verwenden können. Sie müssen 2 Änderungen an den Optionen auf Ihrem Gerät vornehmen:

- Entwickleroptionen aktivieren
- USB-Debugging einschalten

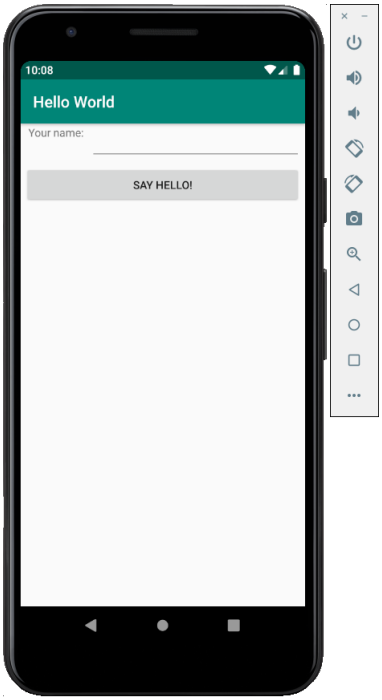


Abb. 4: Demo-App vollständig gestartet

Details zur Durchführung dieser Änderungen können in der Android-Entwicklerdokumentation <https://developer.android.com/studio/debug/dev-options#enable>`__ nachgelesen werden.

Sobald diese Schritte abgeschlossen sind, sollte Ihr Gerät in der Liste der verfügbaren Geräte erscheinen, wenn Sie `briefcase run android` ausführen.

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
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
>
```

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

Select device:

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

>
```

Hier sehen wir ein neues physisches Gerät mit seiner Seriennummer in der Einsatzliste - in diesem Fall ein Pixel 3a. Wenn Sie in Zukunft auf diesem Gerät arbeiten möchten, ohne das Menü zu verwenden, können Sie die Seriennummer des Telefons an Briefcase übergeben (in diesem Fall `briefcase run android -d 94ZZY0LNE8`). Dies wird direkt auf dem Gerät ausgeführt, ohne dass Sie gefragt werden.

Mein Gerät wird nicht angezeigt!

Wenn Ihr Gerät in dieser Liste nicht erscheint, haben Sie entweder das USB-Debugging nicht aktiviert (oder das Gerät ist nicht eingesteckt!).

Wenn Ihr Gerät zwar angezeigt wird, aber als „Unbekanntes Gerät (nicht für die Entwicklung autorisiert)“ aufgeführt ist, wurde der Entwicklermodus nicht korrekt aktiviert. Führen Sie die „Schritte zum Aktivieren der Entwickleroptionen <https://developer.android.com/studio/debug/dev-options#enable>“ erneut aus, und führen Sie „Briefcase run android“ erneut aus.

Nächste Schritte

Wir haben jetzt eine Anwendung auf unserem Telefon! Gibt es noch einen anderen Ort, an dem wir eine BeeWare-Anwendung einsetzen können? Schauen Sie sich *Tutorial 6* an, um das herauszufinden…

2.7 Tutorial 6 - Ins Netz stellen!

Das Toga Widget Toolkit unterstützt nicht nur mobile Plattformen, sondern auch das Web! Mit der gleichen API, die Sie für die Bereitstellung Ihrer Desktop- und mobilen Anwendungen verwendet haben, können Sie Ihre Anwendung als einseitige Webanwendung bereitstellen.

Proof of Concept

Das Toga Web Backend ist das am wenigsten ausgereifte von allen Toga Backends. Es ist ausgereift genug, um ein paar Funktionen zu zeigen, aber es wird wahrscheinlich fehlerhaft sein und viele der Widgets, die auf anderen Plattformen verfügbar sind, fehlen. Zum jetzigen Zeitpunkt sollte Web Deployment als „Proof of Concept“ betrachtet werden - genug, um zu demonstrieren, was getan werden kann, aber nicht genug, um sich auf eine ernsthafte Entwicklung zu verlassen.

Wenn Sie Probleme mit diesem Schritt des Tutorials haben, können Sie zur nächsten Seite wechseln.

2.7.1 Als Webanwendung bereitstellen

Der Prozess der Bereitstellung als einseitige Webanwendung folgt dem gleichen bekannten Muster - Sie erstellen die Anwendung, bauen die Anwendung und führen sie dann aus. Wenn Sie versuchen, eine Anwendung auszuführen und Briefcase feststellt, dass sie nicht für die angestrebte Plattform erstellt oder gebaut wurde, führt es die Schritte zum Erstellen und Bauen für Sie aus. Da wir die Anwendung zum ersten Mal für das Web ausführen, können wir alle drei Schritte mit einem Befehl ausführen:

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
...
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

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

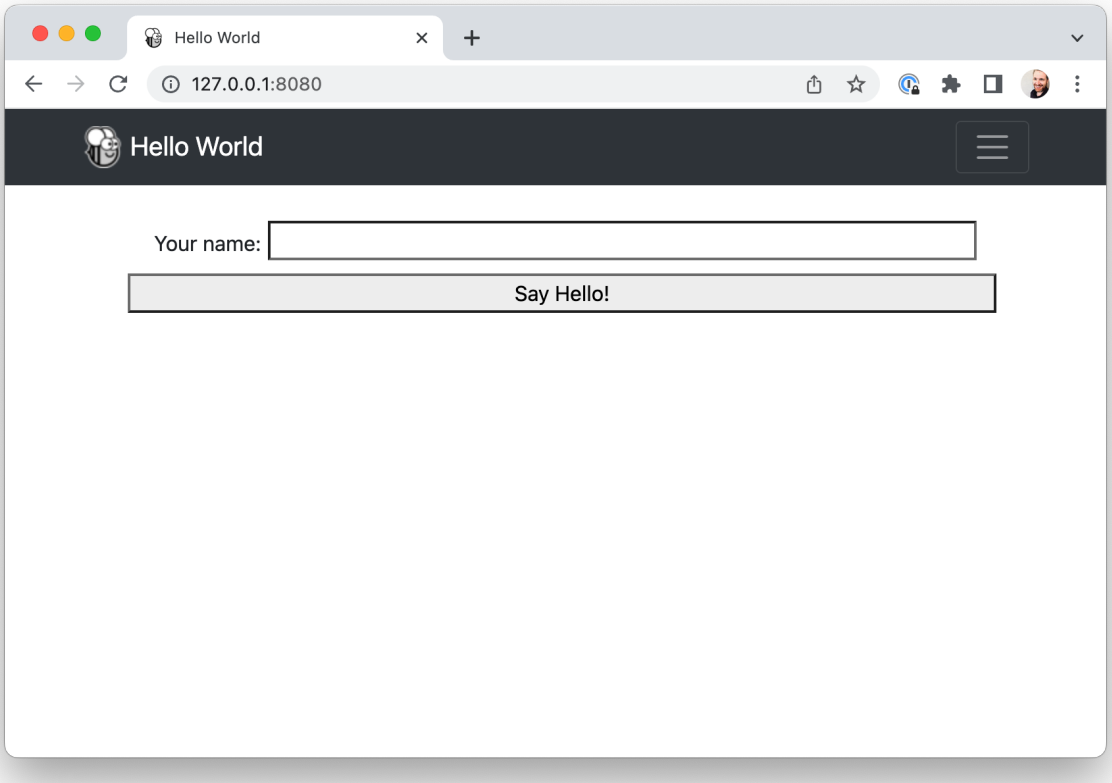
```
(beeware-venv) C:\...>briefcase run web

[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
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

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

Dadurch wird ein Webbrowser geöffnet, der auf http://127.0.0.1:8080 verweist:



Wenn Sie Ihren Namen eingeben und auf die Schaltfläche klicken, wird ein Dialogfeld angezeigt.

2.7.2 Wie funktioniert das?

Diese Webanwendung ist eine statische Website - eine einzelne HTML-Quellseite mit einigen CSS- und anderen Ressourcen. Briefcase hat einen lokalen Webserver gestartet, um diese Seite bereitzustellen, damit Ihr Browser die Seite anzeigen kann. Wenn Sie diese Webseite in Produktion geben möchten, können Sie den Inhalt des Ordners `www` auf einen beliebigen Webserver kopieren, der statische Inhalte bereitstellen kann.

Aber wenn Sie die Taste drücken, führen Sie Python-Code aus… wie funktioniert das? Toga verwendet PyScript, um einen Python-Interpreter im Browser bereitzustellen. Briefcase verpackt den Code Ihrer Anwendung als Rädchen, die PyScript im Browser laden kann. Wenn die Seite geladen wird, läuft der Anwendungscode im Browser und baut die Benutzeroberfläche mit Hilfe des Browser-DOMs auf. Wenn Sie auf eine Schaltfläche klicken, führt diese Schaltfläche den Code zur Ereignisbehandlung im Browser aus.

2.7.3 Nächste Schritte

Obwohl wir diese App jetzt auf dem Desktop, auf dem Handy und im Web bereitgestellt haben, ist die App recht einfach und enthält keine Bibliotheken von Drittanbietern. Können wir Bibliotheken aus dem Python Package Index (PyPI) in unsere App einbinden? Schauen Sie sich *Tutorial 7* an, um das herauszufinden…

2.8 Tutorial 7 - Die (Dritt)-Partei in Gang bringen

Bisher haben wir in der von uns erstellten Anwendung nur unseren eigenen Code sowie den von BeeWare bereitgestellten Code verwendet. In einer realen Anwendung werden Sie jedoch wahrscheinlich eine Bibliothek eines Drittanbieters verwenden wollen, die Sie aus dem Python Package Index (PyPI) herunterladen.

Ändern wir unsere Anwendung so, dass sie eine Bibliothek eines Drittanbieters enthält.

2.8.1 Zugriff auf eine API

Eine häufige Aufgabe, die eine Anwendung erfüllen muss, besteht darin, eine Anfrage an eine Web-API zu stellen, um Daten abzurufen und diese Daten dem Benutzer anzuzeigen. Da es sich hier um eine Spielzeuganwendung handelt, haben wir keine *echte* API, mit der wir arbeiten können, also verwenden wir die {JSON} Placeholder API als Datenquelle.

Die {JSON} Platzhalter-API verfügt über eine Reihe von „gefälschten“ API-Endpunkten, die Sie als Testdaten verwenden können. Einer dieser APIs ist der Endpunkt `/posts/`, der gefälschte Blog-Posts zurückgibt. Wenn Sie `https://jsonplaceholder.typicode.com/posts/42` in Ihrem Browser öffnen, erhalten Sie einen JSON-Payload, der einen einzelnen Beitrag beschreibt - etwas Lorum ipsum Inhalt für einen Blogbeitrag mit der ID 42.

Die Python-Standardbibliothek enthält alle Werkzeuge, die Sie für den Zugriff auf eine API benötigen. Die eingebauten APIs sind jedoch sehr einfach. Sie sind gute Implementierungen des HTTP-Protokolls - aber sie erfordern, dass der Benutzer viele Details auf niedriger Ebene verwaltet, wie URL-Umleitung, Sitzungen, Authentifizierung und Nutzdatenkodierung. Als „normaler Browser-Benutzer“ sind Sie wahrscheinlich daran gewöhnt, diese Details als selbstverständlich anzusehen, da ein Browser diese Details für Sie verwaltet.

Infolgedessen haben Leute Bibliotheken von Drittanbietern entwickelt, die eingebauten APIs umhüllen und eine einfachere API bereitstellen, die der alltäglichen Browsererfahrung besser entspricht. Wir werden eine dieser Bibliotheken verwenden, um auf die {JSON} Platzhalter-API zuzugreifen - eine Bibliothek namens httpx.

Fügen wir einen `httpx`-API-Aufruf zu unserer Anwendung hinzu. Fügen Sie einen Import am Anfang der `app.py` hinzu, um `httpx` zu importieren:

```
import httpx
```

Dann modifizieren Sie den `say_hello()` Callback so, dass er wie folgt aussieht:

```
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"],
    )
```

Dies ändert den `say_hello()` Callback so, dass er, wenn er aufgerufen wird, dies tut:

- eine GET-Anfrage an die JSON-Platzhalter-API stellen, um Beitrag 42 zu erhalten;
- dekodiert die Antwort als JSON;
- den Text der Nachricht zu extrahieren; und
- den Text dieses Beitrags als Text des Dialogs einschließen.

Führen wir unsere aktualisierte Anwendung im Briefcase-Entwicklermodus aus, um zu prüfen, ob unsere Änderung funktioniert.

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
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
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) 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'
```

Was ist passiert? Wir haben `httpx` zu unserem *Code* hinzugefügt, aber wir haben es nicht zu unserer virtuellen Entwicklungsumgebung hinzugefügt. Wir können dies beheben, indem wir `httpx` mit `pip` installieren und dann `briefcase dev` erneut ausführen:

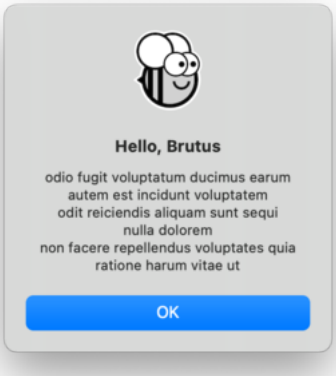
macOS

Linux

Windows

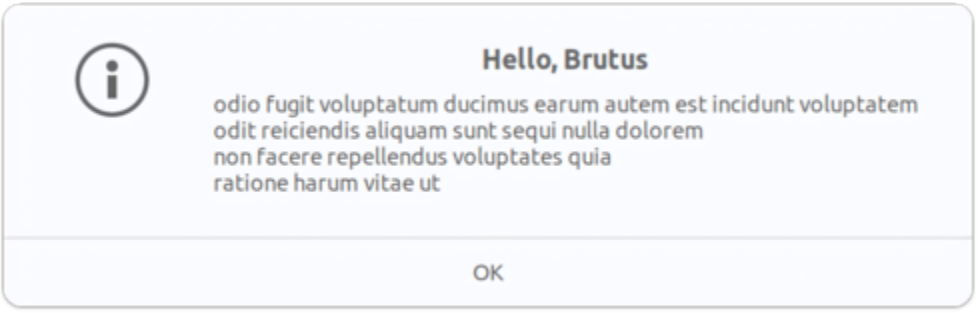
```
(beeware-venv) $ python -m pip install httpx
(beeware-venv) $ briefcase dev
```

Wenn Sie einen Namen eingeben und auf die Schaltfläche drücken, sollte ein Dialogfeld angezeigt werden, das etwa so aussieht:



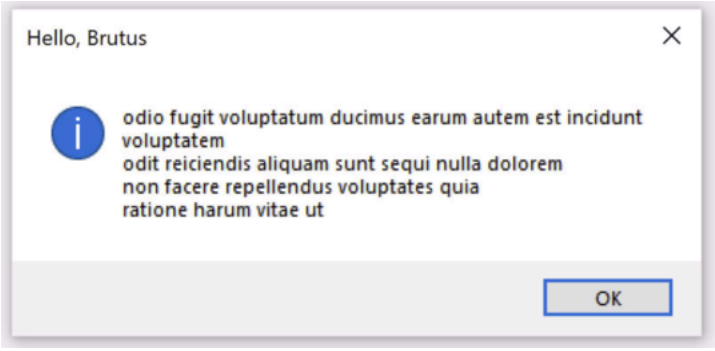

```
(beeware-venv) $ python -m pip install httpx
(beeware-venv) $ briefcase dev
```

Wenn Sie einen Namen eingeben und auf die Schaltfläche drücken, sollte ein Dialogfeld angezeigt werden, das etwa so aussieht:




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

Wenn Sie einen Namen eingeben und auf die Schaltfläche drücken, sollte ein Dialogfeld angezeigt werden, das etwa so aussieht:



Wir haben jetzt eine funktionierende Anwendung, die eine Bibliothek eines Drittanbieters verwendet und im Entwicklungsmodus läuft!

2.8.2 Ausführen der aktualisierten Anwendung

Lassen Sie uns diesen aktualisierten Anwendungscode als eigenständige Anwendung verpacken. Da wir Änderungen am Code vorgenommen haben, müssen wir die gleichen Schritte wie in *Tutorial 4* ausführen:

macOS

Linux

Windows

Aktualisieren Sie den Code in der gepackten Anwendung:

```
(beeware-venv) $ briefcase update

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

[helloworld] Application updated.
```

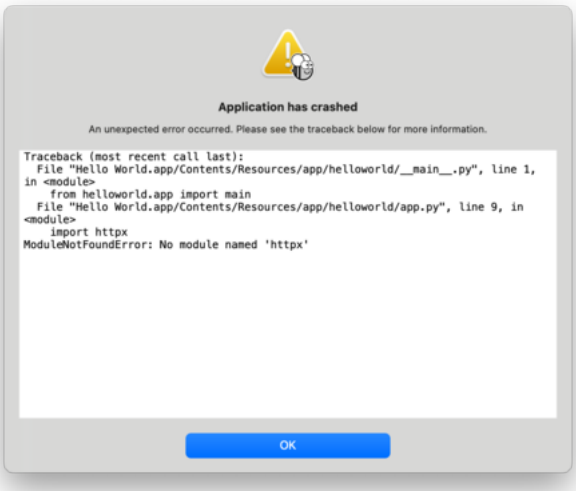
Bauen Sie die Anwendung neu auf:

```
(beeware-venv) $ briefcase build

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

Und schließlich führen Sie die Anwendung aus:

```
(beeware-venv) $ briefcase run

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

Wenn die Anwendung jedoch ausgeführt wird, wird in der Konsole ein Fehler und ein Absturzdialog angezeigt:



Aktualisieren Sie den Code in der gepackten Anwendung:

```
(beeware-venv) $ briefcase update

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

[helloworld] Application updated.
```

Bauen Sie die Anwendung neu auf:

```
(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
```

Und schließlich führen Sie die Anwendung aus:

```
(beeware-venv) $ briefcase run

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

Wenn die Anwendung jedoch ausgeführt wird, wird in der Konsole ein Fehler angezeigt:

```
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.
```

Aktualisieren Sie den Code in der gepackten Anwendung:

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

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

[helloworld] Application updated.
```

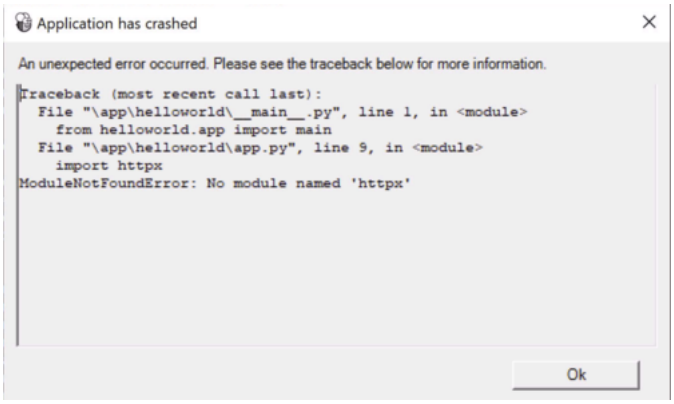
Bauen Sie die Anwendung neu auf:

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

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

Und schließlich führen Sie die Anwendung aus:

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

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

Wenn die Anwendung jedoch ausgeführt wird, wird in der Konsole ein Fehler und ein Absturzdialog angezeigt:



Wieder einmal ist der Start der Anwendung fehlgeschlagen, weil `httpx` installiert wurde - aber warum? Haben wir `httpx` nicht schon installiert?

Das haben wir - aber nur in der Entwicklungsumgebung. Ihre Entwicklungsumgebung befindet sich ausschließlich auf Ihrem Rechner - und wird nur aktiviert, wenn Sie sie explizit aktivieren. Obwohl Briefcase einen Entwicklungsmodus hat, ist der Hauptgrund für die Verwendung von Briefcase, Ihren Code zu verpacken, damit Sie ihn an jemand anderen weitergeben können.

Der einzige Weg, um zu garantieren, dass jemand anderes eine Python-Umgebung hat, die alles enthält, was er braucht, ist, eine vollständig isolierte Python-Umgebung zu erstellen. Das bedeutet, dass es eine komplett isolierte Python-Installation und einen komplett isolierten Satz von Abhängigkeiten gibt. Das ist es, was Briefcase baut, wenn Sie `briefcase build` ausführen - eine isolierte Python Umgebung. Das erklärt auch, warum `httpx` nicht installiert ist - es wurde in Ihrer *Entwicklungsumgebung* installiert, aber nicht in der gepackten Anwendung.

Wir müssen also Briefcase mitteilen, dass unsere Anwendung eine externe Abhängigkeit hat.

2.8.3 Aktualisieren von Abhängigkeiten

Im Hauptverzeichnis Ihrer Anwendung befindet sich eine Datei namens `pyproject.toml`. Diese Datei enthält alle Konfigurationsdetails der Anwendung, die Sie beim Ausführen von `briefcase new` angegeben haben.

`pyproject.toml` ist in Abschnitte unterteilt; einer der Abschnitte beschreibt die Einstellungen für Ihre Anwendung:

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

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

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

Die Option `requires` beschreibt die Abhängigkeiten unserer Anwendung. Es handelt sich um eine Liste von Strings, die Bibliotheken (und optional die Versionen) der Bibliotheken angeben, die Sie in Ihre Anwendung aufnehmen möchten.

Ändern Sie die Einstellung `requires` so, dass sie lautet:

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

Durch Hinzufügen dieser Einstellung teilen wir Briefcase mit: „Wenn du meine Anwendung baust, führe `pip install httpx` in das Anwendungsbündel ein“. Alles, was eine legale Eingabe für `pip install` wäre, kann hier verwendet werden - Sie könnten also angeben:

- Eine bestimmte Bibliotheksversion (z.B. `"httpx==0.19.0"`);
- Eine Reihe von Bibliotheksversionen (z.B. `"httpx>=0.19"`);
- Ein Pfad zu einem Git-Repository (z. B. `"git+https://github.com/encode/httpx"`); oder
- Ein lokaler Dateipfad (Aber Achtung: Wenn Sie Ihren Code an jemand anderen weitergeben, existiert dieser Pfad wahrscheinlich nicht auf dessen Rechner!)

Weiter unten in `pyproject.toml` werden Sie andere Abschnitte bemerken, die vom Betriebssystem abhängig sind, wie `[tool.briefcase.app.helloworld.macOS]` und `[tool.briefcase.app.helloworld.windows]`. Diese Abschnitte haben *auch* eine `requires` Einstellung. Diese Einstellungen erlauben es Ihnen, zusätzliche plattformspezifische Abhängigkeiten zu definieren - wenn Sie also zum Beispiel eine plattformspezifische Bibliothek benötigen, um einen Aspekt Ihrer Anwendung zu behandeln, können Sie diese Bibliothek im plattformspezifischen `requires`-Abschnitt angeben, und diese Einstellung wird nur für diese Plattform verwendet. Sie werden feststellen, dass die `toga`-Bibliotheken alle im plattformspezifischen `requires`-Abschnitt angegeben sind - das liegt daran, dass die Bibliotheken, die für die Anzeige einer Benutzeroberfläche benötigt werden, plattformspezifisch sind.

In unserem Fall wollen wir, dass `httpx` auf allen Plattformen installiert wird, also verwenden wir die Einstellung `requires` auf Anwendungsebene. Die Abhängigkeiten auf Anwendungsebene werden immer installiert; die plattformspezifischen Abhängigkeiten werden *zusätzlich* zu den Abhängigkeiten auf Anwendungsebene installiert.

Einige Binärpakete sind möglicherweise nicht verfügbar

Auf Desktop-Plattformen (macOS, Windows, Linux) kann jede `pip`-Installation zu Ihren Anforderungen hinzugefügt werden. Auf mobilen und Web-Plattformen sind Ihre Möglichkeiten etwas eingeschränkt <https://briefcase.readthedocs.io/en/latest/background/faq.html#can-i-use-third-party-python-packages-in-my-app>`__.

Kurz gesagt: Jedes *reine Python*-Paket (d.h. Pakete, die *kein* binäres Modul enthalten) kann ohne Probleme verwendet werden. Wenn Ihre Abhängigkeit jedoch eine Binärkomponente enthält, muss diese kompiliert werden; derzeit bieten die meisten Python-Pakete keine Kompilierungsunterstützung für Nicht-Desktop-Plattformen.

BeeWare kann Binärdateien für einige beliebte Binärmodule (einschließlich `numpy`, `pandas` und `cryptography`) bereitstellen. Es ist *normalerweise* möglich, Pakete für mobile Plattformen zu kompilieren, aber es ist nicht einfach einzurichten - das würde den Rahmen eines einführenden Tutorials wie diesem sprengen.

Nun, da wir Briefcase über unsere zusätzlichen Anforderungen informiert haben, können wir versuchen, unsere Anwendung erneut zu paketieren. Vergewissern Sie sich, dass Sie Ihre Änderungen in `pyproject.toml` gespeichert haben,

und aktualisieren Sie Ihre Anwendung erneut - dieses Mal mit dem Flag `-r`. Dadurch wird Briefcase angewiesen, die Anforderungen in der gepackten Anwendung zu aktualisieren:

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.
```

Sobald Sie das Update durchgeführt haben, können Sie `briefcase build` und `briefcase run` ausführen - und Sie sollten Ihre gepackte Anwendung mit dem neuen Dialogverhalten sehen.

Bemerkung: Die Option `-r` zum Aktualisieren der Anforderungen wird auch von den Befehlen `build` und `run` beachtet. Wenn Sie also in einem Schritt aktualisieren, bauen und ausführen wollen, können Sie `briefcase run -u -r` verwenden.

2.8.4 Nächste Schritte

Wir haben jetzt eine App, die eine Bibliothek eines Drittanbieters verwendet! Vielleicht haben Sie jedoch bemerkt, dass die App beim Drücken der Taste nicht mehr richtig reagiert. Können wir etwas tun, um das zu beheben? Schau dir *Tutorial 8* an, um das herauszufinden…

2.9 Tutorial 8 - Glatt machen

Wenn Sie keine *wirklich* schnelle Internetverbindung haben, werden Sie vielleicht feststellen, dass die grafische Benutzeroberfläche Ihrer Anwendung beim Drücken der Schaltfläche kurzzeitig blockiert wird. Das liegt daran, dass die Webanforderung, die wir gestellt haben, *synchron* ist. Wenn unsere Anwendung die Webanforderung stellt, wartet sie, bis die API eine Antwort zurückgibt, bevor sie fortfährt. Während sie wartet, erlaubt sie der Anwendung *nicht*, neu zu zeichnen - und das Ergebnis ist, dass die Anwendung blockiert.

2.9.1 GUI-Ereignis-Schleifen

Um zu verstehen, warum das so ist, müssen wir uns mit den Details der Funktionsweise einer GUI-Anwendung befassen. Die Einzelheiten variieren je nach Plattform, aber die grundlegenden Konzepte sind dieselben, unabhängig von der Plattform oder der GUI-Umgebung, die Sie verwenden.

Eine GUI-Anwendung besteht im Grunde aus einer einzigen Schleife, die etwa so aussieht:

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

Diese Schleife wird *Ereignisschleife* genannt. (Dies sind keine tatsächlichen Methodennamen - es ist eine Veranschaulichung dessen, was im „Pseudocode“ vor sich geht).

Wenn Sie auf eine Schaltfläche klicken, eine Bildlaufleiste ziehen oder eine Taste drücken, erzeugen Sie ein „Ereignis“. Dieses „Ereignis“ wird in eine Warteschlange gestellt, und die Anwendung wird die Warteschlange von Ereignissen verarbeiten, wenn sie das nächste Mal die Gelegenheit dazu hat. Der Benutzercode, der als Reaktion auf das Ereignis ausgelöst wird, wird *Ereignishandler* genannt. Diese Event-Handler werden als Teil des `process_events()` Aufrufs aufgerufen.

Sobald eine Anwendung alle verfügbaren Ereignisse verarbeitet hat, wird sie die grafische Benutzeroberfläche `neu zeichnen()`. Dabei werden alle Änderungen berücksichtigt, die Ereignisse an der Anzeige der Anwendung verursacht haben, sowie alles andere, was im Betriebssystem vor sich geht - zum Beispiel können die Fenster einer anderen Anwendung einen Teil des Fensters unserer Anwendung verdecken oder sichtbar machen, und die Neuzeichnung unserer Anwendung muss den Teil des Fensters widerspiegeln, der gerade sichtbar ist.

Ein wichtiges Detail: Während eine Anwendung ein Ereignis verarbeitet, *kann sie nicht neu zeichnen* und *kann sie keine anderen Ereignisse verarbeiten*.

Das bedeutet, dass jede in einem Event-Handler enthaltene Benutzerlogik schnell abgeschlossen werden muss. Jede Verzögerung bei der Fertigstellung des Event-Handlers wird vom Benutzer als Verlangsamung (oder Stopp) der GUI-Aktualisierungen wahrgenommen. Wenn diese Verzögerung lang genug ist, kann Ihr Betriebssystem dies als Problem melden - die macOS-„Beachball“- und Windows-„Spinner“-Symbole sind das Betriebssystem, das Ihnen mitteilt, dass Ihre Anwendung in einem Event-Handler zu lange braucht.

Einfache Operationen wie „Aktualisieren eines Etiketts“ oder „Neuberechnung der Gesamtsumme der Eingaben“ sind leicht und schnell zu erledigen. Es gibt jedoch eine Reihe von Vorgängen, die nicht schnell erledigt werden können. Wenn Sie eine komplexe mathematische Berechnung durchführen, alle Dateien in einem Dateisystem indizieren oder eine große Netzwerkanforderung ausführen, können Sie das nicht „einfach schnell erledigen“ - die Vorgänge sind von Natur aus langsam.

Wie können wir also langlebige Operationen in einer GUI-Anwendung durchführen?

2.9.2 Asynchrone Programmierung

Was wir brauchen, ist eine Möglichkeit, einer Anwendung in der Mitte eines langlebigen Event-Handlers mitzuteilen, dass es in Ordnung ist, die Kontrolle vorübergehend an die Event-Schleife zurückzugeben, solange wir dort fortfahren können, wo wir aufgehört haben. Es liegt an der Anwendung, zu bestimmen, wann diese Freigabe erfolgen kann; aber wenn die Anwendung die Kontrolle regelmäßig an die Ereignisschleife freigibt, können wir einen lang laufenden Event-Handler haben *und* eine reaktionsfähige Benutzeroberfläche beibehalten.

Wir können dies mit Hilfe der *asynchronen Programmierung* erreichen. Asynchrone Programmierung ist eine Art, ein Programm zu beschreiben, das es dem Interpreter erlaubt, mehrere Funktionen gleichzeitig auszuführen und die Ressourcen zwischen allen gleichzeitig laufenden Funktionen zu teilen.

Asynchrone Funktionen (so genannte *Co-Routinen*) müssen ausdrücklich als asynchron deklariert werden. Sie müssen auch intern erklären, wann die Möglichkeit besteht, den Kontext zu einer anderen Co-Routine zu wechseln.

In Python wird die asynchrone Programmierung mit den Schlüsselwörtern `async` und `await`, und dem Modul asyncio in der Standardbibliothek implementiert. Mit dem Schlüsselwort `async` können wir eine Funktion als asynchrone Co-Routine deklarieren. Mit dem Schlüsselwort `await` kann man angeben, wann die Möglichkeit besteht, den Kontext zu einer anderen Co-Routine zu wechseln. Das Modul asyncio bietet einige andere nützliche Werkzeuge und Primitive für asynchrone Programmierung.

2.9.3 Asynchronität des Tutorials

Um unser Tutorial asynchron zu machen, ändern Sie den „say_hello()“-Ereignishandler so, dass er wie folgt aussieht:

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

    payload = response.json()

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

In diesem Code gibt es nur 4 Änderungen gegenüber der vorherigen Version:

1. Die Methode ist als `async def` definiert, anstatt nur als `def`. Dies sagt Python, dass die Methode eine asynchrone Co-Routine ist.
2. Der Client, der erzeugt wird, ist ein asynchroner `AsyncClient()`, anstatt eines synchronen `Client()`. Dies teilt `httpx` mit, dass es im asynchronen Modus und nicht im synchronen Modus arbeiten soll.
3. Der zur Erstellung des Clients verwendete Kontextmanager ist als `async` markiert. Dies sagt Python, dass es eine Möglichkeit gibt, die Kontrolle abzugeben, wenn der Kontextmanager betreten und verlassen wird.
4. Der `get`-Aufruf wird mit einem `await`-Schlüsselwort versehen. Dies weist die Anwendung an, dass sie die Kontrolle an die Ereignisschleife abgeben kann, während wir auf die Antwort des Netzwerks warten.

Toga erlaubt es Ihnen, reguläre Methoden oder asynchrone Co-Routinen als Handler zu verwenden; Toga verwaltet alles hinter den Kulissen, um sicherzustellen, dass der Handler wie gewünscht aufgerufen oder erwartet wird.

Wenn Sie diese Änderungen speichern und die Anwendung erneut ausführen (entweder mit `briefcase dev` im Entwicklungsmodus oder indem Sie die gepackte Anwendung aktualisieren und erneut ausführen), wird es keine offensichtlichen Änderungen an der Anwendung geben. Wenn Sie jedoch auf die Schaltfläche klicken, um den Dialog auszulösen, können Sie eine Reihe von subtilen Verbesserungen feststellen:

- Die Schaltfläche kehrt in einen „nicht angeklickten“ Zustand zurück, anstatt in einem „angeklickten“ Zustand zu verharren.
- Das „Beachball“/“Spinner“-Symbol wird nicht angezeigt
- Wenn Sie das Anwendungsfenster verschieben/vergrößern, während Sie darauf warten, dass der Dialog erscheint, wird das Fenster neu gezeichnet.
- Wenn Sie versuchen, ein Anwendungsmenü zu öffnen, wird das Menü sofort angezeigt.

2.9.4 Nächste Schritte

Wir haben jetzt eine Anwendung, die glatt und reaktionsschnell ist, auch wenn sie auf eine langsame API wartet. Aber wie können wir sicherstellen, dass die Anwendung auch weiterhin funktioniert, wenn wir sie weiter entwickeln? Wie können wir unsere Anwendung testen? Schauen Sie sich *Tutorial 9* an, um das herauszufinden…

2.10 Tutorial 9 - Prüfzeiten

Bei der Softwareentwicklung wird meist kein neuer Code geschrieben, sondern bestehender Code geändert. Die Sicherstellung, dass vorhandener Code weiterhin so funktioniert, wie wir es erwarten, ist ein wichtiger Teil des Softwareentwicklungsprozesses. Eine Möglichkeit, das Verhalten unserer Anwendung sicherzustellen, ist eine *Testsuite*.

2.10.1 Ausführen der Testsuite

Es stellt sich heraus, dass unser Projekt bereits eine Testsuite hat! Als wir unser Projekt ursprünglich erzeugten, wurden zwei Verzeichnisse der obersten Ebene erzeugt: `src` und `tests`. Der Ordner `src` enthält den Code für unsere Anwendung; der Ordner `tests` enthält unsere Testsuite. Innerhalb des Ordners `tests` befindet sich eine Datei namens `test_app.py` mit folgendem Inhalt:

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

Dies ist ein Pytest *Testfall* - ein Codeblock, der ausgeführt werden kann, um ein bestimmtes Verhalten Ihrer Anwendung zu überprüfen. In diesem Fall ist der Test ein Platzhalter und testet nichts über unsere Anwendung - aber es ist ein Test, den wir durchführen können.

Wir können diese Testsuite mit der Option `--test` für `briefcase dev` ausführen. Da dies das erste Mal ist, dass wir Tests ausführen, müssen wir auch die Option `-r` übergeben, um sicherzustellen, dass die Testanforderungen auch installiert werden:

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
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

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

Erfolgreich! Wir haben soeben einen einzigen Test durchgeführt, der bestätigt, dass die Python-Mathematik so funktioniert, wie wir es erwartet haben (was für eine Erleichterung!).

Lassen Sie uns diesen Platzhaltertest durch einen Test ersetzen, der überprüft, ob sich unsere Methode `Gruß()` so verhält, wie wir es erwarten. Ersetzen Sie den Inhalt von `test_app.py` durch den folgenden:

```
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"
```

Dies definiert zwei neue Tests, die beiden erwarteten Verhaltensweisen überprüfen: die Ausgabe, wenn ein Name angegeben wird, und die Ausgabe, wenn der Name leer ist.

Wir können nun die Testsuite erneut ausführen. Diesmal brauchen wir die Option `-r` nicht, da die Testanforderungen bereits installiert wurden; wir müssen nur die Option `--test` verwenden:

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
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

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

Ausgezeichnet! Unsere Utility-Methode `Gruß()` funktioniert wie erwartet.

2.10.2 Testgetriebene Entwicklung

Jetzt, da wir eine Test-Suite haben, können wir sie nutzen, um die Entwicklung neuer Funktionen voranzutreiben. Ändern wir unsere Anwendung, um eine spezielle Begrüßung für einen bestimmten Benutzer zu erhalten. Wir können damit beginnen, indem wir einen Testfall für das neue Verhalten, das wir gerne sehen möchten, am Ende von `test_app.py` hinzufügen:

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

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

Führen Sie dann die Testsuite mit diesem neuen Test aus:

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!'
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
_____________________________________ 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 ==========================
```

Dieses Mal sehen wir einen Testfehler - und die Ausgabe erklärt die Quelle des Fehlers: der Test erwartet die Ausgabe „BeeWare the IDEs of Python!“, aber unsere Implementierung von `Gruß()` gibt „Hallo, Brutus“ zurück. Ändern wir die Implementierung von `Gruß()` in `src/helloworld/app.py`, um das neue Verhalten zu erhalten:

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

Wenn wir die Tests noch einmal durchführen, sehen wir, dass die Tests bestanden wurden:

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

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

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

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

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

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

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

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

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

2.10.3 Laufzeittests

Bislang haben wir die Tests im Entwicklungsmodus ausgeführt. Dies ist besonders nützlich, wenn Sie neue Funktionen entwickeln, da Sie schnell Tests und Code hinzufügen können, damit diese Tests erfolgreich sind. Irgendwann werden Sie jedoch sicherstellen wollen, dass Ihr Code auch als Anwendungspaket korrekt ausgeführt wird.

Die Optionen `--test` und `-r` können auch an den Befehl `run` übergeben werden. Wenn Sie `briefcase run --test -r` verwenden, wird die gleiche Testsuite ausgeführt, allerdings innerhalb des Anwendungspakets und nicht in Ihrer Entwicklungsumgebung:

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

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

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

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

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

Wie bei `briefcase dev --test` wird die Option `-r` nur bei der ersten Ausführung der Testsuite benötigt, um sicherzustellen, dass die Testabhängigkeiten vorhanden sind. Bei späteren Durchläufen können Sie diese Option weglassen.

Sie können auch die Option `--test` auf mobilen Backends verwenden: - so funktionieren `briefcase run iOS --test` und `briefcase run android --test`, die Testsuite auf dem von Ihnen ausgewählten mobilen Gerät ausführen.

2.10.4 Nächste Schritte

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 - Machen Sie diese Anwendung zu Ihrer eigenen

Bisher hat unsere App das Standardsymbol „graue Biene" verwendet. Wie können wir die App aktualisieren, um unser eigenes Symbol zu verwenden?

2.11.1 Hinzufügen eines Symbols

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...
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
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␣
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
↪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
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

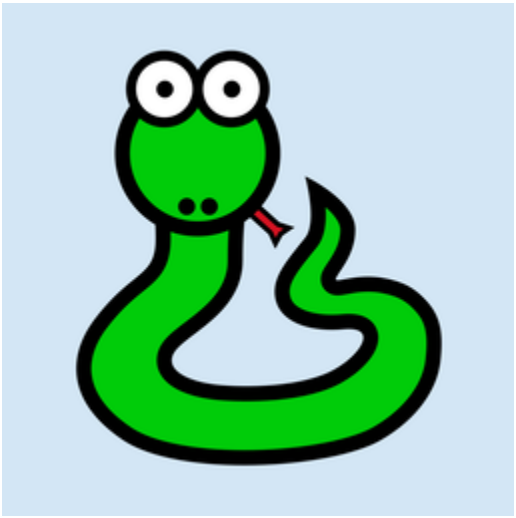
```
  [helloworld] Removing unneeded app content...
  Removing unneeded app bundle content... done

  [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.
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

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

Bemerkung: 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...
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
[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 Nächste Schritte

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.