BeeWare Documentation

Versão 0.1.dev147+gc9c3811

Russell Keith-Magee

06 mai., 2024

Conteúdo

Escreva em Python. Execute em qualquer lugar.

Bem-vindo ao BeeWare! Neste tutorial, vamos construir uma interface gráfica usando Python e implantá-la como um aplicativo para desktop, como um aplicativo móvel e como um aplicativo web de página única (SPA). Também veremos como você pode usar as ferramentas da BeeWare para realizar algumas das tarefas comuns que você precisará fazer como desenvolvedor de apps, por exemplo, testar seu aplicativo.

Traduções estão disponíveis

Se você não se sente confortável com o Português, as traduções deste tutorial estão disponíveis em Deutsch, Español, Français, Italiano, English, e .

CAPÍTULO 1

O que é o BeeWare?

O BeeWare não é um único produto, ferramenta ou biblioteca - é uma coleção de ferramentas e bibliotecas, cada uma das quais trabalha em conjunto para ajudar você a escrever aplicativos Python multiplataforma com uma interface gráfica nativa. Isso inclui:

- Toga, um kit de ferramentas de widget multiplataforma;
- Briefcase, uma ferramenta para empacotar projetos Python como pacotes distribuíveis que podem ser enviados aos usuários finais;
- Bibliotecas (como Rubicon ObjC) para acessar bibliotecas nativas da plataforma;
- Versões pré-compiladas do Python estão disponíveis para plataformas onde instaladores oficiais do Python não estão disponíveis.

Neste tutorial, estaremos utilizando todas essas ferramentas, mas como usuário, você só precisará interagir com as duas primeiras (Toga e Briefcase). No entanto, cada uma das ferramentas também pode ser usada individualmente - por exemplo, você pode usar o Briefcase para implantar um aplicativo sem usar o Toga como um kit de ferramentas de interface gráfica (GUI).

A suíte BeeWare está disponível no macOS, Windows, Linux (usando GTK); em plataformas móveis como Android e iOS; e para a Web.

CAPÍTULO 2

Vamos lá!

Pronto para experimentar o BeeWare por conta própria? *Vamos construir um aplicativo multiplataforma em Python!*

2.1 Tutorial 0 - Vamos nos preparar!

Antes de criarmos nosso primeiro aplicativo BeeWare, precisamos garantir que tenhamos todos os pré-requisitos para executar o BeeWare.

2.1.1 Instalando o Python

A primeira coisa que precisaremos é de um interpretador Python instalado e funcionando.

macOS

Linux

Windows

Se você estiver usando o macOS, uma versão recente do Python já está incluída no Xcode ou nas ferramentas de linha de comando. Para verificar se você já o possui, execute o seguinte comando:

```
$ python3 --version
```

Se o Python estiver instalado, você verá o número da versão. Caso contrário, será solicitado que instale as ferramentas de linha de comando para desenvolvedores.

Se estiver usando o Windows, você pode baixar o instalador oficial no site do Python. Recomendamos usar qualquer versão estável do Python a partir da 3.8. Evite versões alpha, beta e candidatas a lançamento, a menos que você esteja realmente familiarizado com essas versões de desenvolvimento.

Se você estiver usando Linux, será necessário instalar o Python por meio do gerenciador de pacotes do sistema (*apt* no Debian/Ubuntu/Mint, *dnf* no Fedora, ou *pacman* no Arch).

Certifique-se de que a versão do Python do sistema seja 3.8 ou mais recente. Se não for (por exemplo, o Ubuntu 18.04 é fornecido com o Python 3.6), será necessário atualizar a distribuição Linux para uma versão mais recente.

Para o Raspberry Pi, o suporte está limitado neste momento.

Se estiver usando o Windows, você pode baixar o instalador oficial no site do Python. Recomendamos usar qualquer versão estável do Python a partir da 3.8. Evite versões alpha, beta e candidatas a lançamento, a menos que você esteja realmente familiarizado com essas versões de desenvolvimento.

Distribuições Alternativas do Python

Existem várias maneiras diferentes de instalar o Python. Você pode instalar o Python através do homebrew. Pode usar o pyenv para gerenciar várias instalações do Python na mesma máquina. Usuários do Windows podem instalar o Python pela Windows App Store. Usuários com foco em ciência de dados podem preferir utilizar o Anaconda ou o Miniconda.

Se você estiver usando macOS ou Windows, não importa a *forma* que usou ao instalar o Python. o importante é ter um interpretador Python operando ao executar o comando `python3` no prompt de comando/terminal do seu sistema operacional.

Caso esteja no Linux, é recomendado utilizar o Python fornecido pelo seu sistema operacional. Embora seja possível concluir *a maior parte* deste tutorial utilizando uma versão não integrada ao sistema, você não conseguirá empacotar sua aplicação para distribuição a outros.

2.1.2 Instale as dependências

Em seguida, instale as dependências adicionais necessárias para o seu sistema operacional:

macOS

Linux

Windows

Para criar aplicativos BeeWare no macOS, é necessário:

- **Git**, um sistema de controle de versão. Ele está incluso no Xcode ou nas ferramentas de linha de comando, que você instalou acima.

Para dar suporte ao desenvolvimento local, será necessário instalar alguns pacotes do sistema. A lista de pacotes necessários pode variar de acordo com a sua distribuição:

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
```

Para criar aplicativos BeeWare no Windows, é necessário:

- **Git**, um sistema de controle de versão. Você pode baixar o Git em git-scm.org.

Após instalar essas ferramentas, é importante reiniciar qualquer sessão de terminal. No Windows, apenas os terminais iniciados *após* a conclusão da instalação mostrarão as ferramentas recém-instaladas.

2.1.3 Configurar um ambiente virtual

Agora vamos criar um ambiente virtual - uma «sandbox» que podemos usar para isolar nosso trabalho neste tutorial de nossa instalação principal do Python. Se instalarmos pacotes no ambiente virtual, nossa instalação principal do Python (e qualquer outro projeto Python em nosso computador) não será afetada. Se bagunçarmos completamente nosso ambiente virtual, podemos simplesmente excluí-lo e começar novamente, sem afetar nenhum outro projeto Python em nosso computador e sem a necessidade de reinstalar o Python.

macOS

Linux

Windows

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

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

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

Erros ao executar scripts do PowerShell

Se você estiver usando o PowerShell e receber o erro:

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

Sua conta do Windows não possui permissões para executar scripts. Para corrigir isso:

1. Execute o Windows PowerShell como Administrador.
2. Execute `set-executionpolicy RemoteSigned`
3. Selecione `Y` para alterar a política de execução.

Depois de fazer isso, você pode executar novamente o comando `beeware-venv\Scripts\activate.ps1` na mesma janela do PowerShell em que você fez as alterações nas permissões (a janela original) ou em uma nova janela do PowerShell no mesmo diretório.

Se isso funcionou, seu prompt deve estar diferente agora - ele deve ter um prefixo `(beeware-venv)`. Isso indica que você está atualmente no seu ambiente virtual BeeWare. Sempre que estiver trabalhando neste tutorial, certifique-se de que seu ambiente virtual esteja ativado. Se não estiver, execute novamente o último comando (o comando `activate`) para reativar seu ambiente.

Ambientes virtuais alternativos

Caso esteja usando o Anaconda ou Miniconda, você pode estar mais familiarizado com o uso de ambientes conda. Você também deve ter ouvido falar do `virtualenv`, um antecessor do módulo `venv` integrado ao Python. Da mesma forma que as instalações do Python - se você estiver no macOS ou Windows, não importa *como* você cria seu ambiente virtual, desde que tenha um. Se estiver no Linux, é recomendado usar o `venv` e o Python do sistema.

2.1.4 Próximos passos

Agora que configuramos nosso ambiente, estamos prontos para *criar nossa primeira aplicação BeeWare*.

2.2 Tutorial 1 - Seu primeiro app

Estamos prontos para criar nossa primeira aplicação.

2.2.1 Instale as ferramentas do BeeWare

Primeiro, precisamos instalar o **Briefcase**. Briefcase é uma ferramenta do BeeWare que pode ser usada para empacotar sua aplicação para distribuição aos usuários finais - mas também pode ser usada para iniciar um novo projeto. Certifique-se de estar no diretório `beeware-tutorial` que você criou no *Tutorial 0*, com o ambiente virtual `beeware-venv` ativado, e execute:

macOS

Linux

Windows

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

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

Possíveis erros durante a instalação

Se você encontrar erros durante a instalação, é bem provável que alguns dos requisitos do sistema não estejam instalados. Certifique-se de ter *instalado todas os pré-requisitos da plataforma*.

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

Possíveis erros durante a instalação

É importante usar o comando `python -m pip`, em vez de apenas `pip`. Isso porque o Briefcase precisa garantir que as versões do `pip` e `setuptools` estejam atualizadas, pois o `pip` usado sozinho não consegue se atualizar. Se quiser entender melhor o motivo, o Brett Cannon escreveu um post detalhado no blog dele.

Uma das ferramentas do BeeWare é o Briefcase. O Briefcase pode ser usado para empacotar sua aplicação para distribuição aos usuários finais - mas ele também pode ser usado para iniciar um novo projeto.

2.2.2 Iniciar um novo projeto

Vamos começar nosso primeiro projeto BeeWare! Usaremos o comando `new` do Briefcase para criar um aplicativo chamado **Hello World**. Execute o seguinte comando no seu prompt de comando/terminal:

macOS

Linux

Windows

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

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

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

O Briefcase solicitará alguns detalhes sobre nosso novo aplicativo. Para os propósitos deste tutorial, utilize o seguinte:

- **Formal Name** - Aceite o valor padrão: `Hello World`.
- **App Name** - Aceite o valor padrão: `helloworld`.
- **Bundle** - Se você possui seu próprio domínio, insira esse domínio invertido. (Por exemplo, se você possui o domínio «cupcakes.com», insira `com.cupcakes` como nome do bundle). Se você não possui seu próprio domínio, aceite o padrão (`com.example`).
- **Project Name** - Aceite o valor padrão: `Hello World`.
- **Description** - Aceite o valor padrão (ou, se quiser ser realmente criativo, crie sua própria descrição!)
- **Author** - Insira seu próprio nome.
- **Author's email** - Insira seu próprio endereço de e-mail. Ele será usado no arquivo de configuração, em textos de ajuda e em qualquer lugar onde um e-mail seja necessário ao enviar o aplicativo para uma loja de aplicativos.
- **URL** - A URL da página inicial do seu aplicativo. Novamente, se você possui seu próprio domínio, insira uma URL nesse domínio (incluindo o `https://`). Caso contrário, apenas aceite a URL padrão (`https://example.com/helloworld`). Esta URL não precisa realmente existir (por enquanto); ela será usada apenas se você publicar sua aplicação em uma loja de aplicativos.
- **License** - Aceite o valor padrão (BSD). Isso não afetará nada sobre o funcionamento do tutorial. Se você tem preferências particulares em relação à escolha de licença, sinta-se à vontade para escolher outra licença.
- **GUI framework** - Aceite a opção padrão, Toga (a própria biblioteca de interface gráfica da BeeWare).

O Briefcase então gerará uma estrutura básica de projeto para você usar. Se você seguiu este tutorial até agora e aceitou as configurações padrão como descrito, seu sistema de arquivos deve se parecer com algo assim:

```
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
```

Essa estrutura atual já é um aplicativo totalmente funcional sem adicionar mais nada. A pasta `src` contém todo o código do aplicativo, a pasta `tests` contém uma suíte de testes inicial, e o arquivo `pyproject.toml` descreve como empacotar o aplicativo para distribuição. Se você abrir o `pyproject.toml` em um editor, verá os detalhes de configuração que você acabou de fornecer ao Briefcase.

Agora que temos um aplicativo inicial, podemos usar o Briefcase para executar a aplicação.

2.2.3 Inicie o app no modo de desenvolvedor

Vá para o diretório do projeto `helloworld` e peça ao Briefcase para iniciar o projeto no modo de Desenvolvedor (ou `dev`):

macOS

Linux

Windows

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

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

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

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

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

(continues on next page)

(continuação da página anterior)

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

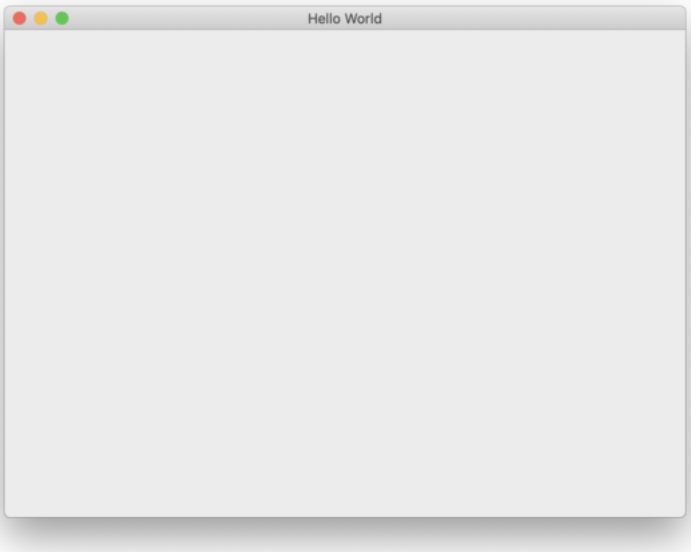
```
(beeware-venv) C:\...>cd helloworld
(beeware-venv) C:\...>briefcase dev

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

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

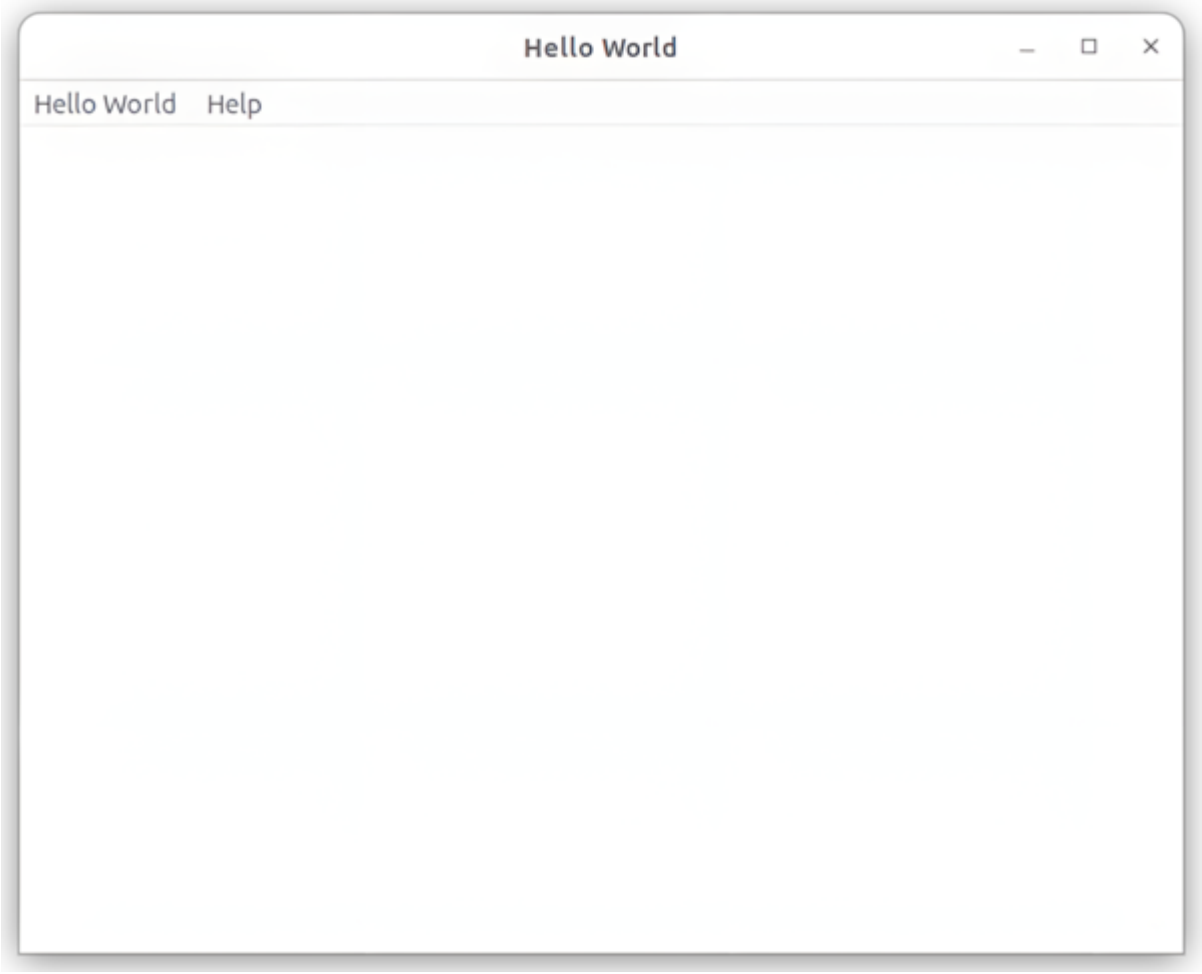
Isso deve abrir uma janela GUI:

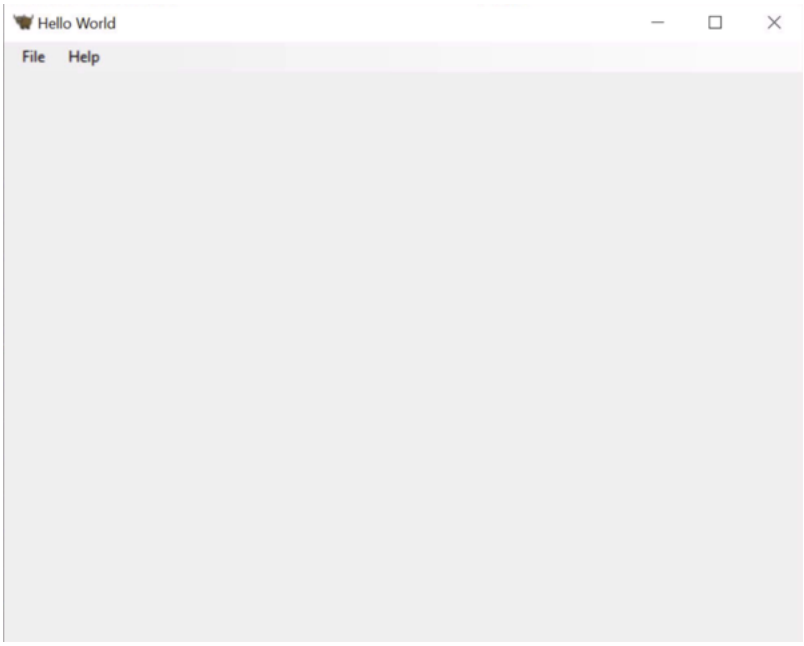
macOS

Linux

Windows







Pressione o botão de fechar (ou selecione `Sair` no menu do aplicativo) e pronto! Parabéns - você acabou de criar um aplicativo autônomo e nativo em Python!

2.2.4 Próximos passos

Agora temos um aplicativo funcional, em execução no modo de desenvolvedor. Podemos adicionar alguma lógica própria para tornar nosso aplicativo um pouco mais interessante. No *Tutorial 2*, vamos colocar uma interface de usuário mais útil em nossa aplicação.

2.3 Tutorial 2 - Tornando as coisas interessantes

No *Tutorial 1*, geramos um projeto inicial que executava, mas não escrevemos nenhum código por conta própria. Vamos dar uma olhada no que foi criado para nós.

2.3.1 O que foi gerado

No diretório `src/helloworld`, você deverá ver 3 arquivos: `__init__.py`, `__main__.py` e `app.py`.

`__init__.py` torna o diretório `helloworld` um módulo Python importável. Trata-se de um arquivo vazio; sua simples existência indica ao interpretador Python que há um módulo definido pelo diretório *helloworld*.

`__main__.py` define o módulo `helloworld` como um tipo especial de módulo - um módulo executável. Ao tentar executar o módulo `helloworld` usando `python -m helloworld`, o Python iniciará a execução no arquivo `__main__.py`. O conteúdo de `__main__.py` é relativamente simples:

```
from helloworld.app import main

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

Isto é - ele importa o método `main` do aplicativo `helloworld`; quando executado como ponto de entrada, esse método main() é chamado, dando início ao loop principal da aplicação. Nesse loop, a aplicação de interface gráfica (GUI) espera pelas interações do usuário (como cliques do mouse e pressionamentos de teclas).

O arquivo mais interessante é o `app.py` - nele está contido a lógica que cria a janela da nossa aplicação:

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

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

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

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

Vamos analisar isso linha por linha:

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

Primeiro, importamos a biblioteca de widgets `toga`, juntamente com algumas classes e constantes que são úteis para o estilo. Por enquanto, nosso código ainda não faz uso delas, mas em breve começaremos a utilizá-las.

Em seguida, definimos uma classe:

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

Cada aplicação Toga possui uma única instância chamada `toga.App`, que representa o aplicativo em si sendo executado. A aplicação pode acabar gerenciando várias janelas; mas para aplicativos simples, haverá uma única janela principal.

Depois, definimos um método chamado `startup()`:

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

A primeira coisa que o método startup() faz é definir um container principal. O esquema de layout do Toga se comporta de maneira semelhante ao HTML. A construção de uma aplicação se dá por meio da criação de uma coleção de containers, cada um contendo outros containers ou os widgets propriamente dito. Depois, você aplica estilos a esses containers para definir como vão usar o espaço na janela.

Neste aplicativo, definimos um único container, mas não colocamos nada dentro dele.

Em seguida, definimos uma janela na qual podemos colocar o container vazio:

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

Isso cria uma instância de `toga.MainWindow`, que terá um título correspondente ao nome do aplicativo. A MainWindow (Janela Principal) é um tipo especial de janela no Toga, pois está intimamente ligada ao ciclo de vida da aplicação. Quando a MainWindow é fechada, a aplicação também se encerra. Além disso, a MainWindow também é a janela que contém o menu do aplicativo (se você estiver em uma plataforma como o Windows, onde as barras de menu fazem parte da janela)

Então, adicionamos nosso container vazio como conteúdo da janela principal e instruímos o aplicativo a mostrar nossa janela:

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

Por último, definimos um método chamado `main()`. Isso é o que cria a instância de nossa aplicação:

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

O método `main()` é aquele que é importado e chamado pelo arquivo `__main__.py`. Ele cria e retorna uma instância da nossa aplicação `HelloWorld`.

Essa é a aplicação Toga mais básica possível. Vamos agora integrar conteúdo personalizado à aplicação e torná-la mais interessante.

2.3.2 Adicionando conteúdo personalizado

Modifique sua classe `HelloWorld` dentro de `src/helloworld/app.py` para que fique assim:

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

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

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

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

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

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

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

Nota: Não remova as importações já feitas no topo do arquivo ou o `main()` ao final. Você só precisa atualizar a classe `HelloWorld`.

Vamos ver em detalhes o que mudou.

Ainda estamos fazendo o container principal; porém, agora estamos aplicando um estilo:

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

O Toga possui um sistema de layout interno denominado «Pack». Seu funcionamento se assemelha ao CSS. Você define objetos em uma hierarquia - no HTML, os objetos são e , e outros elementos do DOM; enquanto no Toga, são widgets e containers. Estilos podem ser atribuídos a cada elemento individualmente. Neste caso específico, estamos indicando que o container é um `'COLUMN'` - isto é, ele ocupará toda a largura disponível e aumentará sua altura à medida que conteúdo for adicionado, porém buscando sempre ser o mais compacto possível.

A seguir, vamos definir dois widgets:

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

(continues on next page)

(continuação da página anterior)

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

Nesta etapa, definimos um Label e um TextInput. Ambos os widgets possuem estilos associados; o Label terá 5px de preenchimento à esquerda e à direita, e nenhum preenchimento na parte superior e inferior. O TextInput é marcado como flexível, ou seja, ele absorverá todo o espaço disponível em seu eixo de layout.

O TextInput é atribuído como uma variável de instância da classe. Isso nos permite acessar facilmente a instância do widget, o que será utilizado em breve.

Após isso, definimos um container para colocar esses dois widgets:

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

A container `name_box` é parecido com o container principal, só que agora ele é do tipo `ROW`. Isso significa que o conteúdo será adicionado horizontalmente e tentará ficar o mais estreito possível. Esse container também possui preenchimento - 5px em todos os lados.

Agora definimos um botão:

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

O botão também possui 5px de preenchimento em todos os lados. Além disso, definimos um *gatilho* - um método a ser invocado quando o botão for pressionado.

Então, adicionamos o container name_box e o botão ao container principal:

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

Com isso, finalizamos o layout; o restante do código do método startup() permanece como antes - definindo uma MainWindow (Janela principal) e a atribuição do container principal como conteúdo dessa janela:

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

A última etapa consiste na definição do gatilho do botão. Um gatilho pode ser qualquer método, gerador ou corrotina assíncrona; ele aceita como argumento o widget que gerou o evento e será invocado sempre que o botão for pressionado:

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

O corpo do método é uma simples instrução de impressão. No entanto, ele consultará o valor atual do campo de entrada de nome e usará esse conteúdo como o texto a ser impresso.

Ao concluirmos essas alterações, podemos visualizar o resultado reiniciando a aplicação. Seguindo o procedimento anterior, utilizaremos o modo de desenvolvedor:

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

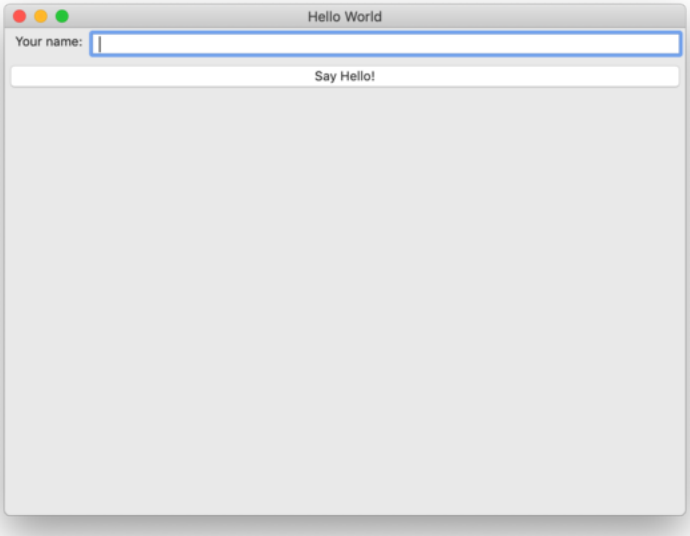
Dessa vez, você vai reparar que as dependências *não são* instaladas. O Briefcase consegue identificar se o app já foi aberto antes e, pra economizar tempo, só executa ele. Se você adicionar novas dependências, pode garantir que elas sejam instaladas usando a opção `-r` quando executar o comando `briefcase dev`.

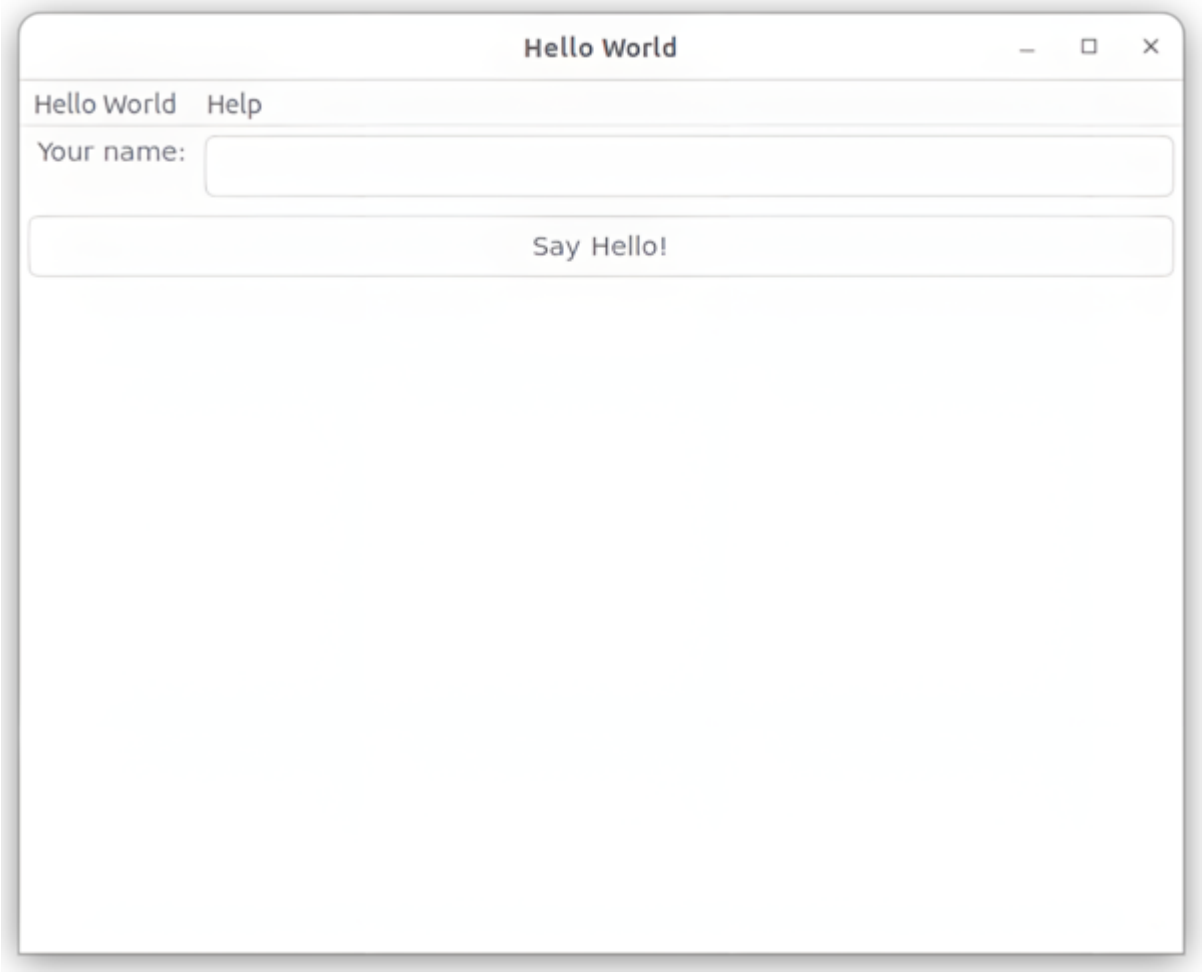
Isso deve abrir uma janela GUI:

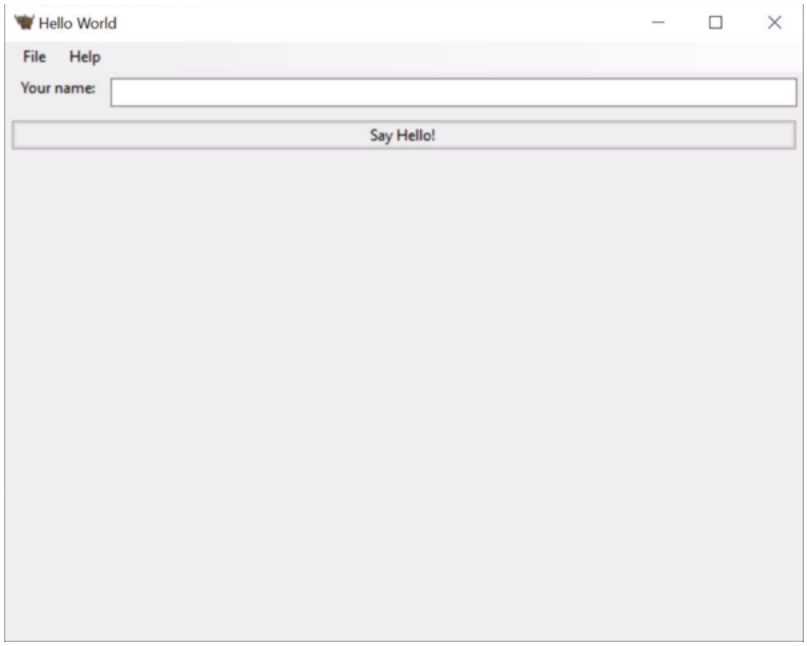
macOS

Linux

Windows







Se você escrever um nome na caixa de texto e clicar no botão da interface, vai ver uma mensagem aparecer no console onde você abriu a aplicação.

2.3.3 Próximos passos

Criamos uma aplicação com recursos mais interessantes. Porém, ela somente funciona em nosso computador. Vamos empacotar esta aplicação para distribuição. No *Tutorial 3*, encapsularemos nossa aplicação em um instalador independente, que poderemos enviar a amigos, clientes ou publicar em uma App Store.

2.4 Tutorial 3 - Empacotando para Distribuição

Até o momento, executamos nossa aplicação no «modo de desenvolvedor». Isso facilita a execução da nossa aplicação localmente - mas o que realmente queremos é poder fornecer nossa aplicação para outras pessoas.

Entretanto, não queremos ter que ensinar aos nossos usuários como instalar o Python, criar um ambiente virtual, clonar um repositório git e executar o Briefcase no modo de desenvolvedor. Queremos é fornecer um instalador, garantindo a execução imediata da aplicação.

O Briefcase pode ser usado para empacotar sua aplicação para distribuição dessa maneira.

2.4.1 Criando a estrutura da sua aplicação

Como esta é a primeira vez que estamos empacotando nossa aplicação, precisamos criar alguns arquivos de configuração e outras estruturas de suporte ao processo de empacotamento. No diretório `helloworld`, execute:

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
```

É provável que você tenha observado uma grande quantidade de texto passando pelo seu terminal… mas o que exata-

mente aconteceu? O Briefcase fez o seguinte:

1. Ele **gerou um modelo de aplicação** (template). Existem muitos arquivos e configurações necessárias para construir um instalador nativo, além do código da sua aplicação. Essa estrutura extra é quase igual para todas as aplicações na mesma plataforma, exceto pelo nome da aplicação que está sendo construída - um modelo de aplicação (template) é fornecido pelo Briefcase para cada plataforma que ele suporta. Esta etapa implementa o modelo, substituindo o nome da sua aplicação, ID do pacote e outras propriedades do seu arquivo de configuração conforme necessário para suportar a plataforma na qual você está construindo.

   Caso o modelo fornecido pelo Briefcase não seja satisfatório, é possível você fornecer um modelo próprio. Entretanto, é recomendável que isso não seja feito antes de se obter maior familiaridade com o modelo padrão do Briefcase.

2. Ele **baixou e instalou um pacote de suporte**. A abordagem de empacotamento adotada pelo Briefcase é melhor descrita como «a coisa mais simples que poderia funcionar» - ele inclui um interpretador Python completo e isolado como parte de cada aplicativo criado. Isso apresenta uma ligeira ineficiência em termos de espaço - se você tiver 5 aplicativos empacotados com o Briefcase, terá 5 cópias do interpretador Python. No entanto, essa abordagem garante que cada aplicação seja completamente independente, usando uma versão específica do Python que comprovadamente funciona com tal aplicação.

   Novamente, o Briefcase fornece um pacote de suporte padrão para cada plataforma; entretanto, caso deseje, você pode fornecer seu próprio pacote de suporte e incluí-lo como parte do processo de construção. Isso pode ser útil se você necessita de opções específicas habilitadas no interpretador Python, ou se deseja remover módulos da biblioteca padrão que não são necessários em tempo de execução.

   O Briefcase mantém um cache local de pacotes de suporte. Portanto, uma vez que você tenha baixado um pacote de suporte específico, essa cópia armazenada será utilizada em compilações futuras.

3. Ele **instalou os requisitos da aplicação**. Seu aplicativo pode especificar quaisquer módulos de terceiros necessários em tempo de execução. Estes serão instalados utilizando o `pip` no instalador da sua aplicação.

4. Ele **instalou o código da sua aplicação**. Sua aplicação terá seu próprio código e recursos (por exemplo, imagens necessárias em tempo de execução); estes arquivos são copiados para o instalador.

5. Ele **instalou os recursos requeridos pela sua aplicação.** Por fim, são adicionados recursos adicionais requeridos pelo próprio instalador. Isso inclui elementos como ícones que serão anexados à aplicação final e imagens de tela de abertura.

Assim que concluído, ao verificar o diretório do projeto, você deverá encontrar um diretório correspondente à sua plataforma (`macOS`, `linux` ou `windows`) contendo arquivos adicionais. Esta é a configuração de empacotamento específica para a plataforma da sua aplicação.

2.4.2 Compilando sua aplicação

Agora você pode compilar sua aplicação. Este passo executa qualquer compilação binária necessária para que sua aplicação seja executável na plataforma de destino.

macOS

Linux

Windows

```
(beeware-venv) $ briefcase build

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

(continues on next page)

(continuação da página anterior)

```
 100.0% • 00:07

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

No macOS, o comando `build` não necessita *compilar* nada, mas precisa assinar o conteúdo do binário para que ele possa ser executado. Essa assinatura é do tipo *ad hoc* - só funcionará no *seu* computador. Se você deseja distribuir a aplicação para outras pessoas, será necessário fornecer uma assinatura completa.

```
(beeware-venv) $ briefcase build

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

Targeting glibc 2.35
Targeting Python3.10

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

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

Após a conclusão dessa etapa, a pasta `build` conterá uma pasta `helloworld-0.0.1` com uma estrutura semelhante ao sistema de arquivos `/usr` do Linux. Ela conterá uma pasta `bin` com um binário `helloworld`, além das pastas `lib` e `share` necessárias para dar suporte ao binário.

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

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

No Windows, o comando `build` não precisa *compilar* nada, mas é preciso registrar alguns metadados para que a aplicação reconheça seu nome, versão e outras coisas assim.

Notificação de antivírus

Como esses metadados estão sendo gravados diretamente no binário pré-compilado que sai do modelo durante o comando `create`, isso pode acionar o software antivírus em execução no computador e impedir que os metadados sejam gravados. Nesse caso, instrua o antivírus a permitir que a ferramenta (denominada `rcedit-x64.exe`) seja executada

e execute novamente o comando acima.

2.4.3 Rodando sua aplicação

Agora você pode utilizar o Briefcase para executar sua aplicação:

macOS

Linux

Windows

```
(beeware-venv) $ briefcase run

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

```
(beeware-venv) $ briefcase run

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

Targeting glibc 2.35
Targeting Python3.10

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

(continues on next page)

(continuação da página anterior)

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

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

[helloworld] Starting app...

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

Isso iniciará a execução de seu aplicativo nativo, usando a saída do comando `build`.

Você pode notar algumas pequenas diferenças na aparência do seu aplicativo quando ele estiver em execução. Por exemplo, os ícones e o nome exibidos pelo sistema operacional podem ser ligeiramente diferentes daqueles vistos durante a execução no modo de desenvolvedor. Isso também ocorre porque você está usando o aplicativo empacotado, e não apenas executando o código Python. Do ponto de vista do sistema operacional, agora você está executando «um aplicativo», não «um programa Python», e isso se reflete na aparência do aplicativo.

2.4.4 Criando o seu instalador

Agora você pode empacotar seu aplicativo para distribuição, usando o comando `package`. O comando package faz qualquer compilação necessária para converter o projeto de andaime em um produto final e distribuível. Dependendo da plataforma, isso pode envolver a compilação de um instalador, a execução de assinatura de código ou outras tarefas de pré-distribuição.

macOS

Linux

Windows

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

[helloworld] Signing app...

```

(continues on next page)

(continuação da página anterior)

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

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

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

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

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

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

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

A pasta `dist` conterá um arquivo chamado `Hello World-0.0.1.dmg`. Se você localizar esse arquivo no Finder e clicar duas vezes em seu ícone, montará o DMG, fornecendo uma cópia do aplicativo Hello World e um link para a pasta Applications para facilitar a instalação. Arraste o arquivo do aplicativo para Applications e você terá instalado o aplicativo. Envie o arquivo DMG para um amigo e ele poderá fazer o mesmo.

Neste exemplo, usamos a opção `--adhoc-sign`, ou seja, estamos assinando nosso aplicativo com credenciais *ad hoc*, credenciais temporárias que só funcionarão em seu computador. Fizemos isso para manter a simplicidade do tutorial. A configuração de identidades de assinatura de código é um pouco complicada, e elas só são *necessárias* se você pretende distribuir o aplicativo para outras pessoas. Se estivéssemos publicando um aplicativo real para ser usado por outras pessoas, precisaríamos especificar credenciais reais.

Quando estiver pronto para publicar um aplicativo real, consulte o guia de instruções da Briefcase sobre Configuração de uma identidade de assinatura de código do macOS

A saída da etapa do pacote será ligeiramente diferente, dependendo de sua distribuição Linux. Se estiver em uma distribuição derivada do Debian, você verá:

```
(beeware-venv) $ briefcase package

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

Targeting glibc 2.35
Targeting Python3.10

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

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

(continues on next page)

(continuação da página anterior)

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

A pasta `dist` conterá o arquivo `.deb` que foi gerado.

Se estiver em uma distribuição baseada em RHEL, você verá:

```
(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
```

A pasta `dist` conterá o arquivo `.rpm` que foi gerado.

Se estiver em uma distribuição baseada no Arch, você verá:

```
(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
```

A pasta `dist` conterá o arquivo `.pkg.tar.zst` que foi gerado.

No momento, não há suporte para o empacotamento de outras distribuições do Linux.

Se quiser criar um pacote para uma distribuição Linux diferente da que você está usando, o Briefcase também pode ajudar, mas você precisará instalar o Docker.

Os instaladores oficiais do Docker Engine estão disponíveis para uma série de distribuições Unix. Siga as instruções para sua plataforma; no entanto, certifique-se de não instalar o Docker no modo «sem raiz».

Depois de instalar o Docker, você deve ser capaz de iniciar um contêiner Linux - por exemplo:

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

mostrará um prompt do Unix (algo como `root@84444e31cff9:/#`) dentro de um contêiner do Docker do Ubuntu 22.04. Digite Ctrl-D para sair do Docker e retornar ao seu shell local.

Depois de instalar o Docker, você pode usar o Briefcase para criar um pacote para qualquer distribuição Linux compatível com o Briefcase, passando uma imagem do Docker como argumento. Por exemplo, para criar um pacote DEB para o Ubuntu 22.04 (Jammy), independentemente do sistema operacional em que você estiver, execute:

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

Isso fará o download da imagem do Docker para o sistema operacional selecionado, criará um contêiner capaz de executar as compilações do Briefcase e compilará o pacote do aplicativo dentro da imagem. Após a conclusão, a pasta `dist` conterá o pacote para a distribuição Linux de destino.

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

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

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

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

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

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

Linking application installer...
Linking... done

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

Neste exemplo, usamos a opção `--adhoc-sign`, ou seja, estamos assinando nosso aplicativo com credenciais *ad hoc*, credenciais temporárias que só funcionarão em seu computador. Fizemos isso para manter a simplicidade do tutorial. A configuração de identidades de assinatura de código é um pouco complicada, e elas só são *necessárias* se você pretende distribuir o aplicativo para outras pessoas. Se estivéssemos publicando um aplicativo real para ser usado por outras pessoas, precisaríamos especificar credenciais reais.

Quando estiver pronto para publicar um aplicativo real, consulte o guia de instruções da Briefcase sobre Configuração de uma identidade de assinatura de código do macOS

Quando essa etapa for concluída, a pasta `dist` conterá um arquivo chamado `Hello_World-0.0.1.msi`. Se você clicar duas vezes nesse instalador para executá-lo, deverá passar por um processo familiar de instalação do Windows. Após

a conclusão da instalação, haverá uma entrada «Hello World» em seu menu Iniciar.

2.4.5 Próximos passos

Agora temos nosso aplicativo empacotado para distribuição em plataformas de desktop. Mas o que acontece quando precisamos atualizar o código em nosso aplicativo? Como podemos colocar essas atualizações em nosso aplicativo empacotado? Consulte o *Tutorial 4* para descobrir...

2.5 Tutorial 4 - Atualizando sua aplicação

No último tutorial, empacotamos nosso aplicativo como um aplicativo nativo. Se estiver lidando com um aplicativo do mundo real, esse não será o fim da história - você provavelmente fará alguns testes, descobrirá problemas e precisará fazer algumas alterações. Mesmo que o seu aplicativo seja perfeito, você eventualmente desejará publicar a versão 2 do seu aplicativo com melhorias.

Então, como você atualiza o aplicativo instalado quando faz alterações no código?

2.5.1 Atualização do código do aplicativo

Atualmente, nosso aplicativo imprime no console quando você pressiona o botão. No entanto, os aplicativos de GUI não devem realmente usar o console para saída. Eles precisam usar caixas de diálogo para se comunicar com os usuários.

Vamos adicionar uma caixa de diálogo para dizer olá, em vez de escrever no console. Modifique o retorno de chamada `say_hello` para que ele tenha a seguinte aparência:

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

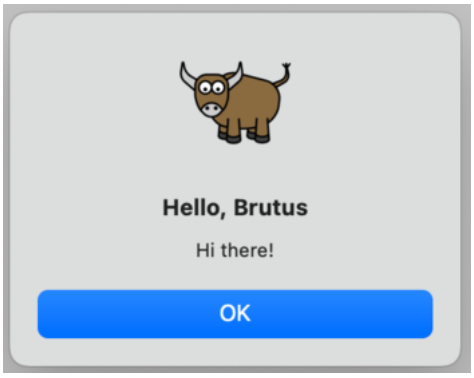
Isso instrui o Toga a abrir uma caixa de diálogo modal quando o botão é pressionado.

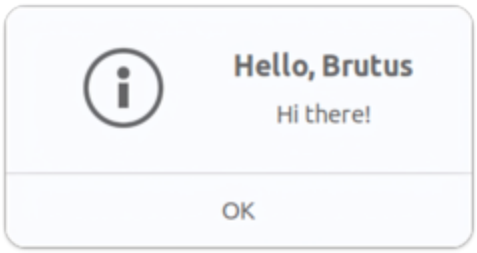
Se você executar `briefcase dev`, digitar um nome e pressionar o botão, verá a nova caixa de diálogo:

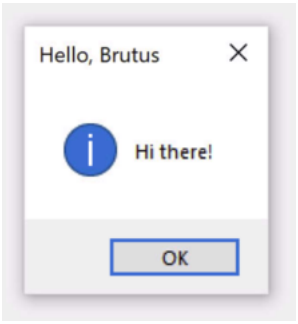
macOS

Linux

Windows







No entanto, se você executar o `briefcase run`, a caixa de diálogo não será exibida.

Por que isso acontece? Bem, o `briefcase dev` opera executando seu código no local - ele tenta produzir o ambiente de tempo de execução mais realista possível para seu código, mas não fornece nem usa nenhuma infraestrutura da plataforma para empacotar seu código como um aplicativo. Parte do processo de empacotamento do aplicativo envolve a cópia do código *para* o pacote de aplicativos e, no momento, o aplicativo ainda contém o código antigo.

Portanto, precisamos dizer ao briefcase para atualizar seu aplicativo, copiando a nova versão do código. Poderíamos fazer isso excluindo o diretório da plataforma antiga e começando do zero. No entanto, o Briefcase oferece uma maneira mais fácil: você pode atualizar o código do seu aplicativo empacotado existente:

macOS

Linux

Windows

```
(beeware-venv) $ briefcase update

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

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

[helloworld] Application updated.
```

```
(beeware-venv) $ briefcase update

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

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

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

(continues on next page)

(continuação da página anterior)

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

[helloworld] Application updated.
```

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

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

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

[helloworld] Application updated.
```

Se o Briefcase não conseguir encontrar o modelo de andaime, ele invocará automaticamente o `create` para gerar um novo andaime.

Agora que atualizamos o código do instalador, podemos executar o `briefcase build` para recompilar o aplicativo, o `briefcase run` para executar o aplicativo atualizado e o `briefcase package` para reembalar o aplicativo para distribuição.

(Usuários do macOS, lembrem-se de que, conforme observado em *Tutorial 3*, para o tutorial, recomendamos executar o `briefcase package` com o sinalizador `--adhoc-sign` para evitar a complexidade da configuração de uma identidade de assinatura de código e manter o tutorial o mais simples possível)

2.5.2 Atualize e execute em uma única etapa

Se estiver iterando rapidamente as alterações no código, provavelmente desejará fazer uma alteração no código, atualizar o aplicativo e executá-lo novamente de imediato. Para a maioria das finalidades, o modo de desenvolvedor (`briefcase dev`) será a maneira mais fácil de fazer esse tipo de iteração rápida; no entanto, se estiver testando algo sobre como o aplicativo é executado como um binário nativo ou procurando um bug que só se manifesta quando o aplicativo está no formato empacotado, talvez seja necessário usar chamadas repetidas para `briefcase run`. Para simplificar o processo de atualização e execução do aplicativo empacotado, o Briefcase tem um atalho para suportar esse padrão de uso - a opção `-u` (ou `--update`) no comando `run`.

Vamos tentar fazer outra alteração. Você deve ter notado que, se não digitar um nome na caixa de entrada de texto, a caixa de diálogo dirá «Hello, «. Vamos modificar a função `say_hello` novamente para lidar com esse caso extremo.

Na parte superior do arquivo, entre as importações e a definição da `classe HelloWorld`, adicione um método utilitário para gerar uma saudação apropriada, dependendo do valor do nome que foi fornecido:

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

Em seguida, modifique a chamada de retorno `say_hello` para usar esse novo método utilitário:

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

Execute seu aplicativo no modo de desenvolvimento (com `briefcase dev`) para confirmar que a nova lógica funciona; em seguida, atualize, crie e execute o aplicativo com um único comando:

macOS

Linux

Windows

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

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

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

[helloworld] Application updated.

[helloworld] Building application...
...

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

[helloworld] Starting app...
```

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

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

Targeting glibc 2.35
Targeting Python3.10

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

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

[helloworld] Application updated.

[helloworld] Building application...
...

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

[helloworld] Starting app...
```

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

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

(continues on next page)

(continuação da página anterior)

```
Installing src/helloworld... done

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

[helloworld] Application updated.

[helloworld] Starting app...
```

O comando package também aceita o argumento `-u`, portanto, se você fizer uma alteração no código do aplicativo e quiser reempacotar imediatamente, poderá executar `briefcase package -u`.

2.5.3 Próximos passos

Agora temos nosso aplicativo empacotado para distribuição em plataformas de desktop e conseguimos atualizar o código em nosso aplicativo.

Mas e quanto ao celular? No *Tutorial 5*, converteremos nosso aplicativo em um aplicativo móvel e o implantaremos em um simulador de dispositivo e em um telefone.

2.6 Tutorial 5 - Levando para o Mobile

Até agora, executamos e testamos nosso aplicativo no desktop. No entanto, o BeeWare também oferece suporte a plataformas móveis, e o aplicativo que escrevemos também pode ser implantado em seu dispositivo móvel!

iOS os aplicativos iOS só podem ser compilados no macOS.

Vamos criar nosso aplicativo para iOS!

Android Os aplicativos Android podem ser compilados no macOS, Windows ou Linux.

Vamos criar nosso aplicativo para Android!

2.6.1 Tutorial 5 - Levando para o Mobile: iOS

Para compilar aplicativos iOS, precisaremos do Xcode, que está disponível gratuitamente na macOS App Store <https://apps.apple.com/au/app/xcode/id497799835?mt=12>`__.

Depois de instalar o Xcode, podemos pegar nosso aplicativo e implantá-lo como um aplicativo iOS.

O processo de implantação de um aplicativo no iOS é muito semelhante ao processo de implantação como um aplicativo de desktop. Primeiro, você executa o comando `create`, mas, desta vez, especificamos que queremos criar um aplicativo iOS:

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

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

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

(continues on next page)

(continuação da página anterior)

```
...

[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
```

Quando isso for concluído, teremos um diretório `build/helloworld/ios/xcode` contendo um projeto Xcode, bem como as bibliotecas de suporte e o código do aplicativo necessários para o aplicativo.

Em seguida, você pode usar o Briefcase para compilar o aplicativo usando `briefcase build iOS`:

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

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

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

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

Agora estamos prontos para executar nosso aplicativo, usando o `briefcase run iOS`. Será solicitado que você selecione um dispositivo para o qual compilar; se você tiver simuladores para várias versões do SDK do iOS instalados, também poderá ser perguntado qual versão do iOS deseja usar como alvo. As opções mostradas a você podem ser diferentes das opções mostradas nesta saída - no mínimo, a lista de dispositivos provavelmente será diferente. Para nossos propósitos, não importa qual simulador você escolherá.

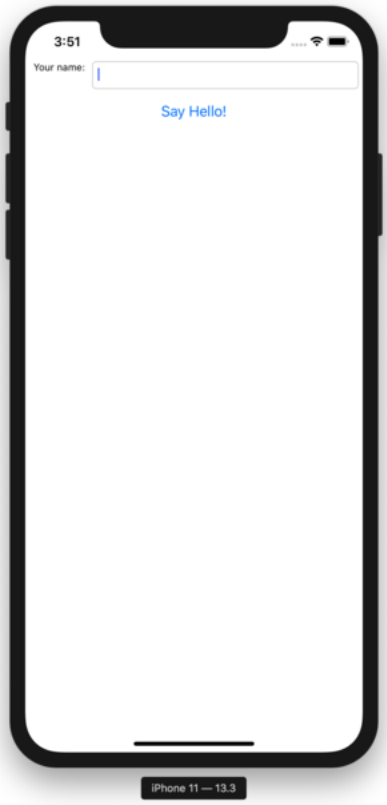
```
(beeware-venv) $ briefcase run iOS

Select simulator device:

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

(continues on next page)

(continuação da página anterior)

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

Isso iniciará o simulador do iOS, instalará seu aplicativo e o iniciará. Você deverá ver o simulador iniciar e, por fim, abrir seu aplicativo iOS:



Se você souber antecipadamente qual simulador de iOS deseja usar, poderá dizer ao Briefcase para usar esse simulador fornecendo uma opção `-d` (ou `--device`). Usando o nome do dispositivo que você selecionou ao criar o aplicativo, execute:

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

Se você tiver várias versões do iOS disponíveis, o Briefcase escolherá a versão mais alta do iOS; se você quiser escolher uma versão específica do iOS, diga a ele para usar essa versão específica:

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

Ou você pode nomear um UDID de dispositivo específico:

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

Próximos passos

Agora temos um aplicativo em nosso telefone! Há algum outro lugar onde possamos implantar um aplicativo BeeWare? Consulte *Tutorial 6* para descobrir...

2.6.2 Tutorial 5 - Levando para o Mobile: Android

Agora, vamos pegar nosso aplicativo e implantá-lo como um aplicativo Android.

O processo de implementação de um aplicativo no Android é muito semelhante ao processo de implementação como um aplicativo de desktop. O Briefcase lida com a instalação de dependências para o Android, incluindo o SDK do Android, o emulador do Android e um compilador Java.

Criar um aplicativo Android e compilá-lo

Primeiro, execute o comando `create`. Isso faz o download de um modelo de aplicativo Android e adiciona seu código Python a ele.

macOS

Linux

Windows

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continuação da página anterior)

```
...

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Quando você executa o `briefcase create android` pela primeira vez, o Briefcase faz o download do Java JDK e do Android SDK. O tamanho dos arquivos e o tempo de download podem ser consideráveis; isso pode demorar um pouco (10 minutos ou mais, dependendo da velocidade da sua conexão com a Internet). Quando o download for concluído, você será solicitado a aceitar a licença do Android SDK do Google.

Quando isso for concluído, teremos um diretório `build\helloworld\android\gradle` em nosso projeto, que conterá um projeto Android com uma configuração de compilação do Gradle. Esse projeto conterá o código do seu aplicativo e um pacote de suporte que contém o interpretador Python.

Em seguida, podemos usar o comando `build` do Briefcase para compilar isso em um arquivo de aplicativo APK para

Android.

macOS

Linux

Windows

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

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

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

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

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

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

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

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

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

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

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

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

O Gradle pode parecer travado

Durante a etapa `briefcase build android`, o Gradle (a ferramenta de compilação da plataforma Android) imprimirá `CONFIGURING: 100%` e parecerá não estar fazendo nada. Não se preocupe, ele não está travado - está baixando mais componentes do Android SDK. Dependendo da velocidade da sua conexão com a Internet, isso pode levar mais 10 minutos (ou mais). Esse atraso só deve ocorrer na primeira vez em que você executar o `build`; as ferramentas são armazenadas em cache e, na próxima compilação, as versões armazenadas em cache serão usadas.

Execute o aplicativo em um dispositivo virtual

Agora estamos prontos para executar nosso aplicativo. Você pode usar o comando `run` do Briefcase para executar o aplicativo em um dispositivo Android. Vamos começar com a execução em um emulador de Android.

Para executar seu aplicativo, execute `briefcase run android`. Ao fazer isso, será exibida uma lista de dispositivos nos quais você pode executar o aplicativo. O último item sempre será uma opção para criar um novo emulador de Android.

macOS

Linux

Windows

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

Select device:

  1) Create a new Android emulator

>
```

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

Select device:

  1) Create a new Android emulator

>
```

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

Select device:

  1) Create a new Android emulator

>
```

Agora podemos escolher o dispositivo desejado. Selecione a opção «Create a new Android emulator» e aceite a opção padrão para o nome do dispositivo (`beePhone`).

O `run` do Briefcase inicializará automaticamente o dispositivo virtual. Quando o dispositivo estiver sendo inicializado, você verá o logotipo do Android:

Quando o dispositivo terminar a inicialização, o Briefcase instalará seu aplicativo no dispositivo. Você verá brevemente uma tela de inicialização:

Em seguida, o aplicativo será iniciado. Você verá uma tela inicial enquanto o aplicativo é iniciado:

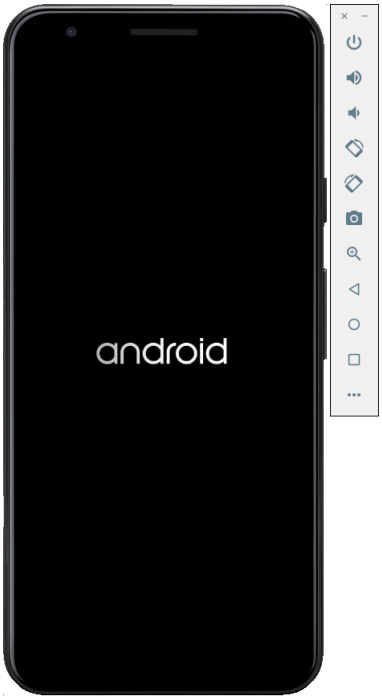


Fig. 1: Inicialização do dispositivo virtual Android

Fig. 2: Dispositivo virtual Android totalmente iniciado, na tela do iniciador

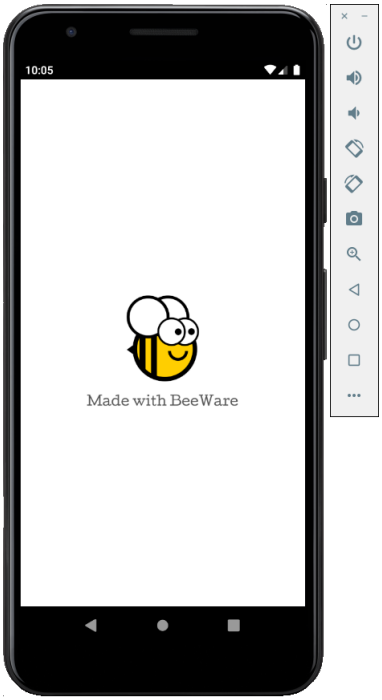


Fig. 3: Tela inicial do aplicativo

O emulador não foi iniciado!

O emulador do Android é um software complexo que depende de vários recursos de hardware e do sistema operacional - recursos que podem não estar disponíveis ou habilitados em máquinas mais antigas. Se tiver alguma dificuldade para iniciar o emulador do Android, consulte a seção «Requisitos e recomendações» da documentação do desenvolvedor do Android.

Na primeira vez que o aplicativo é iniciado, ele precisa ser descompactado no dispositivo. Isso pode levar alguns segundos. Depois de descompactado, você verá a versão para Android do nosso aplicativo para desktop:

Se não conseguir ver o aplicativo sendo iniciado, talvez seja necessário verificar o terminal em que você executou o `briefcase run` e procurar mensagens de erro.

No futuro, se você quiser executar nesse dispositivo sem usar o menu, poderá fornecer o nome do emulador ao Briefcase, usando `briefcase run android -d @beePhone` para executar diretamente no dispositivo virtual.

Execute o aplicativo em um dispositivo físico

Se você tiver um telefone ou tablet Android físico, poderá conectá-lo ao seu computador com um cabo USB e, em seguida, usar o Briefcase para direcionar seu dispositivo físico.

O Android exige que você prepare seu dispositivo antes que ele possa ser usado para desenvolvimento. Você precisará fazer duas alterações nas opções do seu dispositivo:

- Ativar opções de desenvolvedor
- Ativar a depuração USB

Detalhes sobre como fazer essas alterações podem ser encontrados na documentação do desenvolvedor do Android.

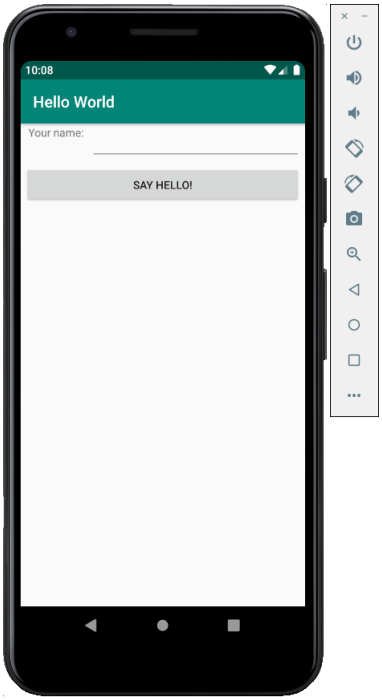


Fig. 4: Aplicativo de demonstração totalmente lançado

Depois que essas etapas forem concluídas, seu dispositivo deverá aparecer na lista de dispositivos disponíveis quando você executar o `briefcase run android`.

macOS

Linux

Windows

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

Select device:

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

>
```

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

Select device:

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

>
```

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

Select device:

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

>
```

Aqui podemos ver um novo dispositivo físico com seu número de série na lista de implantação - neste caso, um Pixel 3a. No futuro, se quiser executar nesse dispositivo sem usar o menu, você poderá fornecer o número de série do telefone ao Briefcase (nesse caso, `briefcase run android -d 94ZZY0LNE8`). Isso será executado diretamente no dispositivo, sem avisos.

Meu dispositivo não aparece!

Se o seu dispositivo não aparecer nessa lista, você não ativou a depuração USB (ou o dispositivo não está conectado!).

Se o seu dispositivo aparecer, mas estiver listado como «Dispositivo desconhecido (não autorizado para desenvolvimento)», o modo de desenvolvedor não foi ativado corretamente. Execute novamente as etapas para ativar as opções de desenvolvedor e execute novamente `briefcase run android`.

Próximos passos

Agora temos um aplicativo em nosso telefone! Há algum outro lugar onde possamos implantar um aplicativo BeeWare? Consulte *Tutorial 6* para descobrir...

2.7 Tutorial 6 - Coloque na web!

Além de oferecer suporte a plataformas móveis, o kit de ferramentas de widget Toga também oferece suporte à Web! Usando a mesma API que você usou para implantar seus aplicativos de desktop e móveis, você pode implantar seu aplicativo como um aplicativo da Web de página única.

Prova de conceito

O backend do Toga Web é o menos maduro de todos os backends do Toga. Ele está maduro o suficiente para exibir alguns recursos, mas é provável que apresente bugs e não tenha muitos dos widgets disponíveis em outras plataformas. Neste momento, a implementação na Web deve ser considerada uma «Prova de Conceito» - o suficiente para demonstrar o que pode ser feito, mas não o suficiente para ser usado em um desenvolvimento sério.

Se você tiver problemas com esta etapa do tutorial, pule para a próxima página.

2.7.1 Implementação como um aplicativo Web

O processo de implementação como um aplicativo da Web de página única segue o mesmo padrão familiar - você cria o aplicativo, constrói o aplicativo e, em seguida, o executa. No entanto, o Briefcase pode ser um pouco inteligente; se você tentar executar um aplicativo e o Briefcase determinar que ele não foi criado ou compilado para a plataforma que está sendo visada, ele executará as etapas de criação e compilação para você. Como esta é a primeira vez que executamos o aplicativo para a Web, podemos executar todas as três etapas com um único comando:

macOS

Linux

Windows

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

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

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

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

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

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

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

[helloworld] Created build/helloworld/web/static

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

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

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

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

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

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

(continues on next page)

(continuação da página anterior)

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

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

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

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

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

[helloworld] Created build/helloworld/web/static

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

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

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

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

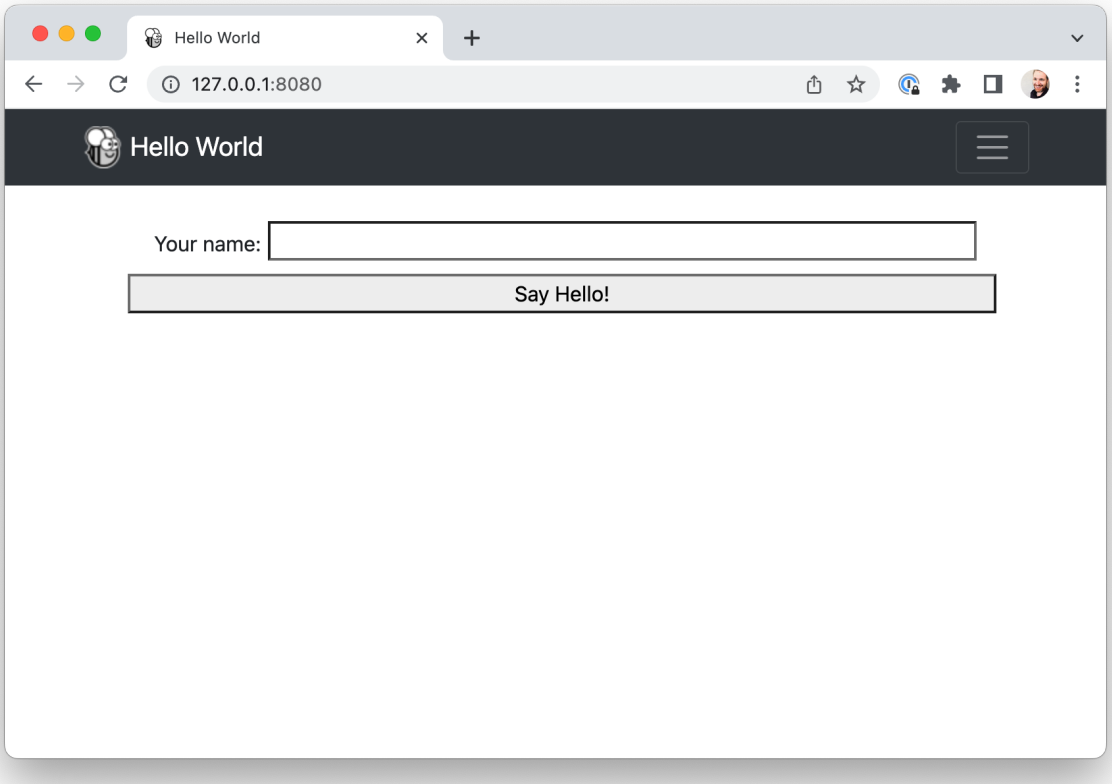
```
(beeware-venv) C:\...>briefcase run web

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

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

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

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

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

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

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

(continues on next page)

(continuação da página anterior)

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

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

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

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

Isso abrirá um navegador da Web, apontando para http://127.0.0.1:8080:



Se você digitar seu nome e clicar no botão, será exibida uma caixa de diálogo.

2.7.2 Como isso funciona?

Este aplicativo da Web é um site estático - uma única página de origem HTML, com alguns recursos CSS e outros. O Briefcase iniciou um servidor da Web local para servir essa página para que o seu navegador possa visualizá-la. Se você quiser colocar essa página da Web em produção, poderá copiar o conteúdo da pasta `www` em qualquer servidor da Web que possa servir conteúdo estático.

Mas quando você pressiona o botão, está executando código Python... como isso funciona? O Toga usa o PyScript para fornecer um interpretador Python no navegador. O Briefcase empacota o código do seu aplicativo como rodas que o PyScript pode carregar no navegador. Quando a página é carregada, o código do aplicativo é executado no navegador, criando a interface do usuário usando o DOM do navegador. Quando você clica em um botão, esse botão executa o código de manipulação de eventos no navegador.

2.7.3 Próximos passos

Embora já tenhamos implantado esse aplicativo no desktop, em dispositivos móveis e na Web, ele é bastante simples e não envolve bibliotecas de terceiros. Podemos incluir bibliotecas do Python Package Index (PyPI) em nosso aplicativo? Consulte o *Tutorial 7* para descobrir...

2.8 Tutorial 7 - Iniciando o uso de bibliotecas externas

Até o momento, o aplicativo que criamos usou apenas o nosso próprio código, além do código fornecido pelo BeeWare. Entretanto, em um aplicativo do mundo real, você provavelmente desejará usar uma biblioteca de terceiros, baixada do Python Package Index (PyPI).

Vamos modificar nosso aplicativo para incluir uma biblioteca de terceiros.

2.8.1 Acesso a uma API

Uma tarefa comum que um aplicativo precisará executar é fazer uma solicitação em uma API da Web para recuperar dados e exibir esses dados ao usuário. Como este é um aplicativo de brinquedo, não temos uma API *real* para trabalhar, portanto, usaremos a {JSON} Placeholder API como fonte de dados.

A API {JSON} Placeholder tem vários endpoints de API «falsos» que você pode usar como dados de teste. Uma dessas APIs é o ponto de extremidade `/posts/`, que retorna publicações de blog falsas. Se você abrir `https://jsonplaceholder.typicode.com/posts/42` em seu navegador, obterá uma carga JSON que descreve uma única publicação - algum conteúdo Lorum ipsum para uma publicação de blog com ID 42.

A biblioteca padrão do Python contém todas as ferramentas necessárias para acessar uma API. Entretanto, as APIs incorporadas são de nível muito baixo. Elas são boas implementações do protocolo HTTP, mas exigem que o usuário gerencie muitos detalhes de baixo nível, como redirecionamento de URL, sessões, autenticação e codificação de carga útil. Como um «usuário normal de navegador», você provavelmente está acostumado a considerar esses detalhes como garantidos, pois o navegador gerencia esses detalhes para você.

Como resultado, as pessoas desenvolveram bibliotecas de terceiros que envolvem as APIs incorporadas e fornecem uma API mais simples que se aproxima da experiência cotidiana do navegador. Vamos usar uma dessas bibliotecas para acessar a API {JSON} Placeholder - uma biblioteca chamada httpx.

Vamos adicionar uma chamada de API `httpx` ao nosso aplicativo. Adicione uma importação na parte superior do `app.py` para importar `httpx`:

```
import httpx
```

Em seguida, modifique a chamada de retorno `say_hello()` para que fique assim:

```
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"],
    )
```

Isso alterará a chamada de retorno `say_hello()` para que, quando for chamada, ela o faça:

- faça uma solicitação GET na API do espaço reservado JSON para obter o post 42;
- decodificar a resposta como JSON;
- extrair o corpo da postagem; e
- incluir o corpo dessa postagem como o texto da caixa de diálogo.

Vamos executar nosso aplicativo atualizado no modo de desenvolvedor do Briefcase para verificar se nossa alteração funcionou.

macOS

Linux

Windows

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

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

(continues on next page)

(continuação da página anterior)

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

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

O que aconteceu? Adicionamos o `httpx` ao nosso *código*, mas não o adicionamos ao nosso ambiente virtual de desenvolvimento. Podemos corrigir isso instalando o `httpx` com o `pip` e, em seguida, executando novamente o `briefcase dev`:

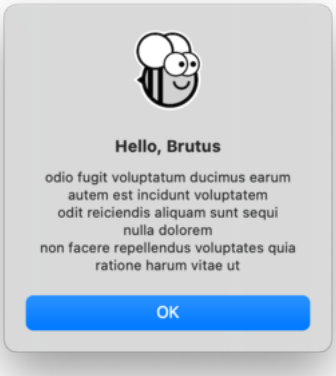
macOS

Linux

Windows

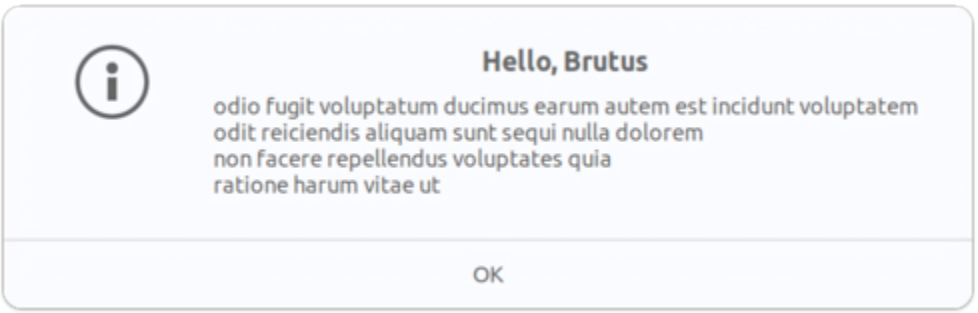
```
(beeware-venv) $ python -m pip install httpx
(beeware-venv) $ briefcase dev
```

Ao inserir um nome e pressionar o botão, você verá uma caixa de diálogo semelhante:



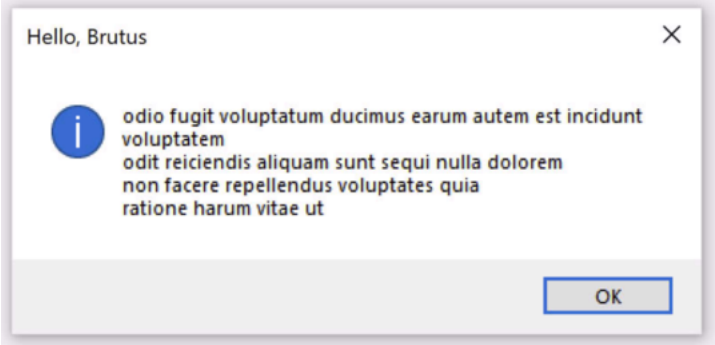

```
(beeware-venv) $ python -m pip install httpx
(beeware-venv) $ briefcase dev
```

Ao inserir um nome e pressionar o botão, você verá uma caixa de diálogo semelhante:




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

Ao inserir um nome e pressionar o botão, você verá uma caixa de diálogo semelhante:



Agora temos um aplicativo funcional, usando uma biblioteca de terceiros, em execução no modo de desenvolvimento!

2.8.2 Executando o aplicativo atualizado

Vamos empacotar esse código de aplicativo atualizado como um aplicativo autônomo. Como fizemos alterações no código, precisamos seguir as mesmas etapas do *Tutorial 4*:

macOS

Linux

Windows

Atualize o código no aplicativo empacotado:

```
(beeware-venv) $ briefcase update

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

[helloworld] Application updated.
```

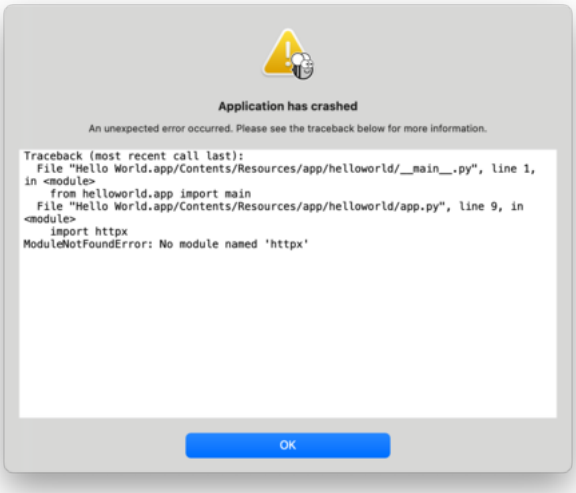
Reconstrua o aplicativo:

```
(beeware-venv) $ briefcase build

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

E, por fim, execute o aplicativo:

```
(beeware-venv) $ briefcase run

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

No entanto, quando o aplicativo for executado, você verá um erro no console, além de uma caixa de diálogo de falha:



Atualize o código no aplicativo empacotado:

```
(beeware-venv) $ briefcase update

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

[helloworld] Application updated.
```

Reconstrua o aplicativo:

```
(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
```

E, por fim, execute o aplicativo:

```
(beeware-venv) $ briefcase run

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

No entanto, quando o aplicativo for executado, você verá um erro no console:

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

Unable to start app helloworld.
```

Atualize o código no aplicativo empacotado:

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

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

[helloworld] Application updated.
```

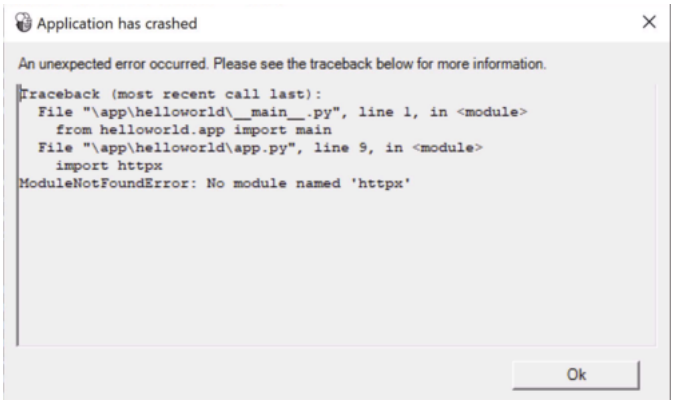
Reconstrua o aplicativo:

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

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

E, por fim, execute o aplicativo:

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

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

No entanto, quando o aplicativo for executado, você verá um erro no console, além de uma caixa de diálogo de falha:



Mais uma vez, o aplicativo não foi iniciado porque o `httpx` foi instalado - mas por quê? Já não instalamos o `httpx`?

Temos, mas somente no ambiente de desenvolvimento. O ambiente de desenvolvimento é totalmente local em seu computador e só é ativado quando você o ativa explicitamente. Embora o Briefcase tenha um modo de desenvolvimento, a principal razão pela qual você usaria o Briefcase é para empacotar o seu código para que você possa fornecê-lo a outra pessoa.

A única maneira de garantir que outra pessoa tenha um ambiente Python que contenha tudo o que ela precisa é criar um ambiente Python completamente isolado. Isso significa que há uma instalação completamente isolada do Python e um conjunto completamente isolado de dependências. Isso é o que o Briefcase está construindo quando você executa o `briefcase build` - um ambiente Python isolado. Isso também explica por que o `httpx` não está instalado - ele foi instalado em seu ambiente de *desenvolvimento*, mas não no aplicativo empacotado.

Portanto, precisamos informar ao Briefcase que nosso aplicativo tem uma dependência externa.

2.8.3 Atualização de dependências

No diretório raiz do seu aplicativo, há um arquivo chamado `pyproject.toml`. Esse arquivo contém todos os detalhes de configuração do aplicativo que você forneceu quando executou originalmente o `briefcase new`.

o `pyproject.toml` é dividido em seções; uma das seções descreve as configurações do seu aplicativo:

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

(continues on next page)

(continuação da página anterior)

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

A opção `requires` descreve as dependências do nosso aplicativo. É uma lista de cadeias de caracteres, especificando as bibliotecas (e, opcionalmente, as versões) das bibliotecas que você deseja incluir em seu aplicativo.

Modifique a configuração `requires` para que fique assim:

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

Ao adicionar essa configuração, estamos dizendo ao Briefcase «quando você compilar meu aplicativo, execute `pip install httpx` no pacote de aplicativos». Qualquer coisa que seja uma entrada legal para `pip install` pode ser usada aqui - portanto, você poderia especificar:

- Uma versão específica da biblioteca (por exemplo, `"httpx==0.19.0"`);
- Um intervalo de versões de biblioteca (por exemplo, `"httpx>=0.19"`);
- Um caminho para um repositório git (por exemplo, `"git+https://github.com/encode/httpx"`); ou
- Um caminho de arquivo local (no entanto, esteja avisado: se você fornecer seu código a outra pessoa, esse caminho provavelmente não existirá no computador dela!)

Mais abaixo no `pyproject.toml`, você notará outras seções que dependem do sistema operacional, como `[tool.briefcase.app.helloworld.macOS]` e `[tool.briefcase.app.helloworld.windows]`. Essas seções *também* têm uma configuração `requires`. Essas configurações permitem definir dependências adicionais específicas da plataforma. Assim, por exemplo, se você precisar de uma biblioteca específica da plataforma para lidar com algum aspecto do seu aplicativo, poderá especificar essa biblioteca na seção `requires` específica da plataforma, e essa configuração será usada somente para essa plataforma. Você notará que as bibliotecas `toga` são todas especificadas na seção `requires` específica da plataforma - isso ocorre porque as bibliotecas necessárias para exibir uma interface de usuário são específicas da plataforma.

No nosso caso, queremos que o `httpx` seja instalado em todas as plataformas, portanto, usamos a configuração `requires` em nível de aplicativo. As dependências no nível do aplicativo sempre serão instaladas; as dependências específicas da plataforma são instaladas *além* das dependências no nível do aplicativo.

Alguns pacotes binários podem não estar disponíveis

Em plataformas de desktop (macOS, Windows, Linux), qualquer `pip` instalável pode ser adicionado aos seus requisitos. Em plataformas móveis e da Web, suas opções são ligeiramente limitadas.

Resumindo: qualquer pacote Python *puro* (ou seja, pacotes que *não* contêm um módulo binário) pode ser usado sem dificuldade. No entanto, se sua dependência contiver um componente binário, ele deverá ser compilado; no momento, a maioria dos pacotes Python não oferece suporte à compilação para plataformas que não sejam de desktop.

O BeeWare pode fornecer binários para alguns módulos binários populares (incluindo `numpy`, `pandas` e `cryptography`). Normalmente, é possível compilar pacotes para plataformas móveis, mas isso não é fácil de configurar, o que está fora do escopo de um tutorial introdutório como este.

Agora que informamos o Briefcase sobre nossos requisitos adicionais, podemos tentar empacotar nosso aplicativo novamente. Certifique-se de que salvou suas alterações no `pyproject.toml` e, em seguida, atualize seu aplicativo novamente - desta vez, passando o sinalizador `-r`. Isso diz ao Briefcase para atualizar os requisitos no aplicativo empacotado:

macOS

Linux

Windows

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

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

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

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

[helloworld] Application updated.
```

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

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

Targeting glibc 2.35
Targeting Python3.10

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

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

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

[helloworld] Application updated.
```

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

(continues on next page)

(continuação da página anterior)

```
[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.
```

Depois de atualizar, você pode executar `briefcase build` e `briefcase run` e verá o aplicativo empacotado com o novo comportamento de diálogo.

Nota: A opção `-r` para atualizar os requisitos também é aceita pelos comandos `build` e `run`, portanto, se quiser atualizar, compilar e executar em uma única etapa, você pode usar `briefcase run -u -r`.

2.8.4 Próximos passos

Agora temos um aplicativo que usa uma biblioteca de terceiros! No entanto, você deve ter notado que, ao pressionar o botão, o aplicativo fica um pouco sem resposta. Podemos fazer algo para corrigir isso? Consulte o *Tutorial 8* para descobrir...

2.9 Tutorial 8 - Suavizando o Processo

A menos que você tenha uma conexão de Internet *muito* rápida, poderá perceber que, ao pressionar o botão, a GUI do seu aplicativo trava um pouco. Isso ocorre porque a solicitação da Web que fizemos é *síncrona*. Quando nosso aplicativo faz a solicitação da Web, ele espera que a API retorne uma resposta antes de continuar. Enquanto espera, ele não permite que o aplicativo seja redesenhado e, como resultado, o aplicativo trava.

2.9.1 Loops de eventos da GUI

Para entender por que isso acontece, precisamos nos aprofundar nos detalhes de como funciona um aplicativo de GUI. Os detalhes variam de acordo com a plataforma, mas os conceitos de alto nível são os mesmos, independentemente da plataforma ou do ambiente de GUI que você estiver usando.

Um aplicativo de GUI é, basicamente, um único loop que se parece com:

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

Esse loop é chamado de *Event Loop*. (Esses não são nomes de métodos reais - é uma ilustração do que está acontecendo no «pseudocódigo»).

Quando você clica em um botão, arrasta uma barra de rolagem ou digita uma tecla, está gerando um «evento». Esse «evento» é colocado em uma fila, e o aplicativo processará a fila de eventos quando tiver a oportunidade de fazê-lo. O código do usuário que é acionado em resposta ao evento é chamado de *manipulador de eventos*. Esses manipuladores de eventos são invocados como parte da chamada `process_events()`.

Depois que um aplicativo tiver processado todos os eventos disponíveis, ele `redraw()` a GUI. Isso leva em conta todas as alterações que os eventos causaram na exibição do aplicativo, bem como qualquer outra coisa que esteja acontecendo no sistema operacional - por exemplo, as janelas de outro aplicativo podem obscurecer ou revelar parte da janela do nosso aplicativo, e o redesenho do nosso aplicativo precisará refletir a parte da janela que está visível no momento.

O detalhe importante a ser observado: enquanto um aplicativo estiver processando um evento, *ele não pode redesenhar* e *não pode processar outros eventos*.

Isso significa que qualquer lógica de usuário contida em um manipulador de eventos precisa ser concluída rapidamente. Qualquer atraso na conclusão do manipulador de eventos será observado pelo usuário como uma desaceleração (ou parada) nas atualizações da GUI. Se esse atraso for longo o suficiente, seu sistema operacional poderá informar isso como um problema - os ícones «bola de praia» do macOS e «botão giratório» do Windows são o sistema operacional informando que seu aplicativo está demorando demais em um manipulador de eventos.

Operações simples como «atualizar um rótulo» ou «recomputar o total das entradas» são fáceis de concluir rapidamente. Entretanto, há muitas operações que não podem ser concluídas rapidamente. Se estiver realizando um cálculo matemático complexo, ou indexando todos os arquivos em um sistema de arquivos, ou realizando uma grande solicitação de rede, não é possível «simplesmente fazer isso rapidamente» - as operações são inerentemente lentas.

Então, como realizamos operações de longa duração em um aplicativo de GUI?

2.9.2 Programação assíncrona

O que precisamos é de uma maneira de informar a um aplicativo no meio de um manipulador de eventos de longa duração que não há problema em liberar temporariamente o controle de volta para o loop de eventos, desde que possamos retomar de onde paramos. Cabe ao aplicativo determinar quando essa liberação pode ocorrer; mas se o aplicativo liberar o controle para o loop de eventos regularmente, poderemos ter um manipulador de eventos de longa duração *e* manter uma interface de usuário responsiva.

Podemos fazer isso usando a *programação assíncrona*. A programação assíncrona é uma maneira de descrever um programa que permite que o intérprete execute várias funções ao mesmo tempo, compartilhando recursos entre todas as funções executadas simultaneamente.

As funções assíncronas (conhecidas como *co-rotinas*) precisam ser explicitamente declaradas como assíncronas. Elas também precisam declarar internamente quando existe uma oportunidade de mudar o contexto para outra co-rotina.

Em Python, a programação assíncrona é implementada usando as palavras-chave `async` e `await` e o módulo asyncio na biblioteca padrão. A palavra-chave `async` nos permite declarar que uma função é uma co-rotina assíncrona. A palavra-chave `await` fornece uma maneira de declarar quando existe uma oportunidade de mudar o contexto para outra co-rotina. O módulo asyncio fornece algumas outras ferramentas e primitivas úteis para codificação assíncrona.

2.9.3 Tornando o tutorial assíncrono

Para tornar nosso tutorial assíncrono, modifique o manipulador de eventos `say_hello()` para que ele tenha a seguinte aparência:

```
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"],
    )
```

Há apenas 4 alterações nesse código em relação à versão anterior:

1. O método é definido como `async def`, em vez de apenas `def`. Isso informa ao Python que o método é uma co-rotina assíncrona.
2. O cliente criado é um `AsyncClient()` assíncrono, em vez de um `Client()` síncrono. Isso informa ao `httpx` que ele deve operar no modo assíncrono, e não no modo síncrono.
3. O gerenciador de contexto usado para criar o cliente é marcado como `async`. Isso informa ao Python que há uma oportunidade de liberar o controle à medida que o gerenciador de contexto entra e sai.
4. A chamada `get` é feita com uma palavra-chave `await`. Isso instrui o aplicativo que, enquanto aguardamos a resposta da rede, ele pode liberar o controle para o loop de eventos.

A Toga permite que você use métodos regulares ou co-rotinas assíncronas como manipuladores; a Toga gerencia tudo nos bastidores para garantir que o manipulador seja chamado ou aguardado conforme necessário.

Se você salvar essas alterações e executar novamente o aplicativo (com o `briefcase dev` no modo de desenvolvimento ou atualizando e executando novamente o aplicativo empacotado), não haverá nenhuma alteração óbvia no aplicativo. No entanto, ao clicar no botão para acionar a caixa de diálogo, você poderá notar uma série de melhorias sutis:

- O botão retorna a um estado «não clicado», em vez de ficar preso em um estado «clicado».
- O ícone «bola de praia»/»botão giratório» não é exibido
- Se você mover/redimensionar a janela do aplicativo enquanto aguarda a exibição da caixa de diálogo, a janela será redesenhada.
- Se você tentar abrir um menu de aplicativo, o menu será exibido imediatamente.

2.9.4 Próximos passos

Agora temos um aplicativo que é elegante e responsivo, mesmo quando está esperando em uma API lenta. Mas como podemos ter certeza de que o aplicativo continuará funcionando à medida que continuarmos a desenvolvê-lo? Como testamos nosso aplicativo? Consulte o *Tutorial 9* para descobrir...

2.10 Tutorial 9 - Hora dos Testes

A maior parte do desenvolvimento de software não envolve a gravação de um novo código, mas sim a modificação do código existente. Garantir que o código existente continue a funcionar da maneira esperada é uma parte fundamental do processo de desenvolvimento de software. Uma maneira de garantir o comportamento do nosso aplicativo é com um *conjunto de testes*.

2.10.1 Executar o conjunto de testes

Acontece que nosso projeto já tem um conjunto de testes! Quando geramos nosso projeto originalmente, foram gerados dois diretórios de nível superior: `src` e `tests`. A pasta `src` contém o código do nosso aplicativo; a pasta `tests` contém nosso conjunto de testes. Dentro da pasta `tests` há um arquivo chamado `test_app.py` com o seguinte conteúdo:

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

Este é um Pytest *caso de teste* - um bloco de código que pode ser executado para verificar algum comportamento do seu aplicativo. Nesse caso, o teste é um espaço reservado e não testa nada sobre nosso aplicativo, mas é um teste que podemos executar.

Podemos executar esse conjunto de testes usando a opção `--test` do `briefcase dev`. Como esta é a primeira vez que estamos executando testes, também precisamos passar a opção `-r` para garantir que os requisitos de teste também sejam instalados:

macOS

Linux

Windows

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

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

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

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

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

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

[helloworld] Installing requirements...
```

(continues on next page)

(continuação da página anterior)

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

Sucesso! Acabamos de executar um único teste que verifica se a matemática Python funciona da maneira esperada (que alívio!).

Vamos substituir esse teste de espaço reservado por um teste para verificar se o nosso método `greeting()` se comporta da maneira esperada. Substitua o conteúdo de `test_app.py` pelo seguinte:

```
from helloworld.app import greeting


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

(continues on next page)

(continuação da página anterior)

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


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

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

Isso define dois novos testes, verificando os dois comportamentos que esperamos ver: a saída quando um nome é fornecido e a saída quando o nome está vazio.

Agora podemos executar novamente o conjunto de testes. Dessa vez, não precisamos fornecer a opção `-r`, pois os requisitos de teste já foram instalados; precisamos apenas usar a opção `--test`:

macOS

Linux

Windows

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continuação da página anterior)

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

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

Excelente! Nosso método utilitário `greeting()` está funcionando como esperado.

2.10.2 Desenvolvimento orientado por testes

Agora que temos um conjunto de testes, podemos usá-lo para impulsionar o desenvolvimento de novos recursos. Vamos modificar nosso aplicativo para ter uma saudação especial para um determinado usuário. Podemos começar adicionando um caso de teste para o novo comportamento que gostaríamos de ver na parte inferior do `test_app.py`:

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

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

Em seguida, execute o conjunto de testes com esse novo teste:

macOS

Linux

Windows

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

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

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

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

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

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

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

(continues on next page)

(continuação da página anterior)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continuação da página anterior)

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

Dessa vez, vemos uma falha no teste - e a saída explica a origem da falha: o teste está esperando a saída «BeeWare the IDEs of Python!», mas nossa implementação de `greeting()` está retornando «Hello, Brutus». Vamos modificar a implementação de `greeting()` em `src/helloworld/app.py` para obter o novo comportamento:

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

Se executarmos os testes novamente, veremos que nossos testes foram aprovados:

macOS

Linux

Windows

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

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

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

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

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

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

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

(continues on next page)

(continuação da página anterior)

```
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 Testes de tempo de execução

Até o momento, estamos executando os testes no modo de desenvolvimento. Isso é especialmente útil quando se está desenvolvendo novos recursos, pois é possível iterar rapidamente na adição de testes e na adição de código para que esses testes sejam aprovados. No entanto, em algum momento, você desejará verificar se o seu código também é executado corretamente no ambiente do aplicativo de pacote.

As opções `--test` e `-r` também podem ser passadas para o comando `run`. Se você usar `briefcase run --test -r`, o mesmo conjunto de testes será executado, mas dentro do pacote de aplicativos empacotados, e não no seu ambiente de desenvolvimento:

macOS

Linux

Windows

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

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

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

[helloworld] Starting test suite...
================================================================================
Configuring isolated Python...
Pre-initializing Python runtime...
PythonHome: /Users/brutus/beeware-tutorial/helloworld/macOS/app/Hello World/Hello World.
↪app/Contents/Resources/support/python-stdlib
```

(continues on next page)

(continuação da página anterior)

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

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

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

[helloworld] Test suite passed!
```

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

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

Targeting glibc 2.35
Targeting Python3.10

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

[helloworld] Updating requirements...
...
[helloworld] Built build/helloworld/linux/ubuntu/jammy/helloworld-0.0.1/usr/bin/
↪helloworld (test mode)

[helloworld] Starting test suite...
================================================================================
============================= test session starts ==============================
...
collecting ... collected 3 items
```

(continues on next page)

(continuação da página anterior)

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

Como no caso do `briefcase dev --test`, a opção `-r` só é necessária na primeira vez em que você executa o conjunto de testes para garantir que as dependências de teste estejam presentes. Nas execuções subsequentes, você pode omitir essa opção.

Você também pode usar a opção `--test` em back-ends móveis: - assim, `briefcase run iOS --test` e `briefcase run android --test` funcionarão, executando o conjunto de testes no dispositivo móvel que você selecionar.

2.10.4 Próximos passos

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 - Crie seu próprio aplicativo

Até agora, nosso aplicativo tem usado um ícone padrão de «abelha cinza». Como podemos atualizar o aplicativo para usar nosso próprio ícone?

2.11.1 Adição de um ícone

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

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

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

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

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

macOS

Linux

Windows

Android

iOS

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

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

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

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

[helloworld] Application updated.
```

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

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

(continues on next page)

(continuação da página anterior)

```
Installing src/helloworld... done

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

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

[helloworld] Application updated.
```

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

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

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

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

[helloworld] Application updated.
```

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

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

[helloworld] Updating application resources...
Unable to find icons/helloworld-round-48.png for 48px round application icon; using␣
↪default
Unable to find icons/helloworld-round-72.png for 72px round application icon; using␣
↪default
Unable to find icons/helloworld-round-96.png for 96px round application icon; using␣
↪default
Unable to find icons/helloworld-round-144.png for 144px round application icon; using␣
↪default
Unable to find icons/helloworld-round-192.png for 192px round application icon; using␣
↪default
Unable to find icons/helloworld-square-48.png for 48px square application icon; using␣
↪default
Unable to find icons/helloworld-square-72.png for 72px square application icon; using␣
↪default
Unable to find icons/helloworld-square-96.png for 96px square application icon; using␣
↪default
Unable to find icons/helloworld-square-144.png for 144px square application icon; using␣
```

(continues on next page)

(continuação da página anterior)

```
↪default
Unable to find icons/helloworld-square-192.png for 192px square application icon; using␣
↪default
Unable to find icons/helloworld-square-320.png for 320px square application icon; using␣
↪default
Unable to find icons/helloworld-square-480.png for 480px square application icon; using␣
↪default
Unable to find icons/helloworld-square-640.png for 640px square application icon; using␣
↪default
Unable to find icons/helloworld-square-960.png for 960px square application icon; using␣
↪default
Unable to find icons/helloworld-square-1280.png for 1280px square application icon;␣
↪using default
Unable to find icons/helloworld-adaptive-108.png for 108px adaptive application icon;␣
↪using default
Unable to find icons/helloworld-adaptive-162.png for 162px adaptive application icon;␣
↪using default
Unable to find icons/helloworld-adaptive-216.png for 216px adaptive application icon;␣
↪using default
Unable to find icons/helloworld-adaptive-324.png for 324px adaptive application icon;␣
↪using default
Unable to find icons/helloworld-adaptive-432.png for 432px adaptive application icon;␣
↪using default

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

[helloworld] Application updated.
```

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

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

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

(continues on next page)

(continuação da página anterior)

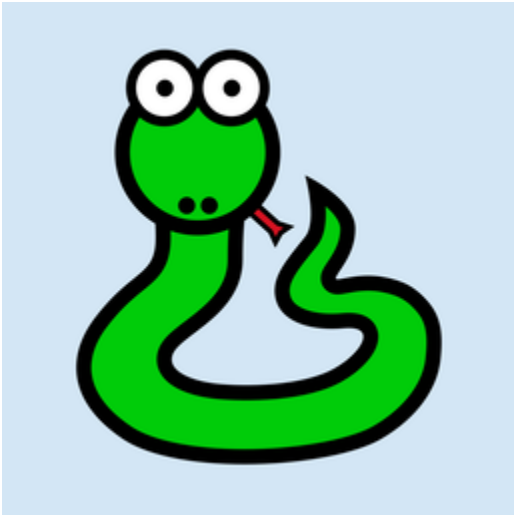
```
  [helloworld] Removing unneeded app content...
  Removing unneeded app bundle content... done

  [helloworld] Application updated.
```

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

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

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

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



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

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

macOS

Linux

Windows

Android

iOS

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

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

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

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

[helloworld] Application updated.

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

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

[helloworld] Starting app...
```

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

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

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

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

[helloworld] Application updated.

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

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

[helloworld] Starting app...
```

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

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

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

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

[helloworld] Application updated.

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

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

[helloworld] Starting app...
```

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

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

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

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

[helloworld] Application updated.
```

(continues on next page)

(continuação da página anterior)

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

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

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

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

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

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

[helloworld] Application updated.

[helloworld] Starting app...
```

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

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

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

macOS

Linux

Windows

Android

iOS

```
(beeware-venv) $ briefcase create

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

[helloworld] Removing old application bundle...

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

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

```
(beeware-venv) $ briefcase create

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

[helloworld] Removing old application bundle...

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

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

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

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

[helloworld] Removing old application bundle...

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

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

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

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

[helloworld] Removing old application bundle...

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

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

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

[helloworld] Removing old application bundle...
```

(continues on next page)

(continuação da página anterior)

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

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

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

2.11.2 Próximos passos

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.