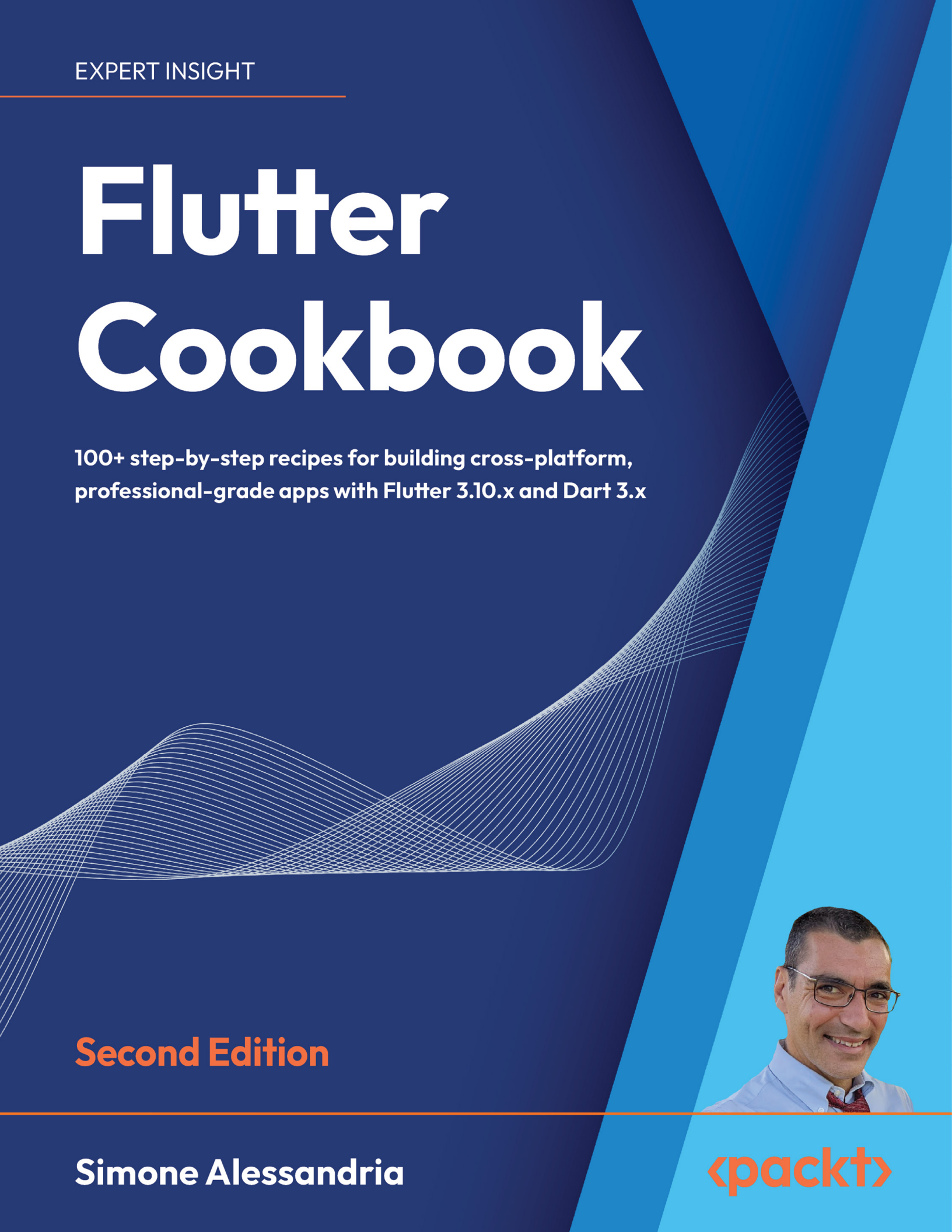
EXPERT INSIGHT

Flutter Cookbook

100+ step-by-step recipes for building cross-platform, professional-grade apps with Flutter 3.10.x and Dart 3.x



Second Edition

Simone Alessandria

packt

EXPERT INSIGHT

Flutter Cookbook

100+ step-by-step recipes for building cross-platform, professional-grade apps with Flutter 3.10.x and Dart 3.x



Second Edition

Simone Alessandria

packt

Table of Contents

Preface

Getting Started with Flutter

Creating Your First Flutter App

Dart: A Language You Already Know

Introduction to Widgets

Mastering Layout and Taming the Widget Tree

Adding Interactivity and Navigation to Your App

Basic State Management

The Future is Now: Introduction to Asynchronous Programming

Data Persistence and Communicating with the Internet

Advanced State Management with Streams

Using Flutter Packages

Adding Animations to Your App

Using Firebase

Firebase Machine Learning

Flutter Web and Desktop

Distributing Your Mobile App

Other Books You May Enjoy

Index

Preface

These recipes cover the most important Flutter features that will allow you to develop real-world apps. In each recipe, you will learn about and immediately use some of the tools that make Flutter so successful: widgets, state management, asynchronous programming, connecting to web services, persisting data, creating animations, using Firebase and machine learning, and developing responsive apps that work on different platforms, including desktop and the web.

Flutter is a developer-friendly, open source toolkit created by Google that you can use to create applications for Android and iOS mobile devices, the web, and desktop.

There are 16 chapters in this book, which you can read independently from one another: each chapter contains recipes that highlight and leverage a single Flutter feature. You can choose to follow the flow of the book or skip to any chapter if you feel confident with the concepts.

Flutter uses Dart as a programming language. *Chapter 3*, *Dart: A Language You Already Know*, is an introduction to Dart, its syntax, and its patterns, and it gives you the necessary knowledge to be productive when using Dart in Flutter.

In later chapters, you'll see recipes that go beyond basic examples; you will be able to play with code and get hands-on experience in using basic, intermediate, and advanced Flutter tools.

Who this book is for

This book is for developers who are familiar with an object-oriented programming language. If you understand concepts such as variables, functions, classes, and objects, this book is for you.

Prior knowledge of Dart is not required as it is introduced in *Chapter 3*, *Dart: A Language You Already Know*.

If you already know and use languages such as Java, C#, Swift, Kotlin, and JavaScript, you will find Dart surprisingly easy to learn.

What this book covers

Chapter 1, *Getting Started with Flutter*, helps you set up your development environment.

Chapter 2, *Creating Your First Flutter App*, shows how to create your first app, and check that your development environment works as expected.

Chapter 3, *Dart: A Language You Already Know*, introduces Dart, its syntax, and its patterns.

Chapter 4, *Introduction to Widgets*, shows how to build simple user interfaces with Flutter.

Chapter 5, *Mastering Layout and Taming the Widget Tree*, shows how to build more complex screens made of several widgets.

Chapter 6, *Adding Interactivity and Navigation to Your app*, contains several recipes that add interactivity to your apps, including interacting with buttons, reading a text from a TextField, changing the screen, and showing alerts.

Chapter 7, *Basic State Management*, introduces the concept of State in Flutter: instead of having screens that just show widgets, you will learn how to build screens that can keep and manage data.

Chapter 8, *The Future is Now: Introduction to Asynchronous Programming*, contains several examples of one of the most useful and interesting features in programming languages: the concept of the asynchronous execution of tasks.

Chapter 9, *Data Persistence and Communicating with the Internet*, gives you the tools to connect to web services and persist data on your device.

Chapter 10, *Advanced State Management with Streams*, shows how to deal with Streams, which are arguably the best tool for creating reactive apps.

Chapter 11, *Using Flutter Packages*, explains how to choose, use, build, and publish Flutter packages.

Chapter 12, *Adding Animations to Your app*, gives you the tools you need to build engaging animations in your apps.

Chapter 13, *Using Firebase*, shows how to leverage a powerful backend without any code.

Chapter 14, *Firebase Machine Learning*, shows how to add machine learning features to your apps by using Firebase.

Chapter 15, *Flutter Web and Desktop*, shows you how to use the same code base to build apps for the web and desktop.

Chapter 16, Distributing Your Mobile app, outlines the steps required to publish an app to the main mobile stores: the Google Play Store and the Apple App Store.

To get the most out of this book

Some experience in at least one object-oriented programming language is strongly recommended.

In order to follow along with the code, you will need a Windows PC, Mac, Linux, or Chrome OS machine connected to the web, with at least 8 GB of RAM and the permissions to install new software.

An Android or iOS device is suggested but not necessary as there are simulators/emulators that can run on your machine. All software used in this book is open source or free to use.

Chapter 1, *Getting Started with Flutter*, explains in detail the installation process; however, you should have the following:

Software/hardware	OS Requirements

Recommended editors: Visual Studio Code, Android Studio, or IntelliJ Idea	Windows, macOS, or Linux
The Flutter SDK	Windows, macOS, or Linux
An emulator/simulator or an iOS or Android device	Windows, macOS, or Linux

In order to create apps for iOS, you will need a Mac.

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

If you like this book or want to share your ideas about it, please write a review on your favorite platform. This will help us make this book better, and you'll also earn the author's and reviewers' everlasting gratitude.

Download the example code files

The code bundle for the book is hosted on GitHub at https://github.com/PacktPublishing/Flutter-Cookbook-Second-Edition/. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://packt.link/WE615.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “Add the latest version of the lint package as a dev dependency in your `pubspec.yaml`, and remove any other linting packages you may find there.”

A block of code is set as follows:

```
dependencies:
  flutter:
    sdk: flutter
  http: ^0.13.5
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
dev_dependencies:
  flutter_test:
    sdk: flutter
  lint: ^2.0.0
```

Any command-line input or output is written as follows:

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
     /etc/asterisk/cdr_mysql.conf
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “ In order to use Google Maps, you need to obtain an **API key**.”

> **Note**
>
> Warnings or important notes appear like this.

> **Pro Tip**
>
> Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit http://www.packtpub.com/submit-errata, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit http://authors.packtpub.com.

Share your thoughts

Once you've read *Flutter Cookbook, Second Edition*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?Is your eBook purchase not compatible with the device of your choice?

Don’t worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don’t stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



https://packt.link/free-ebook/9781803245430

2. Submit your proof of purchase

3. That’s it! We’ll send your free PDF and other benefits to your email directly

Getting Started with Flutter

Whenever you begin developing with a new platform, the first task you need to perform is creating your development environment. In some ways, the ease with which you can go from nothing to building the first “hello world” app can be seen as a test for how your experience with the new platform is going to be. If the environment is difficult and painful to set up, then it may be very likely that it will be difficult and painful to work with.

By the end of this chapter, you will have Flutter fully installed and will be ready to run your first app.

In this chapter, we’ll be covering the following topics:

- Why Flutter?
- Technical requirements
- Installing Flutter: a high-level overview
- How to use Git to manage the Flutter SDK
- Setting up the command line and saving path variables
- Using Flutter doctor to diagnose your environment
- Configuring the iOS SDK
- Setting up CocoaPods (iOS only)
- Configuring the Android SDK setup
- Which IDE/editor should you choose?
- Picking the right channel
- How to choose the platform language for your app

Let’s begin by discussing why Flutter may be a great framework to work with.

Why Flutter?

Flutter is an open-source framework to build apps, created by Google. Before investing time and effort in learning a new framework, there is a critical question we need to answer: *Is Flutter worth learning and using?*

While the answer to this question probably depends on your needs and expectations, there are several reasons that make Flutter a great framework to learn and use. Some of them include:

- **Cross-platform**: With Flutter, you can build apps for iOS and Android, web, desktop, and IoT devices with a single code base.
- **Fast development**: Flutter includes several features that help speed up your development workflow. Among the most important, it's worth mentioning hot reload, hot restart, and sound null safety support.
- **User interface**: Flutter provides a rich set of widgets to create beautiful, high-performance apps.
- **Open source**: Flutter and Dart (the programming language you use in Flutter) are open source and supported by Google.
- **Community**: Flutter has a large and active community. This means you can easily find great resources, third-party packages, and help.
- **Performance**: Flutter uses a reactive programming model and a high-performance rendering engine, and compiles to native code.

There are arguably other points we could add here, but hopefully, you'll discover several more yourself while reading this book.

Technical requirements

Flutter requires a 64-bit operating system. In particular:

- For Windows, you will need Windows 10 or later (64-bit and x86-64 based), Git for Windows, and PowerShell 5 or newer.
- For macOS, OS X Yosemite 10.10 or later is required.
- For Linux and ChromeOS, there is a list of common programs that should be installed in your system before setting up Flutter. More info can be found here: https://docs.flutter.dev/get-started/install/linux (for Linux) and here: https://docs.flutter.dev/get-started/install/chromeos (for ChromeOS).

Building mobile applications can be a taxing task for your computer. While the minimum hardware requirements to develop a Flutter app are relatively low, your developing experience will be greatly enhanced with the following specs:

- 8 GB of **random-access memory** (**RAM**). 16 **gigabytes** (**GB**) is preferred, especially when using a virtual device (emulator or simulator).
- At least 50 GB of available hard drive space.
- A **solid-state drive** (**SSD**) hard drive.
- At least a 2 **gigahertz** (**GHz**) processor.
- A reliable internet connection.
- If you want to build for iOS, you will need a Mac.

Pro Tip

Flutter itself doesn't have strict hardware requirements, but anything less than this may lead to you spending more time waiting rather than working.

Let's see how to install your Flutter development environment next.

Installing Flutter: a high-level overview

You can start developing Flutter apps without any installation or configuration. You can quickly create prototypes and simple apps and share your code with fellow developers with an online tool called **DartPad**, available here: https://dartpad.dev. It is an online open-source tool that runs on any browser.

However, if you are serious about programming with Flutter, there will come a time when you need to develop your apps locally and, therefore, will need to configure your system.

Configuring your Flutter development environment is rather easy. You can divide the process into three parts:

1. First, you must install the Flutter **software development kit** (**SDK**). Currently, you can build Flutter apps from Windows, macOS, Linux, and ChromeOS.
2. Then, depending on your target platform, you have to install a specific platform SDK: Android, iOS, and/or desktop. You can also develop for the web, without installing any specific SDK. Note that:

    - Any supported device can build apps for Android and the web.
    - You can build iOS apps only from a Mac.
    - You can only build desktop apps when the origin and target are the same: a Windows app can only be built from a Windows desktop, a macOS app only from a macOS desktop, etc.

3. The final stage is choosing which editor, or **integrated development environment** (**IDE**), you want to use.

To make this process even easier, Flutter has a tool called Flutter Doctor, which will scan your environment and offer you step-by-step guides for

what you need to do to successfully complete your environment setup.

Installing the Flutter SDK

Before you can build anything, you need to download the **Flutter SDK**.

On the main Flutter website at https://docs.flutter.dev/get-started/install, you'll find the prebuilt packages for Windows, macOS, Linux, or ChromeOS—the currently supported platforms to create Flutter apps.

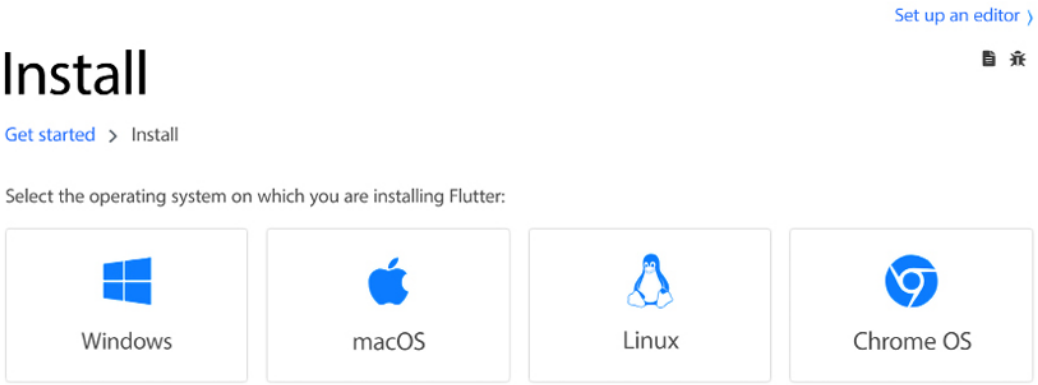


Figure 1.1: The Flutter installation page

Once you select your operating system, whatever your choice, you will find a section called **Get the Flutter SDK**. Here, you will find a compressed file (`.zip` for Windows and macOS, and `.tar.gz` for Linux and ChromeOS).

You should then create a directory for the Flutter SDK, and extract the contents of the compressed file into that directory; for instance, this could be `c:\dev\flutter` on a Windows machine or `~/development` on other platforms.

Note

There is another tool, called **FVM (Flutter Version Management)**, that allows managing multiple versions of Flutter SDKs on your development machine.

If you install it, you can switch between different versions of Flutter, without having to manually download and install each version. This is very useful when you work on projects that require different versions of Flutter.

You'll find FVM here: https://fvm.app/docs/getting_started/installation/.

How to use Git to manage the Flutter SDK

An alternative way to install the Flutter SDK is using Git.

Since Flutter is open source and hosted on GitHub, if you clone the main Flutter repository, you'll already have everything you need. As an added bonus, you can easily change to different versions of the Flutter SDK when needed.

The packages that are available to download on Flutter's website are snapshots from the Git repository. Flutter uses Git internally to manage its versions, channels, and upgrades.

Installing Git

First, you need to make sure you have Git installed on your computer. Git is installed by default on most Mac and Linux machines. For Windows, you can download and install Git here: https://git-scm.com/download/win.

You can also download a Git client to make working with repositories a bit easier. Tools such as `Sourcetree` (https://www.sourcetreeapp.com) or GitHub Desktop (https://desktop.github.com) can simplify working with Git. They are optional, however, and this book will stick to the command line when referencing Git.

To confirm that Git is installed on Linux and macOS, open your Terminal and type `which git`. You should see a `/usr/bin/git` path returned. If you see nothing, then Git is not installed correctly: you may follow the instructions at https://git-scm.com/book/en/v2/Getting-Started-Installing-Git and try to repeat the installation process to solve installation issues.

On all platforms, from your Terminal or Command Prompt, you can type the following command: `git -version`. You'll then see the current Git version installed in your system.

How to do it...

Follow these steps to clone and configure the Flutter SDK:

1. First, choose a directory where Flutter is going to be installed. The specific location does not matter, but make sure you have permissions to read, write, and execute in your chosen location.
2. On Windows, you might choose a folder called `c:\development\flutter`. In this case, in your Command Prompt, type:

```
cd\development\flutter
```

3. On macOS, to install Flutter in the `$HOME` directory, from your Terminal, type in the following command:

```
cd $HOME
```

- Another common path on macOS is the `~` path; in this case, just type:

```
cd ~
```

4. On Linux, you might choose a `/flutter` directory. In this case, just type:

```
cd /flutter
```

5. We can now install Flutter. From your Terminal, type the command:

```
git clone https://github.com/flutter/flutter.git
```

This will download Flutter and all of its associated tools, including the Dart SDK.

Setting up the command line and saving path variables

Now that you have installed the Flutter SDK, there are a few more steps needed to make the software accessible on your computer. Unlike apps with **user interfaces** (**UIs**), Flutter is primarily a command-line tool. Let's quickly learn how to set up the command line on macOS, Windows, and Linux in the following sections.

macOS command-line setup

To use Flutter on macOS, you need to save the location of the Flutter executable to your system's environment variables.

> Note
>
> Newer Macs use the **Z shell** (also known as **zsh**). This is basically an improved version of the older **Bash**, with several additional features.

When using `zsh`, you can add a line to your `zshrc` file, which is a text file that contains the `zsh` configuration. If the file does not exist yet, you can create a new file as follows:

1. Open the `zshrc` file with the following command in your Terminal:

```
nano $HOME/.zshrc
```

2. This will open a basic text editor called `nano` in your Terminal window. There are other popular tools, such as `vim` and `emacs`, that will also work.
3. Type the following command at the bottom of the file:

```
export PATH="$PATH:$HOME/flutter/bin"
```

   If you chose to install Flutter at a different location, then replace `$HOME` with the appropriate directory.

4. Exit `nano` by pressing *Ctrl* + *X*. Don't forget to save your file when prompted.
5. Reload your Terminal session by typing the following command:

```
source ~/.zshrc
```

6. Finally, confirm that everything is configured correctly by typing the following:

```
which flutter
```

You should see the directory where you cloned (or installed) the Flutter SDK printed on the screen, as shown in the following screenshot:

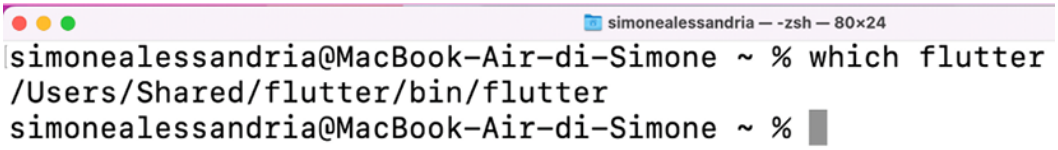


Figure 1.2: The result of the which flutter command on macOS Terminal

Windows command-line setup

To use Flutter on Windows, you need to save the location of the Flutter executable to your system's environment variables.

> **Note**
>
> These instructions assume you are using Windows 10 or 11. The screenshots in this section are taken from a Windows 11 PC.

You will now set up your environment variables for Flutter on Windows:

1. Click on the **Start** button, and type `env`. You should see the **System Properties** dialog window.

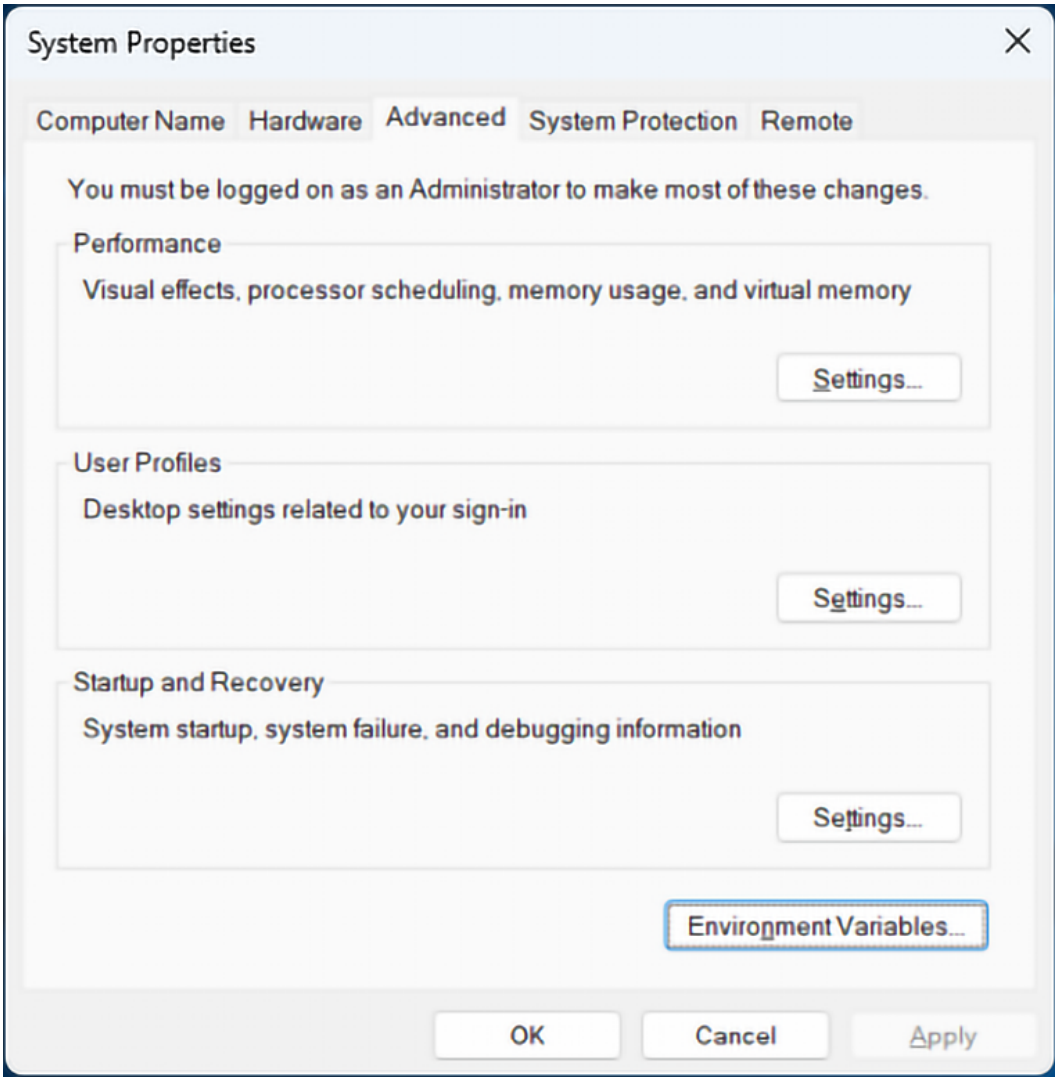


Figure 1.3: System Properties dialog window for Windows

2. At the bottom of the dialog, you will find an **Environment Variables…** button. Click on it.
3. In the next dialog, select the **Path** variable in the **User variables for sales** section and click the **Edit...** button:

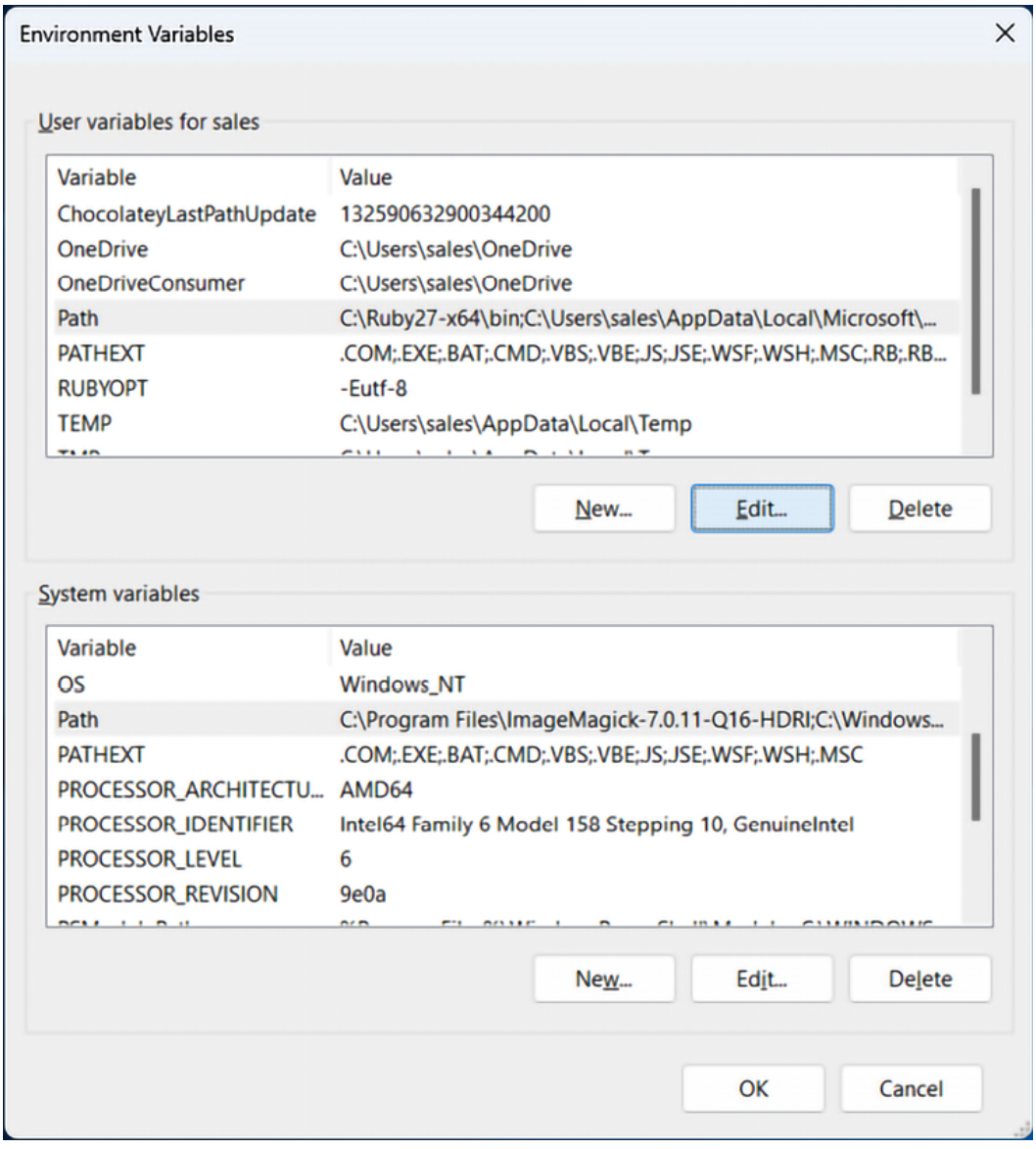


Figure 1.4: The Environment Variables dialog window for Windows

4. Finally, add the location where you installed Flutter to your path:

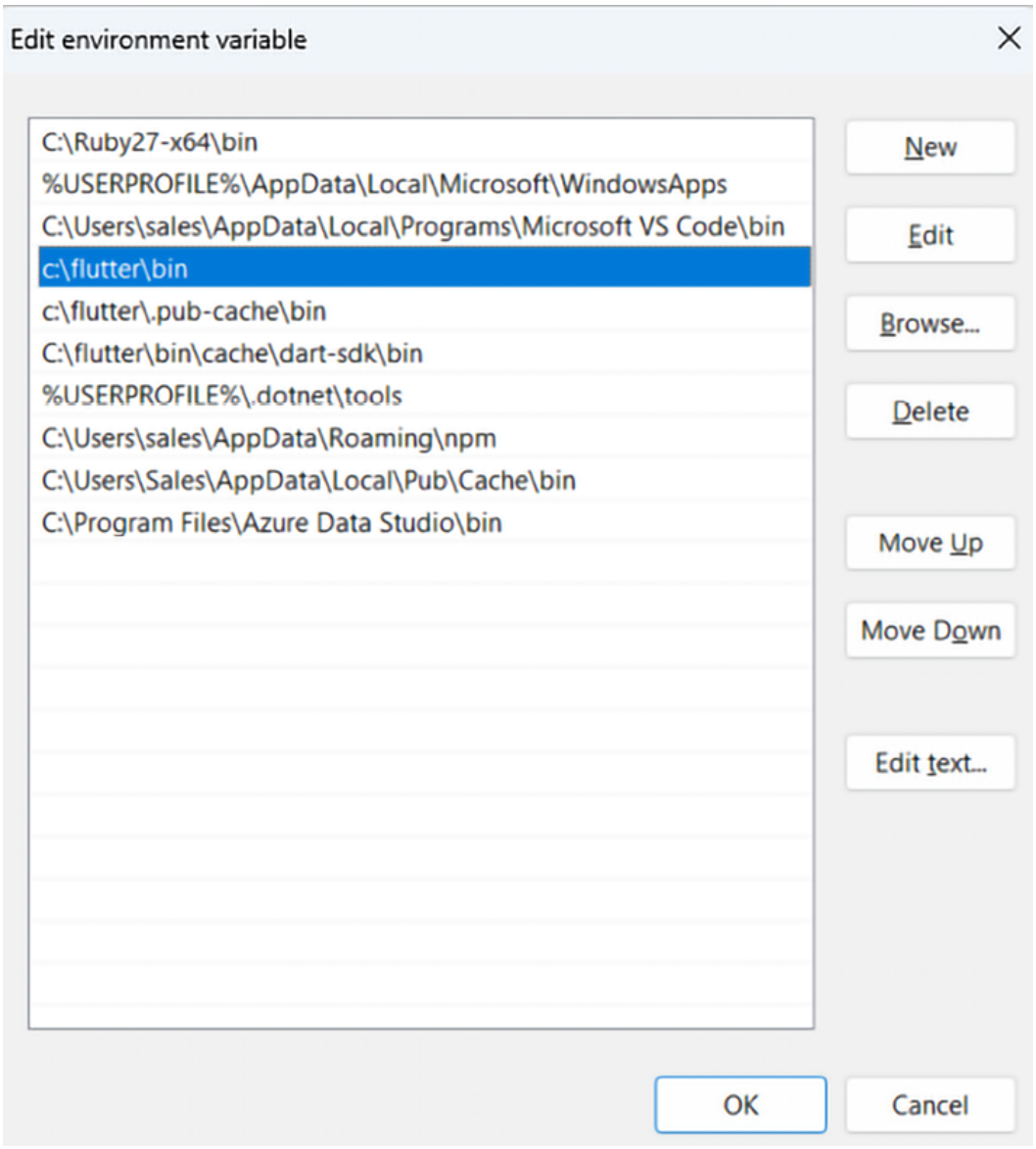


Figure 1.5: The Edit environment variable dialog window for Windows

5. Type `C:\Users\{YOUR_USER_NAME}\flutter\bin`, then select **OK**. Flutter should now be added to your path.
6. Restart your system.
7. Type `flutter` in the command line. You should see a message with some Flutter **command-line interface** (**CLI**) instructions. Flutter might optionally download more Windows-specific tools at this time.

Linux command-line setup

To use Flutter in Linux, you need to update the **PATH** variable in your system:

1. From your shell window, edit the `$HOME/.bashrc` file with your favorite editor.
2. Add the following line to the file:

```
export PATH="$PATH:[FLUTTER_PATH]/bin"
```

3. Open a new Terminal window to automatically source the file.
4. Type the following command:

```
which flutter
```

You should see the directory where you cloned (or installed) the Flutter SDK printed on the screen.

Note

In some cases, the instructions above may not work on your Linux distro. See the **Update path directly** section at https://docs.flutter.dev/get-started/install/linux#update-your-path.

Confirming your environment is correct with Flutter Doctor

Flutter comes with a tool called **Flutter Doctor**. This tool provides a list of everything that needs to be done to make sure that Flutter can run correctly. You are going to use Flutter Doctor as a guide during the installation process. This tool is also invaluable in checking whether your system is up to date.

In your Terminal window, type the following command:

```
flutter doctor
```

Flutter Doctor will tell you whether your platform SDKs are configured properly and whether Flutter can see any devices, including physical mobile devices, simulators and emulators, your desktop, and the web browser.

Configuring the iOS SDK

The iOS SDK is provided by a single application: **Xcode**. Xcode is one behemoth of an application; it controls all the official ways in which you will interact with Apple's platforms and is only available for macOS. As large as Xcode is, there are a few pieces of software that are missing. Two of these are community tools: **CocoaPods** and **Homebrew**. These are **package managers**, or programs that install other programs. You can use both of these tools to build iOS apps.

Downloading Xcode

The iOS SDK comes bundled with Apple's IDE, Xcode. The best place to get Xcode is through the Apple App Store:

1. Press *Command* + *Space* to open **Spotlight** and then type in `app store`:

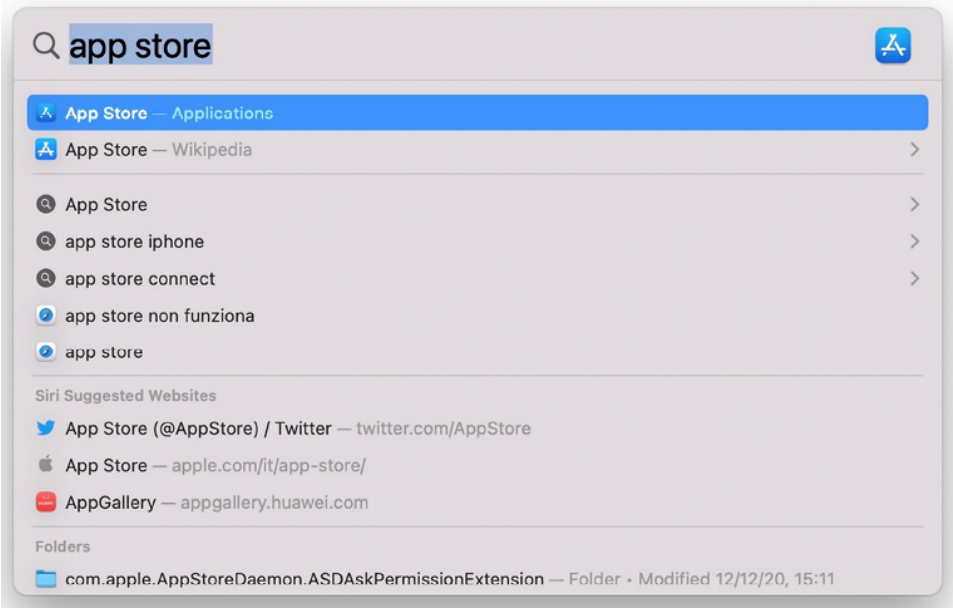


Figure 1.6: App Store results in Spotlight

As an alternative, you can just click on the menu in the top-left corner of your screen and select **App Store**, but keyboard shortcuts are much more fun.

2. After the **App Store** opens, search for `Xcode` and select **Download**:

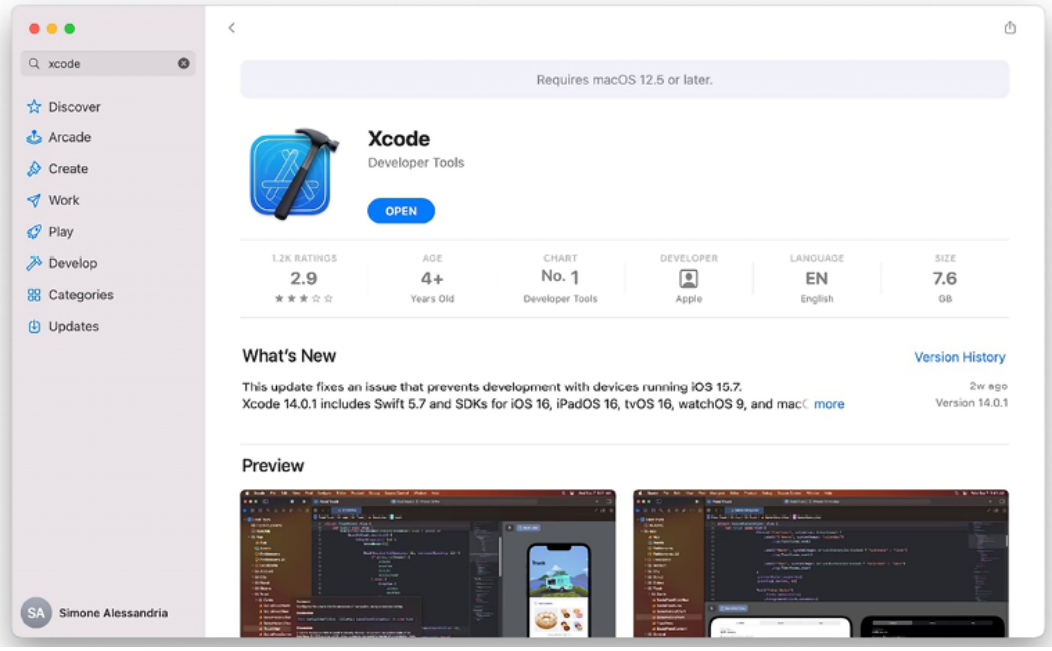


Figure 1.7: Xcode in the App Store

> **Note**
>
> To see the full-color version of Figure 1.7, and all the other images in this book, download the PDF file here: https://packt.link/WE615.

Xcode is a rather large application, so it may take a while to download. While Xcode is installing, you can get some of the smaller tools that you'll need for development. Let's take a look at how to install these tools in the following sections.

CocoaPods

CocoaPods is a popular community-led dependency manager for iOS development, both for *Swift* and *Objective-C* languages.

Basically, a dependency manager is a tool that helps you manage external libraries or packages required by your projects. When you write code, you sometimes need to use code written by other people or teams. A dependency manager automates the tasks to install, update, and delete these third-party packages.

In order to use some iOS-specific plugins in a Flutter app, it is recommended to install CocoaPods. This allows your app to access native libraries provided by iOS, required by certain plugins to function properly.

Flutter requires CocoaPods in its build process to link any libraries you have added to your project:

1. To install (or update) CocoaPods, type the following command in the Terminal:

```
sudo gem install cocoapods
```

> **Note**
>
> Since this command requires administrator privileges, you will likely be prompted to enter your password before continuing. This should be the same password that you use to log in to your computer.

2. After CocoaPods has finished installing, type the following command:

```
pod setup
```

This will configure your local version of the `cocoapods` repository, which can take some time.

CocoaPods can manage and install external libraries in a Flutter app that uses Objective-C or Swift code; it's not strictly required to build Flutter apps, but adding libraries manually, while possible, would be more difficult and time-consuming.

Xcode command-line tools

Command-line tools are used by Flutter to build your apps without needing to open Xcode. They are an extra add-on that requires your primary installation of Xcode to be complete:

1. Verify that Xcode has finished downloading and has been installed correctly.
2. After it is done, open the application to allow Xcode to fully configure itself.
3. Once you see the **Welcome to Xcode** screen appear, you can close the application:



Figure 1.8: Xcode Welcome screen

4. Type the following command in the Terminal window to install the command-line tools:

```
sudo xcode-select --switch
/Applications/Xcode.app/Contents/Developer
```

5. You may also need to accept the Xcode license agreement. Flutter Doctor will let you know if this step is required. You can either open Xcode and will be prompted to accept the agreement on the first launch, or you can accept it via the command line, with the following command:

```
sudo xcodebuild -license accept
```

After completing these steps, you'll be able to use the Xcode command-line tools without needing to open Xcode in your system.

Homebrew

Homebrew is a package manager used to install and manage applications on macOS. If CocoaPods manages packages that are specific to your project, then Homebrew manages packages that are global to your computer.

Homebrew can be installed with this command in your Terminal window:

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/ins
tall)"
```

We will be using Homebrew primarily as a mechanism to download and install other, smaller tools.

You can also find more information about Homebrew on its website: https://brew.sh.

Checking in with the Doctor

Now that we have all the platform tools for iOS, let's run Flutter Doctor one more time to make sure everything is installed correctly.

You may end up seeing this as a result:

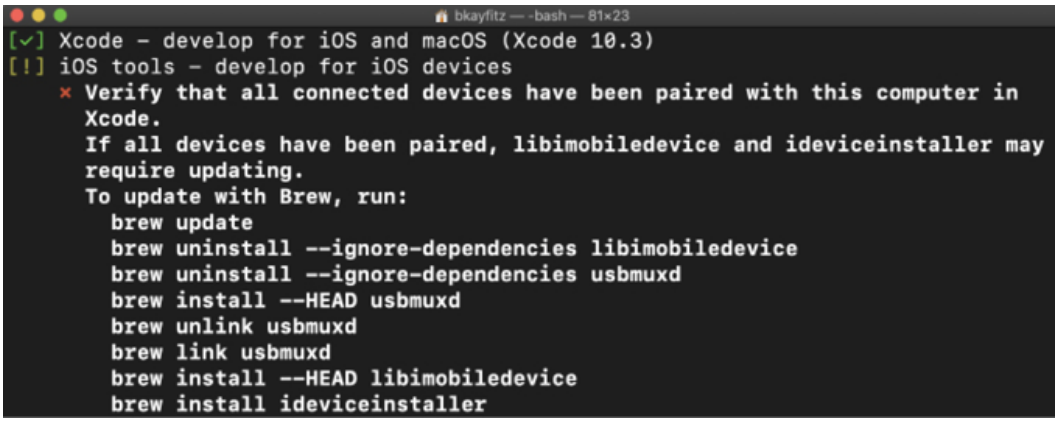


Figure 1.9: Flutter Doctor error

Remember how earlier you installed Homebrew? It's now going to come in handy. You now have two options to solve this problem: you can either copy/paste each one of these `brew` commands one by one into a Terminal window, or you can automate this with a single shell script.

The result will be the same, but let's create a single shell script:

1. Select and copy all the `brew` commands in *Step 2*, then enter `nano` again with the following command:

```
nano update_ios_toolchain.sh
```

2. Add the following commands in the file and then save and exit `nano`:

```
brew update
brew uninstall --ignore-dependencies libimobiledevice
brew uninstall --ignore-dependencies usbmuxd
brew install --HEAD usbmuxd
brew unlink usbmuxd
brew link usbmuxd
brew install --HEAD libimobiledevice
brew install ideviceinstaller
```

3. Run this script with the following command:

```
sh update_ios_toolchain.sh
```

4. To delete the script file after its execution, just type:

```
rm update_ios_toolchain.sh
```

5. When the script finishes, run `flutter doctor` one more time. Everything for the iOS side should now be green.

Configuring the Android SDK setup

Just like with Xcode, Android Studio and the Android SDK come hand in hand, which should make this process fairly easy. But also like Xcode, Android Studio is just the starting point. There are a bunch of tiny tools that you'll need to get everything up and running.

Installing Android Studio

Follow these steps to install Android Studio:

1. You can download Android Studio at https://developer.android.com/studio.

   The website will autodetect your operating system and only show the appropriate download link:

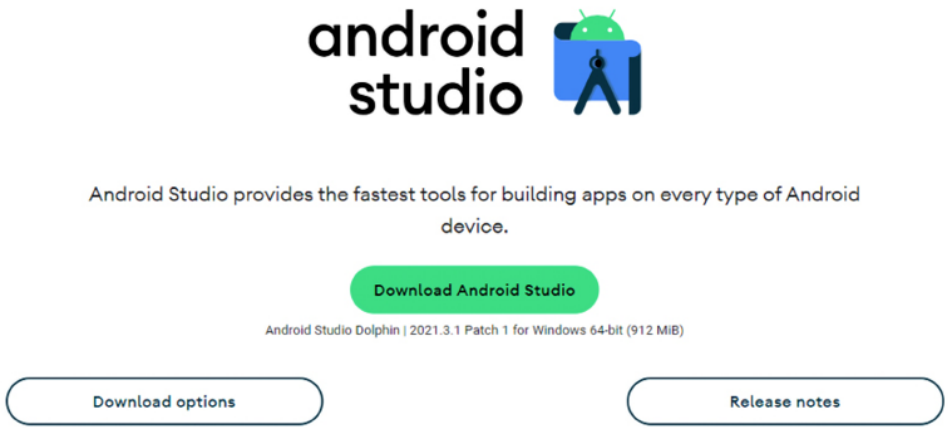


Figure 1.10: Android Studio download page

After Android Studio is installed, you'll need to download your target Android SDK(s). From the **Android Studio** menu, select **Preferences** and then type `android` into the search field:

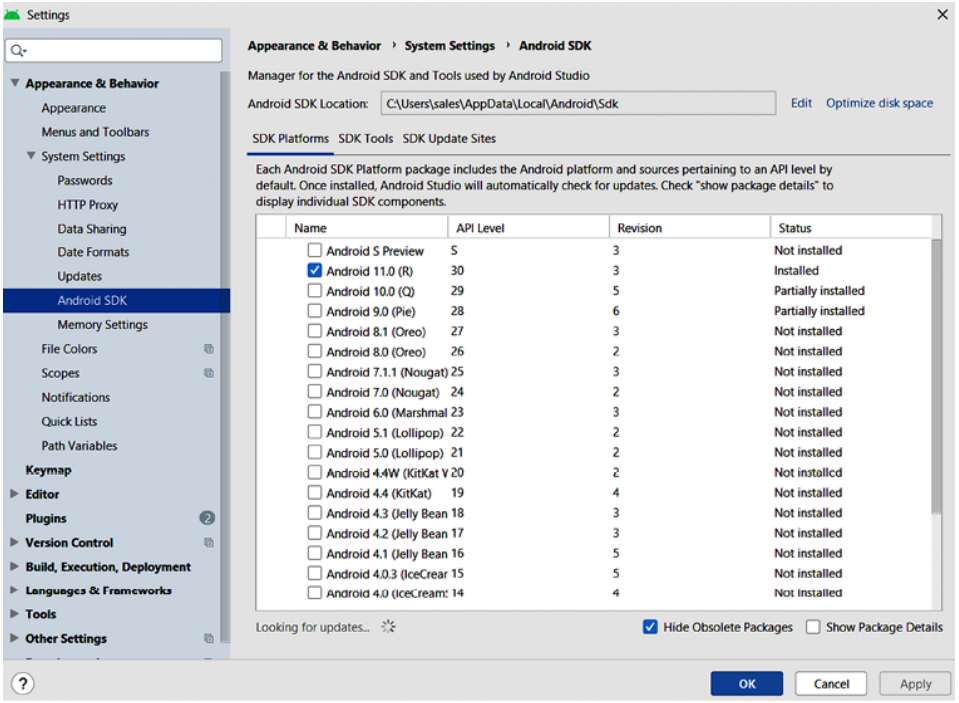


Figure 1.11: Android Studio SDKs screen

Pro Tip

While it may be tempting to grab the most recent version of the Android SDK, you might want to choose the second most recent version, because the Flutter SDK is sometimes a bit behind Android. In most cases, it shouldn't matter, but Android is notorious for breaking compatibility, so be aware of this.

If you ever need to change your version of the Android SDK, you can always uninstall and reinstall it from the screen in *Figure 1.11*

2. You will also need to download the latest build tools, emulator, SDK platform tools, SDK tools, **Hardware Accelerated Execution**

Manager (**HAXM**) installer, and support library.

3. Select the **SDK Tools** tab and make sure the required components are checked. When you click the **Apply** or **OK** button, the tools will begin downloading:

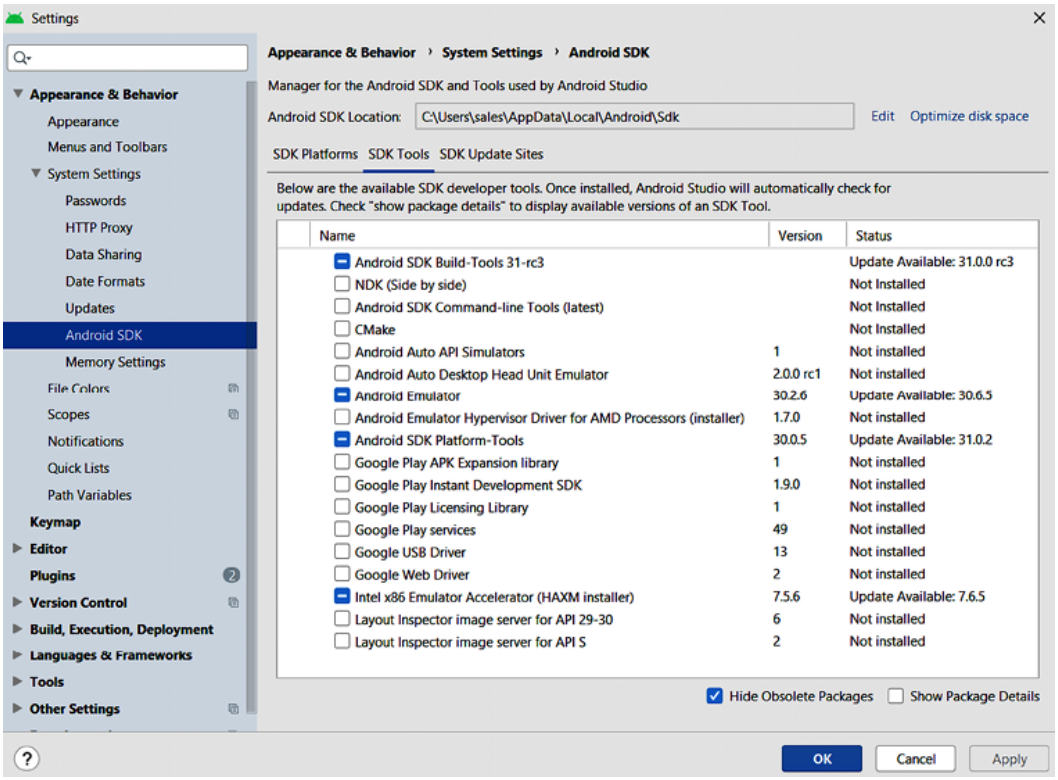


Figure 1.12: Android Studio Android SDK Tools screen

4. After everything finishes installing, run `flutter doctor` to check that everything is working as expected.

Creating an Android emulator

In order to run your app, you are going to need some kind of device to run it on. When it comes to Android, nothing beats the real thing. If you have access to a real Android device, you can use that device for development. It will also be less resource-consuming for your system, as your PC or Mac won't have to supply the RAM, processor, and disk of another device.

However, there are also advantages to using an Android emulator (and an iOS simulator).

When developing, it is often simpler to have a virtual device next to your code rather than having to carry around real devices with their required cables.

Follow these steps to set up your first emulator:

1. Select the **Android Virtual Device Manager** (**AVD Manager**) from the toolbar in **Android Studio**:

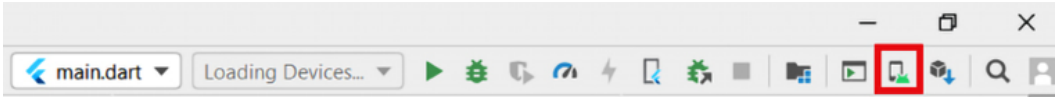


Figure 1.13: Android Studio toolbar

2. The first time you open the AVD Manager, you'll get a splash screen. Select the **Create Virtual Device...** button in the middle to start building your virtual device:

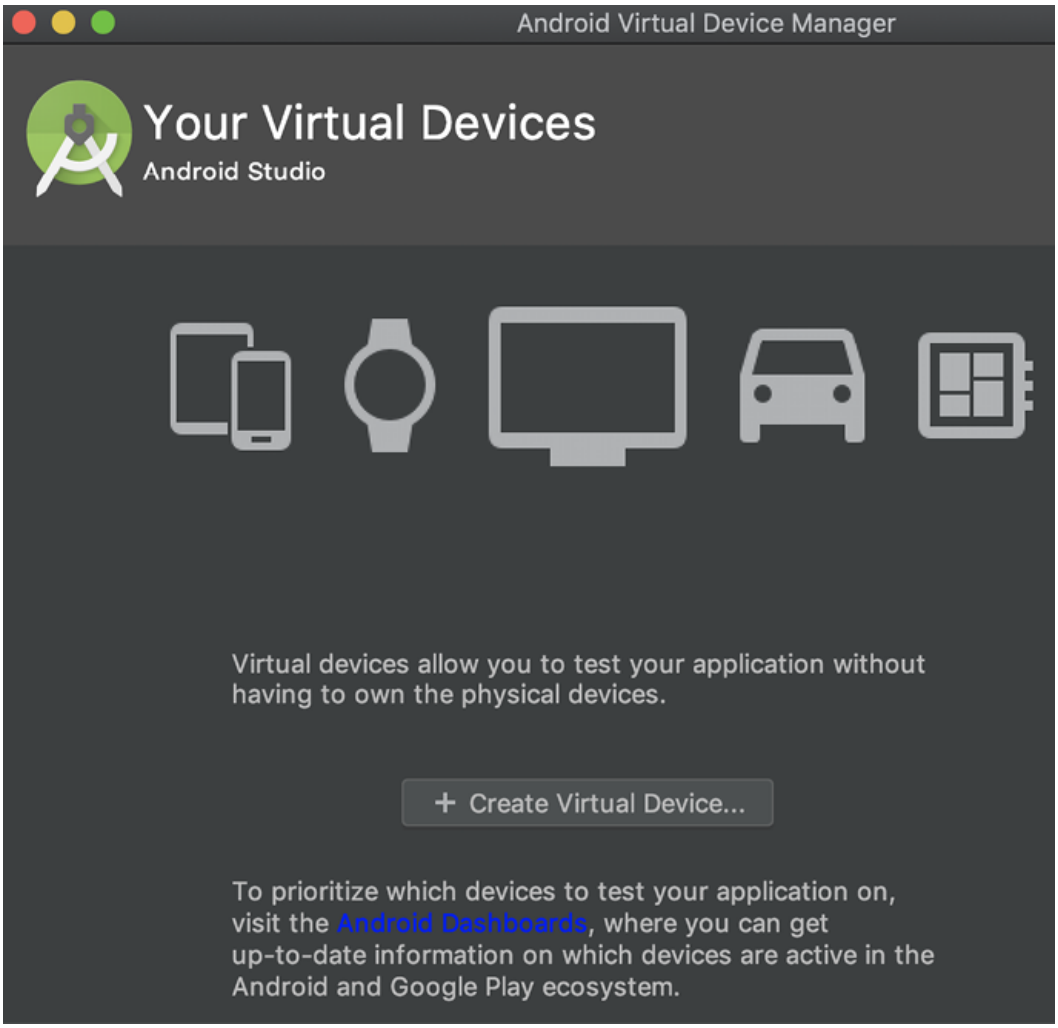


Figure 1.14: Android Studio Android "Your Virtual Devices" screen

3. The next text screen allows you to configure which Android hardware you want to emulate. I recommend using a **Pixel** device (any version will do):

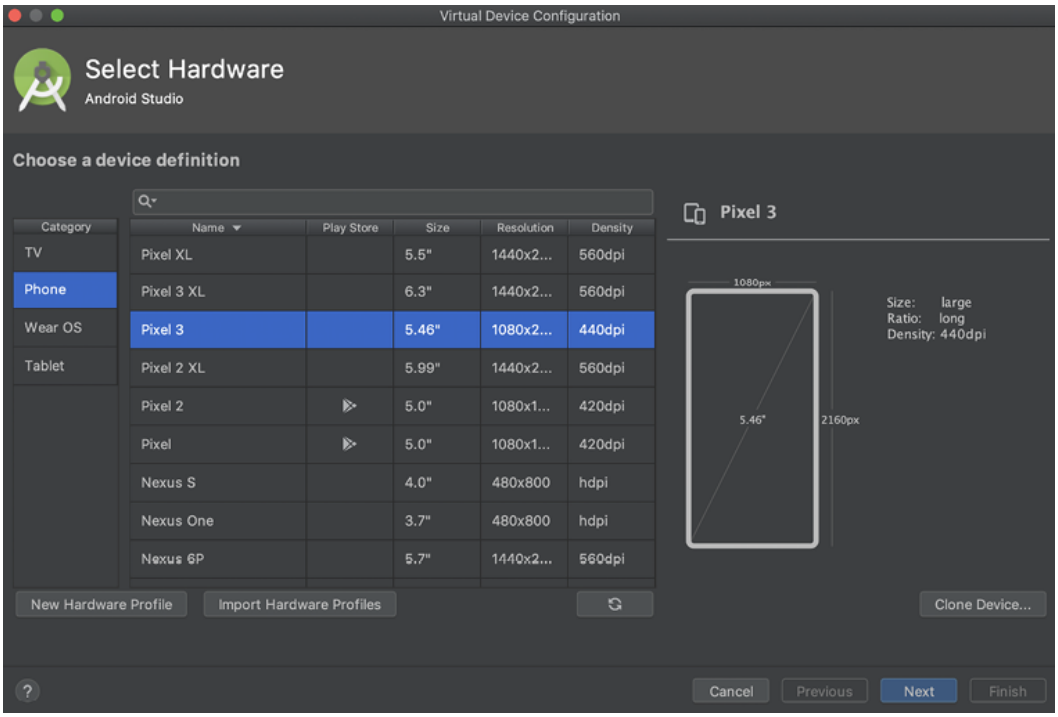


Figure 1.15: Android Studio AVD Select Hardware screen

4. On the next screen, you will have to pull down an Android runtime. For the most part, the most recent image will be sufficient. Each one of these images is several GB in size, so only download what you need:

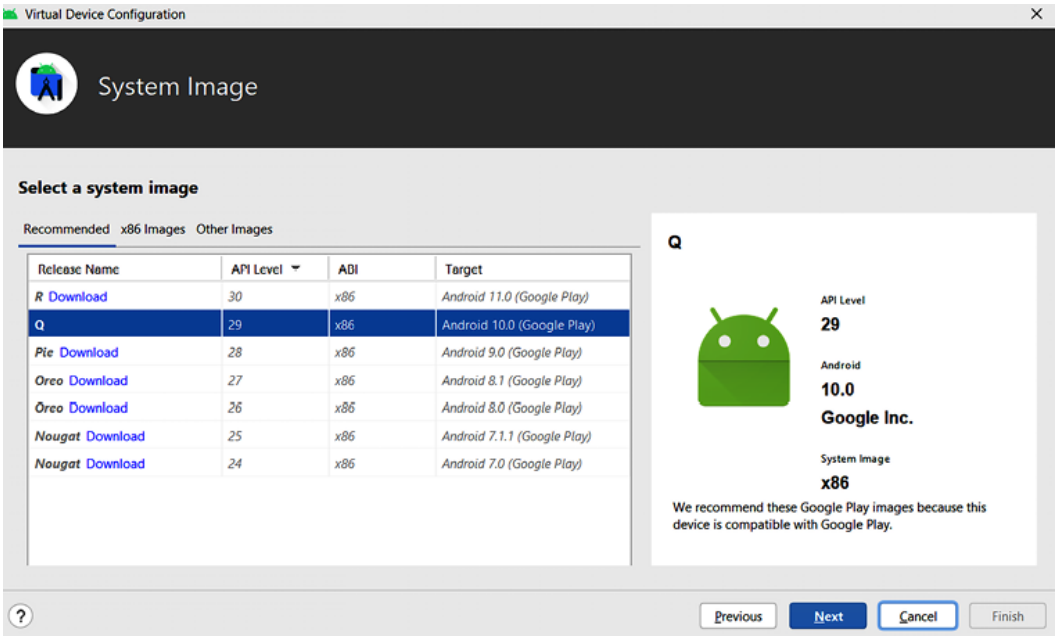


Figure 1.16: Android Studio AVD System Image screen

5. Click **Next** to create your emulator. You can launch the emulator if you want, but this is not necessary.
6. Once again, run `flutter doctor` to check your environment.
7. One final thing that you may have to do is accept all the Android license agreements. You can do this quickly from the Terminal line with this command:

```
flutter doctor --android-licenses
```

8. Keep typing `y` when prompted to accept all the licenses (I have a feeling you aren't reading them anyway).
9. Run `flutter doctor` one more time just for good measure. The Android SDK should now be fully configured.

Congratulations! The Flutter SDK should now be fully set up for both iOS and Android. In the next recipes in this chapter, we are going to explore some optional choices to customize your environment to fit your needs.

Which IDE/editor should you choose?

A developer's IDE is a very personal choice, and developers may engage in heated battles over the best tool to use.

Ultimately, the choice is dependent on which tool you are most productive and more capable in. If you find yourself fighting with the tool rather than just writing code, then it might not be the right choice. As with most things, it's more important to make choices based on what best fits your personal and unique style, rather than follow any prescribed doctrine.

You can develop Flutter apps with any editor, including Notepad and the Flutter CLI, but Flutter provides official plugins for four popular IDEs:

- Android Studio
- Visual Studio Code (VS Code)
- IntelliJ IDEA
- Emacs

Let's compare and configure all four and find out which one might be right for you.

Android Studio

Android Studio is a mature and stable IDE. Since 2014, Android Studio has been promoted as the default tool for developing Android applications. Before that, you would have had to use a variety of plugins for legacy tools such as Eclipse. One of the biggest arguments in favor of using Android Studio is that it's the easiest way to install the Android SDK, so you probably already have it installed.

To add the Flutter plugin, select the **Android Studio** menu, then select **Preferences**:

1. Click on the **Plugins** tab to open the Plugins Marketplace. Search for `Flutter` and install the plugin. This will also install the Dart plugin. You will then be prompted to restart Android Studio:

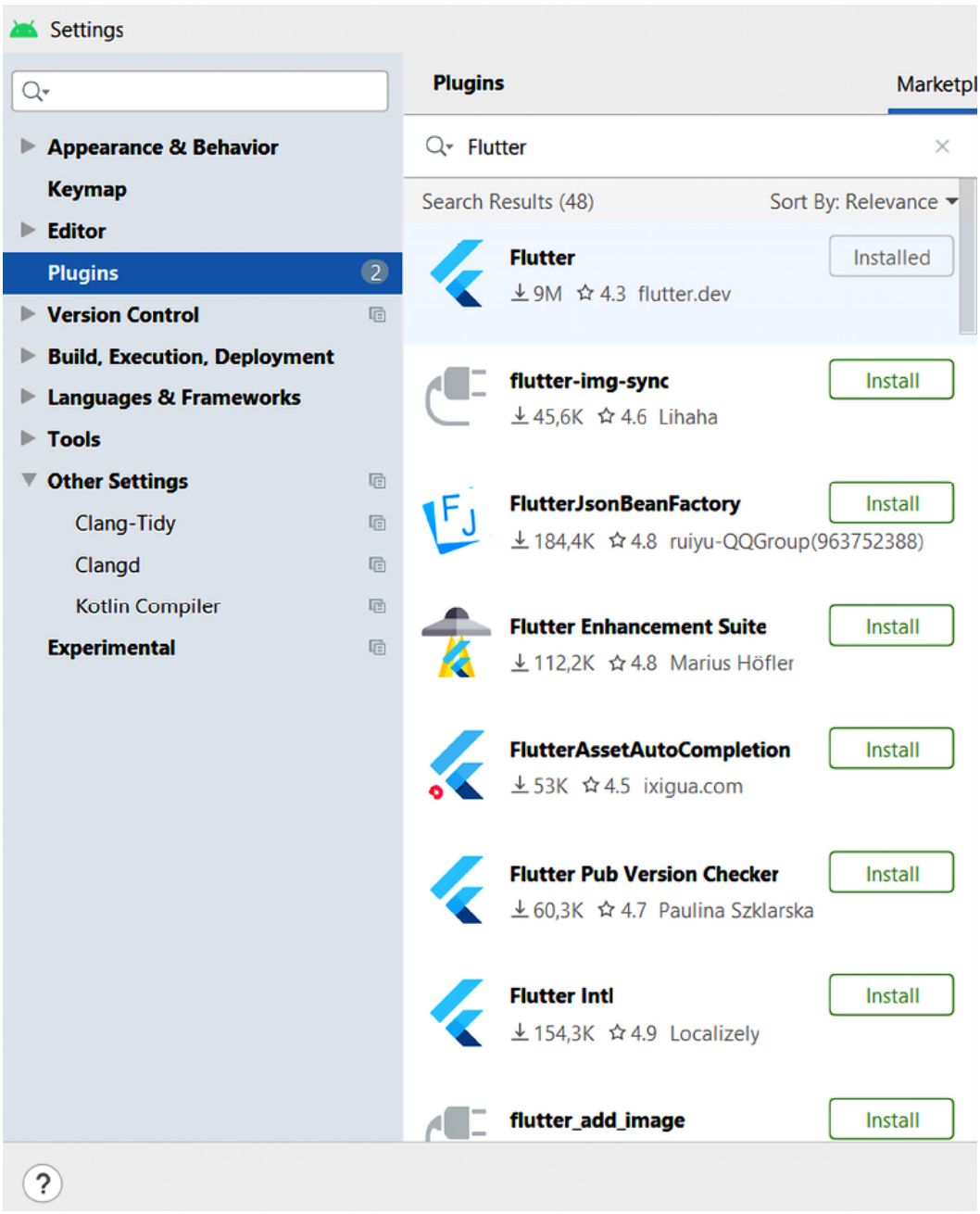


Figure 1.17: Android Studio Plugins screen

Android Studio is a very powerful tool. At the same time, the program can seem intimidating at first, with all the panels, windows, and too many options to enumerate. You will need to spend a few weeks with the program before it starts to feel natural and intuitive.

With all this power comes a consequence: Android Studio is a very demanding application. On a laptop, the IDE can drain your battery very quickly, so keep your power cable nearby. You should also make sure you have a relatively powerful computer; otherwise, you might spend more time waiting than writing code. See the hardware recommendations in the *Technical requirements* section at the beginning of this chapter for further details.

VS Code

VS Code is a free, lightweight, highly extensible tool from Microsoft that can be configured for almost any programming language, including Flutter.

You can download VS Code from https://code.visualstudio.com.

After you've installed the application, click on the fifth button in the left sidebar to open the **Extensions Marketplace**. Search for `flutter` and then install the extension:

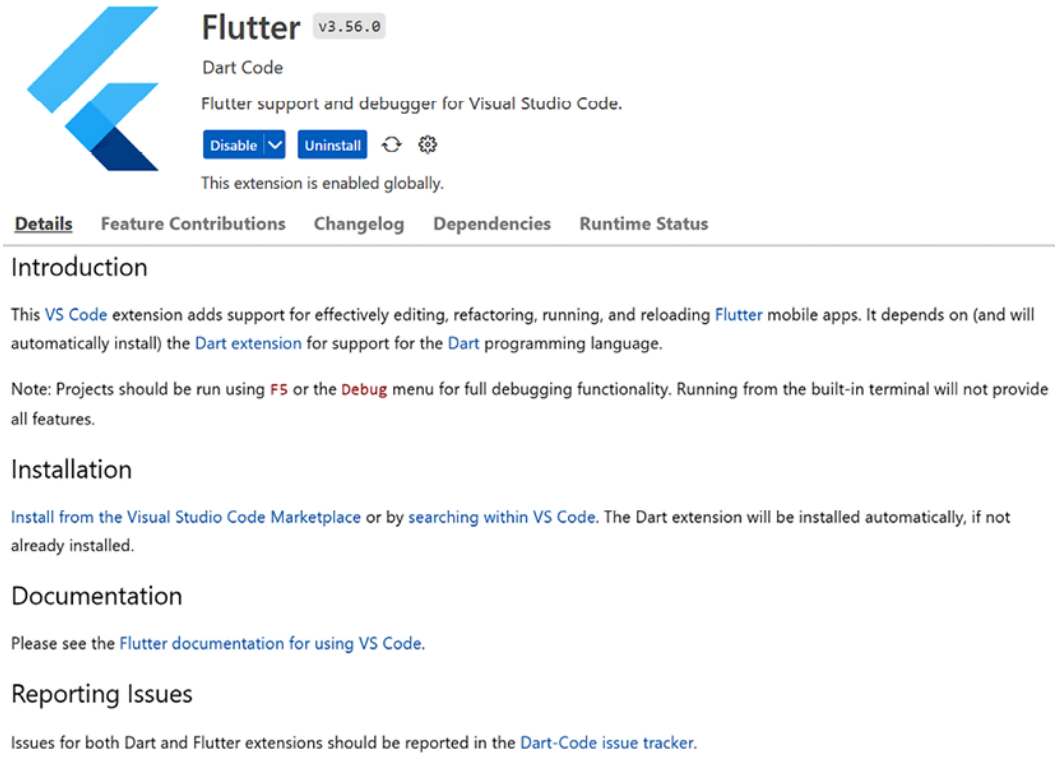


Figure 1.18: Visual Studio Code Flutter extension screen

This will also install the Dart plugin.

VS Code is much kinder on your hardware than Android Studio and has a wide array of community-written extensions. You will also notice that the UI is simpler than Android Studio, and your screen is not covered with panels and menus. This means that most of the features that you would see out in the open in Android Studio are accessible through keyboard shortcuts in VS Code.

Unlike Android Studio, most of the Flutter tools in VS Code are accessible through the **Command Palette**.

Press *Ctrl* + *Shift* + *P* on Windows or *Command* + *Shift* + *P* on a Mac to open the **Command Palette** and type `>Flutter` to see the available options. You can also access the **Command Palette** through the **View** menu:

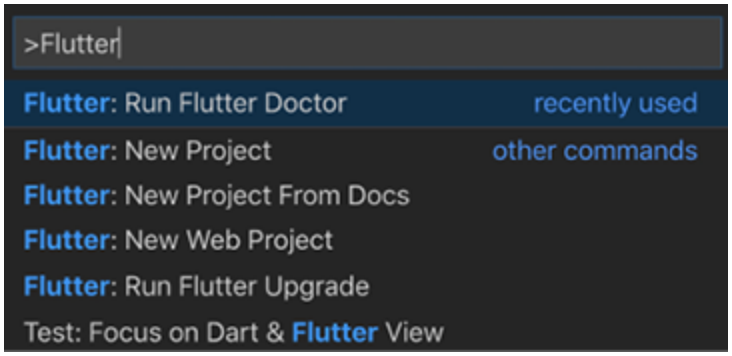


Figure 1.19: VS Code Command Palette

If you want a lightweight but complete environment that you can customize to your needs, then VS Code is the right tool for you. This is my favorite editor when developing Flutter apps.

IntelliJ IDEA

IntelliJ IDEA is another extremely powerful and flexible IDE. You can download the tool from this website: https://www.jetbrains.com/idea/. Currently, this is the only paid-for tool in this group.

If you look carefully, you'll probably notice that IntelliJ IDEA looks very similar to Android Studio, and that is no coincidence. Android Studio is really just a modified version of IntelliJ IDEA. This also means that all the Flutter tools we installed for Android Studio are the exact same tools that are available for IntelliJ IDEA.

So, why would you ever want to use IntelliJ IDEA if you already have Android Studio? Android Studio has removed many of the features in IntelliJ IDEA that aren't related to Android development. This means that if you are interested in web or server development, you can use IntelliJ IDEA to get the same experience.

Emacs

Emacs is the latest addition to the officially supported editors for Flutter. You can download it from https://www.gnu.org/software/emacs/download.html.

After you've installed the application, you need to enable **LSP** (**Language Server Protocol**) mode. This makes sure the editor communicates with a server that contains information about any language. You'll find information about the installation process at this link: https://emacs-lsp.github.io/lsp-mode/page/installation/.

Then, you need to install the `lsp-dart` package (https://emacs-lsp.github.io/lsp-dart/), which enables several great language tools you can leverage when using Flutter.

Emacs' most interesting feature is probably that it is part of the **GNU project**. The goal of the GNU project is to provide a free software environment that can be modified and redistributed by anyone. This means that you can run, copy, distribute, modify, or change Emacs as you see fit. It also means that you are totally in control of how the software works under the hood.

Other than that, Emacs is extremely lightweight, more than any other editor in this list. On the other hand, it's definitely less user-friendly and offers a

more “minimalistic” experience when developing with Flutter, so I would recommend it only to developers who are already familiar with it.

Picking the right channel

One final item we need to cover before diving into building apps is the concept of channels. Flutter segments its development streams into **channels**, which is really just a fancy name for Git branches. Each channel represents a different level of stability for the Flutter framework. Flutter developers will release the latest features to the **master channel** first. As these features stabilize, they will get promoted to the **beta channel**, and then to the **stable channel**.

Note

In previous Flutter versions, there was also a **dev channel**, between master and beta, but this was removed. The Flutter versioning policy will probably be updated in the near future, so please have a look at the Flutter build release page at https://github.com/flutter/flutter/wiki/Flutter-build-release-channels for the latest versioning updates.

Most of the time, you will probably want to stick to the stable channel. This will make sure that your code mostly runs without any issues. If you were interested in cutting-edge features that may not be completely finished, you’d probably be more interested in the master or beta channels.

In order to make sure you are using the stable channel:

1. In your Command Prompt or Terminal, type in the following command:

```
flutter channel
```

2. If you downloaded the Flutter pre-compiled SDK, you’ll probably see output that looks like this:



If you chose to clone the Flutter repository with Git, it defaults to the master channel. You'll generally want to stick to something more reliable.

3. Type in these commands:

```
flutter channel stable
flutter upgrade
```

4. This will switch the Flutter SDK to the stable channel and then make sure that you are running the most recent version.

Note

You may have noticed references to Git when switching channels. This is because, under the hood, Flutter channel is just a fancy name for a Git branch. If you were so inclined, you could switch channels using the Git command line, but you might also desynchronize your Flutter tool. Make sure to always run `flutter upgrade` in your Terminal after switching channels/branches.

Summary

By now, you should have a working Flutter environment set up. It is recommended that you rerun Flutter Doctor from time to time to check the status of your environment. The Doctor will also let you know when a new version of Flutter is available, which—depending on your channel—

happens every few weeks or months. Flutter is still a relatively young framework and growing fast, so you should always keep your environment up to date.

The Flutter team is also very receptive to helping you with any technical issues you might encounter. If you run into an issue that hasn't been documented yet, you can always reach out to the Flutter team directly on GitHub at https://github.com/flutter/flutter/issues.

Learning about the command line will become an invaluable skill. This can range from setting up and configuring your environment to writing build scripts, which we will do later in this book when we automate building and publishing the apps to the stores.

Packt Publishing offers several books and courses on command-line tools. Make sure to check them out at https://www.packtpub.com/application-development/command-line-fundamentals.

Another invaluable skill you should commit to is learning your editor's keyboard shortcuts. For VS Code, you can find a list here: https://code.visualstudio.com/docs/getstarted/keybindings. For Android Studio, have a look here: https://developer.android.com/studio/intro/keyboard-shortcuts.

In the next chapter, you'll see how to create your first Flutter app.

Join us on Discord

Read this book alongside other app developers.

Ask questions, participate in challenges, provide solutions to other readers, network and much more.

Scan the QR code or visit the link to join the community

https://packt.link/fluttercookbook



Creating Your First Flutter App

One of the greatest Flutter features is that it makes app development easy, or at least as easy as is possible today. It provides several pre-built components that can be easily customized to fit your needs.

One of the key elements of a Flutter project is the `lib` folder, which contains the main Dart code for your app. This is where you will be writing most of your code, including both layout and logic. Flutter projects also contain a file called `pubspec.yaml`, where you'll find the project's dependencies and other important configuration information.

In this chapter, you will see how to create the default "Hello World" app, and understand the structure of a Flutter project. By understanding a Flutter project's structure, you will be able to easily navigate and understand the code for your own apps.

In this chapter, you will learn:

- How to create a Flutter app
- How Flutter projects are structured
- How to run a Flutter app
- Hot reload—refresh your app without recompiling
- How to create a simple unit test

How to create a Flutter app

There are two main ways to create a Flutter app: either via the command line or in your preferred IDE. We're going to start by using the command line to get a clear understanding of what is going on when you create a new app.

For later apps, it's perfectly fine to use your IDE, but just be aware that all it is doing is calling the command line under the hood.

Before you begin, it's helpful to have an organized place on your computer to save your projects. This could be anywhere you like, as long as it's consistent. While there are no strict rules for folder names, there are a few conventions you should follow. For directories:

- Use lowercase letters and separate words with underscores: for example, `project_name`.
- Avoid spaces and special characters in your folders and file names.
- When choosing a folder for a Flutter project, you should avoid synchronized spaces like Google Drive or OneDrive, as they might decrease performance and create conflicts when working in teams.

As such, before creating your apps, make sure you have created a directory where your projects will be saved.

How to do it...

Flutter provides a tool called `flutter create` that will be used to generate projects. There are a whole bunch of flags that we can use to configure the app, but for this recipe, we're going to stick to the basics.

Note

If you are curious about what's available for any Flutter command-line tool, in your Terminal simply type `flutter <command> --help`. In this case, it would be `flutter create --help`. This will print a list of all the available options and examples of how to use them.

In the directory you have chosen to create your Flutter projects, type this command to generate your first project:

```
flutter create hello_flutter
```

- This command assumes you have an internet connection since it will automatically reach out to the official repositories to download the project's dependencies.
- If you don't currently have an internet connection, type the following instead:

```
flutter create --offline hello_flutter
```

Pro Tip

You will eventually need an internet connection to synchronize your packages, so it is recommended to check your network connection before creating a new Flutter project.

Now that a project has been created, let's run it and take a look. You have several options for your target:

- Connecting a device to your computer
- Starting an emulator/simulator
- Running the app on a Chrome or Edge web browser
- Running the app on your desktop

1. Type this command to see the devices currently available on your computer:

```
flutter devices
```

2. To specifically see the emulators you have installed on your system, type:

```
flutter emulators
```

3. You should see a list of the available emulators/simulators. Now, to boot an emulator on your system:

   - On Windows/Linux, type the command:

     ```
     flutter emulators --launch [your device name, like:
     Nexus_5X_API_30]
     cd hello_flutter
     flutter run
     ```

   - On a Mac, type the command:

     ```
     flutter emulators --launch apple_ios_simulator
     cd hello_flutter
     flutter run
     ```

4. To run your app on one of the available devices, type the following command:

   ```
   flutter run -d [your_device_name]
   ```

5. After your app has finished building, you should see a demo Flutter project running in your emulator:

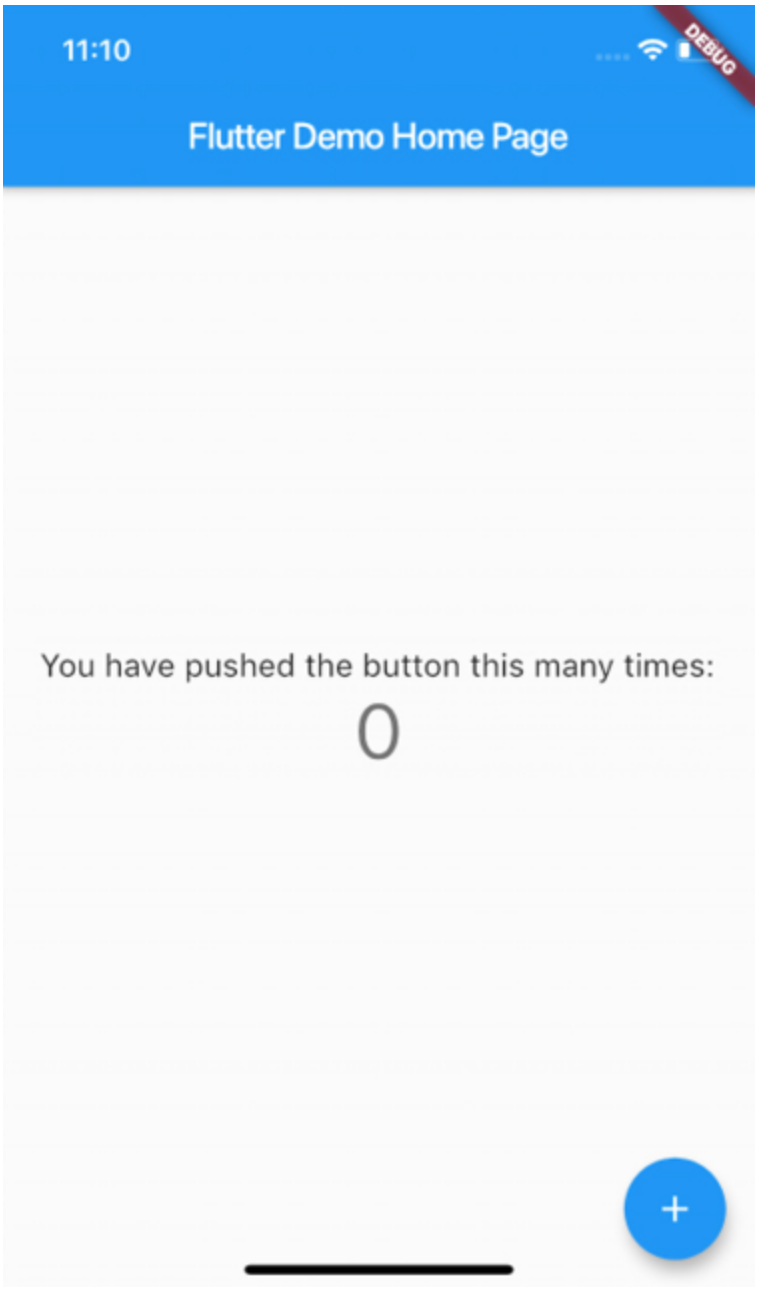


Figure 2.1: Flutter's default "Hello World" app in an iOS Simulator

6. Go ahead and play around with it. The circular button at the bottom right of the screen is called a **Floating Action Button**, and it's a **widget** (much, much more about that in the remainder of this book, starting from *Chapter 4*, *Introduction to Widgets*!).
7. When you are done, type `q` in the terminal to close your app.

How to choose a platform language for your app

Both iOS and Android are currently in the middle of a revolution of sorts. When both platforms started over 10 years ago, they used the Objective-C programming language for iOS, and Java for Android. These are great languages, but sometimes can be a little long and complex to work with.

To solve this, Apple has introduced Swift for iOS, and Google has adopted Kotlin for Android.

These languages are currently automatically selected for new apps.

If you want to use the older languages (for compatibility reasons) when creating an app, enter this command into your Terminal:

```
flutter create \
  --ios-language objc \
  --android-language java \
  hello_older_languages
```

In this way, Objective-C and Java will be chosen. You are also never locked into this decision; if later down the road you want to add some Kotlin or Swift code, there is nothing stopping you from doing so.

It’s important to keep in mind that the majority of your time will be spent writing Dart code. Whether you choose Objective-C or Kotlin, this won’t change much.

Where do you place your code?

The files that Flutter generates when you build a project should look something like this:

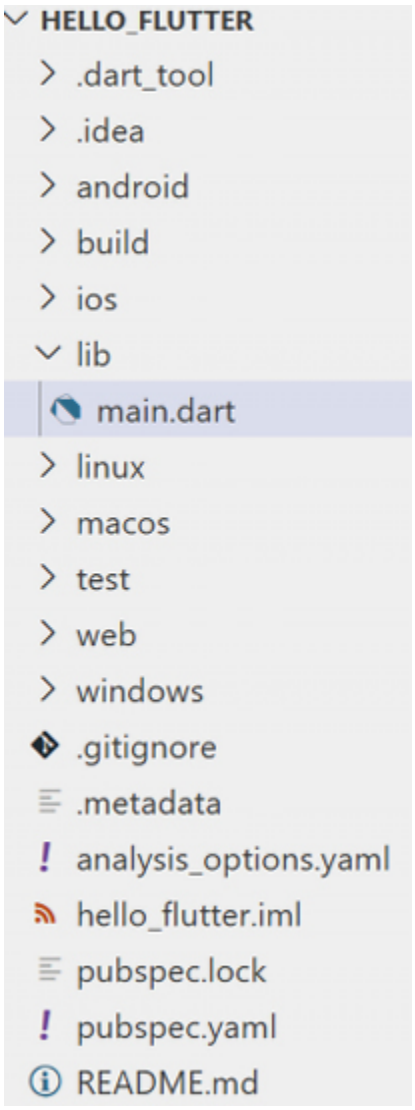


Figure 2.2: A Flutter project structure

The main folders in your projects are listed here:

- `android`
- `build`
- `ios`
- `lib`
- `linux`
- `macos`

- `test`
- `web`
- `windows`

The `android`, `ios`, `linux`, `web`, and `windows` folders contain the platform shell projects that host our Flutter code. You can open the `Runner.xcworkspace` file in Xcode or the `android` folder in Android Studio, and they should run just like normal native apps. Any platform-specific code or configurations will be placed in these folders. While Flutter uses a single code base for all platforms, when you build your app, it generates the specific code for each platform using these folders.

To increase performance when you build your app, Dart and Flutter support both **Just-In-Time** (**JIT**) and **Ahead-Of-Time** (**AOT**) compilation.

JIT compiling takes the source code of an app and translates it into machine code *during execution*. This means *faster development*, as the code can be tested while updating, and your code can be recompiled while the app is running.

With AOT compilation, the code is compiled *before* the program execution. This builds a machine code executable that can then be run on your target machine. This means *better performance* for your apps.

Simplifying things a little, Flutter uses JIT for *debug* builds and AOT for *release* builds, taking the best of the two worlds.

The `build` directory contains all the artifacts that are generated when you compile your app. The contents of this folder should be treated as temporary files since they change every time you run a build. You should also add this folder to your `gitignore` file so that it won't bloat your repository.

Note

The `gitignore` file contains a list of files and folders that Git should ignore when tracking changes in a project. As soon as you initialize a

project with Git, this file will be added to your project's root directory.

The `lib` folder is the heart and soul of your Flutter app. This is where you put all your Dart code. When a project is created for the first time, there is only one file in this directory: `main.dart`. Since this is the main folder for the project, you should keep it organized. We'll be creating plenty of subfolders and recommending a few different architectural styles throughout this book.

The next file, `pubspec.yaml`, holds the configuration for your app. This configuration file uses a markup language called **YAML Ain't Markup Language** (**YAML**), which you can read more about at https://yaml.org. In the `pubspec.yaml` file, you'll declare your app's name, version number, dependencies, and assets. `pubspec.lock` is a file that gets generated based on the contents of your `pubspec.yaml` file. It can be added to your Git repository, but it shouldn't be edited manually.

Finally, the last folder is `test`. Here, you can put your unit and widget tests, which are also just Dart code. As your app expands, automated testing will become an increasingly important technique to ensure the stability of your project. You will create a test in the *Creating a unit test* recipe later in this chapter.

Hot reload—refresh your app without recompiling

Probably one of the most important features in Flutter is **stateful hot reload**. Flutter has the ability to inject new code into your app while it's running, without losing your position (**state**) in the app. The time it takes to update code and see the results in an app programmed in a platform language could take up to several minutes. In Flutter, this edit/update cycle is down to seconds. This feature alone gives Flutter developers a competitive edge.

There is a configuration that allows executing a hot reload every time you save your code, causing the whole feature to become almost invisible.

In **Android Studio/IntelliJ IDEA**, open the **Preferences** window and type `hot` into the search field. This should quickly jump you to the correct setting:

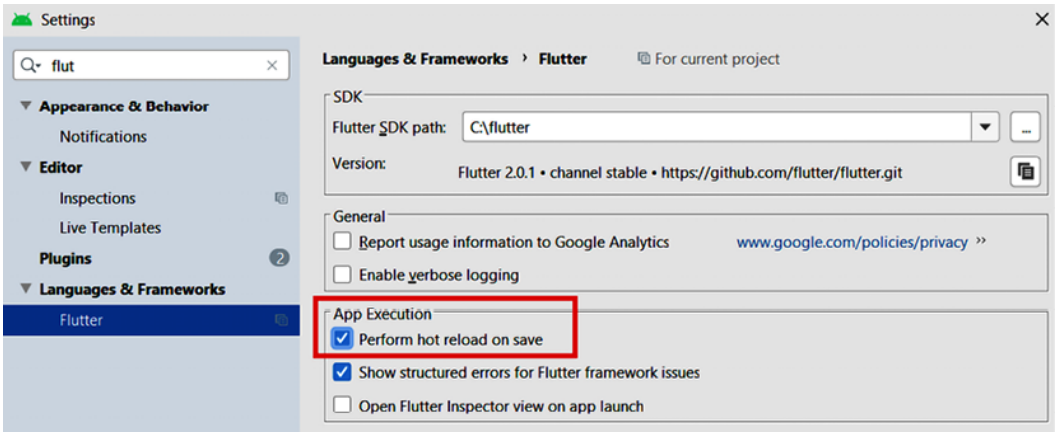


Figure 2.3: Android Studio Flutter hot reload configuration

Verify that the **Perform hot reload on save** setting is checked. While you are there, double-check that **Format code on save** is also checked.

In **VS Code**, open your **Settings** from **File** > **Preferences** > **Settings**. Type `flutter hot reload` in the search field. Here you can specify the **Flutter Hot Reload On Save** option with one of the available values:

- **never**: No hot reload executed when saving
- **all**: Hot reload triggered for both manual save and autosave
- **allDirty**: Hot reload triggered for both manual save and autosave only if the saved file had changes
- **manual**: Hot reload for manual save only
- **manualDirty**: Hot reload for manual save only if the saved file had changes

I would choose the **all** option if performance on your machine does not degrade, as it frees you from having to save manually when you make changes.

Let's see this in action:

1. In Android Studio or IntelliJ IDEA, open the Flutter project you created earlier by selecting **File** > **Open**. Then, select the `hello_flutter` folder.
2. After the project loads, you should see a toolbar in the top-right corner of the screen with a green **play** button. Press that button to run your project:



Figure 2.4: Android Studio toolbox

3. When the build finishes, you should see the app running in the emulator/simulator. For the best effect, adjust the windows on your computer so you can see both, side by side:

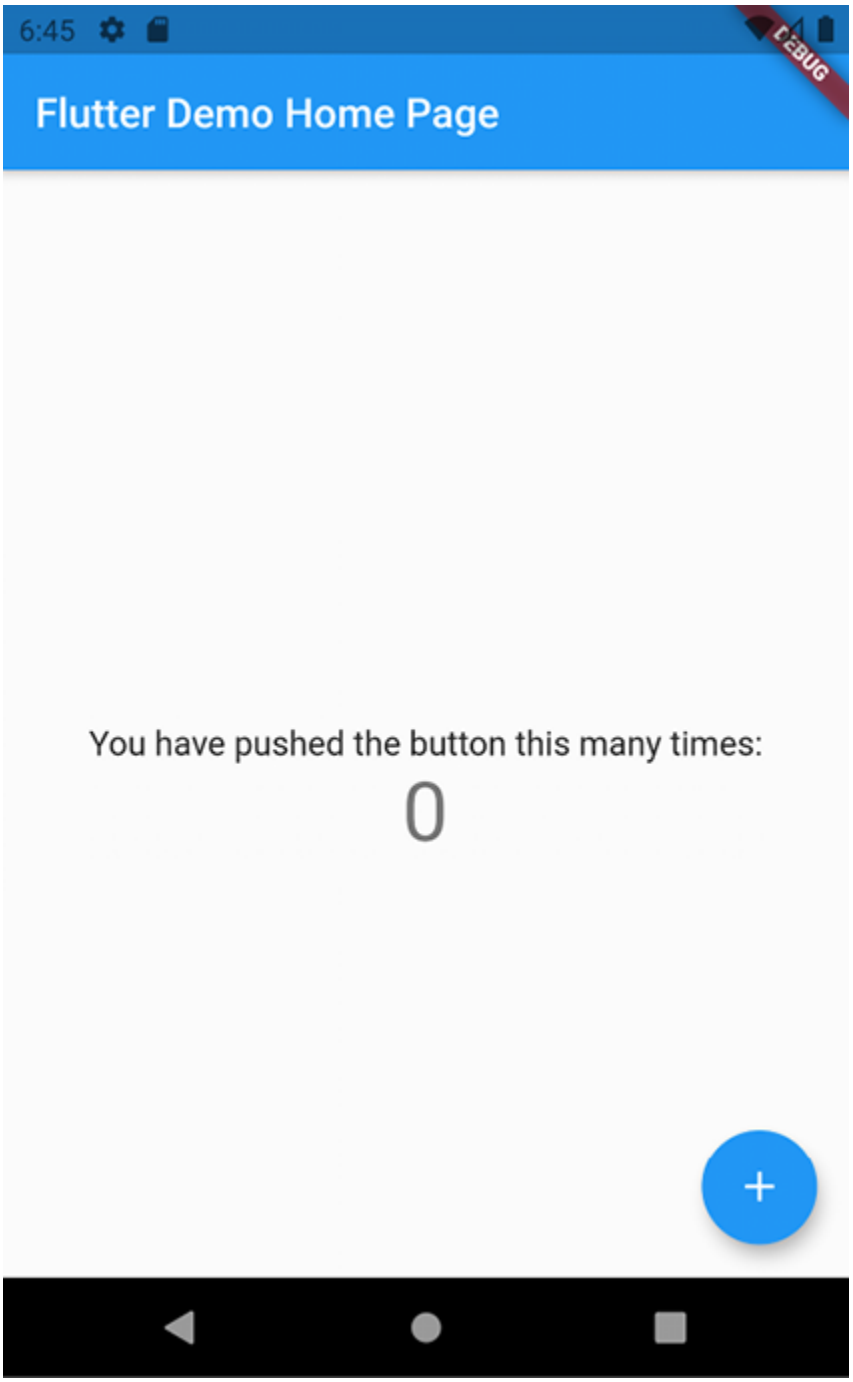


Figure 2.5: The Flutter default project on an Android emulator

4. In the `main.dart` file in the `lib` folder, update the primary swatch to `green`, as shown in the following code snippet, and hit **Save**:

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});
```

```
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.green,
      ),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
```

5. When you save the file, Flutter will repaint the screen:

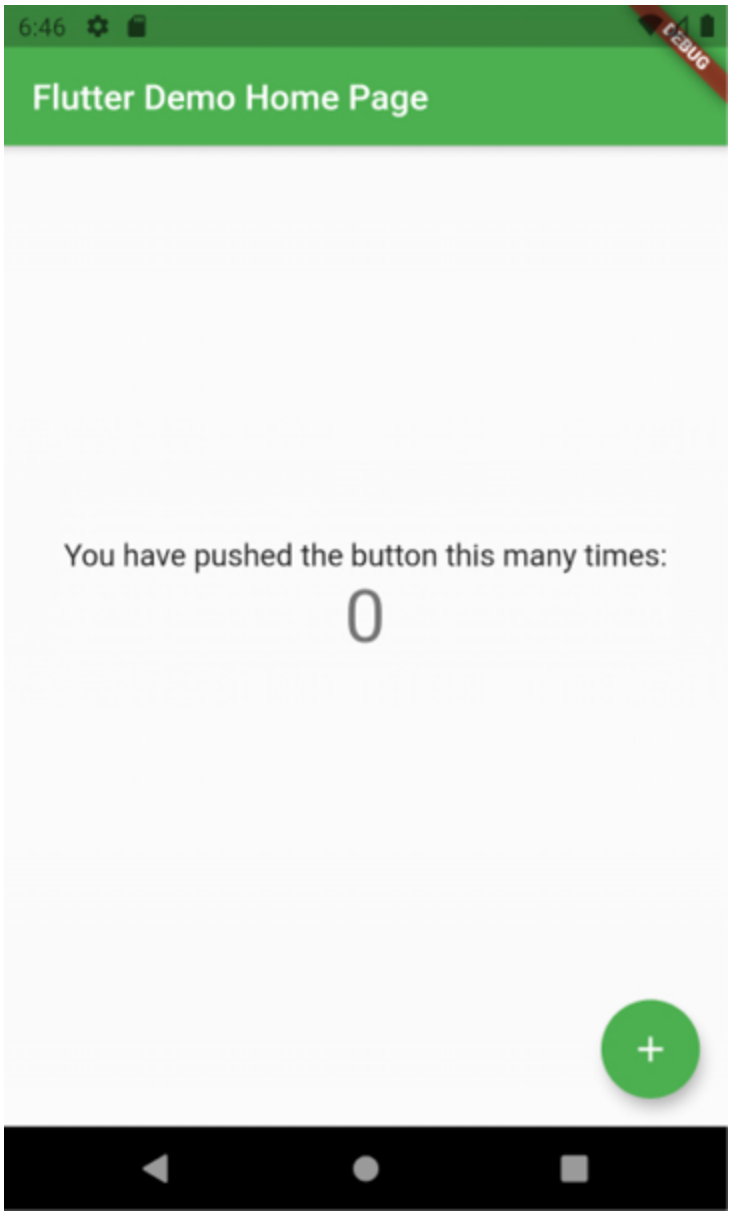


Figure 2.6: The default app after a color change

In VS Code, the pattern is very similar:

6. Click on the triangle on the left of the screen, then on the **Run and Debug** button:

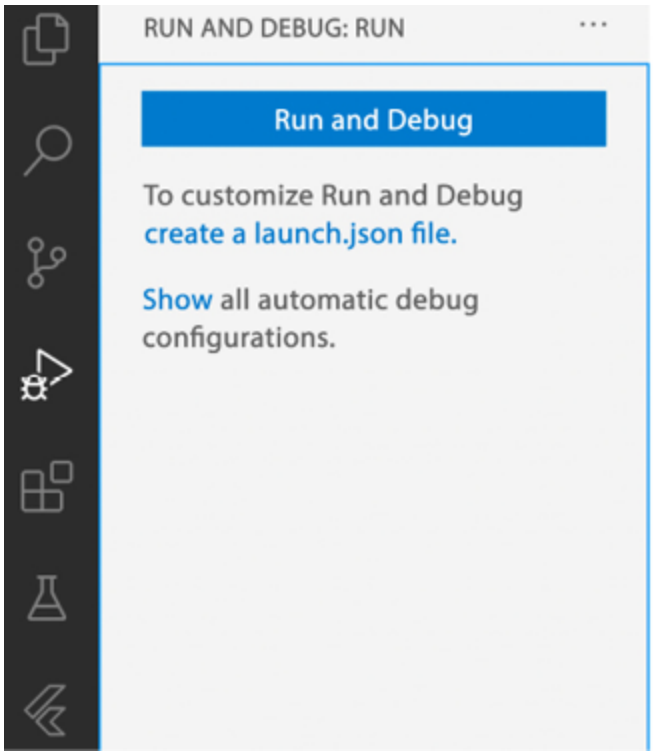


Figure 2.7: Visual Studio Code Run and Debug menu

7. Update the primary swatch to `green`, as shown in the following code snippet, and hit the **Save** button:

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.green,
      ),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
```

8. Only if your app does not update, click on the **Hot reload** button (denoted by a lightning bolt) from the debug toolbar or press *F5*. This will update the color of your app to green:



Figure 2.8: Visual Studio Code debug toolbar

It may seem simple now, but this small feature will save you hours of development time in the future!

Creating a unit test

There are several advantages of using unit tests in Flutter, including improving the overall quality of your code, ease of debugging, and better design. When writing good tests, you may also reduce the time required to maintain your code in the long run. Writing unit tests requires some practice, but it's well worth your time and effort.

In this recipe, you'll see how to write a simple unit test.

Getting ready

You should have created the default Flutter app as shown in the previous recipe before writing a unit test.

How to do it...

You will now update the default Flutter app: you will change the color of **You have pushed the button this many times:**. The text will be red for even numbers, and green for odd numbers, as shown in *Figure 2.9*:

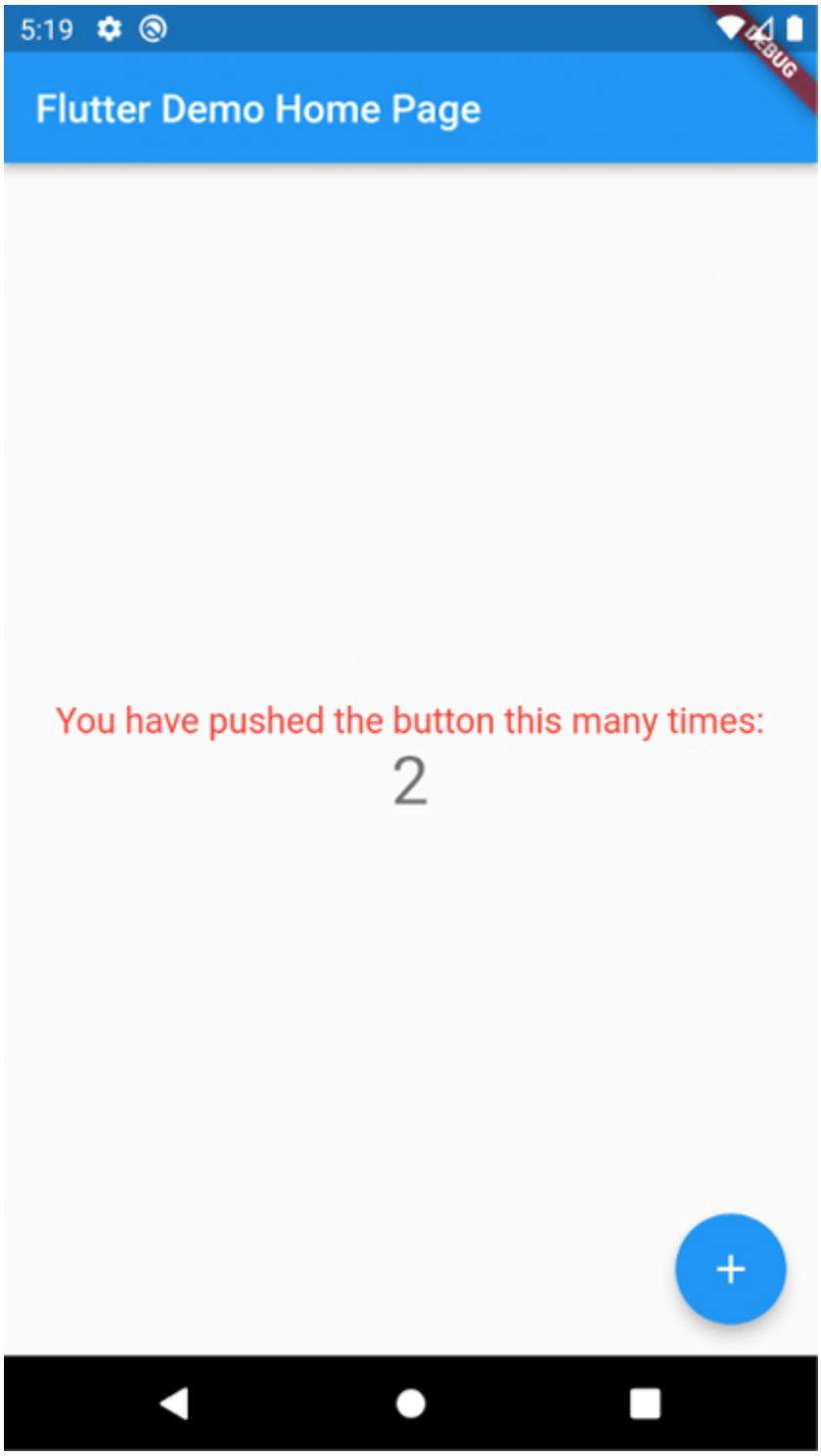


Figure 2.9: Red text for even numbers

1. In the `main.dart` file in the `lib` folder, at the bottom of the file, and out of any class, create a method that returns true when the number

passed is even, and false when the number is odd:

```
bool isEven(int number) {
    if (number % 2 == 0) {
      return true;
    } else {
      return false;
    }
  }
```

2. At the top of the `_MyHomePageState` class, under the `int _counter = 0;` declaration, declare `Color`, and set it to red:

```
Color color = Colors.red;
```

3. In the `_incrementCounter` method, edit the `setState` call, so that it changes the `color` value:

```
void _incrementCounter() {
    setState(() {
      _counter++;
      if (isEven(_counter)) {
        color = Colors.red;
      } else {
        color = Colors.green;
      }
    });
  }
```

4. In the `build` method, edit the text containing **You have pushed the button this many times:**, so that you change its color and size, and remove its `const` keyword, as shown below:

```
Text(
  'You have pushed the button this many times:',
  style: TextStyle(
     color: color,
     fontSize: 18,
  )
),
```

5. Run the app; you should see the color of the text changing each time you press the button.

Now that the app is ready, let's test whether the `isEven` function works as expected.

1. In the tests folder, create a new file, called `unit_test.dart`.
2. At the top of the new file, import the `flutter_test.dart` library and your project's `main.dart` file (note that you may have to change `hello_flutter` to your package name if you named your app differently):

```
import 'package:flutter_test/flutter_test.dart';
import 'package:hello_flutter/main.dart';
```

3. Create a `main` method under the `import` statements:

```
void main() {}
```

4. In the `main` method, call the `test` method, passing `Is Even` as its name and calling the `isEven` method twice, and checking its results, as shown here:

```
void main() {
  test('Is Even', () {
    bool result = isEven(12);
    expect(result, true);
    result = isEven(123);
    expect(result, false);
  });
}
```

5. Run the test by typing `flutter test` in your Terminal. You should see the **All tests passed!** message.
6. Instead of writing a single `test` method, let's create two separate tests:

```
void main() {
    test('Is Even', () {
```

```
      bool result = isEven(12);
      expect(result, true);
    });
    test('Is Odd', () {
      bool result = isEven(123);
      expect(result, false);
    });
}
```

7. Run the tests with the `flutter test` command in your Terminal, and note the success message that appears again in the Terminal: `All tests passed.`
8. Make one of the tests fail:

```
void main() {
    test('Is Even', () {
      bool result = isEven(12);
      expect(result, true);
    });
    test('Is Odd', () {
      bool result = isEven(123);
      expect(result, true);
    });
}
```

9. Run the test again, with the `flutter test` command in your Terminal. You should see an error, as shown in *Figure 2.10*:

```
PS C:\progetti in corso\cookbook\test_app> flutter test
00:01 +1 -1: C:\progetti in corso\cookbook\test_app\test\unit_test.dart: Is Odd [E]
  Expected: <true>
    Actual: <false>

  package:test_api                                  expect
  package:flutter_test/src/widget_tester.dart 460:16  expect
  test\unit_test.dart 12:7                          main.<fn>


To run this test again: c:\flutter\bin\cache\dart-sdk\bin\dart.exe test C:\progetti in corso\cookbook\test_app\test\unit_test.dart -p v
m --plain-name 'Is Odd'
00:02 +3 -1: Some tests failed.
PS C:\progetti in corso\cookbook\test_app>
```

Figure 2.10: Terminal showing failing test

10. Note the error message, showing the name of the failing test (**Is Odd** in this case).
11. Let's group the two tests together using the `group` method:

```
void main() {
  group('Iseven group', () {
    test('Is Even', () {
      bool result = isEven(12);
      expect(result, true);
    });
    test('Is Odd', () {
      bool result = isEven(123);
      expect(result, true);
    });
  });
}
```

12. Run the test and note the error message has changed, adding the group name to the name of the failing test.

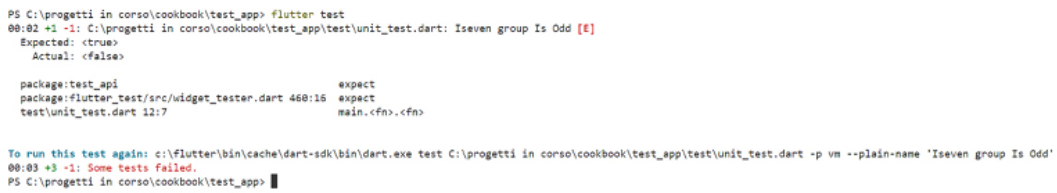


Figure 2.11: Terminal showing failing test with group name

13. Edit the `Is Odd` test as shown below, then run the test again, and note that the error has been fixed.

```
test('Is Odd', () {
      bool result = isEven(123);
      expect(result, false);
    });
```

How it works...

You can use unit tests in Flutter to test specific parts of your code, like functions, methods, and classes.

In this recipe, you wrote a simple function, called `isEven`, that returns `true` when the number passed as an argument is even, and `false` when it's odd:

```
bool isEven(int number) {
    if (number % 2 == 0) {
      return true;
    } else {
      return false;
    }
  }
```

Probably there isn't much need to test a function as simple as this one, but as your business logic evolves and gets more complex, testing your units of code becomes important.

When you create a new project, you'll find a `test` directory in your project's root folder. This is where you should place your test files, including unit tests.

In this recipe, you created a new file, called `unit_test.dart`. When you choose the name for a file containing tests, you might want to add `test` as a prefix or suffix to follow the naming convention for tests. For example, if you are testing functions that deal with parsing JSON, you might call your file `test_json.dart`.

In your test files, you need to import the `flutter_test.dart` package, which contains the methods and classes that you use to run your tests:

```
import 'package:flutter_test/flutter_test.dart';
```

You also need to import the files where the methods and classes you want to test are written. In this recipe's example, the `isEven` method is contained in the `main.dart` file, so this is what you imported:

```
import 'package:hello_flutter/main.dart';
```

Like in any Flutter app, the `main` method is the entry point of your tests. When you execute the `flutter test` command in a Terminal, Flutter will look for this method to start executing your tests.

You write a unit test using the `test` function, which takes two arguments: a string to describe the test, and a callback function that contains the test itself:

```
void main() {
  test('Is Even', () {
    bool result = isEven(12);
    expect(result, true);
  });
}
```

In this case, we are running a test in the `isEven()` function. When you pass an even number to `isEven()`, like 12, you expect `true` to be the return value. This is where you use the `expect()` method. The `expect()` method takes two arguments: the actual value and the expected value. It compares the two values, and if they don't match it throws an exception and the test fails.

In our example, the instruction:

```
expect(result, true);
```

succeeds when `result` is `true`, and fails when `result` is `false`.

To run the tests, you can type the `flutter test` command in your terminal. This will run all tests in your project. You can also run your tests from your editor, like any other Flutter app. If your tests succeed, you get a success message, otherwise, you will see which specific test failed.

When you have several tests, you can group related tests together, using the `group` method:

```
group('Iseven group', () {
    test('Is Even', () {
      bool result = isEven(12);
      expect(result, true);
    });
    test('Is Odd', () {
      bool result = isEven(123);
      expect(result, true);
```

```
    });
  });
```

Like the `test` method, the `group` method takes two arguments. The first is a `String` containing the group description, and the second is a callback function, which in turn contains one or more `test()` functions. You use the `group()` function to better organize your test code, making it easier to read, maintain, and debug.

In general, the goal of unit testing is to test single units of code, like individual functions, to ensure that they are working as expected.

See also

In addition to unit tests, there are other types of tests that you can create for your Flutter application. In particular, widget tests are used to test the functionality of widgets, and integration tests test the integration of different parts of your application. See the official documentation at https://docs.flutter.dev/testing for more information about writing tests in Flutter.

Summary

In this chapter, you've seen how to create a Flutter app and your first unit test. You've seen how directories are organized in a Flutter project: in particular, the `lib` directory contains the `main.dart` file for your app, which is the entry point for your app. Its `main()` method is executed when you run the app.

The `pubspec.yaml` file is the Flutter project's main configuration file. Flutter creates specific target operating system directories in your project, like `android` and `ios`. You've seen how Flutter needs those to compile to different targets, with the same code base.

You've learned how to run the app from the Terminal by using `flutter run` and from an IDE by running the app from your editor's interface.

You've seen the benefits of writing unit tests, which may help you write reliable code that works as expected, and learned how to import the `flutter_test` package.

You've seen the `test()`, `expect()`, and `group()` methods, which are the main building blocks for creating unit tests in Flutter.

In the next chapter, you'll see an introduction to the Dart language.

Dart: A Language You Already Know

At its heart, Dart is a conservative programming language. It was not designed to champion bold new ideas, but rather to create a predictable, stable, and developer-friendly programming environment. Made by Google, the language is relatively young, as it was shown for the first time in 2011, with the goal of unseating JavaScript as the language of the web.

Among its main features, Dart is strongly typed, supports asynchronous programming, and uses both **JIT** (**Just-In-Time**) and **AOT** (**Ahead-Of-Time**) compilation, which allows developers to write and run code quickly during development and produce optimized code for deployment.

JavaScript is a very flexible language, but its lack of a type system (addressed by TypeScript) and misleadingly simple grammar can make projects difficult to manage as they grow. Dart aimed to fix this by finding a halfway point between the dynamic nature of JavaScript and the class-based designs of Java and other object-oriented languages. The language uses a syntax that will be immediately familiar to any developer who already knows a C-style language.

This chapter assumes Dart is not your first programming language. We will be skipping some parts of the Dart language where the syntax is the same as any other C-style language. You will not find anything in this chapter about loops, `if` statements, and `switch` statements; they aren't any different here from how they are treated in other languages you already know. Instead, we will focus on the aspects of the Dart language that make it unique.

Dart 3 contains several improvements. Probably the most important is that Dart is now sound null safe. Dart developers also added new language features, like records, patterns, and switch expressions, that you will see in this chapter

Note

If you want to have a look at the control flow syntax in Dart, see the official guide at https://dart.dev/guides/language/language-tour#control-flow-statements.

In this chapter, we will cover the following recipes, all of which will function as a primer on Dart:

- Declaring variables—`var` versus `final` versus `const`
- Strings and string interpolation
- How to write functions
- How to use functions as variables with closures
- Creating classes and using the class constructor shorthand
- How to group and manipulate data with collections
- Writing less code with higher-order functions
- How to take advantage of the cascade operator
- Using extensions
- Introducing Dart Null Safety
- Using Null Safety in classes

Pro Tip

If you are already aware of how to develop in Dart, feel free to skip this chapter. We will be focusing exclusively on the language here and will then cover Flutter in detail in the next chapter.

Technical requirements

This chapter will focus purely on Dart instead of Flutter. There are two primary options for executing these samples:

- **DartPad** (https://dartpad.dev/): DartPad is a simple web app where you can execute Dart and Flutter code. It's a great playground for trying out new ideas and sharing code and only requires a web browser. This is the option I recommend for this chapter.
- **IDEs**: If you wish to try out the sample code locally, then you can use Android Studio, Visual Studio Code, IntelliJ, or Emacs.

Declaring variables—var versus final versus const

Variables are user-defined symbols that hold a reference to some value. They can range from a single number to large object graphs. It is virtually impossible to write a useful program without at least one variable. You can probably argue that almost every program ever written can be boiled down to taking in some input, storing that data in a variable, manipulating the data in some way, and then returning an output. All of this would be impossible without variables.

Recently, a new trend has appeared in programming that emphasizes immutability. This means that once values are stored in variables, they cannot change. **Immutable variables** are generally safer, produce no side effects, and lead to fewer bugs as a consequence. For example, immutability can help prevent bugs that occur when multiple parts of an app try to modify the same data at the same time.

In this recipe, we will create a small program that will declare variables in the three different ways that Dart allows—`var`, `final`, and `const`.

Getting ready

Currently, Dart officially supports Chrome, Edge, Firefox, and Safari browsers. While DartPad might also work with other browsers, I recommend you use one of the supported ones for this chapter.

To open DartPad, in your browser, navigate to https://dartpad.dev/.

Pro Tip

When working with the code samples in this book, it is *strongly discouraged* that you copy and paste them into your IDE. Instead, you should transcribe the samples manually. The act of writing code, not copying/pasting, will allow your brain to absorb the code and see how tools such as code completion make it easier for you to type code. If you copy and paste, you'll get a working program, but you will probably learn less.

How to do it...

Let's get started with our first Dart project. Make sure you have opened DartPad as mentioned in the *Getting ready* section. We will start from a blank canvas:

1. From the **Pad** in DartPad, delete everything. At this point, the **Pad** should be completely empty. Now, let's add the `main` function, which is the entry point for every Dart program:

```
void main() {
 variablePlayground();
}
```

2. This code won't compile yet because we haven't defined that `variablePlayground` function. This function will be the hub for all the different examples in this recipe:

```
void variablePlayground() {
 basicTypes();
}
```

We added the `void` keyword in front of this function, which is the same as saying that this function returns nothing.

3. Now, let's implement the first example. In this method, all these variables are mutable; they can change once they've been defined:

```
void basicTypes() {
 int four = 4;
 double pi = 3.14;
 num someNumber = 24601;
 bool yes = true;
 bool no = false;
 int? nothing;
 print(four);
 print(pi);
 print(someNumber);
 print(yes);
 print(no);
 print(nothing == null);
}
```

4. Click on the **Run** button at the top of the screen. The result should look like the screenshot below:

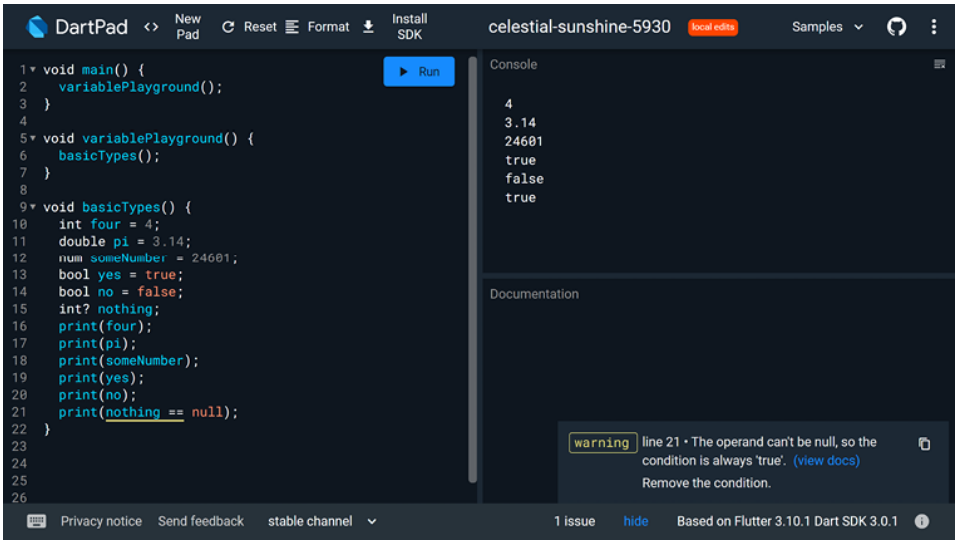


Figure 3.1: DartPad console after calling the *basicTypes* function

The syntax for declaring a mutable variable should look very similar to other programming languages. First, you declare the type and then

the name of the variable. You then supply a value for the variable after the assignment operator. Note that the `num` type can contain either `int` or `double` values.

If you don't supply any value, that variable will be set to `null`, but this won't work unless you put a question mark near the type or postpone the initialization (this is because of *null safety*: more about that in the *Introducing Dart Null Safety* recipe later in this chapter).

5. Dart has a special type called `dynamic`, which is a sort of "get out of jail free" card from the type system. You can annotate your variables with this keyword to imply that the variable can be anything. It is useful in some cases, but for the most part, it should be avoided. Add a new `untypedVariables` method after the end of the `basicTypes` method:

```
void untypedVariables() {
 dynamic something = 14.2;
 print(something.runtimeType); //outputs 'double'
}
```

6. Add a call to the `untypedVariables` method in the `variablePlayground` method:

```
void variablePlayground() {
 basicTypes();
 untypedVariables();
}
```

7. Run the app with the **Run** button; you should see **double** as the type of the `something` variable.
8. Dart can also infer types with the `var` keyword. `var` is not the same as `dynamic`. Once a value has been assigned to the variable during initialization, Dart will remember the type and it cannot be changed later. The values, however, are still mutable. Add a new method, like shown below, in the `variablePlayground` method (optionally, comment out the previous calls):

```
void typeInterpolation() {
 var anInteger = 15;
 var aDouble = 27.6;
 var aBoolean = false;
 print(anInteger.runtimeType);
 print(anInteger);
 print(aDouble.runtimeType);
 print(aDouble);
 print(aBoolean.runtimeType);
 print(aBoolean);
}
```

9. Run the app. You should see the result shown in the screenshot:

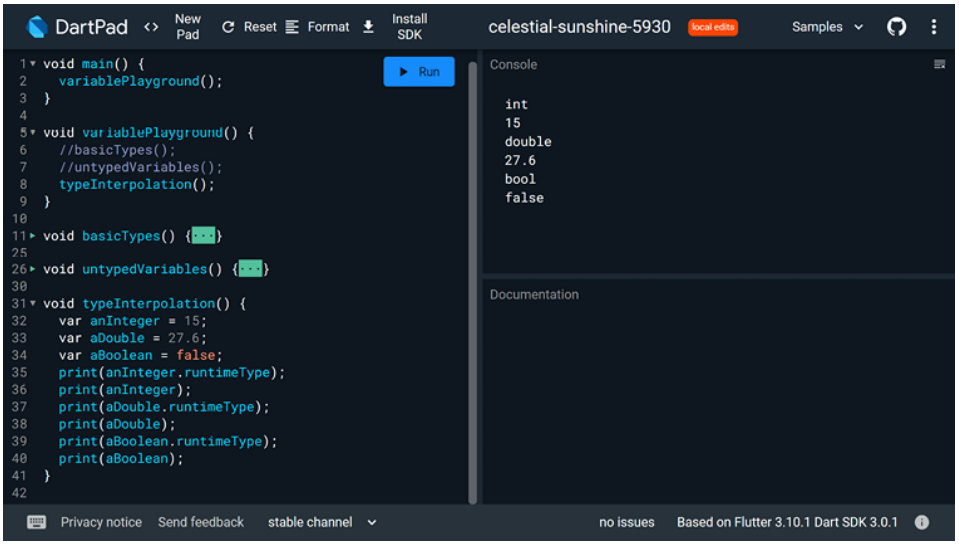


Figure 3.2: DartPad console after calling the *typeInterpolation* function

Finally, we have our immutable variables. Dart has two keywords that can be used to indicate immutability—`final` and `const`.

> **Note**
>
> The main difference between `final` and `const` is that `const` *must* be determined at compile time; for example, you cannot have `const` containing `DateTime.now()` since the current date and time can only be determined at runtime, not at compile time. See the *How it works...* section of this recipe for more details.

10. Add the following function in the `variablePlayground` method, and run your code:

```
void immutableVariables() {
 final int immutableInteger = 5;
 final double immutableDouble = 0.015;

 // Type annotation is optional
 final interpolatedInteger = 10;
 final interpolatedDouble = 72.8;
 print(interpolatedInteger);
 print(interpolatedDouble);
 const aFullySealedVariable = true;
 print(aFullySealedVariable);
}
```

How it works...

An assignment statement in Dart follows the same grammar as other languages in the C language family:

```
// (optional modifier) (optional type) variableName = value;
final String name = 'Donald'; //final modifier, String type
```

First, you can optionally declare a variable as either `var`, `final`, or `const`, like so:

```
var animal = 'Duck';
final numValue = 42;
const isBoring = true;
```

These modifiers indicate whether or not the variable is mutable. `var` is mutable as its value can be reassigned at any point. `final` variables can only be assigned once, but by using objects, you can change the value of their fields. `const` variables are compile-time constants and are fully immutable; nothing about these variables can be changed once they've been assigned.

After the `final` modifier, you can optionally declare the variable type, from simple built-in types such as `int`, `double`, and `bool` to your own, more complex custom types. This notation is standard for languages such as Java, C, C++, Objective-C, and C#.

Explicitly annotating the type of a variable is the traditional way of declaring variables in languages such as Java and C, but Dart can also interpolate the type based on its assignment. In the `typeInterpolation` example, we decorated the types with the `var` keyword; Dart was able to figure out the type based on the value that was assigned to the variable (this notation is standard in JavaScript). For example, `15` is an integer, while `27.6` is a double.

In most cases, there is no need to explicitly reference the type; the compiler is smart enough to figure this out. This allows us, as developers, to write succinct, script-like code and still take advantage of the inherent gains that we get from a type-safe language.

The difference between `final` and `const` is subtle but important. A `final` variable must have a value assigned to it in the same statement where it was declared (unless you use the `late` keyword, as mentioned in the *Introducing Dart Null Safety* recipe later in this chapter) and that variable cannot be reassigned to a different value:

```
final meaningOfLife = 42;
meaningOfLife = 64; // This will throw an error
```

While the top-level value of a final variable cannot change, its internal contents can. In a list of numbers that have been assigned to a final variable, you can change the internal values of that list, but you cannot assign a completely new list.

`const` takes this one step further. `const` values must be determined at compile time. New values are blocked from being assigned to `const` variables, and the internal contents of that variable are also completely sealed. This is indicated by having the object have a `const` constructor, which only allows immutable values to be used. Since their value is already

determined at compile time, `const` values also tend to get better performance than variables.

There's more...

In recent years, there has been a trend in development that favors immutable values over mutable ones. Immutable data cannot change. Once it has been assigned, that's it. There are two primary benefits to preferring immutable data, as follows:

1. It's faster. When you declare a `const` value, the runtime has less work to do. It only has to allocate memory for that variable once and doesn't need to worry about reallocating if the variable is reassigned. This may seem like an infinitesimal gain, but as your programs grow, your performance gain grows as well.
2. Immutable data does not have side effects. One of the most common sources of bugs in programming is when the value is changed in one place, and it causes an unexpected cascade of changes. If the data cannot change, then there will be no cascade. And in practice, most variables tend to only be assigned once anyway, so why not take advantage of immutability?

See also

Have a look at the following resource.

The Dart website provides a great tour of all its language features and provides a deeper explanation of every built-in variable type: https://dart.dev/guides/language/language-tour.

Strings and string interpolation

A **String** is simply a variable that holds human-readable text. The reason why they're called strings instead of text has more to do with history than

practicality. From a computer's perspective, a `String` is actually a list of integers. Each integer represents a character.

For example, the number `U+0041` (Unicode notation, `65` in decimal notation) is the letter `A`. These numbers are put together in a *string* to create text.

In this recipe, we will continue with the console application in order to define and work with strings.

Getting ready

To follow along with this recipe, you should write this code in DartPad (https://dartpad.dev/). Remove any content inside the `main()` method.

How to do it...

Just like in the previous recipe, you are going to create a "hub" function. Inside it, every sub-function will show a different way of using strings:

1. Type in the following code and use it as the hub for all the other string examples:

```
void stringPlayground() {
 basicStringDeclaration();
 multiLineStrings();
 combiningStrings();
}
```

2. This first section shows different ways to declare string literals. Write the following function into your code, just under the `stringPlayground` function:

```
void basicStringDeclaration() {
  // With Single Quotes
  print('Single quotes');
  final aBoldStatement = 'Dart isn\'t loosely typed.';
  print(aBoldStatement);
  //raw string
  print('Raw String');
```

```
  final rawString = r'Show an escape \ character';
  print(rawString);
  // With Double Quotes
  print("Hello, World");
  final aMoreMildOpinion = "Dart's popularity has
skyrocketed with
Flutter!";
  print(aMoreMildOpinion);
  // Combining single and double quotes
  final mixAndMatch =
      'Every programmer should write "Hello, World"
when learning
a new language.';
  print(mixAndMatch);
}
```

3. Dart also supports multi-line strings. The following example gets a little Shakespearean:

```
void multiLineStrings() {
  final withEscaping = 'One Fish\nTwo Fish\nRed
Fish\nBlue Fish';
  print(withEscaping);
// pay attention to any spaces left at the beginning
// of the lines when using '''
  final hamlet = '''
  To be, or not to be, that is the question:
  Whether 'tis nobler in the mind to suffer
  The slings and arrows of outrageous fortune,
  Or to take arms against a sea of troubles
  And by opposing end them.
  ''';
  print(hamlet);
}
```

4. Finally, one of the most common tasks programmers perform with strings is composing them to make more complex strings. Dart supports the traditional method of concatenation with the + operator, but it's recommended to use string interpolation when appropriate. Type in the following blocks of code to get a feel for both techniques:

```
void combiningStrings() {
 traditionalConcatenation();
 modernInterpolation();
```

```
}
void traditionalConcatenation() {
 final hello = 'Hello';
 final world = "world";
 final combined = hello + ' ' + world;
 print(combined);
}
void modernInterpolation() {
 final year = 2011;
 final interpolated = 'Dart was announced in $year.';
 print(interpolated);
 final age = 42;
 final howOld = 'I am $age ${age == 1 ? 'year' :
'years'} old.';
 print(howOld);
}
```

5. Now, in the `main` method, call `stringPlayground`():

```
main() {
 stringPlayground();
}
```

How it works...

There are two ways of declaring string literals in Dart—using a single quote or double quotes. It doesn't matter which one you use, as long as both begin and end a string with the same character. Depending on which character you chose, you would have escaped that character if you wanted to insert it in your string.

For example, to write a string stating *Dart isn't loosely typed* with single quotes, you would have to write the following:

```
// With Single Quotes
final aBoldStatement = 'Dart isn\'t loosely typed.';
// With Double Quotes
final opinion = "Dart's popularity has skyrocketed with
Flutter!";
```

Notice how we had to write a backslash in the first example but not in the second. That backslash is called an **escape character**. Here, we are telling the compiler that even though it sees an apostrophe, this is not the end of the string, and the apostrophe should be included as part of the string.

The two ways in which you can write a string are helpful when you're writing strings that contain single quotes/apostrophes or quotation marks. If you declare your string with the symbol that is *not* in your string, then you will not have to add any unnecessary characters to your code, which ultimately improves legibility.

It has become a convention to prefer single quote strings over doubles in Dart, which is what we will follow in this book, except if that choice forces us to add escape characters.

Sometimes you might also want to print the escape character in a string. In this case, you can use a *raw string*:

```
final rawString = r'I want to show an escape \ character';
```

You use raw strings to treat backslashes (`\`) as literal characters. To create a raw string, you add a prefix with the letter "`r`" before the string. Raw strings can also be useful whenever you want to treat your characters literally, like in regular expressions or file paths.

One other interesting feature of strings in Dart is multi-line strings. If you've ever had a larger block of text that you didn't want to put into a single line, you would have to insert the newline character, `\n`, as you saw in this recipe's code:

```
final withEscaping = 'One Fish\nTwo Fish\nRed Fish\nBlue
Fish';
```

The newline character (`\n`) has served us well for many years, but more recently, another option has emerged. If you write three quotation marks (single or double), Dart will allow you to write free-form text without

having to inject any non-rendering control characters, as shown in the following code block:

```
final hamlet = '''
  To be, or not to be, that is the question:
  Whether 'tis nobler in the mind to suffer
  The slings and arrows of outrageous fortune,
  Or to take arms against a sea of troubles
  And by opposing end them.
  ''';
```

In this example, every time you press *Enter* on the keyboard, it is the equivalent of typing the control character, `\n`, in your string. Also, any spaces at the beginning of the lines will also be included in the string.

There's more...

On top of simply declaring strings, the more common use of this data type is to concatenate multiple values to build complex statements. Dart supports the traditional way of concatenating strings; that is, by simply using the addition (`+`) symbol between multiple strings, like so:

```
final sum = 1 + 1; // 2
final concatenate = 'one plus one is ' + sum;
```

While Dart supports this method of constructing strings, it's now recommended to use interpolation syntax. The second statement can be updated to look like this:

```
final sum = 1 + 1;
final interpolate = 'one plus one is $sum'
```

The dollar sign notation only works for single values, such as the integer in the preceding snippet. If you need anything more complex, you can add curly brackets after the dollar sign and write any Dart expression. This can range from something simple, such as accessing a member of a class, to a complex ternary operator.

Let's break down the following example:

```
  final age = 42;
  final howOld = 'I am $age ${age == 1 ? 'year' : 'years'}
old.';
  print(howOld);
```

The first line declares an integer called `age` and sets its value to `42`. The second line contains both types of string interpolation. First, the value is just inserted with `$age`, but after that, there is a ternary operator inside the string to determine whether the word `year` or `years` should be used:

```
age == 1 ? 'year' : 'years'
```

This statement means that if the value of `age` is `1`, then use the singular word `year`; otherwise, use the plural word `years`. In other words, it's exactly like writing:

```
If (age == 1) {
    howOld += 'year';
} else {
    howOld += 'years';
}
```

When you run this code, you'll see the following output:

```
I am 42 years old.
```

Over time, this will become natural. Just remember that legible code is usually better than shorter code, even if it takes up more space.

It's probably worth mentioning another way to perform concatenation tasks, which is using the `StringBuffer` object. Consider the following code:

```
List fruits = ['Strawberry', 'Coconut', 'Orange', 'Mango',
'Apple'];
StringBuffer buffer = StringBuffer();
for (String fruit in fruits) {
  buffer.write(fruit);
```

```
  buffer.write(' ');
}
print (buffer.toString()); // prints: Strawberry Coconut
Orange Mango Apple
```

You can use a `StringBuffer` to incrementally build a string. This is better than using string concatenation as it performs better. You add content to a `StringBuffer` by calling its `write` method. Then, once it's been created, you can transform it into a `String` with the `toString` method.

See also

Check out the following resources for more details on strings in Dart:

- The Dart language guide's entry on strings: https://dart.dev/guides/language/language-tour#strings
- Effective Dart suggestions on the proper usage of strings: https://dart.dev/guides/language/effective-dart/usage#strings
- Official documentation on the `String` class: https://api.flutter.dev/flutter/dart-core/String-class.html

How to write functions

Functions are the basic building blocks of any programming language and Dart is no different. The basic structure of a function is as follows:

```
optionalReturnType functionName(optionalType parameter1,
optionalType parameter2...) {
  // code
}
```

You have already written a few functions in previous recipes. In fact, you really can't write a proper Dart application without them.

Dart also has some variations of this classical syntax and provides full support for optional parameters, optionally named parameters, default parameter values, annotations, closures, generators, and asynchronicity decorators. This may seem like a lot to cover in one recipe, but with Dart, most of this complexity will disappear.

Let's explore how to write functions and closures in this recipe.

Getting ready

To follow along with this recipe, you can write the code in DartPad. Begin with a clean Pad, with an empty `main` function.

How to do it...

We'll continue with the same pattern from the previous recipe:

1. Start by creating the hub function for the different features we are going to cover:

```
void functionPlayground() {
  classicalFunctions();
  optionalParameters();
}
```

2. Now, add some functions that take parameters and return values:

```
void printMyName(String name) {
  print('Hello $name');
}
int add(int a, int b) {
  return a + b;
}
int factorial(int number) {
  if (number <= 0) {
    return 1;
  }
  return number * factorial(number - 1);
}
void classicalFunctions() {
```

```
  printMyName('Anna');
  printMyName('Michael');
  final sum = add(5, 3);
  print(sum);
  print('10 Factorial is ${factorial(10)}');
}
```

You can also use *optional positional parameters*. If you wrap your function's parameter list in square brackets, then those parameters can be omitted without the compiler throwing errors.

Note

The question mark after a parameter, such as in `String? name`, tells the Dart compiler that the parameter itself can be `null`.

3. Write this code immediately after the previous example:

```
void unnamed([String? name, int? age]) {
  final actualName = name ?? 'Unknown';
  final actualAge = age ?? 0;
  print('$actualName is $actualAge years old.');
}
```

Note

The double question mark (`??`) is the **null-aware coalescing operator**. It is a quick way to check if a value is null and provide a default value if it is.

In the instruction:

```
final actualName = name ?? 'Unknown';
```

If `name` contains null, `actualName` takes the string `'Unknown'`, otherwise it will take the value of `name`.

Dart also supports **named optional parameters**, with curly brackets.

> **Note**
>
> When calling a function with named parameters, you need to specify the **parameter name**. You can call the parameters *in any order*; for example, `named(greeting: 'hello!');`.

4. Add this function right after the `unnamed` function:

```
void named({String? greeting, String? name}) {
  final actualGreeting = greeting ?? 'Hello';
  final actualName = name ?? 'Mystery Person';
  print('$actualGreeting, $actualName!');
}
```

5. Optional parameters and optional named parameters also support default values. If the parameter is omitted when the function is called, the default value will be used instead of `null`. You can also place a set of required parameters first, followed by a list of optionals. Add the following code to see how this can be accomplished:

```
String duplicate(String name, {int times = 1}) {
  final merged = StringBuffer(name);
  for (var i = 1; i < times; i++) {
    merged.write(' $name');
  }
  return merged.toString();}
```

6. Now, implement the `playground` function to show all these pieces in action:

```
void optionalParameters() {
  unnamed('Huxley', 3);
  unnamed();
  // Notice how named parameters can be in any order
  named(greeting: 'Greetings and Salutations');
  named(name: 'Sonia');
  named(name: 'Alex', greeting: 'Bonjour');
  final multiply = duplicate('Mikey', times: 3);
  print(multiply);
}
```

7. Finally, update the `main` method so that these functions can be executed:

```
main() {
  functionPlayground();
}
```

How it works...

With Dart, you can write functions with *positional*, *named*, and *unnamed optional* parameters.

Named parameters can also remove ambiguity from what each parameter is supposed to do. Take a look at the following line from the preceding code example:

```
unnamed('Huxley', 3);
```

Now, compare it with this line:

```
duplicate('Mikey', times: 3);
```

In the first example, it isn't immediately clear what the purpose of each parameter is. In the second example, the `times` parameter immediately tells you that the text `Mikey` will be duplicated three times. This can go a long way with functions that have rather long parameter lists, where it can be difficult to remember the expected order of the parameters. Take a look at how this syntax is put to work in the Flutter framework:

```
Container(
  margin: const EdgeInsets.all(10.0),
  color: Colors.red,
  height: 48.0,
  child: const Text('Named parameters are great!'),
)
```

This isn’t even all the properties that are available for containers—it can get much longer. Without named parameters, this sort of syntax could be almost impossible to read.

Note

Type annotation for Dart functions is optional.

You can completely omit it if you are so inclined. However, for any parameter or even function name that does not have type annotation, Dart will assume that it is of the **dynamic** type. Since we would like to exploit Dart’s type system for all it’s worth, dynamic types should be avoided. That is why we always strive to add the `void` keyword in front of any function that doesn’t return a value.

How to use functions as variables with closures

Closures, also known as first-class functions, are an interesting language feature that emerged from *lambda calculus*, a mathematical logic system first developed in the 1930s. The basic idea is that *a function is also a value* that can be passed around to other functions as a parameter. These types of functions are called **closures**, but there is really no difference between a function and a closure.

Closures can be saved to variables and used as parameters for other functions.

Getting ready

To follow along with this recipe, you can write the code in DartPad.

How to do it...

To implement a closure in Dart, follow these steps:

1. To add a closure to a function, you have to essentially define another function signature inside a function:

```
void callbackExample(void Function(String value)
callback) {
  callback('Hello Callback');
}
```

2. Under the `callbackExample` method, create another plain function that prints its parameter:

```
void printSomething(String value) {
  print(value);
}
```

3. Defining closures inline can get quite verbose. To simplify this, Dart uses the `typedef` keyword to create a custom type alias that will represent the closure. Under the `main()` method, create a `typedef` called `NumberGetter`, which will be a function that returns an integer:

```
typedef NumberGetter = int Function();
```

4. The following function will take in a `NumberGetter` as its parameter and invoke it in its function:

```
int powerOfTwo(NumberGetter getter) {
 return getter() * getter();
}
```

5. Let's put this all together with a function that will use all these closure examples:

```
void consumeClosure() {
 int getFour() => 4;
 final squared = powerOfTwo(getFour);
 print(squared);
 callbackExample(printSomething);
}
```

6. Finally, add an invocation to `consumeClosure` in your `main` method:

```
consumeClosure();
```

How it works...

A modern programming language wouldn't be complete without closures, and Dart is no exception. Basically, a closure is a function that is saved to a variable that can be called later. Closures are often used for callbacks, such as when the user taps a button or when the app receives data from a network call.

You've seen two ways to define closures in this recipe:

1. Function prototypes
2. `typedefs`

Note the following syntax:

```
void callbackExample(void Function(String value) callback) {
   callback('Hello Callback');
}
```

The `callbackExample` function takes another function as its parameter; this in turn takes a `String` as its parameter, and returns `void`.

You call it with the instruction:

```
callbackExample(printSomething);
```

Note that `printSomething` is passed *without* the parentheses, and without a parameter. This is because it's in the `callbackExample` itself that the function gets called, with its parameter.

The easiest and most maintainable way to work with closures is with the `typedef` keyword. This is especially true if you are planning on reusing the

same closure type multiple times; then, using `typedefs` will make your code more succinct:

```
typedef NumberGetter = int Function();
```

This defines a closure type called `NumberGetter`, which is a function that is expected to return an integer:

```
int powerOfTwo(NumberGetter getter) {
  return getter() * getter();
}
```

The closure type is then used in this function, which will call the closure twice and then multiply the result:

```
int getFour() => 4;
final squared = powerOfTwo(getFour);
```

In this line, we call the function and provide our closure, which returns the number `4`. This code also uses the fat arrow syntax, which allows you to write any function that takes up a single line without braces. For single-line functions, you can use the arrow syntax, `=>`, instead of brackets.

The `getFour` line without the arrow is equivalent to writing the following:

```
int getFour () {
  return 4;
};
// this is the same as: int getFour() => 4;
```

Arrow functions are very helpful for removing unneeded syntax, but they should only be used for simple single-line statements. For longer and more complex functions, you should use the block function syntax.

Closures are probably one of the most cognitively difficult programming concepts. It may seem awkward to use them at first, but the only way for it to become natural is to practice using them several times.

Using Switch Expressions, Records and Patterns

A switch statement is a very common control flow statement that allows you to perform different actions based on the value of a value. Dart 3 also introduced **switch expressions**, that allow writing switch statements that return values in an efficient and concise way. In this recipe you will see how to write a switch statement and then transform it into a switch expression.

Getting ready

To follow along with this recipe, you can write the code in DartPad.

How to do it...

To implement a **switch expression** in Dart, follow these steps:

1. In DartPad, delete all the code inside the main method.
2. Add a new method under the `main()` method, calling it `getDay()`, that takes an integer and returns a String:

```
String getDay(int day) {
}
```

3. In the `getDay()` method, add a switch statement, that checks the number that was passed, and based on the value, returns the day of the week as a string:

```
switch (day) {
    case 1:
      return 'Monday';
    case 2:
      return 'Tuesday';
    case 3:
```

```
      return 'Wednesday' ;
    case 4:
      return 'Thursday' ;
    case 5:
      return 'Friday' ;
    case 6:
      return 'Saturday' ;
    case 7:
      return 'Sunday' ;
    default:
      return 'Invalid day' ;
  }
```

Note

Note that usually each case in a switch should end with a break instruction:

case 1:

```
   //do something
   break;
 case 2:
```

In the example in this recipe you omit break only because the return statement exits the function.

4. In the main method, call the `getDay()` method and print the result:

```
int dayOfWeek = 7;
String myDay = getDay(dayOfWeek);
print(myDay);
```

5. Run the code and note that Sunday appears in the Console.
6. Comment out the call to `getDay()` and add the following code in the main method:

```
int dayOfWeek = 7;
 //String myDay = getDay(dayOfWeek);
```

```
var myDay = switch (dayOfWeek) {
    1 => 'Monday',
    2 => 'Tuesday',
    3 => 'Wednesday',
    4 => 'Thursday',
    5 => 'Friday',
    6 =>  'Saturday',
    7 => 'Sunday'  ,
    _ => 'Invalid day' //Default value
 };
print(myDay);
```

7. Run the code again, and note that again, Sunday is printed in the console.

How it works…

You use switch statements to perform different actions based on different conditions. In the example in this recipe, you check the values of the variable day:

```
switch (day) {
    case 1:
      return 'Monday';
    case 2:
      return 'Tuesday';
    case 3:
      return 'Wednesday' ;
    case 4:
      return 'Thursday' ;
    case 5:
      return 'Friday' ;
    case 6:
      return 'Saturday' ;
    case 7:
      return 'Sunday' ;
    default:
      return 'Invalid day' ;
  }
```

Based on the value of day, you return a different String, and if the number does not match an integer between 1 and 7, it returns a default value.

You can achieve the same result using a Switch Expression:

```
var myDay = switch (dayOfWeek) {
      1 => 'Monday',
      2 => 'Tuesday',
      3 => 'Wednesday',
      4 => 'Thursday',
      5 => 'Friday',
      6 =>  'Saturday',
      7 => 'Sunday'  ,
      _ => 'Invalid day' //Default value
  };
```

In this case, you can omit the case keyword, and in the body of the case you specify a single expression instead of a series of statements. Cases are separated by a comma, and the default case should use an underscore (_).

As you can see this is a much more compact way to write a switch when a single expression is what you need.

There's more...

One of the main changes that was introduced in Dart 3 is *records*. Records are types that hold custom data. Consider this code:

```
var person = (name: 'Clark', age: 42);
print (person.name);
```

In this case `person` is a *record expression*: an immutable set of fields (named, like in this example, or positional), separated by a comma. As you can see, they are a very efficient way to bundle together different values in a single type.

You can also use *record type annotations* for variable declarations:

```
({String name, int age}) person =  (name: 'Clark', age: 42);
print (person.name);
```

In this case `person` has been explicitly declared with its type.

Now consider this other example:

```
void main() {
  var (String name, int age) = getPerson({'name': 'Clark',
'age': 42});
  print('$name is $age years old.');
}
(String, int) getPerson(Map<String, dynamic> json) {
  return (json['name'] as String, json['age'] as int);
}
```

In this case, `getPerson` is a method that returns a record of type `String`, `int`, but instead of returning a Map or a full object, it returns a record, with its values in parenthesis.

The instruction:

```
var (String name, int age) = getPerson({'name': 'Clark',
'age': 42});
```

is called a *pattern*: it transforms a record into two new variables: `name` and `age`.

Creating classes and using the class constructor shorthand

Classes in Dart are not dramatically different from what you would find in other **object-oriented programming** (**OOP**) languages.

Note

OOP is a programming paradigm that is based on the idea of “objects.” An object contains data and methods (functions in an object) that act on that data.

Objects are created from a “class,” a blueprint for creating objects. The class defines the properties (data) and behaviors (methods) that all objects created from it will have.

This makes it easy to reuse the same code and make your code modular.

In this recipe, we’re going to define a class hierarchy around formal and informal names.

Getting ready

As with the other recipes in this chapter, add your code in DartPad. You may start with an empty `main` method.

How to do it...

Let’s start building our own custom types in Dart:

1. First, define a class called `Name`, which is an object that stores a person’s first and last names:

```
class Name {
 final String first;
 final String last;
 Name(this.first, this.last);
 @override
 String toString() {
   return '$first $last';
 }
}
```

2. Now, let’s define a subclass called `OfficialName`. This will be just like the `Name` class, but it will also have a title:

```
class OfficialName extends Name {
 // Private properties begin with an underscore
 final String _title;
```

```
  // You can add colons after constructor
  // to parse data or delegate to super
  OfficialName(this._title, String first, String last)
  : super(first, last);
  @override
  String toString() {
    return '$_title. ${super.toString()}';
  }
}
```

3. Now, we can see all these concepts in action by using the `playground` method:

```
void classPlayground() {
  final name = OfficialName('Mr', 'Clark', 'Kent');
  final message = name.toString();
  print(message);
}
```

4. Finally, add a call to `classPlayground` in the `main` method:

```
main() {
  classPlayground();
}
```

The final result should look like *Figure 3.3*:

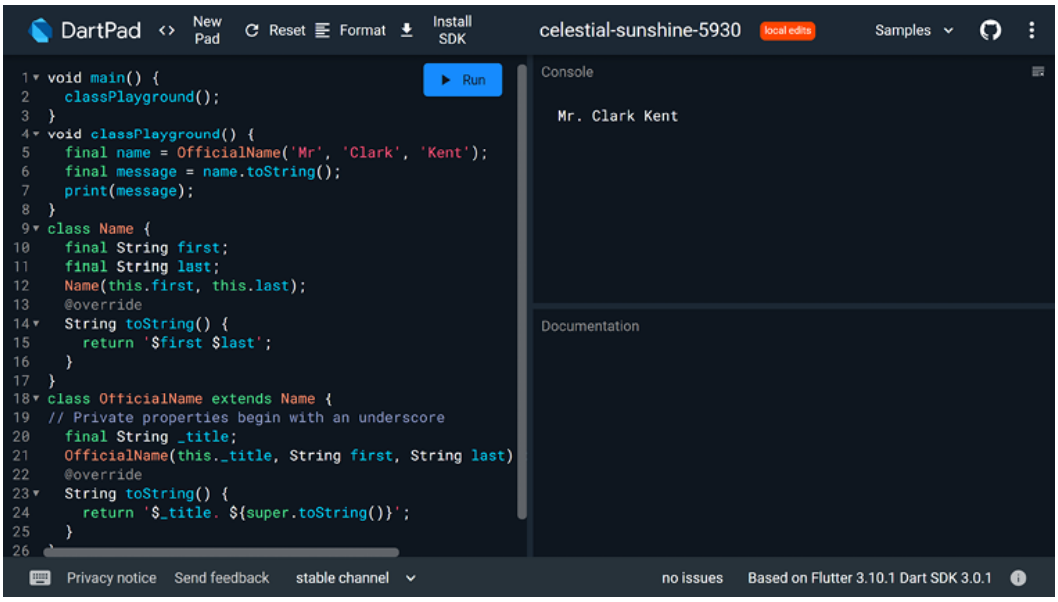


Figure 3.3: DartPad console after calling the *OfficialName* function

How it works...

Just like functions, Dart implements the expected behavior for classical OOP.

In this recipe, you used **inheritance**, which is a building block of OOP. Consider the following class declaration:

```
class OfficialName extends Name {
...
```

This means that `OfficialName` inherits all the properties and methods that are available in the `Name` class and may add more or override existing ones.

One of the more interesting syntactical features in Dart is the constructor shorthand. This allows you to automatically assign members in constructors by simply adding the `this` keyword, which is demonstrated in the `Name` class, as shown in the following code block:

```
const Name(this.first, this.last) : super(first, last);
```

Note the `super` keyword: this is used to refer to the parent class from a subclass. In this example, `OfficialName`, the subclass, calls its parent `Name` constructor.

In this way, subclasses inherit properties and methods from their parent class and can customize or extend them when it makes sense.

The Dart plugin for Android Studio and Visual Studio Code also has a handy shortcut for generating constructors, so you can make this process go even faster.

The building blocks of OOP

Where Dart does deviate from other OOP languages is in its lack of explicit keywords for interfaces and abstract classes.

There are three keywords for building relationships among classes:

`extends`	**Class Inheritance** Use this keyword with any class where you want to extend the superclass's functionality. A class can only extend one class. Dart does not support multiple inheritance.

`implements`	**`Interface Conformance`**
	`You can use implements when you want to create your own implementation of another class, as` **`all classes are implicit interfaces`**`. When the FullName class implements the Name class, all the functions defined in the Name class must be implemented. This means that when you implement a class,` **`you do not inherit any code`**`, just the definition.`
	`Classes can implement any number of interfaces (but try keeping the number reasonable and not making that list too long).`

`with`	**`Apply Mixin`** `In Dart, a class can only extend another class. Mixins allow you to reuse a class's code in multiple class hierarchies. This means that mixins allow you to get blocks of code without needing to create subclasses.`

Table 3.1: Dart keywords to build relationships among classes

Note

Dart 2.1 added the `mixin` keyword to the language. Previously, mixins were also just abstract classes, and they can still be used in that manner if desired. See *Chapter 12*, *Adding Animations to Your App*, for examples of using mixins.

See also

This recipe touched on a bunch of topics that warrant more detail. For a better look at classes, objects, and Dart patterns, see:

- *Mastering Dart*: https://www.packtpub.com/product/mastering-dart/

How to group and manipulate data with collections

All programming languages possess ways to organize data. We've already covered the most common way—objects. These class-based structures allow you to define how you want to model your data and manipulate it with methods.

If you want to model groups of similar data, **collections** are your solution. A collection contains a group of elements. There are many types of collections in Dart, but we are going to focus on the three most popular ones: `List`, `Map`, and `Set`:

1. **Lists** are linear collections where the order of the elements is maintained.
2. **Maps** are a non-linear collection of values that can be accessed by a unique key.
3. **Sets** are a non-linear collection of unique values where the order is not maintained.

These three main types of collections can be found in almost every programming language, but sometimes by a different name. If Dart is not your first programming language, then this matrix should help you correlate collections to equivalent concepts in other languages:

Dart	Java	Swift	JavaScript

List	ArrayList	Array	Array
Map	HashMap	Dictionary	Object
Set	HashSet	Set	Set

Table 3.2: Data collections

Getting ready

You can type the code for this recipe in DartPad. You may start with an empty `main` method.

How to do it...

Follow these steps to understand and use Dart collections:

1. Create the `Playground` function that will call the examples for each collection type we're going to cover:

```
void collectionPlayground() {
  listPlayground();
  mapPlayground();
  setPlayground();
  collectionControlFlow();
}
```

2. First up is **lists**, more commonly known as arrays in other languages. This function shows how to declare, add, and remove data from a list:

```
void listPlayground() {
  // Creating with list literal syntax
  final List<int> numbers = [1, 2, 3, 5, 7];
  numbers.add(11);
  numbers.addAll([8, 17, 35]);
  // Assigning via subscript
  numbers[1] = 15;
  print('The second number is ${numbers[1]}');
```

```
  for (int number in numbers) {
    print(number);
  }}
```

3. Maps store two points of data per element—a **key** and a **value**. Keys are used to write and retrieve the values stored in the `Map`. Add this function to see `Map` in action:

```
void mapPlayground() {
 // Map Literal syntax
 final Map<String, int> ages = {
   'Clark': 26,
   'Peter': 35,
   'Bruce': 44,
 };
 // Subscript syntax uses the key type.
 // A String in this case
 ages['Steve'] = 48;
 final ageOfPeter = ages['Peter'];
 print('Peter is $ageOfPeter years old.');
 ages.remove('Peter');
 ages.forEach((String name, int age) {
   print('$name is $age years old');
 });}
```

4. Sets are the least common collection type, but still very useful. They are used to store values where the order is not important, but all the values in the collection must be unique. The following function shows how to use sets:

```
void setPlayground() {
 // Set literal, similar to Map, but no keys
 final Set<String> ministers = {'Justin', 'Stephen',
'Paul', 'Jean',  'Kim', 'Brian'};
 ministers.addAll({'John', 'Pierre', 'Joe', 'Pierre'});
//Pierre is a  duplicate, which is not allowed in a set.
 final isJustinAMinister = ministers.contains('Justin');
 print(isJustinAMinister);
 // 'Pierre' will only be printed once
 // Duplicates are automatically rejected
 for (String primeMinister in ministers) {
   print('$primeMinister is a Prime Minister.');
 }
}
```

5. Another Dart feature is the ability to include control flow statements directly in your collection. This feature is also one of the few examples where Flutter directly influences the direction of the language. You can include `if` statements, `for` loops, and spread operators directly inside your collection declarations.

   We will be using this style of syntax extensively when we get to Flutter in the next chapter. Add this function to get a feel for how control flows work on more simplistic data:

```
void collectionControlFlow() {
  final addMore = true;
  final randomNumbers = [
    34,
    232,
    54,
    32,
    if (addMore) ...[
      123,
      258,
      512,
    ],
  ];
  final doubled = [
    for (int number in randomNumbers) number * 2,
  ];
  print(doubled);}
```

6. Add the call to `collectionPlayground` in the `main` method:

```
void main() {
  collectionPlayground();
}
```

How it works...

Each of these examples shows elements in collections that can be added, removed, and enumerated. When choosing which collection type to use, there are three questions you need to answer:

1. Does the order matter? Choose a list.
2. Should all the elements be unique? Choose a set.
3. Do you need to access elements from a dataset quickly with a key? Choose a map.

Of these three types, `Set` is probably the most underused collection, but you should not dismiss it so easily. Since sets require elements to be unique and they don't have to maintain an explicit order, they can also be significantly faster than lists. For relatively small collections (~100 elements), you will not notice any difference between the two, but once the collections grow (~10,000 elements), the power of a set will start to shine.

You can explore this further by looking into **Big-O notation**, a method of measuring the complexity and efficiency of computer algorithms. See https://en.wikipedia.org/wiki/Big_O_notation for further information.

Subscript syntax

Subscripts are a way to quickly access elements in a collection, and they tend to work identically in most languages:

```
numbers[1] = 15;
```

The preceding line assigns the second value in the numbers list to `15`. Lists in Dart use a zero offset to access the element. If the list is 10 elements long, then element 0 is the first element and element 9 is the last. If you were to try and access element 10, then your app would throw an **out-of-bounds** exception because element 10 does not exist.

Sometimes, it is safer to use the `first` and `last` accessors on the list instead of accessing the element directly:

```
final firstElement = numbers.first;
final lastElement = numbers.last;
```

Note that if your set is empty, `first` and `last` will throw an exception as well:

```
final List mySet = [];
print (mySet.first); //this will throw a Bad state: No
element error
```

For maps, you can access the values not only with integers but also with custom types, like strings:

```
ages['Tom'] = 48;
final myAge = ages['Brian']; //This will be null
```

However, unlike arrays, if you try to access a value with a key that is not on the map, then it will just return null. It will not throw an exception.

There's more...

One exciting language feature that was added to Dart in version 2.3 is the ability to put control flows inside collections. This will be of particular importance when we start digging into Flutter build methods.

These operators work mostly like their normal control flow counterparts, except you do not add brackets and you only get a single line to yield a new value in the collection:

```
  final doubled = [
    for (int number in randomNumbers) number * 2,
  ];
```

In this example, we are iterating through the `randomNumbers` list and yielding double the value. Notice that there is no return statement; the value is immediately added to the list.

However, the single line requirement can be very restrictive. To remedy this, Dart has also borrowed the spread operator from JavaScript:

```
final randomNumbers = [
    34,
    232,
    54,
    32,
    if (addMore) ...[
      123,
      258,
      512,
    ],
```

In this example you use the `spread` operator, written with three dots (…), to unpack the elements from a list of numbers [`123, 258, 512`] and add them to the `randomNumbers` list. The resulting list will contain all the initial values and the additional values specified in the `spread` operator.

If `addMore` is `false`, nothing will be added to the `randomNumbers` list.

You can use this technique to add more than one value inside a *collection-if* or *collection-for* statement. Spread operators can also be used anywhere you wish to merge lists; they are not limited to `collection-if` and `collection-for`.

See also

Refer to these resources for a more in-depth explanation of collections:

1. Collection library: https://api.dart.dev/stable/2.18.0/dart-collection/dart-collection-library.html
2. Big-O notation: https://en.wikipedia.org/wiki/Big_O_notation

Writing less code with higher-order functions

One of the main tasks every programmer should master is dealing with data. Our apps receive data from a source, be it a web service or some local

database, and then we transform that data into user interfaces where we can collect more information and then send it back to the source. There is even an acronym for this—**Create, Read, Update, and Delete** (**CRUD**).

Throughout your life as a programmer, you will spend a lot of your time writing data manipulation code. It doesn't matter if you are working with 3D graphics or training machine learning models—dealing with data will consume a good part of your life as a developer.

In Dart, **higher-order functions** are functions that take one or more functions as their arguments, or return a function. These provide a powerful tool to manipulate data because they allow you to write reusable and modular code that can be easily adapted to different scenarios. Higher-order functions are one of the primary tools of functional programming.

Getting ready

Create a new file in your project or type this code in DartPad. Begin with an empty `main` method.

How to do it...

Higher-order functions can be divided into categories. This recipe will explore several different ways to use them. Let's get started:

1. Define the `playground` function that will define all the other types of higher-order functions that this recipe will cover:

```
void higherOrderFunctions() {
 final names = mapping();
 names.forEach(print);
 sorting();
 filtering();
 reducing();
 flattening();
}
```

2. Create a global variable called `data` that contains all the content that we will manipulate.

> **Pro Tip**
>
> You can create a global variable by adding it to the top of the file where you are working. In DartPad, just add it to the top of the screen, before the `main` method. If you are in a project, you can also add it to the top of the `main.dart` file.

3. The data in the following code block is random. You can replace this with whatever content you want:

```
List<Map> data = [
 {'first': 'Nada', 'last': 'Mueller', 'age': 10},
 {'first': 'Kurt', 'last': 'Gibbons', 'age': 9},
 {'first': 'Natalya', 'last': 'Compton', 'age': 15},
 {'first': 'Kaycee', 'last': 'Grant', 'age': 20},
 {'first': 'Kody', 'last': 'Ali', 'age': 17},
 {'first': 'Rhodri', 'last': 'Marshall', 'age': 30},
 {'first': 'Kali', 'last': 'Fleming', 'age': 9},
 {'first': 'Steve', 'last': 'Goulding', 'age': 32},
 {'first': 'Ivie', 'last': 'Haworth', 'age': 14},
 {'first': 'Anisha', 'last': 'Bourne', 'age': 40},
 {'first': 'Dominique', 'last': 'Madden', 'age': 31},
 {'first': 'Kornelia', 'last': 'Bass', 'age': 20},
 {'first': 'Saad', 'last': 'Feeney', 'age': 2},
 {'first': 'Eric', 'last': 'Lindsey', 'age': 51},
 {'first': 'Anushka', 'last': 'Harding', 'age': 23},
 {'first': 'Samiya', 'last': 'Allen', 'age': 18},
 {'first': 'Rabia', 'last': 'Merrill', 'age': 6},
 {'first': 'Safwan', 'last': 'Schaefer', 'age': 41},
 {'first': 'Celeste', 'last': 'Aldred', 'age': 34},
 {'first': 'Taio', 'last': 'Mathews', 'age': 17},
];
```

4. For this example, we will use the `Name` class, which we implemented in a previous section of this chapter:

```
class Name {
 final String first;
 final String last;
```

```
 Name(this.first, this.last);
 @override
 String toString() {
   return '$first $last';
 }
}
```

5. The first higher-order function is `map`. Its purpose is to take data in one format and quickly transform it into another format. In this example, we're going to use the `map` function to transform the raw `Map` of key-value pairs into a list of strongly typed `Name` objects:

```
List<Name> mapping() {
 // Transform the data from raw maps to a strongly
typed model
 final names = data.map<Name>((Map rawName) {
 final first = rawName['first'];
 final last = rawName['last'];
 return Name(first, last);
 }).toList(); //don't forget the toList() method!
 return names;
}
```

6. Now that the data is strongly typed, we can take advantage of the known schema to sort the list of names. Add the following function to use the `sort` function in order to alphabetize the names with just a single line of code:

```
void sorting() {
 final names = mapping();
 // Sort the list by last name
 names.sort((a, b) => a.last.compareTo(b.last));
 print('');
 print('Alphabetical List of Names');
 names.forEach(print);
}
```

7. You will often run into scenarios where you need to pull out a subset of your data. The following higher-order function will return a new list of names that only begin with the letter *M*:

```
void filtering() {
 final names = mapping();
 final onlyMs = names.where((name) =>
name.last.startsWith('M'));
 print('');
 print('Filters name list by M');
 onlyMs.forEach(print);
}
```

8. Reducing a list is the act of deriving a single value from the entire collection. In the following example, we're going to reduce to help calculate the average age of all the people on the list:

```
void reducing() {
  // Merge an element of the data together
  final allAges = data.map<int>((person) =>
person['age']);
  final total = allAges.reduce((total, age) => total +
age);
  final average = total / allAges.length;
  print('The average age is $average');
}
```

9. The final tool solves the problem you may encounter when you have collections nested within collections and need to remove some of that nesting. This function shows how we can take a 2D matrix and flatten it into a single linear list:

```
void flattening() {
  final matrix = [
    [1, 0, 0],
    [0, 0, -1],
    [0, 1, 0],
  ];
  final linear = matrix.expand<int>((row) => row);
  print(linear);
}
```

How it works...

Each of these functions operates on a list of data and executes a function on each element in this list. You can achieve the same result with `for` loops, but you would have to write a lot more code.

Mapping

In the first example, we used the `map` function. `map` expects you to take the data element as the input of your function and then transform it into something else. It is very common to map some JSON data that your app received from an API or a database to a strongly typed Dart object:

```
// Without the map function, we would usually write
// code like this
final names = <Name>[];
for (Map rawName in data) {
  final first = rawName['first'];
  final last = rawName['last'];
  final name = Name(first, last);
  names.add(name);
}
// But instead it can be simplified and it can
// actually be more performant on more complex data
final names = data.map<Name>((Map rawName) {
  final first = rawName['first'];
  final last = rawName['last'];
  return Name(first, last);
}).toList();
```

Both samples achieve the same result. In the first option, you create a list that will hold the names. Then, you iterate through each entry in the data list, extract the elements from `Map`, initialize a named object, and then add it to the list.

The second option is certainly easier for the developer. Iterating and adding are delegated to the `map` function. All you need to do is tell the `map` function how you want to transform the element. In this case, the transformation was extracting the values and returning a `Name` object. `map` is also a generic function. Consequently, you can add some typing information—in this case,

`<Name>`—to tell Dart that you want to save a list of names, not a list of dynamics.

This example is also purposefully verbose, although you could simplify it even more:

```
final names = data.map<Name>(
  (raw) => Name(raw['first'], raw['last']),
).toList();
```

This may not seem like a big deal for this simple example, but when you need to parse complex graphs of data, these techniques can save you a lot of work and time.

Don't forget to call `toList` after `map`: it's `toList` that actually creates the `List` of elements in the new format generated by `map`.

Sorting

The second higher-order function you saw in action is `sort`. Unlike the other functions in this recipe, `sort` in Dart is a **mutable function**, which is to say, it alters the original data.

A `sort` function follows this signature:

```
int sortPredicate<T>(T elementA, T elementB);
```

The function will get two elements in the collection and it is expected to return an integer to help Dart figure out the correct order:

-1	Less than
0	Same
1	Greater than

Table 3.3: Sort return values

In our example, we delegated to the string's `compareTo` function, which will return the correct integer. All this can be accomplished with a single line:

```
names.sort((a, b) => a.last.compareTo(b.last));
```

Filtering

Another common task that can be easily solved with higher-order functions is filtering. There are several cases where you or your users are only interested in a subset of your data. In these cases, you can use the `where()` function to filter your data:

```
final onlyMs = names.where((name) =>
name.last.startsWith('M'));
```

This line iterates through every element in the list and returns `true` or `false` if the last name starts with an *M*. The result of this instruction will be a new list of names that only contains the filtered items.

A function that takes an input value and returns a Boolean value is called a **predicate**. Predicates are often used to test whether an element of a collection satisfies specific conditions. The `where` function here accepts a predicate as a parameter.

For higher-order functions that expect a function that returns a Boolean, you can refer to the provided function as either a test or a predicate.

`where()` is not the only function that filters data. There are a few others, such as `firstWhere()`, `lastWhere()`, `singleWhere()`, `indexWhere()`, and `removeWhere()`, which all accept the same sort of predicate function.

Reducing

Reducing is the act of taking a collection and simplifying it down to a single value. For a list of numbers, you might want to use the `reduce` function to quickly calculate the sum of those numbers. For a list of strings, you can use `reduce` to concatenate all the values.

A `reduce` function will provide two parameters, the previous result and the current elements:

```
final total = allAges.reduce((total, age) => total + age);
```

The first time this function runs, the `total` value will be 0. The function will return 0 plus the first age value, 10. In the second iteration, the total value will 10. That function will then return 10 + 9. This process will continue until all the elements have been added to the `total` value.

Since, in many cases, you use higher-order functions as an abstraction on top of loops, we could write this code without the `reduce` function, like so:

```
int sum = 0;
for (int age in allAges) {
  sum += age;
}
```

Just like with `where()`, Dart also provides alternative implementations of `reduce` that you may want to use. The `fold()` function allows you to provide an initial value for the reducer. This may be helpful if you do not want your code to start reducing from the first element:

```
final oddTotal = allAges.fold<int>(-1000, (total, age) =>
total + age);
```

Flattening

The purpose of the `expand()` function is to look for nested collections inside your collection and flatten them into a single list. This is useful when you need to start manipulating nested data structures, such as a matrix or a

tree. There, you will often need to flatten the collection as a data preparation step before you can extract useful insights from the values:

```
final matrix = [
  [1, 0, 0],
  [0, 0, -1],
  [0, 1, 0],
];
final linear = matrix.expand<int>((row) => row);
```

In this example, every element in the matrix list is another list. The `expand` function will loop through every element, and if the function returns a collection, it will transform that collection into a linear list of values.

There's more...

There are two interesting lines in this recipe that we should pay attention to on top of explaining the higher-order functions:

```
// What is going on here?
names.forEach(print);
// Why do we have to do this?
.toList();
```

First-class functions

`names.forEach(print);` implements a pattern called **first-class functions**. This pattern dictates that functions can be treated just like any other variable. They can be stored as closures or even passed around to different functions.

The `forEach()` function expects a function with the following signature:

```
void Function<T>(T element)
```

The `print()` function has the following signature:

```
void Function(Object? object)
```

Since both of these expect a function parameter and the `print` function has the same signature, we can just provide the `print` function as the parameter!

```
// Instead of doing this
data.forEach((value) {
  print(value);
});
// We can do this
data.forEach(print);
```

This language feature can make your code more readable.

Iterables and chaining higher-order functions

If you inspected the source code of the `map` and `where` functions, you probably noticed that the return type of these functions is not a `List`, but another type called `Iterable`. This abstract class represents an intermediary state before you decide what concrete data type you want to store. It doesn't necessarily have to be a `List`. You can also convert your iterable into a `Set` if you want.

The advantage of using iterables is that they are *lazy*. Programming is one of the only professions where laziness is a virtue. In this context, laziness means that **the function will only be executed when it's needed, not earlier**. This means that we can take multiple higher-order functions and chain them together, without stressing the processor with unnecessary cycles.

We could reduce the sample code even further and add more functions for good measure:

```
final names = data
    .map<Name>((raw) => Name(raw['first'], raw['last']))
    .where((name) => name.last.startsWith('M'))
    .where((name) => name.first.length > 5)
    .toList(growable: false);
```

Each of these functions is cached in our `Iterable` and only runs when you make the call to `toList()`. Here, you are serializing the data in a model, checking whether the last name starts with M, and then checking whether the first name is longer than five letters.

This is executed in a single iteration through the list.

When you set `growable` to be `false`, the list is fixed-length.

Note

Higher-order functions and first-class functions are both used in functional programming. The difference between the two is rather subtle:

- A first-class function can be treated as a value: it can be assigned to a variable, passed as an argument, or returned as a value.
- A higher-order function takes one or more functions as arguments or returns a function as a result.

This means that first-class functions can be treated as values, and higher-order functions take or return functions as input or output.

See also

Check out these resources for more information on higher-order functions:

1. Iterable documentation: https://api.dart.dev/stable/2.18.0/dart-core/Iterable-class.html

2. *Top 10 Array Utility Methods*, by Jermaine Oppong: https://codeburst.io/top-10-array-utility-methods-you-should-know-dart-feb2648ee3a2

How to take advantage of the cascade operator

So far, you've seen how Dart follows many of the same patterns of other modern languages. Dart, in some ways, takes the best ideas from multiple languages—you have the expressiveness of JavaScript and the type safety of Java.

However, there are some features that are unique to Dart. One of those features is the **cascade** (**..**) operator.

Getting ready

Before we dive into the code, let's diverge briefly to the builder pattern. You can use this pattern to build complex objects with many properties. It can get to a point where standard constructors become impractical and unwieldy because they are too large. This is the problem the builder pattern solves. It is a special kind of class whose only job is to configure and create other classes.

This is how we would accomplish the builder pattern without the cascade operator:

```
class UrlBuilder {
  String? _scheme;
  String? _host;
  String? _path;
  UrlBuilder setScheme(String value) {
    _scheme = value;
    return this;
  }
  UrlBuilder setHost(String value) {
    _host = value;
```

```
    return this;
  }
  UrlBuilder setPath(String value) {
    _path = value;
    return this;
  }
  String build() {
    assert(_scheme != null);
    assert(_host != null);
    assert(_path != null);
    return '$_scheme://$_host/$_path';
  }
}
void main() {
  final url = UrlBuilder()
      .setScheme('https')
      .setHost('dart.dev')
      .setPath('/guides/language/language-tour#cascade-
notation-')
      .build();

  print(url);
}
```

This is very verbose. Dart can implement this pattern without any setup.

You can type the code of this recipe in DartPad.

How to do it...

Let's continue with the aforementioned less-than-optimal code and reimplement it with cascades:

1. Recreate the `UrlBuilder` class but without any of the extra methods that are usually required to implement this pattern. This new class will not look that different from a `mutable` Dart object:

```
class UrlBuilder {
  String scheme = '';
  String host = '';
  List<String> routes = [];
  @override
  String toString() {
```

```
    final paths = [host, if (routes != []) ...routes];
    final path = paths.join('/');
    return '$scheme://$path';
  }
}
```

2. Next, use the cascade operator to get the builder pattern for free. Write this function just after the declaration of the `UrlBuilder` class:

```
void cascadePlayground() {
  final url = UrlBuilder()
    ..scheme = 'https'
    ..host = 'dart.dev'
    ..routes = [
      'guides',
      'language',
      'language-tour#cascade-notation',
    ];
  print(url);
}
```

3. The cascade operator is not exclusively used for the builder pattern. It can also be used to, well, *cascade* similar operations on the same object. Add the following code inside the `cascadePlayground` function:

```
final numbers = [42, 88, 53, 232, 55]
    ..insert(0, 8)
    ..sort((a, b) => a.compareTo(b));

  print('The largest number in the list is
${numbers.last}');
```

4. Add the call to `cascadePlayground` in the `main` method:

```
void main() {
  cascadePlayground();
}
```

How it works...

Cascades are pretty elegant. They allow you to **chain** methods together that were never intended to be chained. Dart is smart enough to know that all these consecutive lines of code are operating **on the same object**. Let's have a look at the `numbers` example:

```
final numbers = [342, 23423, 53, 232, 534]
    ..insert(0, 10)
    ..sort((a, b) => a.compareTo(b));
```

Both the `insert` and `sort` methods are void functions. Declaring these objects with cascades simply allows you to remove the call to the `numbers` object:

```
final numbers = [342, 23423, 53, 232, 534];
numbers.insert(0, 10);
numbers.sort((a, b) => a.compareTo(b));
```

With the cascade operator, you can merge unrelated statements in a simple fluent chain of function calls. In our example, `UrlBuilder` is just a plain old Dart object. Without the cascade operator, we would have to write the same builder code like this:

```
final url = UrlBuilder();
url.scheme = 'https';
url.host = 'dart.dev';
url.routes = ['guides', 'language', 'language-tour#cascade-
notation'];
```

But with cascades, that code can now be simplified, like so:

```
final url = UrlBuilder()
    ..scheme = 'https'
    ..host = 'dart.dev'
    ..routes = ['guides', 'language', 'language-tour#cascade-
notation'];
```

Notice that this was accomplished without changing a single line in our class.

See also

The following resource may help you understand the builder pattern:

- Builder pattern: https://en.wikipedia.org/wiki/Builder_pattern

Using extensions

Extensions allow you to add methods and properties to existing classes, without modifying the original class.

With extensions, you *extend* the functionality of classes, and this is especially useful when you extend classes that you cannot modify.

Getting ready

To follow along with this recipe, you can write the code in DartPad.

How to do it...

To create a new method that extends the `String` class, follow these steps:

1. Create an `extension` called `StringExtensions` for the `String` class that adds a method called `toBool`, which returns `false` when the string is empty, and `true` when it's not:

```
extension StringExtensions on String {
  bool toBool() {
    return isNotEmpty;
  }
}
```

2. Create a method that calls the `toBool()` method on two strings, one empty and another with some content:

```
void testExtension() {
  String emptyString = "";
  String nonEmptyString = "Hello Extensions!";
  print(emptyString.toBool());       //--> false
  print(nonEmptyString.toBool());    //--> true
}
```

3. In the `main` method, call `testExtension()`:

```
void main() {
  testExtension();
}
```

4. Run your code. You should see `false` and `true` in the console:

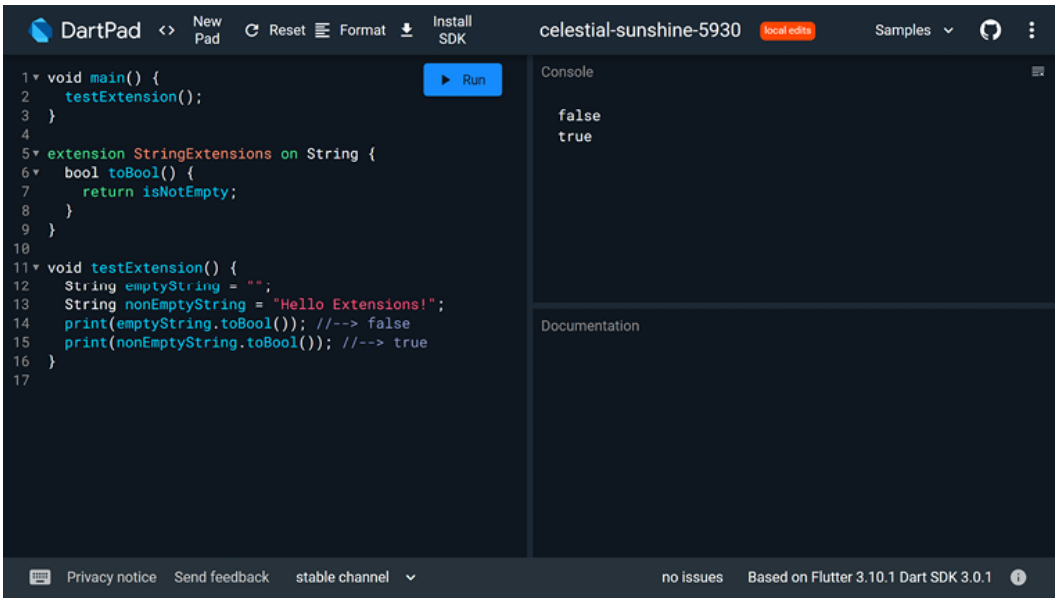


Figure 3.4: DartPad console after using an extension

How it works...

You define an extension by using the `extension` keyword, and then adding the name of the extension and the name of the class that you want to extend. In the example in this recipe, you define an extension named `StringExtensions` for the `String` class, and there you create a method called `toBool`:

```
extension StringExtensions on String {
  bool toBool() {
    return isNotEmpty;
  }
}
```

The `toBool` method checks whether the string is **not** empty, using the `isNotEmpty` property, and returns `true` if it is not empty, and `false` otherwise. In other words, it transforms a `String` into a `Boolean`.

After you define an `extension` method, you can call it on any `String`:

```
String emptyString = "";
String nonEmptyString = "Hello Extensions!";
print(emptyString.toBool());      //--> false
print(nonEmptyString.toBool());   //--> true
```

Creating an extension allows you to add methods to it without changing that class. This is a great feature whenever you want to extend classes that you cannot or don't want to change.

Introducing Dart Null Safety

When Dart version 2.12 was shipped in Flutter in March 2021, an important language feature was added that impacts how you should view `null` values for your variables, parameters, and fields: this is **sound null safety**.

Generally speaking, variables that have no value are `null`, and this may lead to errors in your code. If you have been programming for any length of time, you are probably already familiar with null exceptions in your code. The goal of null safety is to help you prevent execution errors raised by the improper use of null.

With null safety, by default, *your variables cannot be assigned a null value*.

There are obviously cases when you want to use `null`, but you have to explicitly allow null values in your apps. In this recipe, you will see how to

use null safety to your advantage, and how to avoid null safety errors in your code.

Getting ready

Create a new Pad in DartPad.

How to do it...

Let's see an example of null **unsafe** code, and then fix it. To do that, follow these steps:

1. Remove the default code in the `main` method, and add the following instructions:

```
void main() {
 int someNumber;
 increaseValue(someNumber);
}
```

2. Create a new method under `main` that takes an integer and prints the value that was passed, incremented by `1`:

```
void increaseValue(int value) {
 value++;
 print (value);
}
```

3. Note the compile-time null error in the console, as shown in the following screenshot:

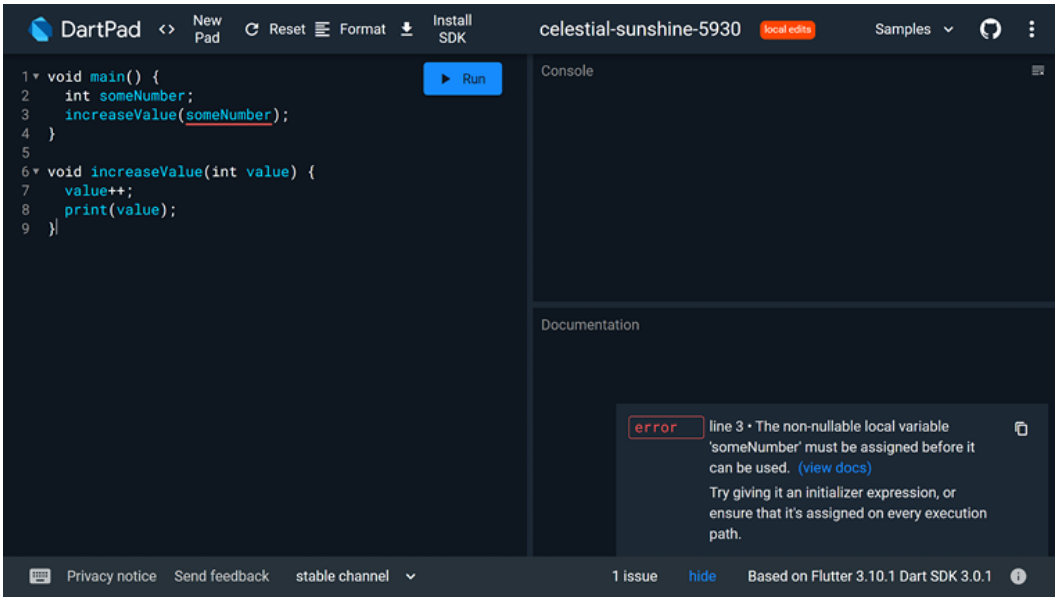


Figure 3.5: DartPad console when passing a null value

4. Add a question mark after the two `int` declarations:

```
void main() {
 int? someNumber;
 increaseValue(someNumber);
}
void increaseValue(int? value) {
 value++;
 print (value);
}
```

5. Note that the error has changed to: `" Operator '+' cannot be called on 'int?' because it is potentially null.`

6. Edit the `increaseValue` method so that you check whether the value is `null` before incrementing it; otherwise, you just return `1`:

```
void increaseValue(int? value) {
 if (value != null) {
   value++;
 } else {
   value = 1;
 }
 print (value);
}
```

7. Run the app and note that you find the value `1` in the console.
8. Edit the `increaseValue` method again. This time, use the null-check operator:

```
void increaseValue(int? value) {
 value = value ?? 0;
 value++;
 print (value);
}
```

9. Run the app and note that, in the console, you still find the value `1`.
10. Remove the question mark from the `value` parameter, and force the call to `increaseValue` with an exclamation mark:

```
void main() {
 int? someNumber;
 increaseValue(someNumber!);
}
void increaseValue(int value) {
 value++;
 print (value);
}
```

11. Run the app and note that you get an execution null exception.
12. Finally, fix the code by initializing `someNumber` with an integer value:

```
void main() {
 int someNumber = 0;
 increaseValue(someNumber);
}
void increaseValue(int value) {
 value++;
 print (value);
}
```

13. Now you should see the value `1` in the console again.

How it works...

The main reason behind the addition of null safety in Dart is that errors caused by unexpected null values are frequent and not always easy to debug.

Without null safety, you can assign to a variable a value of its type, **or null**.

With null safety, by default, you **cannot assign a null value to any variable, field, or parameter**. For instance, in the following code snippet, the second line will prevent your app from compiling:

```
int someNumber = 42; //an int value for an int variable: OK
int someOtherNumber = null; //you cannot assign null to an
int!
```

In most cases, this should not impact your code. Actually, consider the last code snippet that you wrote for this recipe, which is as follows:

```
void main() {
 int someNumber = 0;
 increaseValue(someNumber);
}
void increaseValue(int value) {
 value++;
 print (value);
}
```

This is *null-safe* code that should cover most scenarios. Here you make sure that a variable actually has a value as follows:

```
int someNumber = 0;
```

So when you pass `someNumber` to the function, you (and the compiler) can be sure that the `value` parameter will contain a valid integer, and not `null`.

There are cases though where you may need to use null values and, of course, Dart and Flutter allow you to do that. Only, you must be explicit about it. In order to make a variable, field, or parameter nullable, you can use a question mark after the type:

```
int? someNumber;
```

With the preceding code, `someNumber` becomes nullable, and therefore you can assign either an integer or a `null` value to it.

Dart will still not compile the following code though:

```
void main() {
 int? someNumber;
 increaseValue(someNumber);
}
void increaseValue(int? value) {
 value++;
 print (value);
}
```

This is probably the most interesting part of this recipe: `someNumber` is explicitly nullable, and so is the `value` parameter, but still this code will not compile. The Dart parser is smart enough to note that when you write `value++`, you risk an error, as `value` can be `null`, and therefore you are required to check whether `value` is null before incrementing it. The most obvious way to do this is with an `if` statement:

```
if (value != null) {
   value++;
} else {
   value = 1;
 }
```

But this may add several lines of code to your projects.

Another, more concise way to achieve the same result is to use the **null-coalescing operator**, which you write with a double question mark:

```
value = value ?? 0;
```

In the preceding instruction, `value` takes `0` only if `value` itself is null; otherwise, it keeps its own value.

Another very interesting code snippet we used in this recipe is the following:

```
void main() {
 int? someNumber;
 increaseValue(someNumber!);
}
void increaseValue(int value) {
 value++;
 print (value);
}
```

In the preceding code, `someNumber` may be null (`int? someNumber`), but the `value` parameter cannot be null (`int value`). The exclamation mark (`!`), also called the **bang operator** (`someNumber!`), will explicitly force the `value` parameter to accept `someNumber`. Basically, here you are telling the compiler, "Don't worry, I will make sure `someNumber` is valid, so do not raise any error." And after running the code, you get a **runtime error**.

Implementing null safety is a good way to write code, and may help you create solid apps.

See also

A thorough and complete resource for understanding null safety in Dart and Flutter is available at https://dart.dev/null-safety.

Using Null Safety in classes

While you've seen the basics of null safety in the previous recipe, you should also be aware of a few rules that should drive the way you design classes in Dart.

Getting ready

Create a new pad in DartPad and remove the default code in the `main` method.

How to do it...

Let's see an example of a plain Dart class, with two properties. Follow these steps:

1. Add a new class, with two `Strings` and no value:

```
class Person {
  String name;
  String surname;
}
```

2. Note the compile error saying that the two fields are non-nullable.
3. In the `Person` class, create a constructor that gives a value to the fields:

```
  Person(this.name, this.surname);
```

4. Note that the error is now fixed.
5. Add a named constructor that takes a map of `String, dynamic` and creates an instance of `Person`:

```
Person.fromMap(Map<String, dynamic> map) {
    name = map['name'];
    surname = map['surname'];
}
```

6. Note the compile errors that show in the console, requiring initialization of the two fields.
7. Add the `late` keyword to the two class fields, and note that this solves the errors (marking a field as `late` means that you'll initialize it later, before using it):

```
late String name;
late String surname;
```

8. Add another field to the class, this time explicitly setting it as nullable:

```
int? age;
```

9. Add the `age` field to the `Person` constructor, as follows:

```
Person(this.name, this.surname, {this.age} );
```

10. Update the `main` method, as shown here, and note the compile error:

```
void main() {
  var bruce = Person('Bruce', 'Wayne', age: 42);
  if (bruce.age < 18) {
    print ('Minor');
  }
}
```

11. Fix the error by adding a null-coalescing (`??`) operator that will return 0 if `bruce.age` is null:

```
void main() {
  var bruce = Person('Bruce', 'Wayne', age: 42);
  if ((bruce.age ?? 0) < 18) {
    print ('Minor');
  }}
```

12. At the bottom of the `main` method, add a nullable instance of `Person`, print its name, and note the error:

```
Person? person;
print (person.name);
```

13. Fix the error by adding a question mark (`?`) near `person` and run the code:

```
Person? person;
print (person?.name);
```

14. Delete the question mark and add an exclamation mark instead, run the code again, and note the error:

```
Person? person;
print (person!.name);
```

How it works...

In this recipe, you've seen a few examples of how null safety can influence the way you design classes. Having class fields that are declared and not initialized has never been a great pattern, but now you simply cannot do this anymore. The code snippet:

```
class Person {
  String name;
  String surname;
}
```

just won't compile. You could simply give some value to the fields (for example, an empty string), and this would solve the issue. Another way is creating a constructor that sets the fields:

```
Person(this.name, this.surname);
```

But then you might wonder why this does not work with the second constructor:

```
Person.fromMap(Map<String, dynamic> map) {
    name = map['name'];
    surname = map['surname'];
}
```

Why is this code raising an error? After all, it might be quite similar to the first constructor. The difference here is that with the unnamed `Person`

constructor, you are setting the fields *within the parameters* of the constructor; in the `Person.fromMap` named constructor, you set the fields in the **body** of the constructor. And this is too late.

A simple solution here can be adding the `late` keyword to the fields.

```
late String name;
late String surname;
```

When you mark a field as late, you are telling the compiler that you'll initialize it *later*, but *before* using it.

Another effect of `late` is **lazy initialization**. Dart will initialize a `late` field only when it's used for the first time.

There are cases where you want to have null fields in classes. In this case, like for any other variable, you can explicitly tell the compiler that a field is nullable with the question mark:

```
int? age;
```

In this case, you must be aware that, at any time, `age` can be null. For instance, you can't use `age` in an `if` statement unless you also use a null-coalescing (`??`) operator:

```
if ((bruce.age ?? 0) < 18) {…
```

In this case, if `bruce.age` is null, the expression will return `0`.

The last example you saw in this recipe is accessing object members when the object itself can be null. For instance, note this code:

```
Person? person;
print (person.name);
```

The `person` object can be null, and you cannot directly access any field of a nullable object. This is why `person.name` raises a compile error.

There are two ways to solve this: the first one is the "`?.`" null-aware operator. The expression `person?.name` returns null when `Person` is null, and the `name` value when `person` is *not* null.

The other way is using the bang `!` operator. With the expression `person!.name`, you are telling the compiler that `person` will never be null at that point in the code. If you are wrong, you'll get a runtime error.

See also

You can find an interesting and in-depth overview of null-aware operators at https://flutterbyexample.com/lesson/null-aware-operators.

Summary

In this chapter, you have seen an introduction to some of the core concepts of Dart programming, starting with the basics of variable declaration, where you have seen how to use `var`, `final`, and `const`, and their differences.

You have seen how to use strings and string interpolation, which allow you to add expressions within a string, making it easier to create dynamic strings. Dart supports string interpolation using the `$` symbol for variables and `${}` to evaluate an expression within a string.

You have seen different ways to use functions, and their *positional*, *optional*, and *named* parameters. You have also treated functions as values in *first-class* functions, and seen how to use *higher-order functions*, which allow you to write less code by passing functions as arguments to other functions.

You have seen how to work with collections, including `List`, `Set`, and `Map`.

You have used the cascade operator, which allows you to chain multiple method calls on an object, and the builder pattern, which allows you to simplify the creation of complex objects.

This chapter also covered an introduction to null safety, which helps prevent null runtime errors. You have seen how to leverage null safety in functions and classes, including how to use the `late` keyword.

Starting from the next chapter, you'll finally get to build real Flutter apps!

Introduction to Widgets

It’s time to finally start playing with Flutter!

In Flutter, everything you see is a **widget.** Everything your users see and interact with, such as text, a button, an image, a table, or a scrolling list, is a widget. Even the app itself is, in fact, a widget.

Most widgets in Flutter are supposed to perform a single small task. On their own, widgets are classes that perform tasks on the user interface. A `Text` widget displays text. A `Padding` widget adds space between widgets. A `Scaffold` widget provides a structure for a screen.

The real power of widgets comes not from any individual class, but from how you can compose them together to create expressive interfaces. All the widgets on a screen, when combined together, form a **widget tree**.

In Flutter, you do not have any markup language to design the UI, like XML or HTML: you create widgets in Dart.

This chapter will cover the following recipes:

- Creating immutable widgets
- Using a Scaffold
- Using the Container widget
- Printing stylish text on the screen
- Importing fonts and images into your app

Technical requirements

All the code for this project can be downloaded from https://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_04.

Creating immutable widgets

There are two kinds of visual widget in Flutter: **stateless** and **stateful**.

A **stateless widget** is simple, lightweight, and performant. Flutter can render hundreds of stateless widgets without breaking a sweat.

Stateless widgets are **immutable**. Once they are created, they cannot be modified. Flutter only has to concern itself with these widgets once. It doesn't have to maintain any complex life cycle states or worry about a block of code modifying them. In fact, the only way to modify a stateless widget is by deleting it and creating a new one.

How to do it...

Start off by creating a brand-new Flutter project called `flutter_layout`, either via your IDE or the command line. Don't worry about the sample code generated. We're going to delete it and start from scratch:

1. Open `main.dart` and delete everything.
2. Then, type the following code into the editor:

```
void main() => runApp(const StaticApp());
class StaticApp extends StatelessWidget {
  const StaticApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const ImmutableWidget(),
    );
```

```
    }
}
```

Note that you have a bunch of red underlines. We need to fix this by importing the `material.dart` library. This can be done manually, but it's more fun to let your IDE do that job. Move your cursor over the word `StatelessWidget`.

3. In VS Code, press *Ctrl* + . or *Command* + . on a Mac. In Android Studio/IntelliJ, press *Alt* + *Enter* or *Option* + *Enter* on a Mac. This will bring up a dialog where you can choose which file to import.
4. Alternatively, you can also click with your mouse on the light bulb that appears on the left of the screen. Choose the file to import from the dialog. Select `material.dart` and most of the errors will go away:

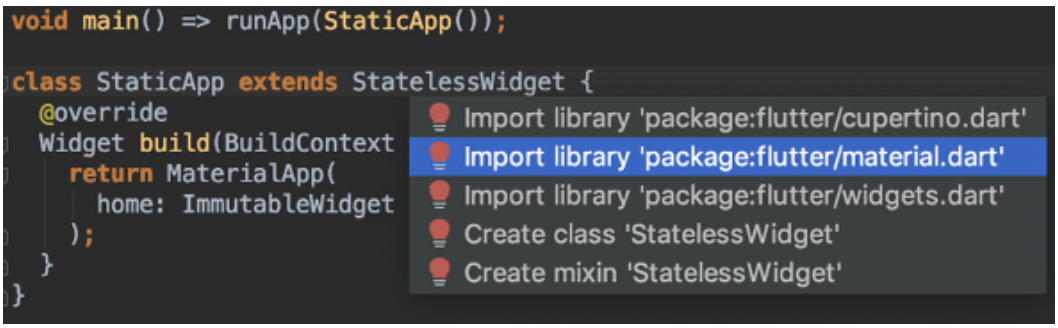


Figure 4.1: The Import helper in Android Studio

5. We'll get to the remaining error in just a second.
   The `ImmutableWidget` class can't be imported, as it doesn't exist yet.
6. Create a new file, `immutable_widget.dart`, in the project's `lib` folder. You should see a blank file.
7. There is some boilerplate with stateless widgets, but you can create them with a simple code snippet. Just type `stless`. Hit *Enter* on your keyboard and the template will appear:

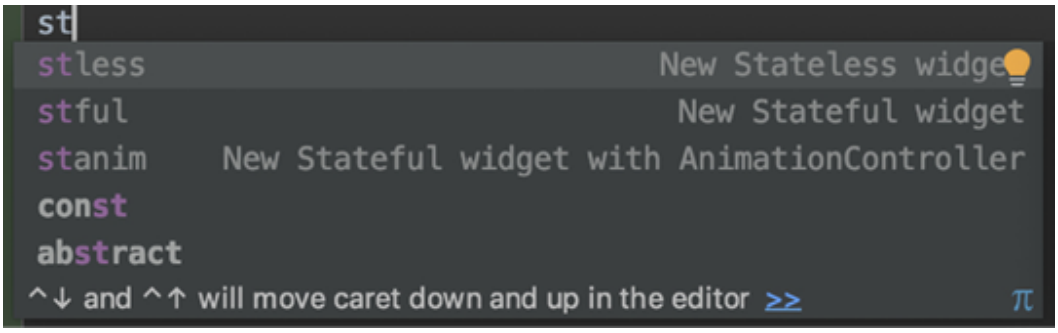


Figure 4.2: stless shortcut

8. Type in the name `ImmutableWidget` and import the material library again just as you did in *Step 2*. Now, type the following code to create a new stateless widget:

```
import 'package:flutter/material.dart';
class ImmutableWidget extends StatelessWidget {
  const ImmutableWidget({super.key});
  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.green,
      child: Padding(
        padding: const EdgeInsets.all(40),
        child: Container(
          color: Colors.purple,
          child: Padding(
            padding: const EdgeInsets.all(50.0),
            child: Container(
              color: Colors.blue,
            ),
          ),
        ),
      ),
    );
  }
}
```

9. Now that this widget has been created, we can go back to `main.dart` and import the `immutable_widget.dart` file. Move your cursor over to the constructor for `ImmutableWidget` and then click the light bulb or just type the following code at the top of the `main.dart` file:

```
import './immutable_widget.dart';
```

Note

There are two ways to import files within your project: relative and package imports. Relative imports depend on the location of the current file.

For example, with the instruction:

```
import './immutable_widget.dart';
```

the `immutable_widget.dart` file must be in the same directory as the current file. This is called a **relative import**.

You can also import the same file with the instruction:

```
import 'package:flutter_layout/immutable_widget.dart';
```

In this case, you specify your package name and the name of the file you want to import.

10. And that's it! Hit the run button to run your app in either the iOS Simulator or Android Emulator. You should see three boxes nested inside one another:



Figure 4.3: The app at the end of this recipe

Pro Tip

To see the full-color version of Figure 4.3, and all the other images in this book, download the PDF file here: https://packt.link/WE615.

Congratulations! You have now created your first custom Flutter app. That wasn't so hard, was it?

How it works...

Just like every Dart application starts with the `main` function, so does every Flutter app. But in a Flutter app, you also need to call the `runApp` function:

```
void main() => runApp(const StaticApp());
```

This line initializes the Flutter framework and places a `StaticApp`, which is just a stateless widget, at the root of the widget tree.

Our root class, `StaticApp`, is a widget. This class will be used to set up any global data that needs to be accessed by the rest of our app. However, in this case, it will just be used to kick off our widget tree, which consists of a `MaterialApp` (a widget) and the custom `ImmutableWidget` (another widget).

When we say that *Everything you see is a widget*, this implies two things:

- Every visual item in Flutter inherits from the `Widget` class. If you want it on the screen, it's a widget. Boxes are widgets. Padding is a widget. Even screens themselves are widgets.
- The core data structure in a Flutter app is the tree. Every widget can have a child or children, which can have other children, which can have other children... This nesting is referred to as a widget tree.

Note

DevTools is a set of tools to debug and measure performance on Flutter apps. They include the Flutter inspector, which you see in this recipe, but also other tools, including code analysis and diagnostics.

To learn more about DevTools, see https://flutter.dev/docs/development/tools/devtools/overview.

You can see and explore the widget tree using one of the most useful features of the Flutter DevTools from your IDE. To open the inspector while your app is running, in VS Code, perform the following steps:

1. Open the command palette (*Ctrl* + *Shift* + *P*, or *Cmd* + *Shift* + *P* on a Mac).
2. Select the **Flutter: Open DevTools Widget Inspector Page** command.

3. Alternatively, in the bottom panel, hover over the `{}` icon and click on the **Launch** button near **Dart DevTools**.

In Android Studio/IntelliJ, perform the following steps:

1. Click on the **Flutter Inspector** tab on the right of your screen.
2. Here you can see an image of the Flutter widget inspector:

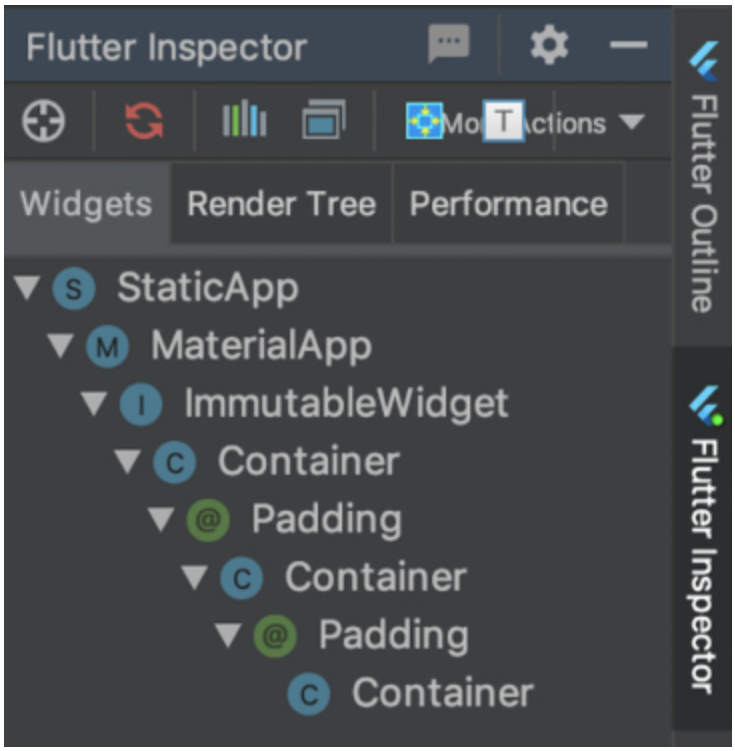


Figure 4.4: The widget tree in Android Studio

This is just a very basic app with only single-child widgets.

The core of every `StatelessWidget` is the `build()` method. Flutter will call this method every time it needs to repaint a given set of widgets. Since we are only using stateless widgets in this recipe, that should never happen, unless you rotate your device/emulator or close the app.

Let's walk through the two build methods in this example:

```
@override
Widget build(BuildContext context) {
   return MaterialApp(
     home: const ImmutableWidget(),
   );
 }
```

This first build method returns `MaterialApp`, which contains our `ImmutableWidget`. Material apps are one of the primary building blocks for apps that follow Google's Material Design specification. This widget implements a lot of features required in an app, such as navigation, theming, and localization. You can also use `CupertinoApp` if you want to follow Apple's design language, or `WidgetsApp` if you want to create your own.

We will typically use material apps at the root of the Flutter tree in this book.

Note

There are some widgets that use different property names to describe their children. `MaterialApp` uses `home`, and `Scaffold` uses `body`. Even though these are named differently, they are still basically the same as the `child` property that you will see on most of the widgets in the Flutter framework.

Let's now have a look at the second `build` method, in the `immutable_widget.dart` file:

```
 @override
 Widget build(BuildContext context) {
   return Container(
     color: Colors.green,
     child: Padding(
       padding: const EdgeInsets.all(40),
       child: Container(
         color: Colors.purple,
         child: Padding(
           padding: const EdgeInsets.all(50.0),
           child: Container(
```

```
              color: Colors.blue,
            ),
          ),
        ),
      ),
    );  }
```

This method returns a `Container` widget (green), which contains a `Padding` widget, which in turn contains another `Container` widget (purple), which contains another `Padding` widget, which contains the last `Container` widget (blue) of this tree.

A `Container` is rendered as a box that has many styling options. The three `Container` widgets are separated by two paddings of slightly different sizes.

Note

In this example, we have chosen to create `Padding` (with an uppercase P) as a widget. `Container` widgets also have a `padding` (lowercase p) property that can specify some padding for the container itself. For example, you can write the following:

```
child: Container(
padding: EdgeInsets.all(24),
color: Colors.blue,)
```

The `Padding` widget will adjust the spacing of its child, which can be any widget of any shape or size.

At the top of the `ImmutableWidget` class, note the line:

```
const ImmutableWidget({super.key});
```

This is a **constructor**. The `const` modifier makes sure that only one global instance of this object is created and no field values can be changed during the life cycle of the object.

`super.key` is a parameter that is passed to the constructor, but because it is enclosed in curly braces, it is an **optional parameter**. In fact, from the `home` property of `MaterialApp` we just call `ImmutableWidget` without passing any parameters. `super` refers to the parent class, i.e., `StatelessWidget`, and `key` is one of its properties. In particular, `key` is used to uniquely identify a widget.

Note

The code in this recipe, while correct, could also be written in a more efficient way. As `Container` has a `padding` property, we could remove the `Padding` widget. And if `color` is the only property of a `Container`, you can also use a `ColoredBox` widget instead of `Container`. So you could achieve the same results with the code shown here:

```
return Container(
      color: Colors.green,
      padding: const EdgeInsets.all(40),
      child: Container(
        color: Colors.purple,
        padding: const EdgeInsets.all(50.0),
        child: const ColoredBox(
          color: Colors.blue,
        ),
      ),
    );
```

Using a Scaffold

Android and iOS user interfaces are based on two different design languages. Android uses **Material Design**, while Apple has created the **Human Interface Guidelines** for iOS, but the Flutter team calls the iOS design pattern **Cupertino**, in honor of Apple's hometown. These two packages, Material Design and Cupertino, provide a set of widgets that mirror the user experience of their respective platforms.

These frameworks use a widget called `Scaffold` (`CupertinoScaffold` in the Cupertino framework) that provides a basic structure of a screen.

In this recipe, you are going to give your app some structure. You will be using the `Scaffold` widget to add an `AppBar` to the top of the screen and a slide-out drawer that you can pull from the left.

Getting ready

You should have completed the previous recipe in this chapter before following along with this one.

Create a new file in the project called `basic_screen.dart`. Note that *you can have the app running while you make these code changes*. You could also adjust the size of your IDE so that the iOS Simulator or Android Emulator can fit beside it:

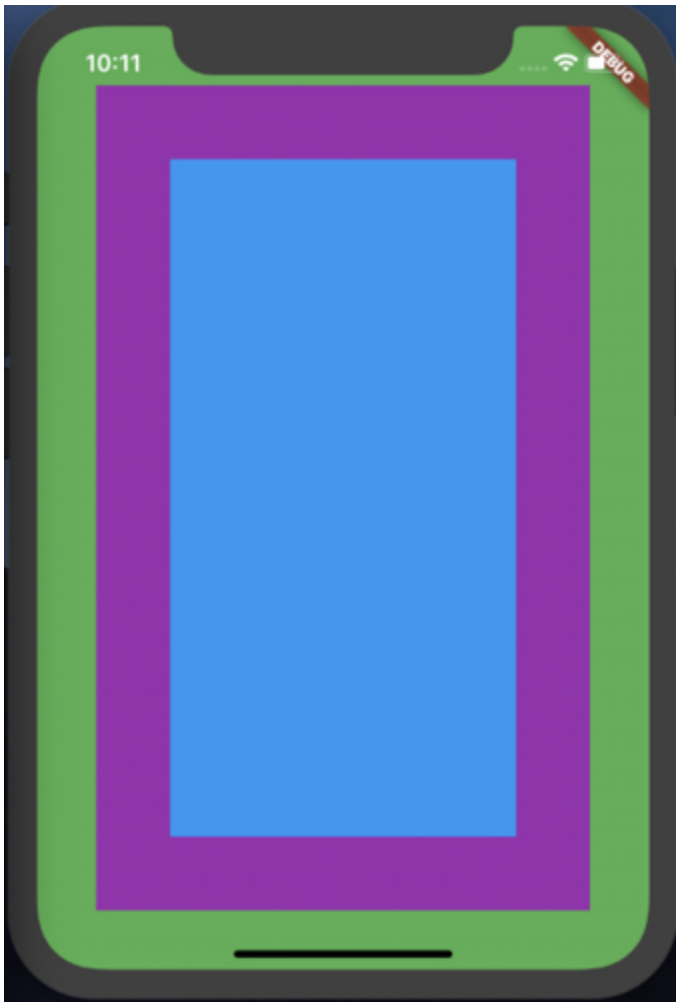


Figure 4.5: The app running on the iOS Simulator

By setting up your workspace in this way, it will be much easier to see code changes automatically injected into your app. (If you are lucky enough to be using two monitors, this does not apply, of course.)

How to do it...

Let's start by setting up a `Scaffold` widget:

1. In `basic_screen.dart`, type `stless` to create a new stateless widget and name that widget `BasicScreen`. Don't forget to import the Material Design library as well:

```
import 'package:flutter/material.dart';
class BasicScreen extends StatelessWidget {
  const BasicScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

2. Now, in `main.dart`, replace `ImmutableWidget` with `BasicScreen`. Hit the save button to hot reload and your simulator screen should be completely black:

```
import 'package:flutter/material.dart';
import './basic_screen.dart';
void main() => runApp(const StaticApp());
class StaticApp extends StatelessWidget {
  const StaticApp({super.key});
  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: BasicScreen(),
    );
  }
}
```

3. It's time to bring in the `Scaffold` widget. In `basic_screen.dart`, we're going to add the widget that was created in the previous recipe, but bring it under control with the `AspectRatio` and `Center` widgets:

```
import 'package:flutter/material.dart';
import './immutable_widget.dart';
class BasicScreen extends StatelessWidget {
  const BasicScreen ({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: AspectRatio(
          aspectRatio: 1.0,
          child: ImmutableWidget(),
        ),
      ),
    );  }
}
```

The screen should now look like this:

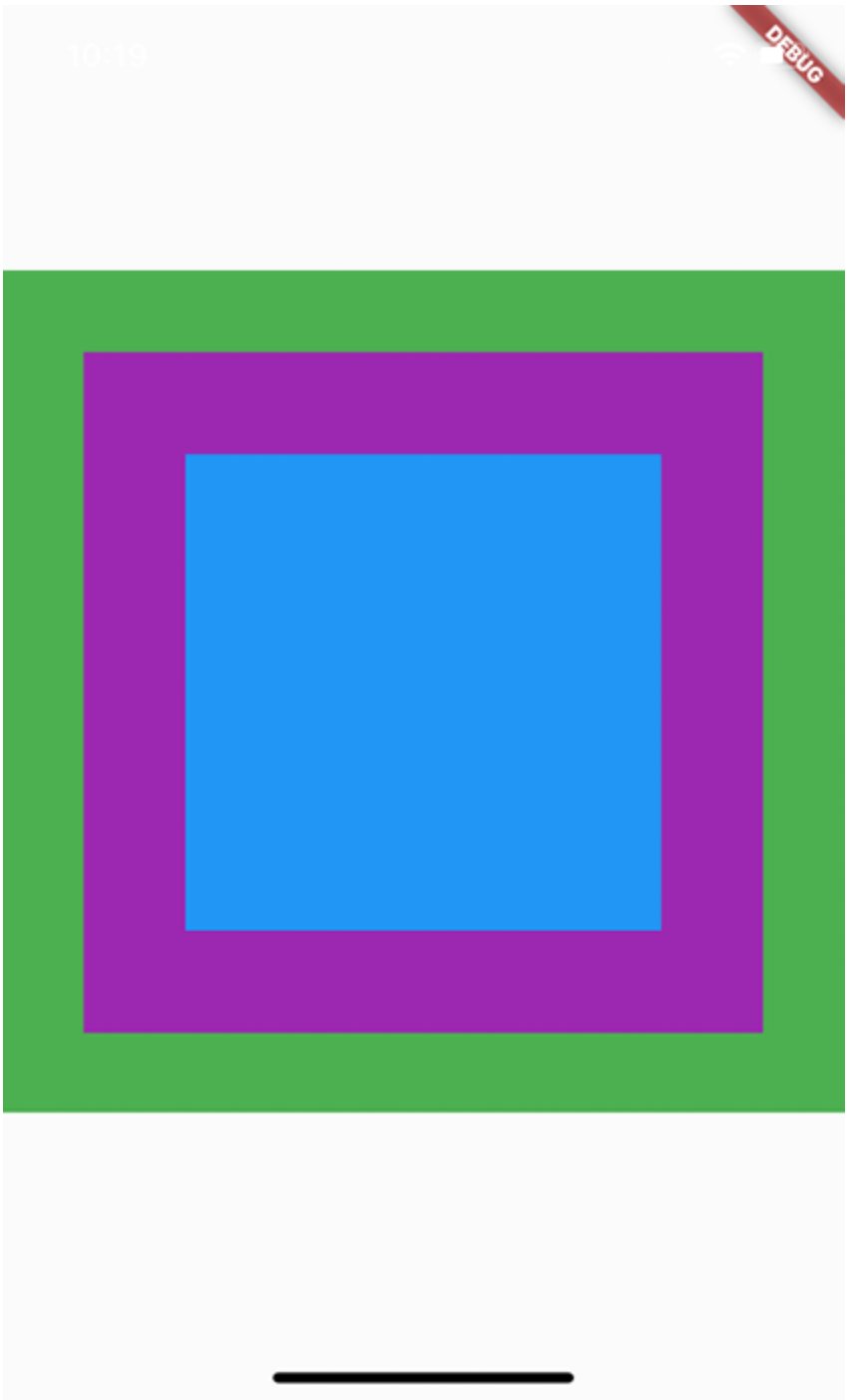


Figure 4.6: The app with a 1.0 aspectRatio

4. Probably one of the most popular widgets in an app is **AppBar**. This is a persistent header that lives at the top of the screen and helps you navigate from one screen to the next. Add the following code to the scaffold:

```
return Scaffold(
  appBar: AppBar(
        backgroundColor: Colors.indigo,
        title: const Text('Welcome to Flutter'),
        actions: const [
           Padding(
            padding: EdgeInsets.all(10.0),
            child: Icon(Icons.edit),
          ),
        ],  ),
  body: Center(
...
```

5. Hot reload the app and you will now see an app bar at the top of the screen:

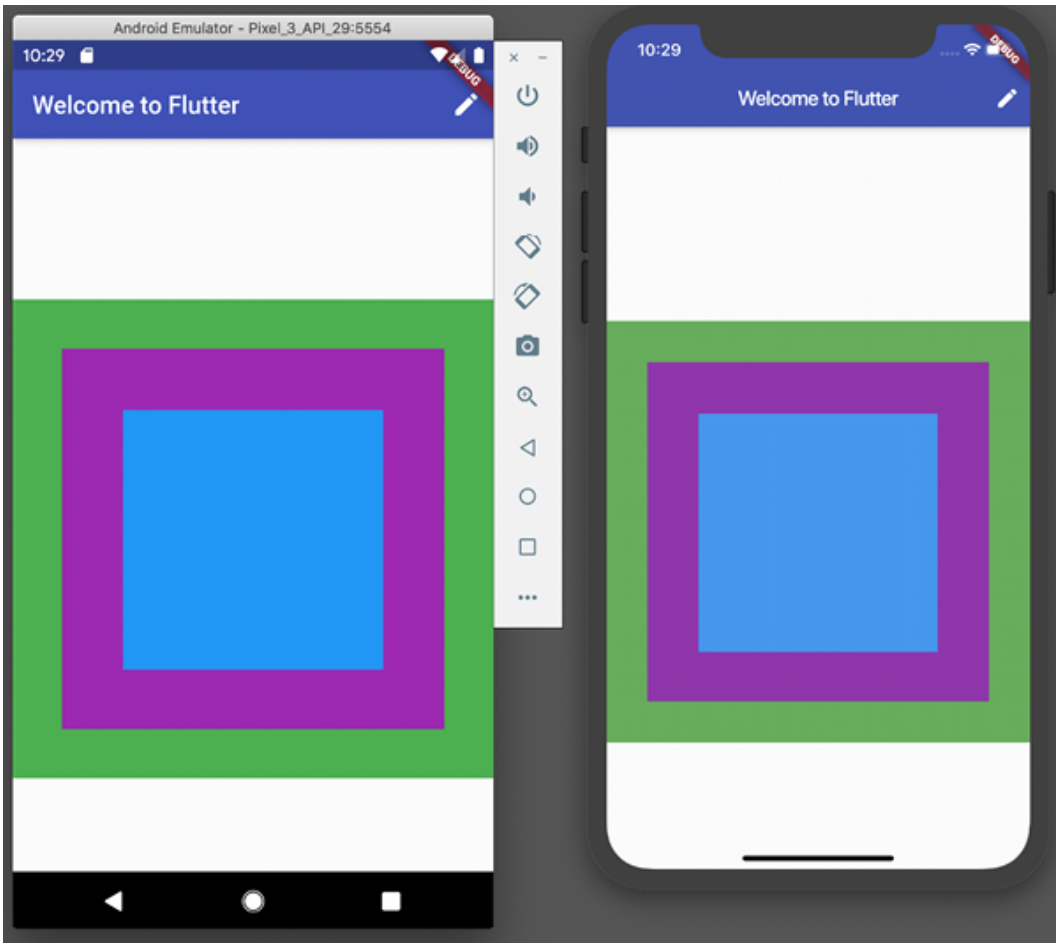


Figure 4.7: The app running on the Android Emulator and iOS Simulator

6. Finally, let's add a drawer to the app. Add this code to `Scaffold`, just after `body`:

```
body: Center(
  child: AspectRatio(
    aspectRatio: 1.0,
    child: ImmutableWidget(),
  ),
),
drawer: Drawer(
  child: Container(
       color: Colors.lightBlue,
         child: const Center(
         child: Text("I'm a Drawer!"),
    ),
),
),
```

7. The final app should now have a "hamburger" icon (the three lines in the top left, stacked to look like a hamburger) in the `AppBar`. If you press it, the drawer will be shown:

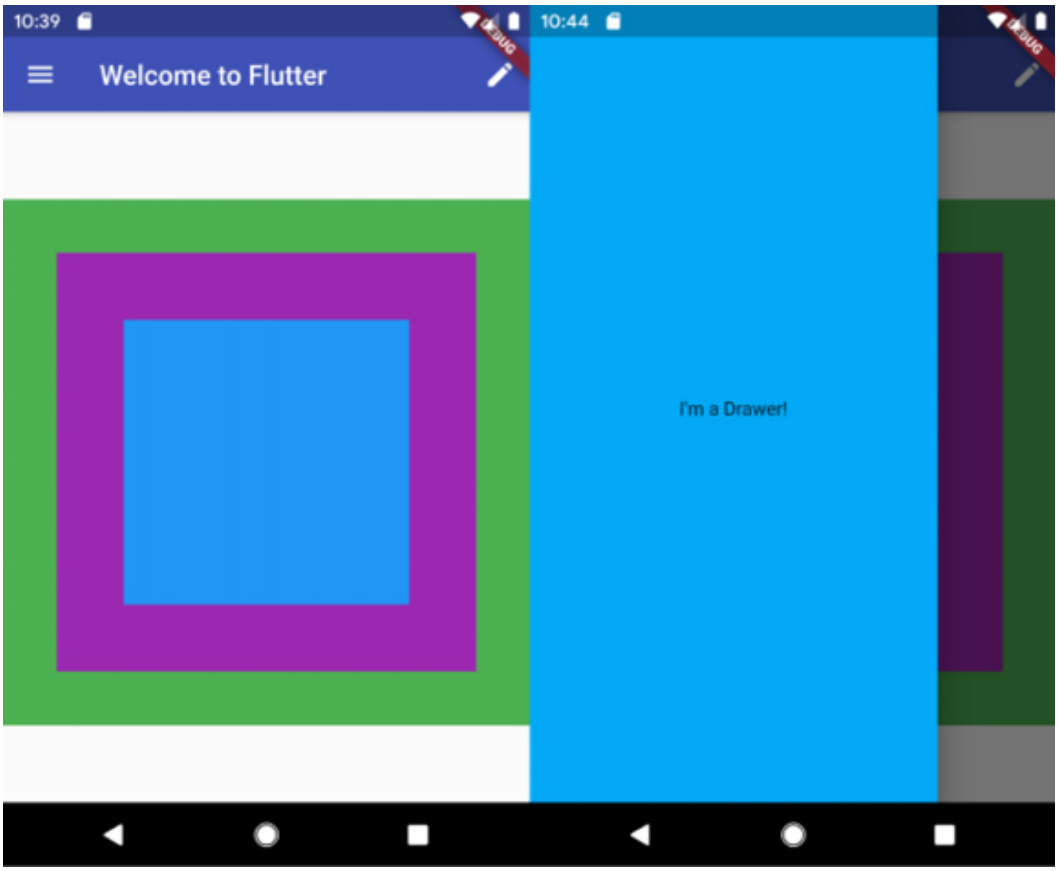


Figure 4.8: A drawer showing after pressing the “hamburger” button

How it works...

`Scaffold` is, as you may guess, a widget. It is usually recommended to use a `Scaffold` widget as the root widget for your screen, as you have in this recipe, even if it is not strictly required. You generally use a `Scaffold` widget when you want to create a screen. Widgets that do not begin with `Scaffold` are intended to be *components* used to compose screens.

Scaffolds are also aware of your device’s specifics. `AppBar` will render differently depending on whether you are on iOS or Android. These are known as **platform-aware widgets**. When you add an app bar and you run your app on iOS, `AppBar` formats itself to avoid the iPhone’s notch. If you run the app on an iOS device that doesn’t have a notch, like the iPhone 8 or an iPad, the extra spacing added for the notch is automatically removed.

There are also other tools for `Scaffold` that we will cover in the next chapters.

> **Pro Tip**
>
> Even if you don't plan on using any of the components that `Scaffold` provides, it is recommended to start every screen with a scaffold to bring consistency to your app's layout.

There are two other widgets you used in this recipe: `Center` and `AspectRatio`.

A **Center widget** centers its child both horizontally and vertically.

You can use the **AspectRatio widget** when you want to size a widget following a specific aspect ratio. The `AspectRatio` widget tries the largest width possible in its context, and then sets the height, applying the chosen aspect ratio to the width. For instance, an `AspectRatio` of `1` will set the height to be equal to the width.

Using the Container widget

We've already played around a bit with the `Container` widget in the previous recipes, so now we will build upon what you've seen before and add other features of this versatile widget. In this recipe, you will add some new effects to the existing `ImmutableWidget`.

Getting ready

Before following this recipe, you should have completed the two previous recipes in this chapter, *Creating immutable widgets*, and *Using a Scaffold*.

I suggest you also leave the app running while you are typing your code, so you can see your changes via hot reload every time you save your file.

How to do it...

Let's start by updating the small box in the center and turning it into a circle:

1. In the `ImmutableWidget` class, replace the third `Container` with this method:

```
@override
Widget build(BuildContext context) {
  return Container(
    color: Colors.green,
    child: Padding(
      padding: EdgeInsets.all(40),),
      child: Container(
        color: Colors.purple,
        child: Padding(
          padding: const EdgeInsets.all(50.0),
          child: _buildShinyCircle(),
        ),
      ),
    ),
  );
}
```

2. Write the `_buildShinyCircle()` method. You will be adding `BoxDecoration` to a `Container`, which can include gradients, shapes, shadows, borders, and even images.

> **Pro Tip**
>
> After adding `BoxDecoration`, you should make sure to remove the original `color` property on the container; otherwise, you will get an exception. Containers can have a `decoration` or a `color`, but not both.

3. Add the following code at the end of the `ImmutableWidget` class:

```
Widget _buildShinyCircle() {
   return Container(
```

```
      decoration: const BoxDecoration(
        shape: BoxShape.circle,
        gradient: RadialGradient(
          colors: [
            Colors.lightBlueAccent,
            Colors.blueAccent,
          ],
          center: Alignment(-0.3, -0.5),
        ),
        boxShadow: [
          BoxShadow(blurRadius: 20),
        ],
      ),
    );
  }
}
```

Circles are only one kind of shape that can be defined in a `Container`. You can create rounded rectangles and give these shapes an explicit size instead of letting Flutter figure out how large the shape should be.

4. Add an `import` statement to the top of the screen for Dart's math library and give it an alias of `Math`:

```
import 'dart:math' as math;
```

5. Now, update the second container (the purple one) with the decoration below and wrap it in a `Transform` widget. To make your life easier, you can use your IDE to insert another widget inside the tree. Move your cursor to the declaration of the second `Container` and then, in VS Code, press *Ctrl* + . (*Command* + . on a Mac) and in Android Studio, press *Alt* + *Enter* (*Option* + *Enter* on a Mac) to bring up the following context dialog:

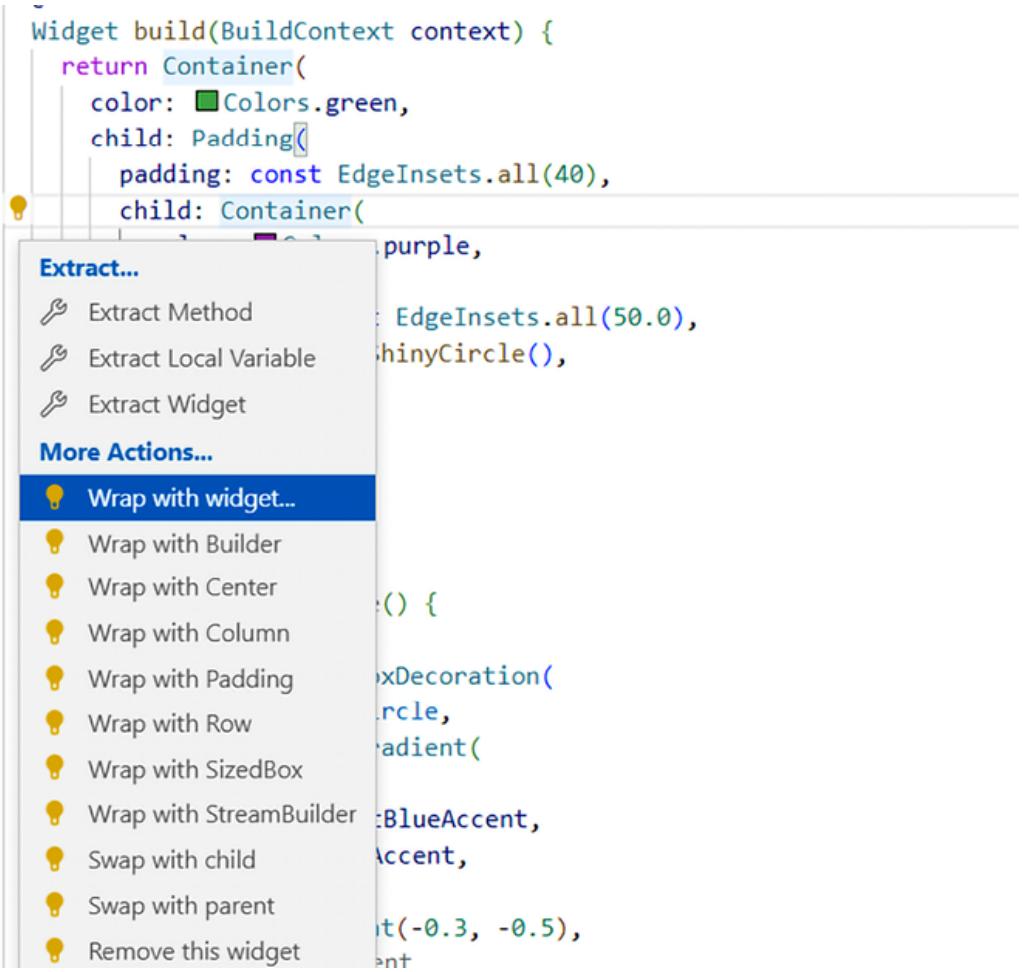


Figure 4.9: A menu showing Flutter code actions

6. Select **Wrap with widget...** or **Wrap with a new widget**, which will insert a placeholder in your code. Replace the placeholder with `Transform.rotate` and add the missing properties, as shown in the updated code here:

```
return Container(
  color: Colors.green,
  child: Center(
    child: Transform.rotate(
     angle: 180 / Math.pi, // Rotations are supplied in
radians
      child: Container(
        width: 250,
        height: 250,
        decoration: BoxDecoration(
          color: Colors.purple,
```

```
          boxShadow: [
            BoxShadow(
              color: Colors.deepPurple.withAlpha(120),
              spreadRadius: 4,
              blurRadius: 15,
              offset: Offset.fromDirection(1.0, 30),
            ),
          ],
          borderRadius: const
BorderRadius.all(Radius.circular(20)),
        ),
        child: Padding(
          padding: const EdgeInsets.all(50.0),
          child: _buildShinyCircle(),
        ),
      ),
    ),
  ),
);
```

7. You will now add some style to the top widget. Containers actually support two types of decorations: **foreground decorations** and **background decorations**. These two types of decorations can even be blended together to create interesting effects. Add this code to the root container in `ImmutableWidget`:

```
return Container(
  decoration: const BoxDecoration(color: Colors.green),
      foregroundDecoration: const BoxDecoration(
        backgroundBlendMode: BlendMode.colorBurn,
        gradient: LinearGradient(
          begin: Alignment.topCenter,
          end: Alignment.bottomCenter,
          colors: [
            Color(0xAA0d6123),
            Color(0x00000000),
            Color(0xAA0d6123),
          ],
        ),
      ),
    ),
 child: […]
```

Your final project should look similar to that in *Figure 4.10*:

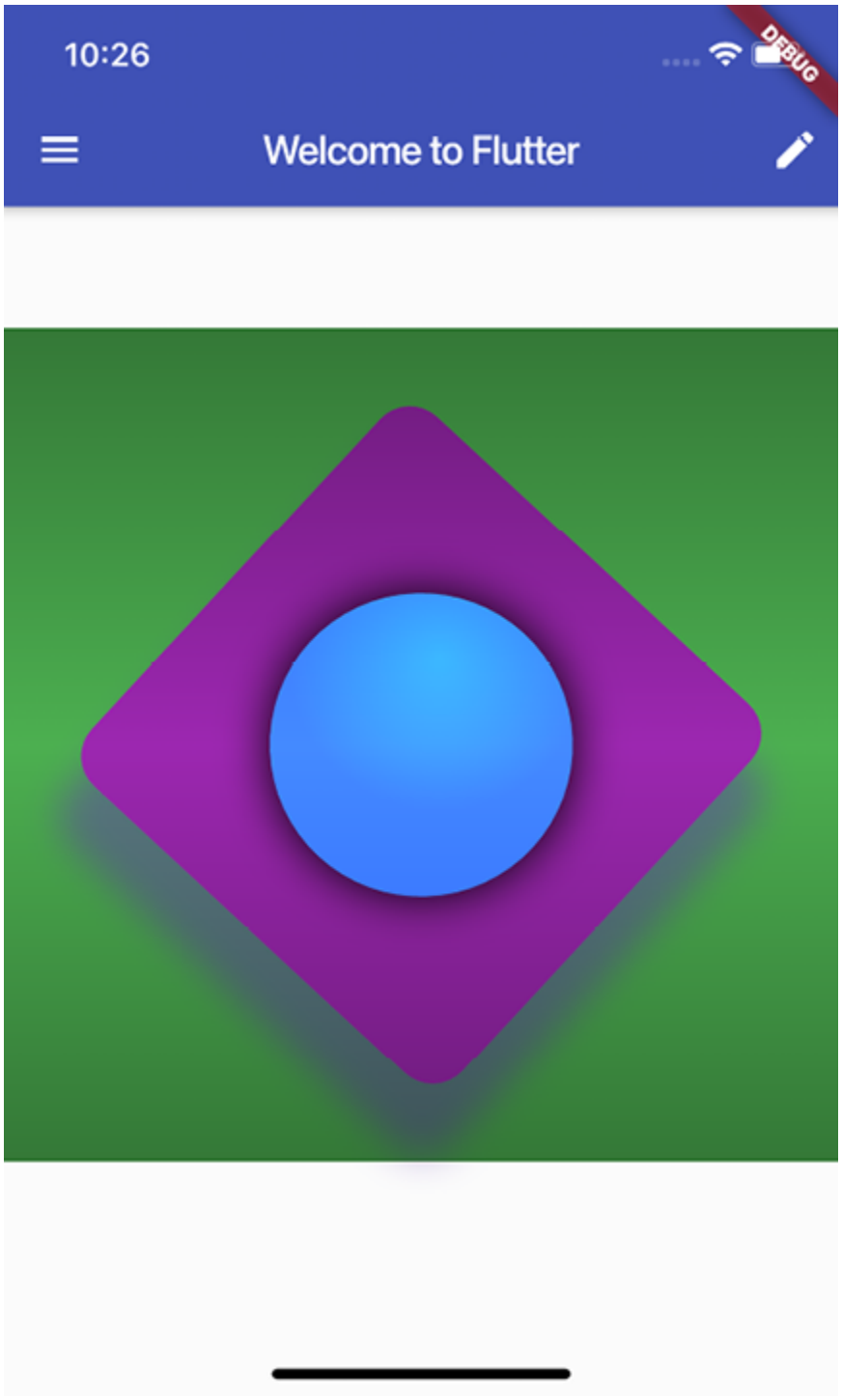


Figure 4.10: The screen after completing this recipe

How it works...

`Container` widgets can add several effects to their children. Like Scaffolds, they enable customizations that can be explored and experimented with.

A `BoxDecoration` can draw:

- Borders
- Shadows
- Colors
- Gradients
- Images
- Shapes (rectangle or circles)

The `Container` itself supports two decorations—the primary background decoration and a foreground decoration, which is painted on top of the container's child.

Containers can also provide their own transforms (like how you rotated the second container), paddings, and margins. Sometimes, you may prefer adding properties such as `padding` inside a container. In other cases, you may use a `Padding` widget and add `Container` as its child, as we did in this recipe. Both achieve the same result.

In this recipe, we *could* also have rotated the box by supplying a `Matrix4` to the `transform` property of the container, but delegating that task to a separate widget follows Flutter's ideology: widgets should only do one very small thing and be composed to create complex designs.

Pro Tip

Don't worry about how deep your widget tree gets. Flutter can take it. Widgets are extremely lightweight and are optimized to support hundreds of layers. The widget itself doesn't do any rendering; it just provides a configuration for the renderer. The actual rendering is done in two more parallel trees, the `Element` tree and the `RenderObject` tree. Flutter uses these internal trees to talk to the GPU.

Printing stylish text on the screen

Almost every app has to display some text. Flutter has a powerful and fast text engine that can render all the rich text that you'd expect from a modern mobile framework.

In this recipe, we will be drawing text with Flutter's two primary widgets—`Text` and `RichText`. The `Text` widget is the most common way to quickly print text on the screen, but you will also occasionally need `RichText` when you want to add even more style within a single line.

Getting ready

To follow along with this recipe, you should have completed all of the previous recipes in this chapter.

Create a new file called `text_layout.dart` in your project's `lib` directory.

As always, make sure that your app is running in either a simulator/emulator or an actual device to see the changes in your app in real time with the hot reload feature.

How to do it...

Let's get started with some basic text:

1. In your `text_layout.dart` file, add the shell for a class called `TextLayout`, which will extend `StatelessWidget`. Import all the requisite packages:

```
import 'package:flutter/material.dart';
class TextLayout extends StatelessWidget {
  const TextLayout({super.key});
  @override
  Widget build(BuildContext context) {
```

```
    return Container();
  }
}
```

2. Open `basic_screen.dart` and perform these updates so that the `TextLayout` widget will be displayed underneath `ImmutableWidget`.

   For the sake of brevity, `AppBar` and the `Drawer` code have been removed and an ellipsis (…) is shown instead:

```
import 'package:flutter/material.dart';
import 'package:flutter_layout/immutable_widget.dart';
import 'package:flutter_layout/text_layout.dart';
class BasicScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(...),
      body: Column(
        children: const [
          AspectRatio(
            aspectRatio: 1.0,
            child: ImmutableWidget(),
          ),
          TextLayout(),
        ],
      ),
 drawer: Drawer(...),
    );
  }
}
```

3. Now, moving back to the `text_layout.dart` file, let's edit the `build` method, adding a column containing three `Text` widgets:

```
  @override
  Widget build(BuildContext context) {
    return Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: [
        const Text(
          'Hello, World!',
          style: TextStyle(fontSize: 16),
```

```
        ),
        Text(
          'Text can wrap without issue',
          style: Theme.of(context).textTheme.headline6,
        ),
//make sure the Text below is all in one line:
        const Text(
            'Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Etiam at mauris massa. Suspendisse
potenti. Aenean aliquet eu nisl vitae tempus.'),
      ],
    );  }
```

When you run the app, it should look like this:


10:09
DEBUG
Welcome to Flutter
Hello, World!
Text can wrap without issue
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam at mauris massa. Suspendisse potenti. Aenean aliquet eu nisl vitae tempus.

Figure 4.11: Formatted text

4. All of these `Text` widgets can take a single style object, but what if you want to add multiple styles to different parts of your sentences? That is the job of `RichText`. Add these new widgets just after the last widget in your column:

```
const Divider(),
   RichText(
      text: const TextSpan(
      text: 'Flutter text is ',
      style: TextStyle(fontSize: 22, color:
Colors.black),
        children: <TextSpan>[
          TextSpan(
            text: 'really ',
            style: TextStyle(
              fontWeight: FontWeight.bold,
              color: Colors.red,
            ),
            children: [
              TextSpan(
                text: 'powerful.',
                style: TextStyle(
                  decoration: TextDecoration.underline,
                  decorationStyle:
TextDecorationStyle.double,
                  fontSize: 40,
                ),
              ),
            ],
          ),
        ],
      ),),
```

This is what the final screen should look like:

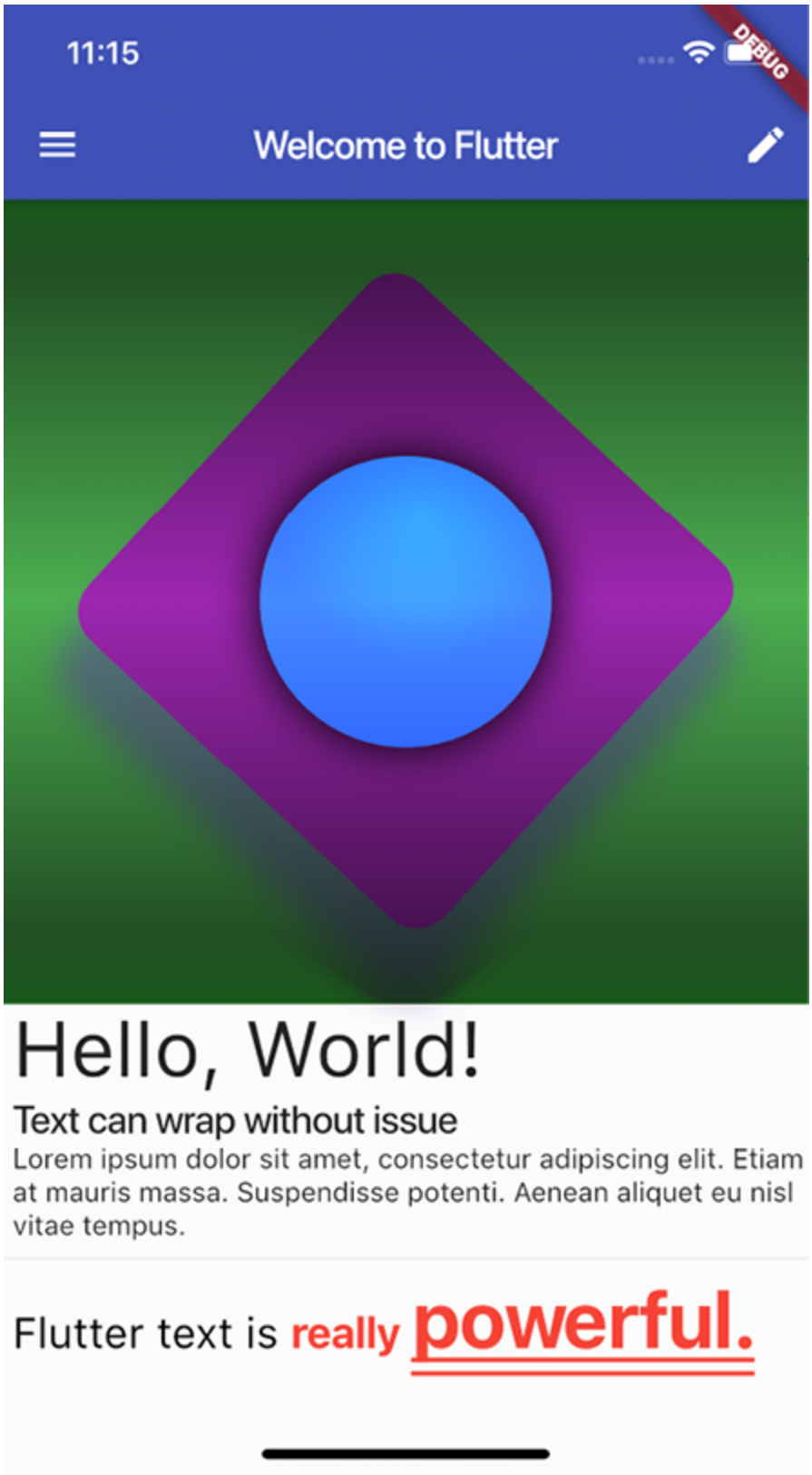


Figure 4.12: The screen after completing this recipe

How it works...

Hopefully, most of the code for the `Text` widget should be self-evident. It's just a matter of creating hundreds of these widgets over time, which will eventually create fluency with this API. The `Text` widget has some basic properties that warrant discussion, such as text alignment and setting a maximum number of lines, but the real meat is in the `TextStyle` object. There are several properties in `TextStyles` that are exhaustively covered in the official documentation, but you will most frequently be adjusting the font size, color, weight, and font.

As a bonus, `Text` widgets are accessibility-aware out of the box. There is no extra code that you'll need to write. `Text` widgets respond to the text-to-speech synthesizers and will even scale their font size up and down if the user decides to adjust the system's font size.

The `RichText` widget creates another tree of `TextSpan`, where each child inherits its parent's style, but can override it with its own.

We have three spans in the example and each one adds a bit more to the inherited style:

- Font size 22, colored black
- Font weight bold, colored red
- Font size 40, double underline

The final span will be styled with the sum of all its parent spans in the tree.

There's more...

At the beginning of the recipe, did you notice this line?

```
Theme.of(context).textTheme.headline6,
```

This is a very common Flutter design pattern, and we will call it the **of-context** pattern in this book. This is used to access data from other parts higher up the widget tree.

Every build method in every widget is provided with a `BuildContext` object, which is a very abstract-sounding name. `BuildContext`, or `context` for short, can be thought of as a marker of your widget's location in the tree. This `context` can then be used to travel up and down the widget tree.

In this case, we're handing our context to the static `Theme.of(context)` method, which will then search up the tree to find the closest `Theme` widget. The theme has predetermined colors and text styles that can be added to your widgets so that they will have a consistent look in your app. This code is adding the global `headline6` style to this text widget.

See also

If you want to learn more about how to set themes in your apps and even create your own custom ones, have a look at the official guide at https://docs.flutter.dev/cookbook/design/themes.

Importing fonts and images into your app

Text and colors are nice, but pictures are often worth a thousand words. The process of adding custom images and fonts to your app is a little more complex than you might expect. Flutter has to work within the constraints of its host operating systems, and since iOS and Android like to do similar things in different ways, Flutter creates a unified abstraction layer on top of both of their filesystems.

In this recipe, we will be using asset bundles to add a photo at the top of the screen and use a custom font.

Getting ready

You should have completed the previous recipe in this chapter, *Printing stylish text on the screen*, before following along with this one.

You will add an image to the app. You can get some great free stock photography from Unsplash. Download this beach image by Khachik Simonian as well: https://unsplash.com/photos/nXOB-wh4Oyc.

How to do it...

Let's update the previous recipe's code with some new fonts:

1. Open the `pubspec.yaml` file in the root folder of your project.
2. In the `pubspec.yaml` file, add the `google_fonts` package in the `dependencies` section, but be careful. YAML is one of the few languages where white space matters, so be sure to put the `google_fonts` dependency precisely with the same indent as `flutter`, as shown here:

```
dependencies:
  flutter:
    sdk: flutter
  google_fonts: ^3.0.1
```

3. You can also add a dependency from the terminal, typing:

```
flutter pub add google_fonts
```

4. After this is done, run `flutter pub get` to rebuild your asset bundle, or alternatively press the **Get Packages** button in your editor (in Android Studio/IntelliJ you'll find it in the action ribbon at the top of `pubspec.yaml`, in VS Code it's an icon at the top right of the screen).
5. We can now add any Google font to the text widgets in `text_layout.dart`. Add the `google_fonts` import at the top of the file:

```
import 'package:google_fonts/google_fonts.dart';
```

6. Update the first `Text` widget to reference the `leckerliOne` font:

```
const Text(
    'Hello, World!',
    style: GoogleFonts.leckerliOne(fontSize: 40),
),
```

The `leckerliOne` font will now be rendered on the screen:



Figure 4.13: Hello World with the leckerliOne font

7. Now you will add a picture to the screen. Stop the app. Then, at the root of your project, create a new folder called `assets`.
8. Rename the file you have downloaded (refer to the *Getting ready* section of this recipe) to something simple, such as `beach.jpg`, and drag the image to the `assets` folder.

9. Update the `pubspec.yaml` file once again. Locate and uncomment the `assets` section of the file to include the image folder in your project:

```
# To add assets to your application, add an assets
section, like this:
  assets:
    - assets/
```

10. In `basic_screen.dart`, replace `body` of the `Scaffold` with this code:

```
body: Column(
  crossAxisAlignment: CrossAxisAlignment.start,
  children: [
    Image.asset('assets/beach.jpg'),
    const TextLayout(),
  ],
),
```

11. Wrap the image with a `Semantics` widget, and add a description of the image you are showing:

```
Semantics(
     image: true,
     label: 'A beautiful beach',
     child: Image.asset('assets/beach.jpg')),
```

The final layout should show the image at the top of the screen:



Figure 4.14: An image showing on the screen

How it works...

In this recipe, you have seen two common features in Flutter: choosing fonts for your `Text` widgets and adding images to your screens.

You may use any custom font in Flutter. When working with Google Fonts, adding them to your project is extremely easy. You just need to add the `google_fonts` package dependency in the `pubspec.yaml` file to your app:

```
google_fonts: ^3.0.1
```

Note

There are currently about 1,000 fonts you can use for free in your apps with Google Fonts. Have a look at the official site, https://fonts.google.com/, to choose the right one for your apps.

When you want to use the `google_fonts` package in one or more of your screens, you need to import the package at the top of the file. You can do this in the `text_layout.dart` file with the help of the following command:

```
import 'package:google_fonts/google_fonts.dart';
```

From there, you just need to use the package. You add the `GoogleFonts` widget to the `style` property of your `Text`:

```
 style: GoogleFonts.leckerliOne(fontSize: 40),
```

When adding the image to the `pubspec.yaml` file, you have provided Flutter with instructions on how to build an asset bundle. The bundles then get converted to their platform equivalents (`NSBundle` on iOS and `AssetManager` on Android) where they can be retrieved through the appropriate file API.

For listing assets, we thankfully do not have to explicitly reference every single file in our assets directory:

```
- assets/
```

This shorthand is equivalent to saying that you want to add each file in the `assets` directory to the asset bundle.

You can also write this:

```
- assets/beach.jpg
```

This notation will only add the file that you specify in that exact directory.

If you have any sub-directories, they will also have to be declared in `pubspec`. For example, if you have `images` and `icons` folders in the `assets` folder, you should write the following:

```
- assets/
- assets/images/
- assets/icons/
```

You might want to keep all your project assets in one flat directory because of this and not go too crazy with the file organizations.

The last step in this recipe was wrapping the image within a `Semantics` widget:

```
Semantics(
     image: true,
     label: 'A beautiful beach',
     child: Image.asset('assets/beach.jpg')),
```

You can use the `Semantics` widget to describe the purpose or meaning of a widget to users with accessibility needs. In our example, we provide a label to describe the content of the image. The text description in the label can be read by screen readers and other accessibility tools.

See also

For a comprehensive guide on how to add assets in Flutter, have a look at the assets guide: https://docs.flutter.dev/development/ui/assets-and-images.

Summary

In this chapter, you have had an introduction to building user interfaces in a Flutter app.

You have seen how to create immutable widgets and use them in your apps.

The chapter covered the use of the `Scaffold` widget, which provides a basic structure for screens in your apps.

You have also used Containers, powerful widgets that provide several layout-related properties and effects.

You have used the `Text` widget to print text on the screen, and seen how to customize the font, color, size, and other properties of the text.

You've seen how to import fonts and images into a Flutter app: in particular, you added and used Google fonts and displayed an image from the assets directory in your project.

Finally, you have seen how to describe onscreen content using the `Semantics` widget: this can help to improve the accessibility of your app.

In general, you are now familiar with the basics of creating user interfaces. The skills that you have acquired in this chapter will provide the foundation for building engaging and performant UIs in Flutter, in both simple and more complex apps.

Mastering Layout and Taming the Widget Tree

A tree data structure is a favorite of computer engineers (especially in job interviews!). Trees elegantly describe hierarchies with a parent-child relationship. You can find **user interfaces** (**UIs**) expressed as trees everywhere. **HyperText Markup Language** (**HTML**) and the **Document Object Model** (**DOM**) are trees. UIViews and their subviews are trees. The Android **Extensible Markup Language** (**XML**) and the .NET **Extensible Application Markup Language** (**XAML**) layouts are also trees. While developers are subconsciously aware of this data structure, it's not nearly as present in the foreground as it is with Flutter. If you do not live and breathe trees, at some point, you will get lost among the leaves.

This is why managing your widget trees becomes more important as your app grows. You could, in theory, create one single widget that is tens of thousands of layers deep, but maintaining that code would be a nightmare!

This chapter will cover various techniques that you can use to bring the widget tree to heel. We will explore layout techniques with columns and rows, as well as refactoring strategies that will be critical to keeping your widget tree easy to maintain.

In this chapter, we will be covering the following recipes:

- Placing widgets one after another
- Proportional spacing with the `Flexible` and `Expanded` widgets
- Drawing shapes with `CustomPaint`
- Nesting complex widget trees
- Refactoring widget trees to improve legibility
- Applying global themes

Placing widgets one after another

Writing layout code, especially when we are dealing with devices of all sorts of shapes and sizes, can get complicated.

Thankfully, Flutter makes writing layout code relatively simple. As Flutter is a rather young framework, it has the advantage of learning from previous layout solutions that have been used on the web, desktop, iOS, and Android. Armed with the knowledge of the past, the Flutter engineers were able to create a layout engine that is flexible, responsive, and developer friendly.

In this recipe, we're going to create a prototype for a profile screen for dogs.

Getting ready

1. Create a new Flutter project with your favorite editor. You can call it `dogs`.
2. You're going to need a picture for the profile image. You can choose any image you want, or you can just download this image of a dog created by `@charlesdeluvio` from *Unsplash*: https://unsplash.com/photos/Mv9hjnEUHR4.
3. Rename the file `dog.jpg` and add it to the `assets` folder (create an `assets` in the root folder of your project).
4. We will also reuse the beach picture by Khachik Simonian that we used in the previous chapter: you can download this at https://unsplash.com/photos/nXOB-wh4Oyc. Rename it `beach.jpg`.
5. Don't forget to add the `assets` folder to your `pubspec.yaml` file:

```
assets:
 - assets/
```

6. You are also going to need a new file for this page. Create a `profile_screen.dart` file in the `lib` folder.
7. You can leave the app running while inputting the code to take advantage of hot reload.

How to do it...

You are now going to use a `Column` widget and then a `Stack` widget to place elements on the screen:

1. In the `profile_screen.dart` file, import the `material.dart` library.
2. Type `stless` to create a new stateless widget, and call it `ProfileScreen`:

```
import 'package:flutter/material.dart';
class ProfileScreen extends StatelessWidget {
  const ProfileScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

3. In the `main.dart` file, add the import to `profile_screen.dart` and remove the `MyHomePage` class. Then use the new `ProfileScreen` class as the home of `MyApp`:

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: ProfileScreen(),
    );
  }
}
```

4. In the `profile_screen.dart` file, add this shell code. This won't do anything yet, but it gives us three places to add the elements for this screen:

```
class ProfileScreen extends StatelessWidget {
  const ProfileScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Column(
        children: const [
          ProfileImage(),
          ProfileDetails(),
          ProfileActions(),
        ],
      ),
    );
  }
}
class ProfileImage extends StatelessWidget {
  const ProfileImage({super.key});
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
class ProfileDetails extends StatelessWidget {
  const ProfileDetails({super.key});
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
class ProfileActions extends StatelessWidget {
  const ProfileActions({super.key});
  @override
  Widget build(BuildContext context) {
    return Container();
  }}
```

5. Now, update the `build()` method in the `ProfileImage` widget to actually show the image of the dog:

```
  Widget build(BuildContext context) {
    return ClipOval(
```

```
      child: Image.asset(
        width: 200,
        height: 200,
        'assets/dog.jpg',
        fit: BoxFit.fitWidth,
      ),
    );
  }
```

6. Replace the `build()` method in the `ProfileDetails` widget with this code. This code also includes the `Column` widget's horizontal sibling, `Row`:

```
Widget build(BuildContext context) {
  return Padding(
    padding: const EdgeInsets.all(20.0),
    child: Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: [
        const Text(
          'Wolfram Barkovich',
          style: TextStyle(fontSize: 35, fontWeight:
          FontWeight.w600),
        ),
        _buildDetailsRow('Age', '4'),
        _buildDetailsRow('Status', 'Good Boy'),
      ],
    ),
  );
}
Widget _buildDetailsRow(String heading, String value) {
  return Row(
    children: [
      Text(
        '$heading: ',
        style: const TextStyle(fontWeight:
FontWeight.bold),
      ),
      Text(value),
    ],
  );
}
```

7. Let's add some fake controls that simulate interactions with our dog. Replace the `build()` method in the `ProfileActions` widget with this

code block:

```
Widget build(BuildContext context) {
  return Row(
    mainAxisAlignment: MainAxisAlignment.center,
    children: <Widget>[
      _buildIcon(Icons.restaurant, 'Feed'),
      _buildIcon(Icons.favorite, 'Pet'),
      _buildIcon(Icons.directions_walk, 'Walk'),
    ],
  );
}
Widget _buildIcon(IconData icon, String text) {
  return Padding(
    padding: const EdgeInsets.all(20.0),
    child: Column(
      children: <Widget>[
        Icon(icon, size: 40),
        Text(text),
      ],
    ),
  );
}
```

8. Run the app—your device screen should now look similar to this:

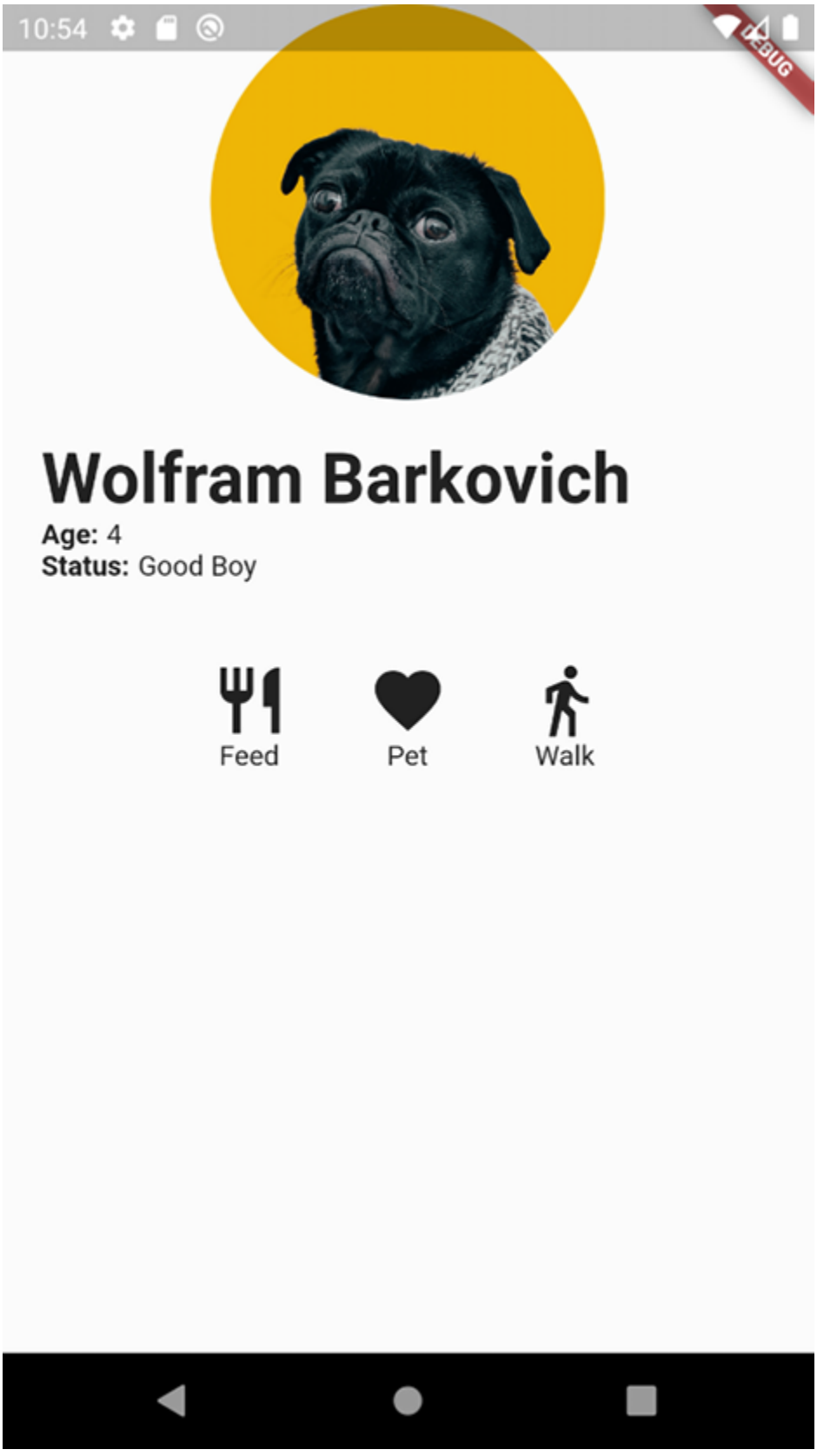
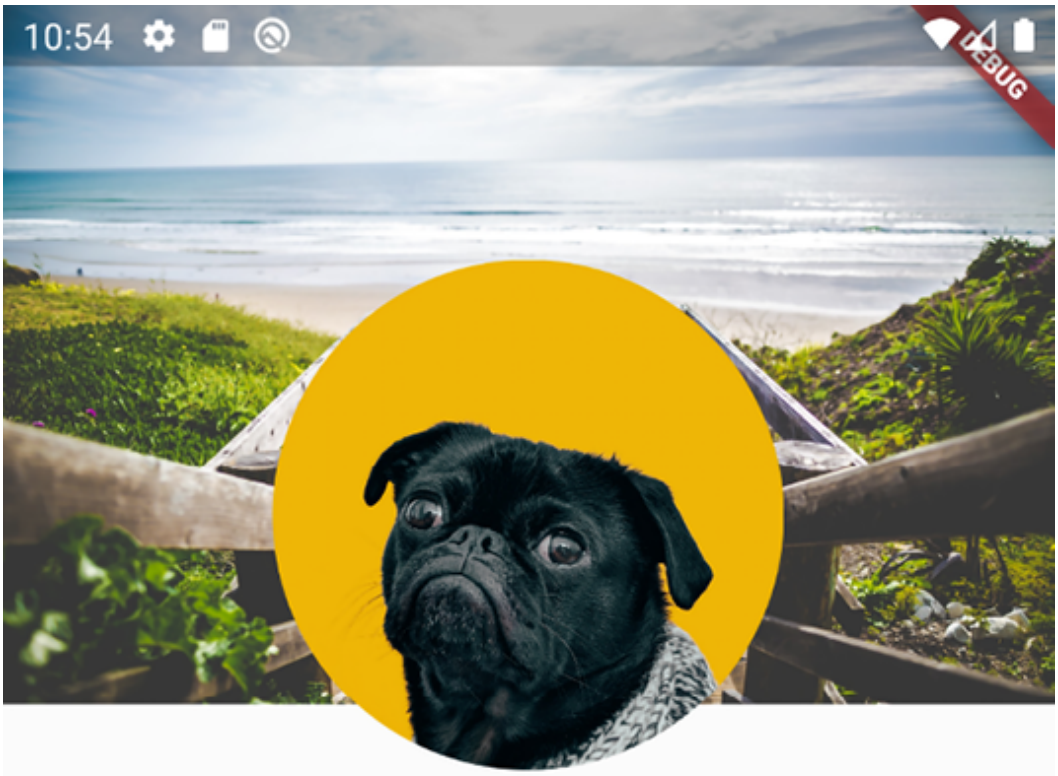

10:54
DEBUG
Wolfram Barkovich
Age: 4
Status: Good Boy
Feed
Pet
Walk

Figure 5.1: UI with a Column widget

9. In order to place widgets on top of each other, you can use a `Stack` widget. Replace the code in the `build` method in `ProfileScreen` to add a billboard behind the dog photo:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
      body: Stack(children: [
        Image.asset('assets/beach.jpg'),
        Transform.translate(
          offset: const Offset(0, 100),
          child: Column(
            children: const [
              ProfileImage(),
              ProfileDetails(),
              ProfileActions(),
            ],
          ),
        ),
      ]),
    );
}
```

The final screen will look like *Figure 5.2*:



Wolfram Barkovich

Age: 4
Status: Good Boy

Feed Pet Walk



Figure 5.2: UI with a Stack widget

How it works...

In Flutter, you use `Columns` and `Rows` to display lists of widgets. The only real difference between a `Row` widget and a `Column` widget is the axis in which they lay out their children. It's interesting that you can insert a `Row` widget into a `Column` widget, and vice versa.

There are also two properties on `Column` and `Row` widgets that can modify how Flutter lays out your widgets:

- `CrossAxisAlignment`
- `MainAxisAlignment`

These are abstractions on the *x* and *y* axes. They are also referring to different axes depending on whether you are using a `Row` widget or a `Column` widget, as shown in the following diagram:

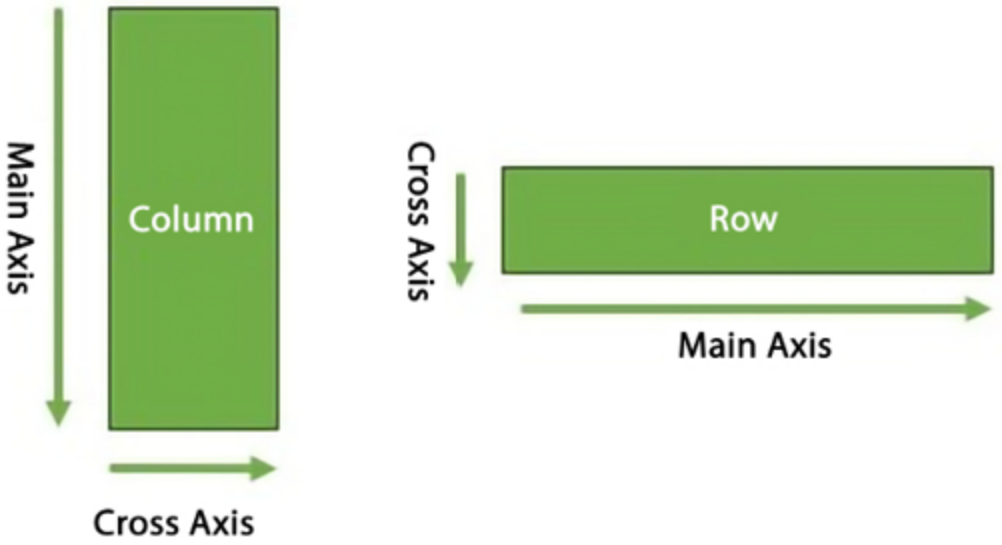


Figure 5.3: Main and Cross axes

With these properties, you can adjust whether the `Column` widget or `Row` widget is centered, evenly spaced, or aligned to the start or end of the widget. It will take a bit of experimentation to get the perfect look, but you are equipped with hot reload, so you can experiment at your will to see how they impact the layout.

The `Stack` widget is different. It expects you to provide your own layout widgets using `Align`, `Transform`, and `Positioned` widgets.

Note that it's now recommended to use `SizedBox` instead of `Container` in most cases, as SizedBox is a lighter widget.

From the Widget Inspector in your editor, you can activate the **Show Guidelines** option to help you fix layout issues, as shown in *Figure 5.4*:

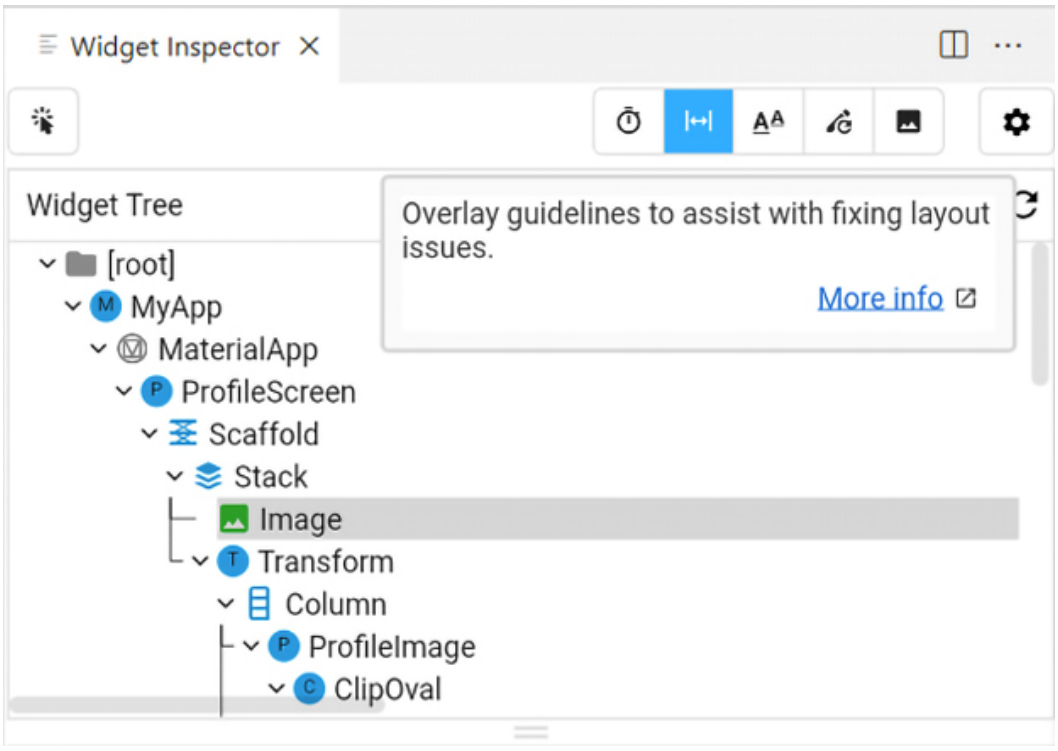
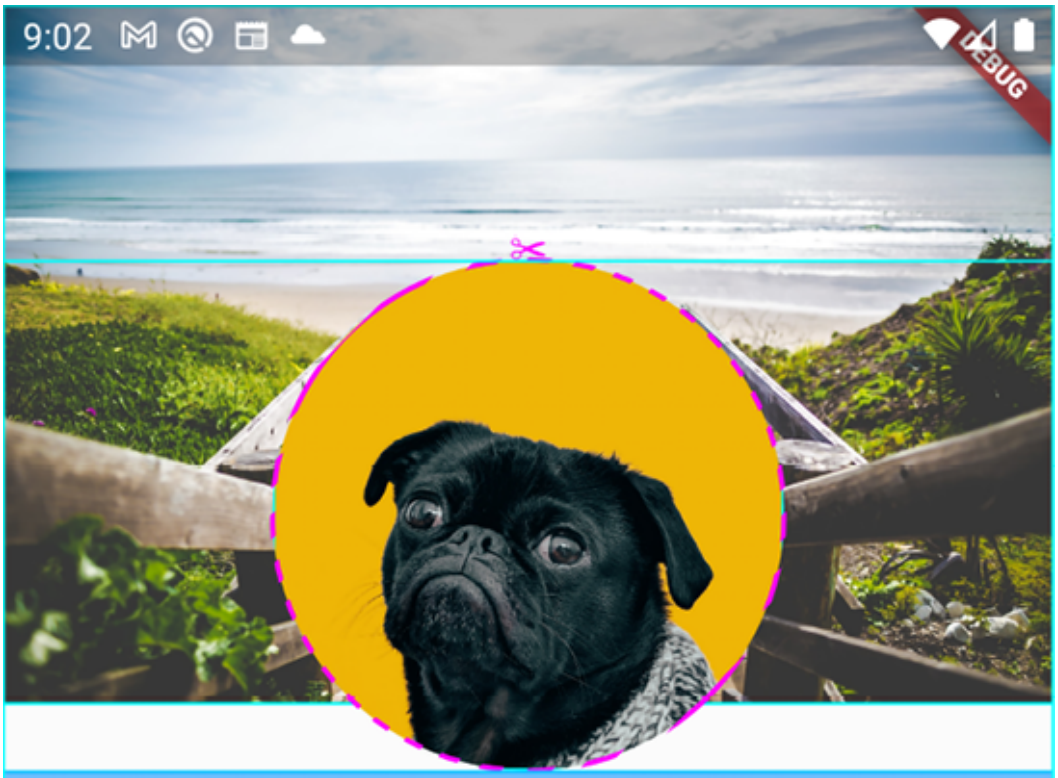


Figure 5.4: Activating the overlay guidelines from the Flutter Inspector

This feature will draw lines around your widgets so that you can see in more detail how they are being rendered, which can be useful to catch

layout errors, as shown in *Figure 5.5*:



Wolfram Barkovich

Age: 4

Status: Good Boy

Feed

Pet

Walk



Figure 5.5: Guidelines around widgets

Proportional spacing with the Flexible and Expanded widgets

Today, almost every device has a different height and width. Some devices also have a notch at the top of the screen that insets into the available screen space. In short, you cannot assume that your app will look the same on every screen——your user interfaces have to be flexible.

In this recipe, we will demonstrate two ways to develop proportional widgets: `Flexible` and `Expanded` widgets.

Getting ready

You should have completed the project in the previous recipe before starting this one.

Create a new Dart file called `flex_screen.dart` and create the requisite `StatelessWidget` subclass called `FlexScreen`. Also, just like the last recipe, replace the `home` property in `main.dart` with `FlexScreen`.

How to do it...

Before we can show off `Expanded` and `Flexible`, we need to create a simple helper widget:

1. Create a new file, called `labeled_container.dart`, and import `material.dart`.
2. Add the following code in the `labeled_container.dart` file:

```
import 'package:flutter/material.dart';
class LabeledContainer extends StatelessWidget {
  final String text;
  final double? width;
  final double? height;
  final Color? color;
  final Color? textColor;
  const LabeledContainer({
    required this.text,
    this.width,
    this.height = double.infinity,
    this.color,
    this.textColor,
    super.key,
  }) ;
  @override
  Widget build(BuildContext context) {
    return Container(
      width: width,
      height: height,
      color: color,
      alignment: Alignment.center,
      child: Text(
        text,
        style: TextStyle(
          color: textColor,
          fontSize: 20,
        ),
      ),
    );
  }
}
```

3. In the `flex_screen.dart` file, add this code:

```
import 'package:flutter/material.dart';
import 'label_container.dart';
class FlexScreen extends StatelessWidget {
  const FlexScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Flexible and Expanded'),
      ),
      body: Column(
```

```
        crossAxisAlignment: CrossAxisAlignment.start,
        children: <Widget>[
          ..._header(context, 'Expanded'),
          const DemoExpanded(),
          ..._header(context, 'Flexible'),
          const DemoFlexible(),
        ],
      ),
    );
  }
  Iterable<Widget> _header(BuildContext context, String
text) {
    return [
      const SizedBox(height: 20),
      Text(
        text,
        style:
Theme.of(context).textTheme.headlineSmall,
      ),
    ];
  }
}
class DemoExpanded extends StatelessWidget {
  const DemoExpanded({super.key});
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
class DemoFlexible extends StatelessWidget {
  const DemoFlexible({super.key});
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
class DemoFooter extends StatelessWidget {
  const DemoFooter({super.key});
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

4. Now, fill in the `build()` method of the `DemoExpanded` widget:

```
Widget build (BuildContext context) {
  return SizedBox(
    height: 100,
    child: Row(
      children:const [
        LabeledContainer(
          width: 100,
          color: Colors.green,
          text: '100',
        ),
        Expanded(
          child: LabeledContainer(
            color: Colors.purple,
            text: 'The Remainder',
            textColor: Colors.white,
          ),
        ),
        LabeledContainer(
          width: 40,
          color: Colors.green,
          text: '40',
        )
      ],
    ),
  );
}
```

5. Fill in the `build()` method of the `DemoFlexible` widget. Don't forget to hot reload while writing this code:

```
Widget build(BuildContext context) {
  return SizedBox(
    height: 100,
    child: Row(
      children: const[
        Flexible(
          flex: 1,
          child: LabeledContainer(
            color: Colors.orange,
            text: '25%',
          ),
        ),
        Flexible(
          flex: 1,
          child: LabeledContainer(
            color: Colors.deepOrange,
```

```
            text: '25%',
          ),
        ),
        Flexible(
          flex: 2,
          child: LabeledContainer(
            color: Colors.blue,
            text: '50%',
          ),
        )
      ],
    ),
  );
}
```

6. Update the `build()` method in the `DemoFooter` widget to create a rounded banner:

```
Widget build(BuildContext context) {
  return Center(
      child: Container(
        decoration: BoxDecoration(
          color: Colors.yellow,
          borderRadius: BorderRadius.circular(40),
        ),
        padding: const EdgeInsets.symmetric(
          vertical: 15.0,
          horizontal: 30,
        ),
        child: Text(
          'Pinned to the Bottom',
          style: Theme.of(context).textTheme.titleSmall,
        ),
      ),
    );
}
```

7. In order to push this widget to the bottom of the screen, we have to add an `Expanded` widget to the root `Column` widget to eat up all the remaining space. Insert this line in the main `build` method after calling the `DemoFlexible` widget:

```
DemoFlexible(context),
const Expanded(
 child: Spacer(),
```

```
),
const DemoFooter(context)
```

8. When running the app, you should see a screen similar to *Figure 5.6*:

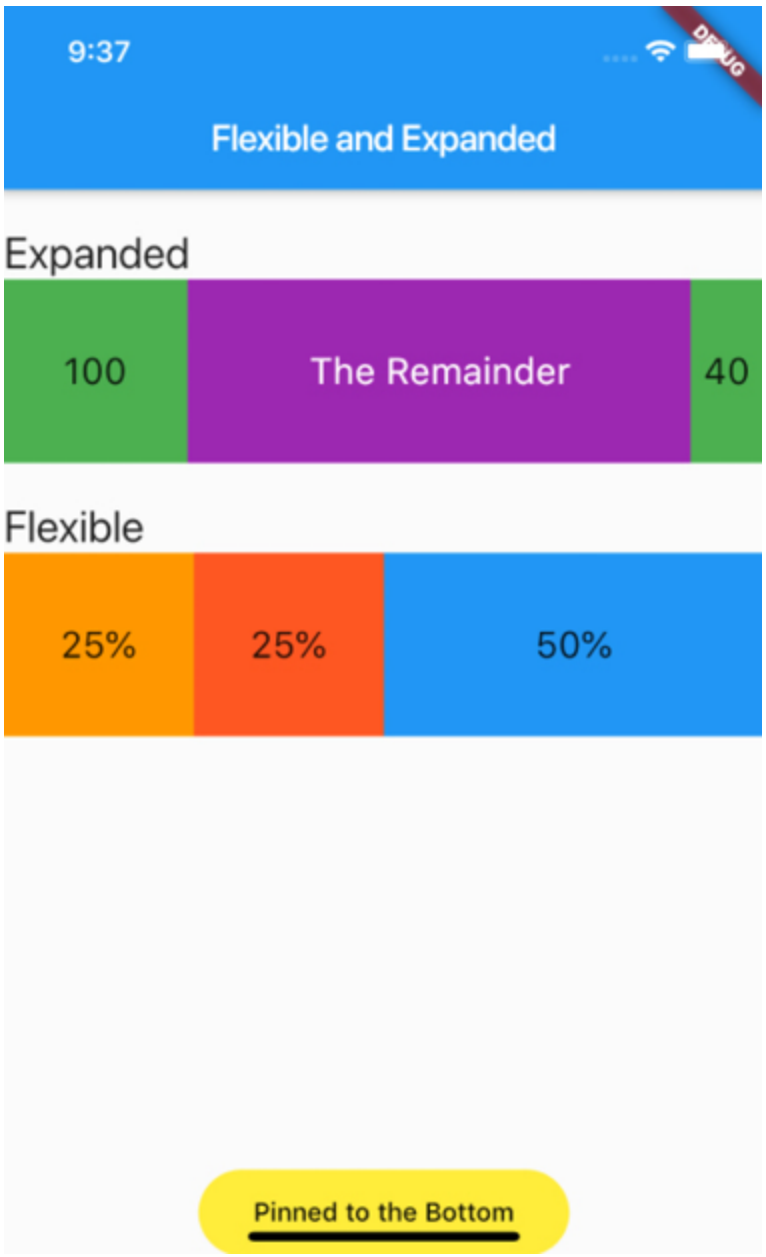


Figure 5.6: Flexible and Expanded widgets

9. On some devices, the header or footer covers up some software or hardware features (such as the notch or the Home indicator). To fix this, just put a `SafeArea` widget in the `Scaffold` :

```
return Scaffold(
      appBar: AppBar(
        title: const Text('Flexible and Expanded'),
      ),
      body: SafeArea(
        child: Column(
```

10. Perform a hot reload and everything should now render correctly, as shown here:

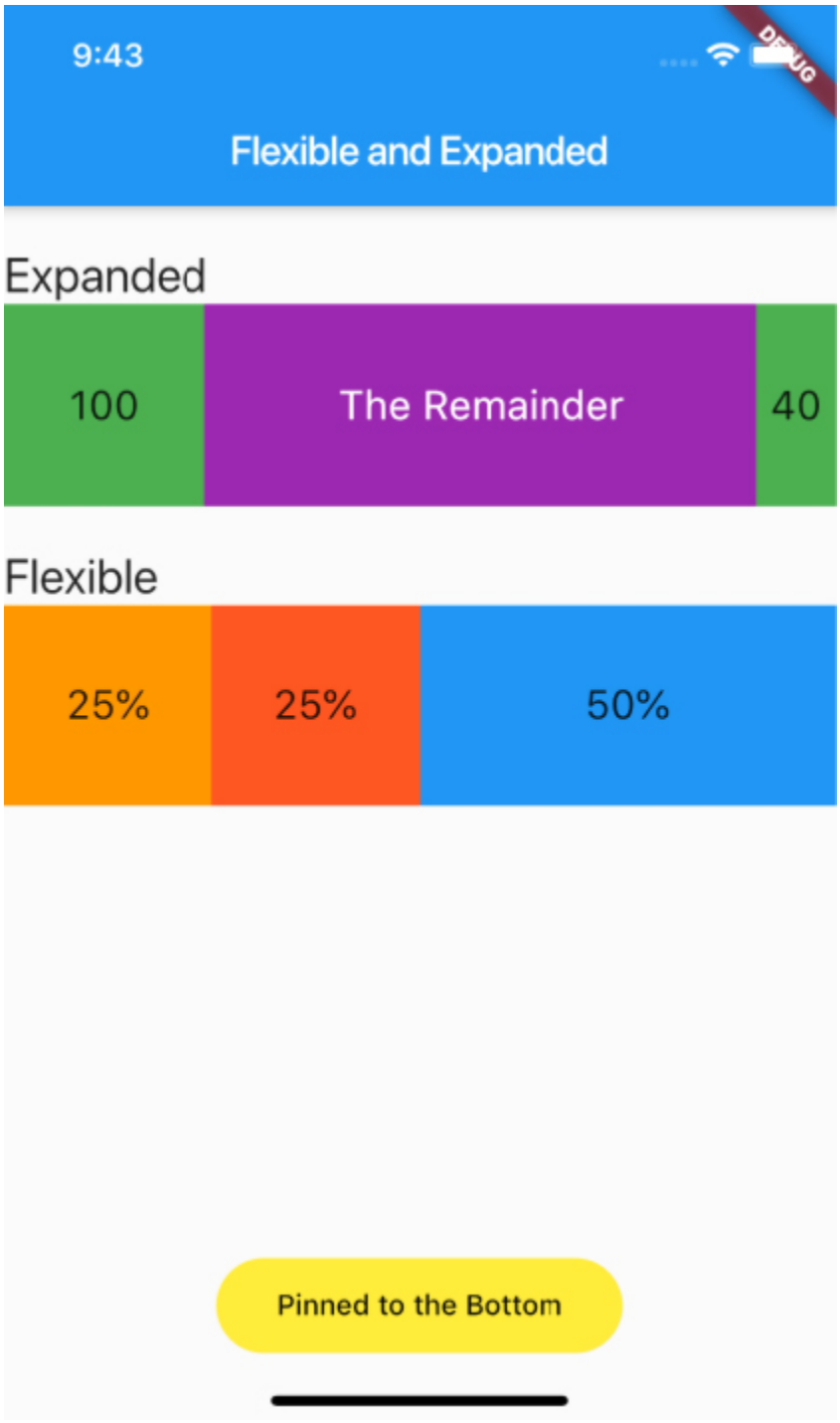


Figure 5.7: SafeArea

How it works...

When you break them down, the `Flexible` and `Expanded` widgets are quite beautiful in their simplicity.

The `Expanded` widget will take up all the remaining unconstrained space from a `Row` or a `Column`. In the preceding example, we placed three containers (`SizeBox`) in the first row. The container was given a width of 100 device-independent pixels.

Note

Unlike pixels, which depend on the specific screen where you are running your apps, **device-independent pixels** remain constant regardless of the screen resolution. Different screens can have vastly different **pixel densities**, meaning that an object or text that looks a certain size on one screen may appear much smaller or larger on another.

For example, let's say we have a 100-pixel square that we want to display on two different devices with different pixel densities.

The first device has a pixel density of 100 **pixels per inch** (**PPI**), and the second has a pixel density of 200 PPI.

On the first device, the 100-pixel square would take an area of 1 inch x 1 inch, because there are 100 pixels per inch.

On the second device, the 100-pixel square would only take up an area of 0.5 inches x 0.5 inches, because there are 200 pixels per inch. So the square would appear to be much smaller.

To avoid this inconsistency, you use **device-independent pixels** (**DIPs**) to specify the size of the square. In our example, the 100 DIPs will be converted into actual pixel values based on the pixel density of the device. On the first device, 100 DIPs will take 100 pixels (1 inch), and on the second, 100 DIPs will take 200 pixels.

In this way, the square would appear to be about the same size on both devices, despite the different pixel densities.

The last container was given a width of 40 DIPs. The middle container is wrapped in an `Expanded` widget, so it consumes **all the remaining space in the row**. These explicit values are referred to as **constrained spacing**.

The width calculation for the middle container would look like this:

Width = ParentWidth – 100 - 40

These types of widgets can be very useful when you need to push widgets to the other edges of the screen, such as when you pushed the footer banner to the bottom of the screen:

```
const Expanded(
 child: Spacer(),
),
```

It is very common to create an `Expanded` widget with a `Spacer` that will simply consume all the available remaining space in a `Row` or `Column`.

The `Flexible` widget behaves similarly to the `Expanded` widget, but it does not force the child to fill the available space.

When Flutter lays out `Flexible` widgets, it takes the sum of the `flex` values to calculate the percentage of the proportional space that needs to be applied to each widget. In our example, we gave the first two flexible widgets a value of 1 and the second one a value of 2. The sum of our flex values is 4. This means that the first two widgets will get 1/4 of the available width, and the last widget will get 1/2 of the available width.

Pro Tip

It's usually a good idea to keep your `flex` values small so that you don't have to do any complicated arithmetic to figure out how much space your widget is going to take.

Now, just for fun, let's look at the code for how `Expanded` widgets are implemented:

```
class Expanded extends Flexible {
  const Expanded({
    super.key,
    super.flex,
    required super.child,
  }) : super(fit: FlexFit.tight);
}
```

That's it! An `Expanded` widget is actually just a `Flexible` widget with a `flex` value of 1. We could, in theory, replace all references to `Expanded` with `Flexible` and the app would be unchanged. Again, the difference is that `Expanded` will always take up all the available space.

Note

`Flexible` and `Expanded` widgets have to be a child of the `Flex` subclass; otherwise, you will get an error. This means they can be a child of a `Column` or a `Row` widget. But if you place one of these widgets as a child of a `Container` widget, expect to see a red error screen. If you want to know more about handling these kinds of errors, skip ahead to *Chapter 7*, *Basic State Management*, which is dedicated to solving what happens when code fails.

See also

Box constraints are a very important topic when dealing with space in Flutter. Have a look at https://docs.flutter.dev/development/ui/layout/box-constraints for more information on these.

Drawing shapes with CustomPaint

So far, we've been limiting ourselves to very boxy shapes. `Rows`, `Columns`, `Stacks`, `SizedBoxes` and `Containers` are just boxes. Boxes will cover the majority of UIs that you would want to create, but sometimes you may need to break free from the tyranny of the quadrilateral.

Enter `CustomPaint`. Flutter has a full-featured vector drawing engine that allows you to draw pretty much whatever you want. You can then reuse these shapes in your widget tree to make your app stand out from the crowd.

In this recipe, we will be creating a star rating widget to see what `CustomPaint` is capable of.

Getting ready

This recipe will update the `ProfileScreen` widget that was created in the *Placing widgets one after another* recipe in this chapter. Make sure that in `main.dart`, `ProfileScreen()` is set in the `home` property.

Also, create a new file called `star.dart` in the `lib` directory and set up a `StatelessWidget` subclass called `Star`.

How to do it...

We need to start this recipe by first creating a shell that will hold the `CustomPainter` subclass:

1. Update the `Star` class to use a `CustomPaint` widget:

```
import 'package:flutter/material.dart';
class Star extends StatelessWidget {
  final Color color;
  final double size;
    const Star({required this.color, required
this.size, super.key});
  @override
  Widget build(BuildContext context) {
    return SizedBox(
      width: size,
      height: size,
      child: CustomPaint(
        painter: _StarPainter(color),
      ),
    );
```

```
  }
}
```

2. This code will throw an error because the `_StarPainter` class doesn't exist yet. Create it now and make sure to override the required methods of `CustomPainter`:

```
class _StarPainter extends CustomPainter {
 final Color color;
 _StarPainter(this.color);
 @override
 void paint(Canvas canvas, Size size) {
 }
 @override
 bool shouldRepaint(CustomPainter oldDelegate) {
 return false;
 }
}
```

3. Update the `paint` method to include this code to draw a five-pointed star:

```
@override
void paint(Canvas canvas, Size size) {
 final paint = Paint()..color = color;
 final path = Path()
      ..moveTo(size.width * 0.5, 0)
      ..lineTo(size.width * 0.618, size.height * 0.382)
      ..lineTo(size.width, size.height * 0.382)
      ..lineTo(size.width * 0.691, size.height * 0.618)
      ..lineTo(size.width * 0.809, size.height)
      ..lineTo(size.width * 0.5, size.height * 0.7639)
      ..lineTo(size.width * 0.191, size.height)
      ..lineTo(size.width * 0.309, size.height * 0.618)
      ..lineTo(size.width * 0.309, size.height * 0.618)
      ..lineTo(0, size.height * 0.382)
      ..lineTo(size.width * 0.382, size.height * 0.382)
      ..close();
 canvas.drawPath(path, paint);
}
```

4. Create another class that uses these stars to create a rating. For the sake of brevity, we can include this new widget in the same file, as shown here:

```
class StarRating extends StatelessWidget {
  final Color color;
  final int value;
  final double starSize;
  const StarRating({
    super.key;
    required this.value,
    this.color = Colors.deepOrange,
    this.starSize = 25,
  });
  @override
  Widget build(BuildContext context) {
    return Row(
      children: List.generate(
        value,
        (_) => Padding(
          padding: const EdgeInsets.all(2.0),
          child: Star(
            color: color,
            size: starSize,
          ),
        ),
      ),
    );
  }
}
```

5. That should wrap up the stars. In `profile_screen.dart`, update the `build` method in `ProfileDetails` to add the `StarRating` widget:

```
const Text(
  'Wolfram Barkovich',
  style: TextStyle(fontSize: 35, fontWeight:
FontWeight.w600),
),
const StarRating(
 value: 5,
),
_buildDetailsRow('Age', '4'),
```

The final app, as shown, should now have five stars under the dog's name:

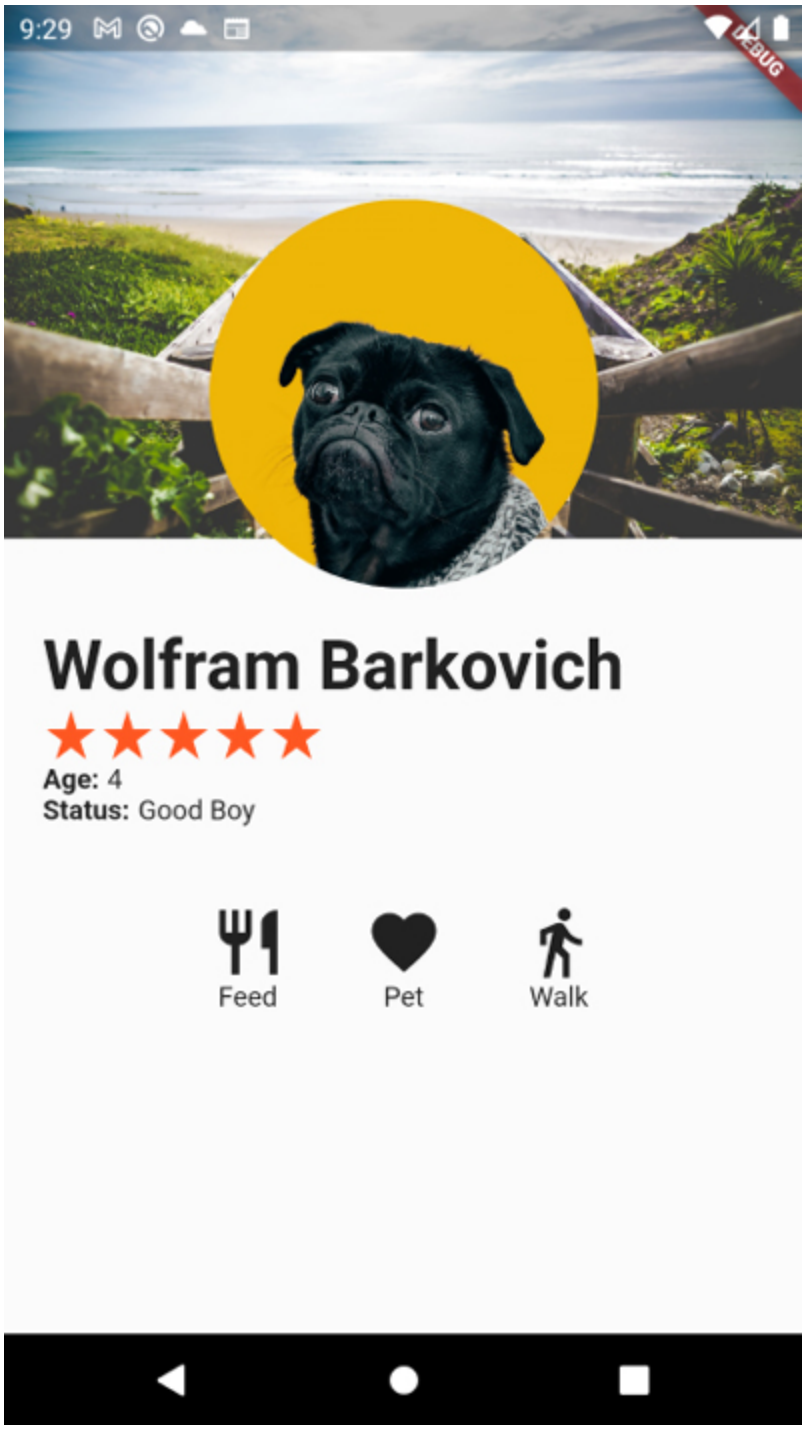


Figure 5.8: CustomPaint example

Pro Tip

To see the full-color version of Figure 5.8, and all the other images in this book, download the PDF file here: https://packt.link/WE615.

How it works...

This recipe is composed of quite a few small pieces that work together to create the custom star. Let's break this down into small pieces. Here's the first part of the code:

```
const Star({required this.color, required this.size,
super.key});
```

This custom constructor is asking for a color and a size, which will then be passed down to the painter.

Note the `required` keyword, which makes sure the caller provides values for all "required" parameters, otherwise the app won't even compile.

The `Star` widget doesn't impose any size restrictions on the star. Let's look at the next part:

```
return SizedBox(
  width: size,
  height: size,
  child: CustomPaint(
    painter: _StarPainter(color),
  ),
);
```

This `build` method returns a `SizedBox` that is constrained by this `size` property and then uses `CustomPaint` as its child.

The real work is now done in the `CustomPainter` subclass, which is not a widget. `CustomPainter` is an *abstract* class, meaning you cannot instantiate it directly, as it can only be used through inheritance. There are two methods that you have to override in the subclass: `paint` and `shouldRepaint`.

The second method, `shouldRepaint`, is there for optimization purposes. Flutter will call this method whenever it needs to determine whether a `CustomPaint` widget should be repainted. This makes the framework determine whether the current state of the widget has changed since the last time it was painted and whether a repaint is necessary.

The `paint` method is where the shape is actually drawn. You are given a `Canvas` object. From there, you can use the `Path` **application programming interface** (**API**) to draw the star as a vector shape. Since we want the star to look good at any size, instead of typing explicit coordinates for each vector point, we performed a simple calculation to draw on the percentage of the canvas instead of absolute values.

That is why we wrote this:

```
..lineTo(size.width * 0.309, size.height * 0.618);
```

We wrote the preceding code instead of the following:

```
..lineTo(20, 15);
```

If you only provide absolute coordinates, the shape would only be usable at a specific size. Now, this star can be infinitely large or infinitely small, and it will still look good.

Determining these coordinates can be virtually impossible without the aid of a graphics program. These numbers are not just guessed out of thin air. You will probably want to use a program such as Sketch, Figma, or Adobe Illustrator to draw your image first and then transcribe it to code. There are even some tools that will automatically generate drawing code that you can copy into your project.

Note

If you are familiar with other vector graphics engines such as **Scalable Vector Graphics** (**SVG**) or Canvas2D on the web, this API should seem familiar. In fact, both Flutter and Google Chrome are powered by the same C++ drawing framework: **Skia**. These drawing commands that we

write in Dart are eventually passed down to Skia, which in turn will use the **graphics processing unit** (**GPU**) to draw your image.

Soon, this will change (or, depending on when you are reading this, may have already changed), as the Flutter team is working on a new rendering engine called Impeller. For more information, have a look here: https://github.com/flutter/flutter/wiki/Impeller.

Finally, after we have our shape completed, we commit to the canvas with the `drawPath` method:

```
canvas.drawPath(path, paint);
```

This will take the vector shape and rasterize it (convert it to pixels) using a `Paint` object to describe how it should be filled.

It may seem like a lot of work just to draw a star, and that would not be an entirely incorrect assumption. If you can accomplish your desired look using a simpler API such as the `BoxDecoration` API, then you don't need a `CustomPaint` widget. But once you bump up against the limits of the higher-level APIs, there is more flexibility (and more complexity) waiting for you with a `CustomPainter`.

There's more...

There is one more Dart language feature that we used in this recipe that we should quickly explain:

```
List.generate(
  value,
  (_) => Padding(...)
),
```

This is the expected syntax for the generator closure:

```
E generator(int index)
```

Every time this closure is called, the framework will be passing an index value that can be used to build the element. However, in this case, the index is not important. It's so unimportant that we don't need it at all and will never reference it in the generator closure.

In these cases, you can replace the name of the index with an underscore, and that will mean we are ignoring this value, while still complying with the required API.

We could explicitly reference the index value with this code:

```
(index) => Padding(...)
```

But it's usually considered a faux pas to declare variables and not use them. By replacing the `index` symbol with an underscore symbol (_), you can sidestep this issue.

See also

For more information about the `CustomPaint` class, see https://api.flutter.dev/flutter/widgets/CustomPaint-class.html.

The Flutter `Canvas` class is a powerful tool: see the official guide at https://api.flutter.dev/flutter/dart-ui/Canvas-class.html.

Nesting complex widget trees

The speed of making changes significantly impacts the effectiveness of any given platform. Hot reload helps with this exponentially. Being able to quickly edit properties on a widget, hit **Save**, and almost instantly see your results without losing state is wonderful. This feature enables you to experiment.

It also allows you to make mistakes and quickly undo them without wasting precious compile time.

But there is one area where Flutter's nested syntax can slow your progress. Throughout this chapter, we have used the phrase *wrap in a widget* frequently. This implies that you are going to take an existing widget and make it the child of a new widget, essentially pushing it one level down the tree. This can be error-prone if done manually, but thankfully, the Flutter tools help you manipulate your widget trees with speed and effectiveness.

In this recipe, you are going to explore the IDE tools that will allow you to build deep trees without worrying about mismatching your parentheses.

Getting ready

For this recipe, you should have completed the first recipe in this chapter: *Placing widgets one after another* (and/or any other recipes in this chapter).

Let's create a new file called `deep_tree.dart`, import the `material.dart` library, and make a `StatelessWidget` subclass called `DeepTree`. Call an instance of this widget in the `home` property of `MaterialApp`, in the `main.dart` file.

How to do it...

Let's start by running the app. You should see a blank canvas to play with:

1. Add some text to the `DeepTree` class:

```
class DeepTree extends StatelessWidget {
  const DeepTree({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
         body: Text('Its all widgets!'),
    );
  }
}
```

2. Compile and run the code, ignoring the `const` warning message. We'll fix this throughout this recipe. You'll get the following result:

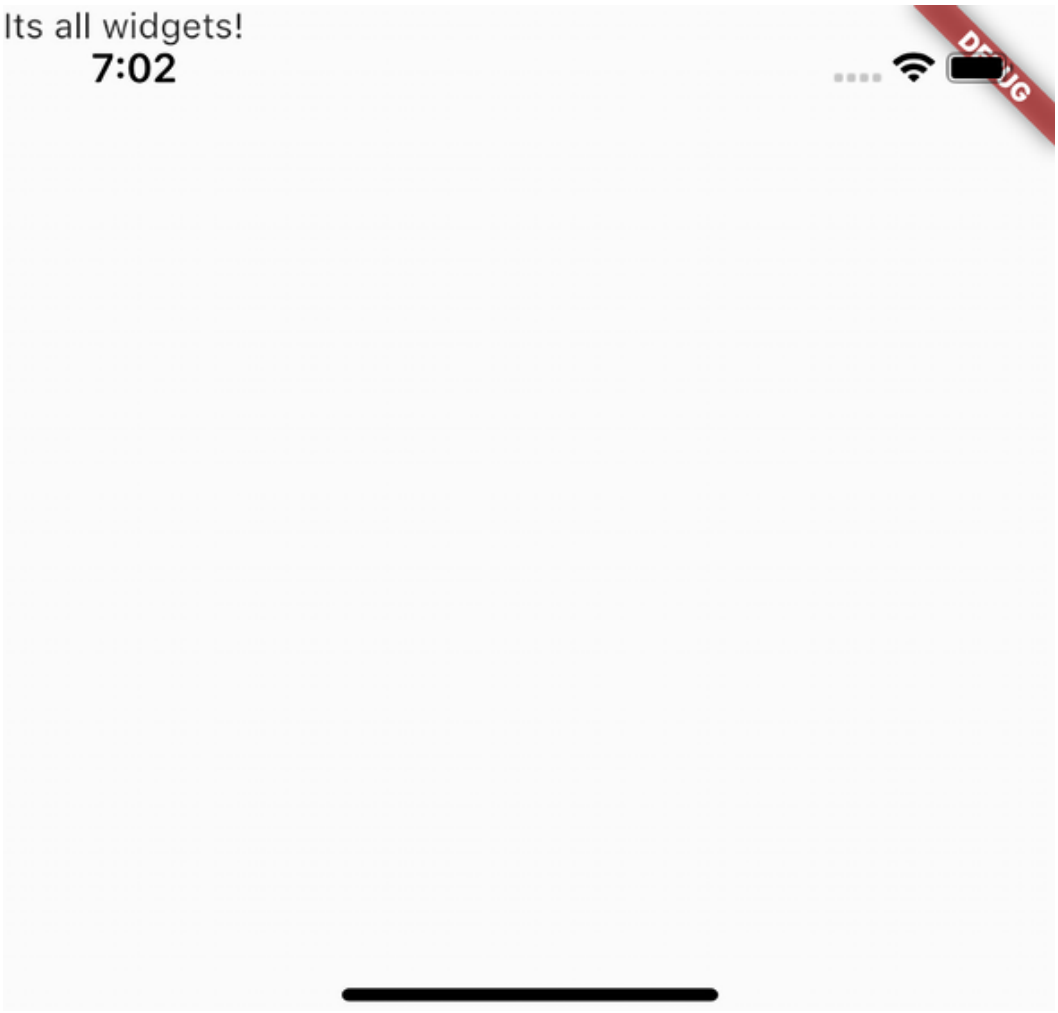


Figure 5.9: Text in a scaffold

We can see that, in the preceding screenshot, the text is in the top corner and it can barely be seen. How about we wrap it in a `Center` widget?

1. Move your cursor over the `Text` constructor and type the following:

> - In Android Studio: *Ctrl* + *Enter* (or *Command* + *Enter* on a Mac).
> - In Visual Studio Code (VS Code): *Ctrl* + . (or *Command* + . on a Mac) to bring up the intentions dialog. Then, select **Center widget / Wrap with Center**.

2. Perform a hot reload, and the text will move to the center of the screen:

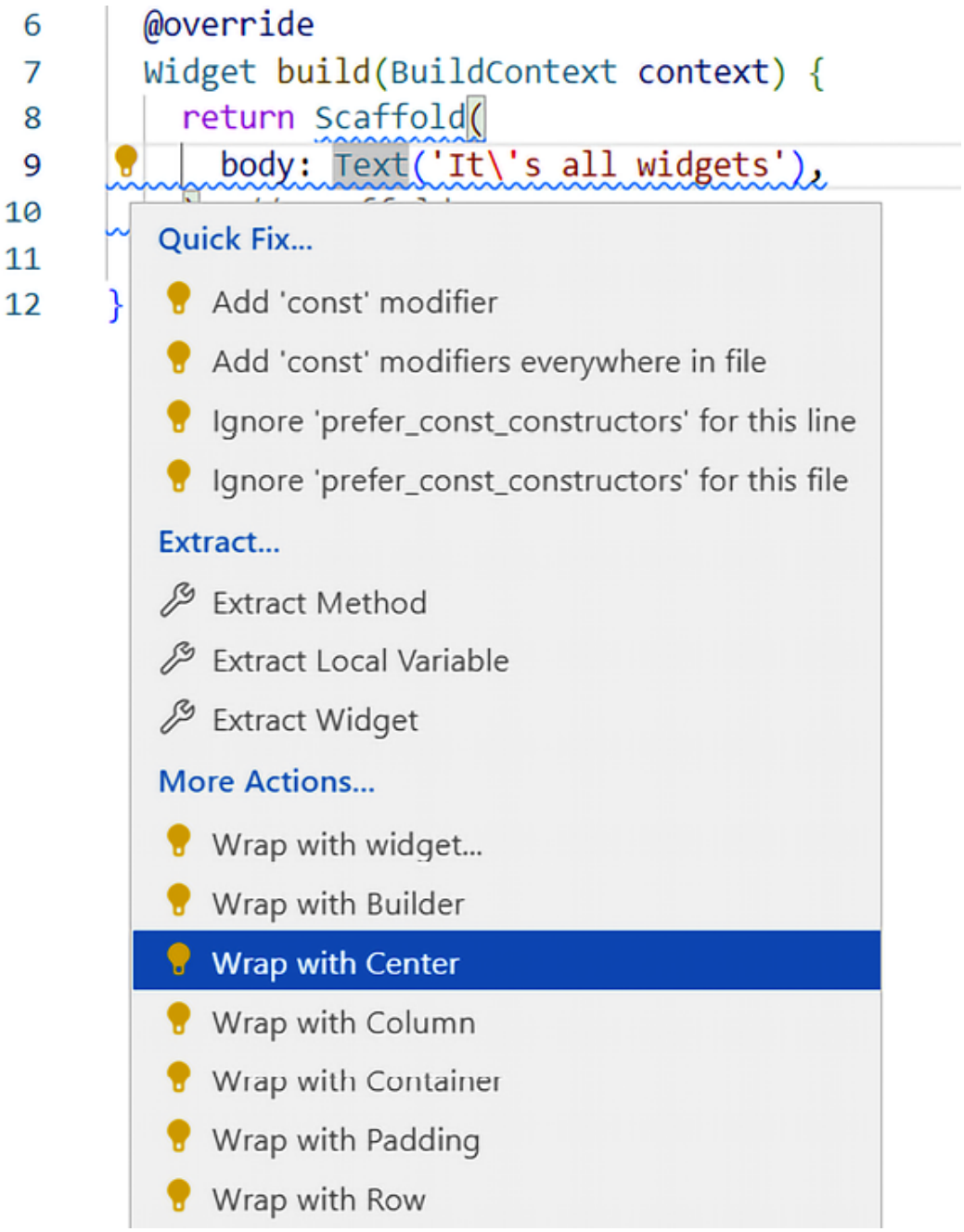


Figure 5.10: VS Code intentions dialog (quick fix)

3. That looks slightly better, but how about we change that single widget to a `Column` widget and add some more text? Once again,

move your cursor over the `Text` constructor and type the editor shortcut to get the helper methods.

> **Pro Tip**
>
> You will be using these shortcuts a lot during this recipe: to bring in the intentions methods, you use *Ctrl*/*Command* + . in VS Code and *Ctrl*/*Command* + *Enter* in Android Studio. This time, select **Wrap with Column**.

4. You can now remove the `Center` widget. Put your cursor on the `Center` widget and call the intentions methods, then select **Remove this widget**.
5. Add two more `Text` widgets, as shown here:

```
class DeepTree extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
          child: Column(
        children: [
          Text('Its all widgets!'),
          Text('Flutter is amazing.'),
          Text('Let\'s find out how deep the rabbit
hole goes.'),
        ],
      )),
    );
  }
}
```

6. Uh oh! The text has moved to the top again. We can fix that by setting the main axis on the column back to the center:

```
Column(
  mainAxisAlignment: MainAxisAlignment.center,
  children: [
    Text('Its all widgets!'),
```

7. Keep building up the tree by wrapping the middle `Text` widget with a `Row` widget and adding a `FlutterLogo` widget:

```
Text('Its all widgets!'),
Row(
 mainAxisAlignment: MainAxisAlignment.center,
 children: const [
 FlutterLogo(),
 Text('Flutter is amazing.'),
 ],
),
Text('Let\'s find out how deep the rabbit hole goes.'),
```

That middle row might look better if it were the first row. We *could* just cut it and place it first in the `Column` widget's list, but that could lead to errors if we didn't copy the whole row. Instead, let's see what the Flutter tools offer:

1. Bring up the intentions dialog and select **Move widget up**:

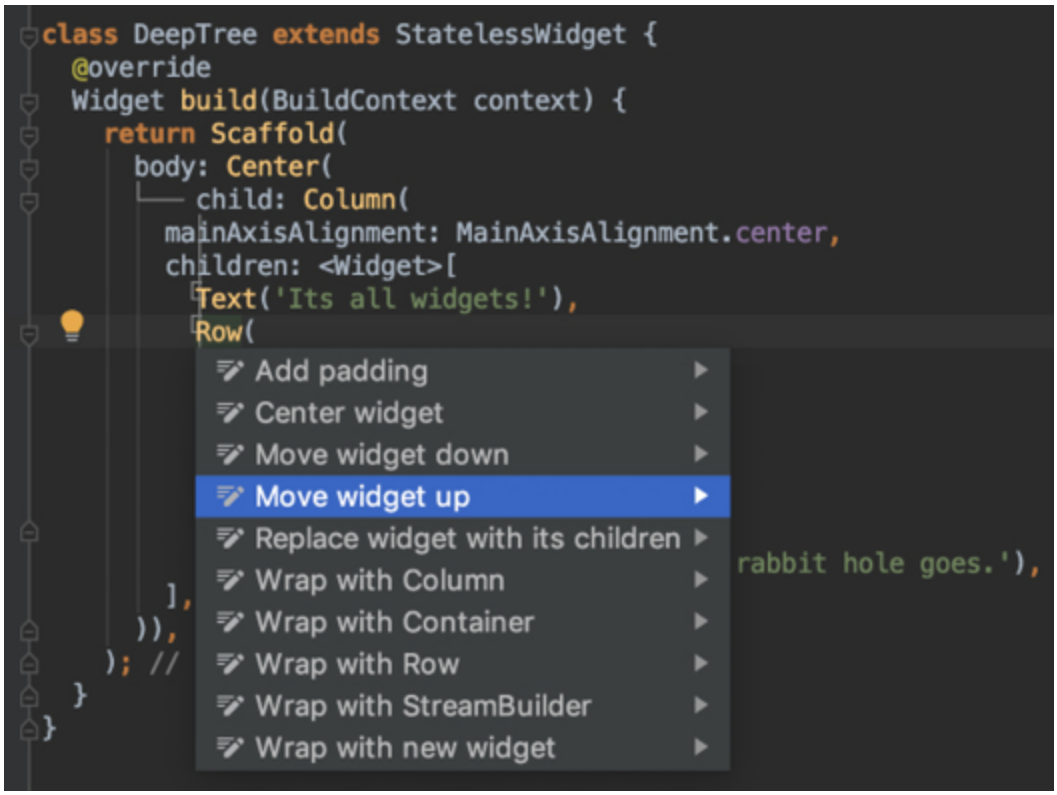


Figure 5.11: Android Studio quick fix dialog

2. Add a purple container between the row and the `'It's all widgets!' Text` widget:

```
Row(
  mainAxisAlignment: MainAxisAlignment.center,
  children: const [
    FlutterLogo(),
    Text('Flutter is amazing.'),
  ],
),
Expanded(
  child: Container(
    color: Colors.purple,
  ),
),
Text('It's all widgets!'),
```

3. This will push everything to the edges of the screen, making the text almost impossible to read again. We can use the `SafeArea` widget to bring back some legibility.
4. Bring up the intentions methods and select **Wrap with (a new) widget**. A placeholder will appear.
5. Replace the word `widget` with the `SafeArea` widget:

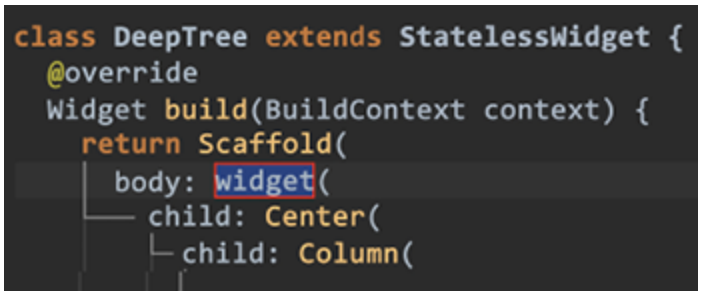


Figure 5.12: Wrap with widget command

How it works...

Mastering this tool is critical to getting into a flow state when developing your UIs. A common criticism of Flutter is that deeply nested widget trees

can make code very hard to edit. This is the reason why tools such as the quick fix exist.

As your Flutter knowledge grows, you should strive to become more reliant on these tools to improve your efficiency and accuracy. If you want, try running through this recipe again, but this time, don't use the helper tools—just edit the code manually. Notice how hard it is, and how easy it is to mismatch your parentheses, breaking the whole tree?

For the curious, the quick-fix dialog feature is powered by a separate program called the Dart Analysis Server. This is a background process that isn't tightly coupled to your IDE. When you open the intentions dialog, your IDE sends the token you are currently highlighting to a language analysis server, which returns the appropriate options. This server is constantly running while you are editing your code. It checks for syntax errors, helps with code autocomplete, and provides syntax highlighting and other features.

It is unlikely that you will ever interact with the Dart Analysis Server directly, since that's the IDE's job, not yours. But you can rest peacefully knowing that it has your back.

See also

Dart Analysis Server, the engine that is powering this feature, is described at https://github.com/dart-lang/sdk/tree/master/pkg/analysis_server.

Refactoring widget trees to improve legibility

There is a delightfully sardonic anti-pattern in coding known as the **pyramid of doom**. This pattern describes code that is excessively nested (such as 10+ nested `if` statements and control flow loops). You end up getting code that, when you look at it from a distance, resembles a pyramid.

Pyramid-like code is highly bug-prone. It is hard to read and, more importantly, hard to maintain.

Widget trees are not immune to this deadly pyramid; in fact, quite the opposite. In this chapter, we've tried to keep our widget trees fairly shallow, but none of the examples so far are really indicative of production code—they are simplified scenarios to explain the fundamentals of Flutter. The tree only grows deeper from here.

To fight the pyramid of doom, we're going to use a weapon known as **refactoring**. This is a process of taking code that is not written ideally and updating the code without changing its functionality. We can take our *n*-layer deep widget trees and refactor them toward something easier to read and maintain. This will give you the added benefit of improving your app's performance, as there will be fewer widgets layers to repaint.

In this recipe, we're going to take a large and complicated widget tree and refactor it toward something cleaner.

Getting ready

Download the sample code that comes with this book at https://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_05 and look for the `e_commerce_screen_before.dart` file. Copy that file and the `woman_shopping.jpg` and `textiles.jpg` images to your `assets` folder in the project. Update `main.dart` to set the `home` property to `ECommerceScreen()`.

How to do it...

This recipe is going to focus on taking some existing code and updating it to something more manageable. Open `e_commerce_screen_before.dart` and notice that the whole class is one single unbroken `build` method.

In order to reduce the number of nested widgets in this code, the first thing we're going to do with this file is divide it up using a refactoring tool called the **extract method**:

1. Move your cursor over `AppBar`, right-click, and select **Refactor | Extract Method** on VS Code or **Refactor | Extract | Extract Method** on Android Studio to open the **Extract…** dialog. Call the method `_buildAppBar` and press *Enter*:

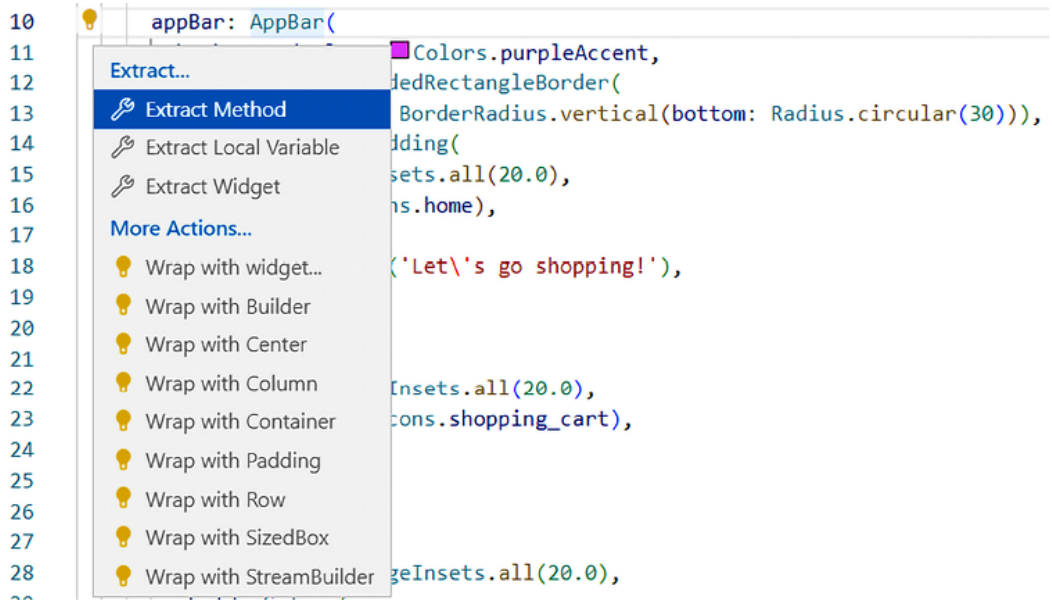


Figure 5.12: The Extract Method quick-fix option

2. Perform an extract of the first `Row` widget in the `Column` widget using the same technique. Call this method `_buildToggleBar()`.
3. The next step is going to be a bit more tricky. Collapse the code for the two next rows by tapping the collapse icon just to the right of the line numbers:



Figure 5.13: The Collapse icon

4. With both rows collapsed, highlight the three `SizedBox` instances and the two rows and call the intentions dialog. Select **Wrap with Column**:

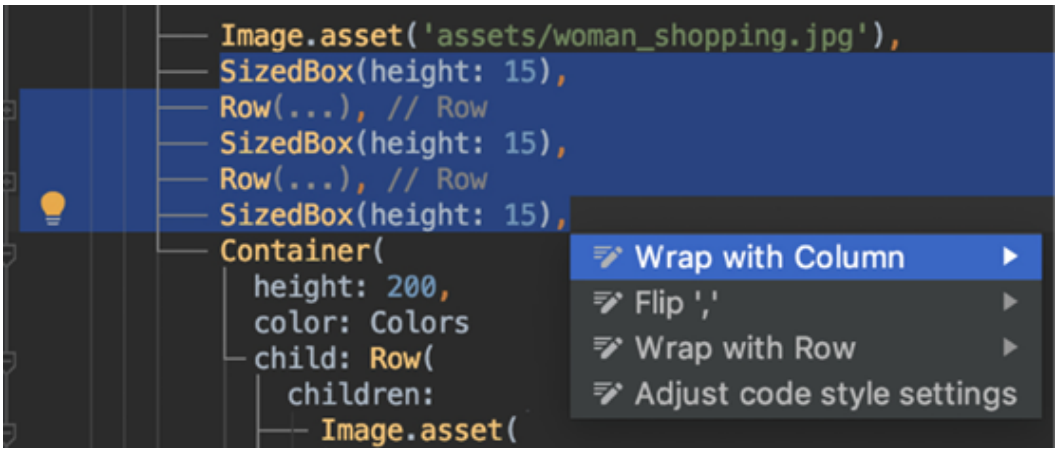


Figure 5.14: Wrap with Column option

5. The whole group of widgets can now be extracted. This time, extract the `Column` widget to a Flutter widget by right-clicking and selecting **Refactor** | **Extract** | **Flutter Widget**. Call the new widget `DealButtons`.

6. Finally, extract the last `Container` widget into a method called `_buildProductTile`. After all of this is done, the `build` method should be short enough to fit on the screen:

```
@override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.purple,
      appBar: _buildAppBar(),
      body: Padding(
        padding: const EdgeInsets.all(20.0),
        child: Column(
          children: <Widget>[
            _buildToggleBar(),
            SizedBox(
                height: 100, child:
Image.asset('assets/woman_shopping.jpg')),
```

```
            const DealButtons(),
            const _buildProductTile(),
          ],
        ),
      ),
    );
  }
```

7. Extracting is only one aspect of refactoring. There is also some redundant code that can be reduced. Scroll down to the newly created `DealButtons` widget.
8. Extract the third `Expanded` widget, the one that says **Must buy in summer**, into another Flutter widget called `DealButton`.
9. Add two properties at the top of the `DealButton` class and update the widget as shown in the following code snippet:

```
class DealButton extends StatelessWidget {
  final String text;
  final Color color;
  const DealButton({
    required this.text,
    required this.color,
    Key? key,
  }) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Expanded(
      child: Container(
        height: 80,
        decoration: BoxDecoration(
            color: color,
            borderRadius: BorderRadius.circular(20)),
        child: Center(
            child: Text(
          text,
          style: const TextStyle(
            color: Colors.white,
            fontSize: 20.0,
            fontWeight: FontWeight.bold,
          ),
        )),
      ),
    );
```

```
  }
}
```

10. The `build` method for the `DealButtons` widget can now be significantly simplified. Replace that verbose repetitive method with this more efficient one:

```
  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        const SizedBox(height: 15),
        Row(
          children: const [
            DealButton(
              text: 'Best Sellers',
              color: Colors.orangeAccent,
            ),
            SizedBox(width: 10),
            DealButton(
              text: 'Daily Deals',
              color: Colors.blue,
            )
          ],
        ),
        const SizedBox(height: 15),
        Row(
          children:const [
            DealButton(
              text: 'Must buy in summer',
              color: Colors.lightGreen,
            ),
            SizedBox(width: 10),
            DealButton(
              text: 'Last Chance',
              color: Colors.redAccent,
            )
          ],
        ),
        const SizedBox(height: 15),
      ],
    );  }
}
```

If you run the app, you should not see any visual difference, but the code is now much easier to read.

How it works...

There are two extraction techniques that we highlighted in this recipe that bear delving deeper into:

- Extracting methods
- Extracting widgets

The first technique should be relatively straightforward. The Dart Analysis Server will inspect all the elements of your highlighted code and the Flutter code refactoring features and simply pull it out into a new method in the same class.

You have code like this:

```
class RefactoringExample extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Row(
          children:[
            Text('Widget A'),
            Text('Widget B'),
          ],
        ),
        Row(
          children: [
            Text('Widget C'),
            Text('Widget D'),
          ],
        ),
      ],
    );
  }
}
```

You can go one step further beyond extracting, and simplify the code as well. Both of these rows have the same widget structure, but their text is different. You can update the widget to use a configurable `build` method, as shown here:

```
class RefactoringExample extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        buildRow('Widget A', 'Widget B'),
        buildRow('Widget C', 'Widget D'),
      ],
    );
  }
  Widget buildRow(String textA, String textB) {
    return Row(
      children: [
        Text(textA),
        Text(textB),
      ],
    );
  }
}
```

Not only is this code easier to read, but it's also reusable. You can add as many rows as you want with a different text. As an added bonus, you can update the `buildRow` method to add some padding around the text or change the text style, and these changes will automatically be applied every time you invoke the `buildRow` method.

One of the main rules in programming is: **Don't repeat yourself**.

If you have widget trees or statements that are repeated throughout your code, you should refactor your code. If you can accomplish the same result with less code, then you will have fewer bugs and be able to change your code faster.

Extracting widget trees into Flutter widgets is similar to extracting methods. Instead of generating a method, the IDE will generate a whole new `StatelessWidget` subclass that is then instantiated instead of invoked. This

is generally considered a best practice and should be your default approach when creating your apps.

See also

If you want to learn more about the pyramid of doom, see https://en.wikipedia.org/wiki/Pyramid_of_doom_(programming).

See *Refactoring* by Martin Fowler, the quintessential book on this subject: https://martinfowler.com/books/refactoring.html.

Applying global themes

Consistency is at the heart of any good design. Every screen in your app should look as if it were designed as a single unit. Your font selections, color palettes, and even text padding are all part of your app's identity. When users look at your app, branding consistency is critical for recognition. Apple products *look* like Apple products, with their white backgrounds and sleek curves. Google's Material Design is a colorful splash of primary shapes and shadows.

To make all their products look like they belong to the same design system, these companies use detailed documents that explicitly describe the schematics of how UIs should be designed. On the programmatic side, we have themes. These are widgets that live at the top of the tree and influence all of their children. You don't need to declare styling for every single widget. You just need to make sure that it respects the theme.

In this recipe, we will take the e-commerce mock-up screen and simplify it even more by using themes to express the text and color styling.

Getting ready

Make sure that you have completed all the refactoring tasks from the previous recipe before beginning. If not, you can use the

`e_commerce_screen_after.dart` file as your base.

How to do it...

Open your IDE and run the app. Let's start theming the e-commerce screen by adding a theme to your `MaterialApp` widget:

1. Open `main.dart` and add the following code:

```
class StaticApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      theme: ThemeData(
             primarySwatch: Colors.green,
      ),
      home: ECommerceScreen(),
    );
  }
}
```

2. Now that the theme has been declared, the rest of the recipe will be about deleting code so that the theme traverses down to the appropriate widgets.
3. In the `Scaffold` of the `EcommerceScreen` class, delete the property and value for `backgroundColor`. In the `_buildAppBar` method, also delete these two lines:

```
backgroundColor: Colors.purpleAccent,
elevation: 0,
```

4. When you hot reload, the `AppBar` will be green, respecting the `primaryColor` property of the app's theme.
5. The toggle bar could use a bit more refactoring, along with removing more styling information. Extract one of the `Padding` widgets in `_buildToggleBar` to a method called `_buildToggleItem`.
6. Then, update the code so that the extracted method is parametrized:

```
Widget _buildToggleBar() {
  return Row(
    children: [
      _buildToggleItem(context, 'Recommended',
selected: true),
      _buildToggleItem(context, 'Formal Wear'),
      _buildToggleItem(context, 'Casual Wear'),
    ],
  );
}
Widget _buildToggleItem(BuildContext context, String
text,
    {bool selected = false}) {
  return Padding(
    padding: const EdgeInsets.all(8.0),
    child: Text(
      text,
      style: TextStyle(
        fontSize: 17,
        color: selected
            ? null
            :
Theme.of(context).textTheme.titleMedium?.color?.withOpac
ity(0.5),
            fontWeight: selected ? FontWeight.bold :
null,
      ),
    ),
  );
}
```

7. The theme needs access to `BuildContext` to work correctly, but the original `_buildToggleBar` method doesn't have access to it.
8. The context has to be passed down from the root `build` method. Update the signature of `_buildToggleBar` to accept a context:

```
Widget _buildToggleBar(BuildContext context) { ... }
```

> **Note**
>
> In a `Widget` **class**, `BuildContext` is automatically available, so you don't need to pass it to the method.

9. Change the `build` method to pass down the context:

```
Column(
  children: [
    _buildToggleBar(context),
    Image.asset('assets/woman_shopping.jpg'),
    DealButtons(),
    _buildProductTile(context),
  ],
),
```

10. Let's add a dark theme to our app. In the `main.dart` file, in the `MaterialApp` theme, add the `Brightness.dark` property:

```
ThemeData(
  brightness: Brightness.dark,
  primaryColor: Colors.green,
 ),
```

11. If you hot-reload the app, you'll notice that all the text in the product tile has disappeared. That's a bit of an illusion because the theme we selected is rendering white text on a white background.
12. Update `_buildProductTile` to make the text visible:

```
Widget _buildProductTile(BuildContext context) {
  return Container(
    height: 200,
    color: Theme.of(context).cardColor,
```

13. The screen should be fully responding to the app theme. But we can take it further.
14. Update the theme in `main.dart` to assign a global style for every `AppBar`:

```
ThemeData(
  brightness: Brightness.dark,
  primaryColor: Colors.green,
  appBarTheme: AppBarTheme(
          elevation: 10,
          titleTextStyle: const TextTheme(
            headline6: TextStyle(
```

```
              fontFamily: 'LeckerliOne',
              fontSize: 24,
            ),
          ).titleLarge,
        ),),
```

15. In order to use the Leckerli One font, download it from https://fonts.google.com/specimen/LeckerliOne, add it to your `assets` folder, and in your `pubspec.yaml` file, enable it with the following instructions:

```
fonts:
    - family: LeckerliOne
      fonts:
        - asset: assets/LeckerliOne-Regular.ttf
```

16. Everything in the app is very dark, not so becoming for an e-commerce app. Quickly adjust the theme's brightness back to `light`:

```
brightness: Brightness.light,
```

17. Perform a hot reload and feast your eyes on the final themed product:

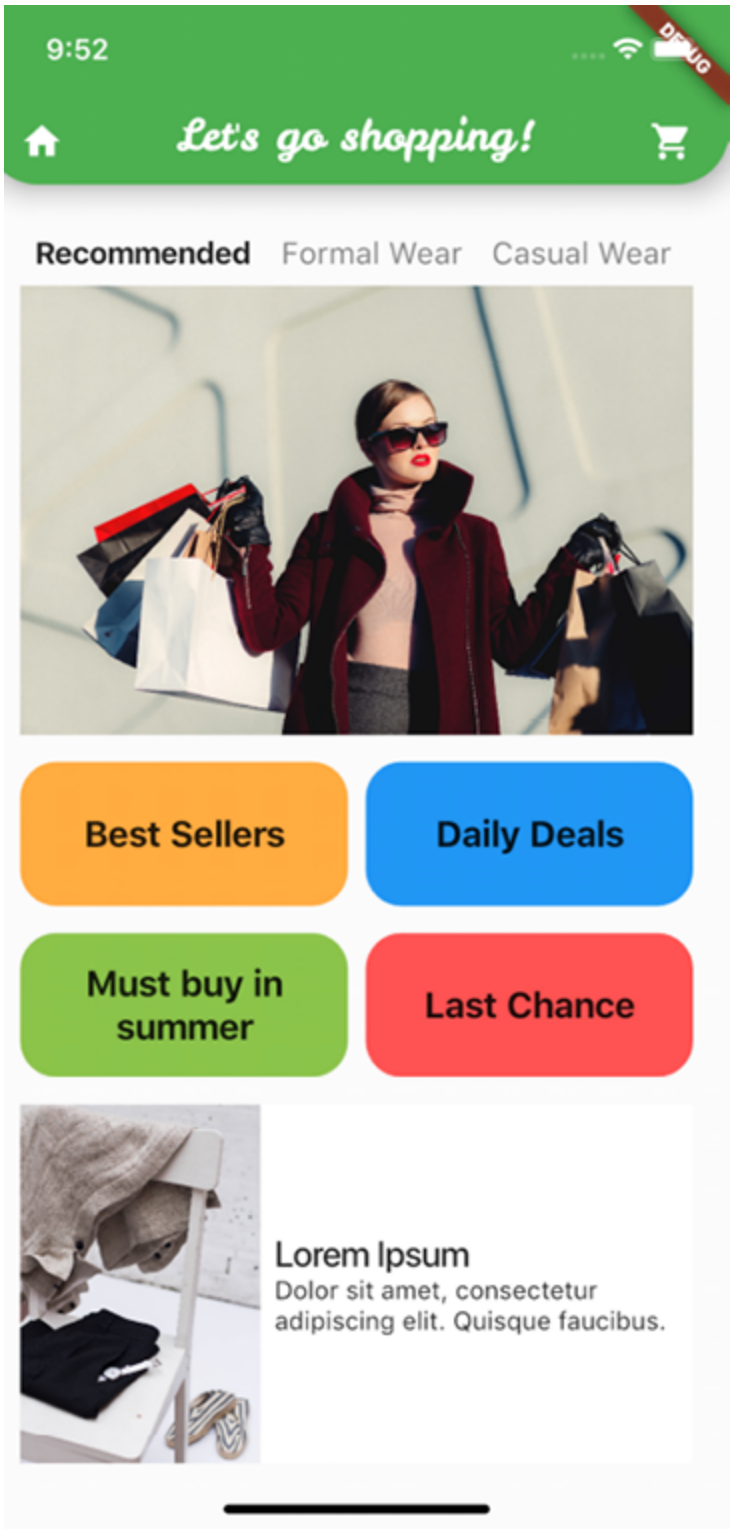


Figure 5.15: A complex widget tree in action

How it works...

`MaterialApp` widgets (and `CupertinoApp` widgets) are not just a single widget; they compose a widget tree that contains many elements that build the UI for your app.

The `ThemeData` class is just a plain old Dart object that stores properties and sub-themes, such as a `TextTheme` and `AppBarTheme` sub-theme. These sub-themes are also just models that store values.

The interesting part happens with this line:

```
Theme.of(context)
```

`Theme` is an `InheritedWidget`. This means that when you create a `Theme` widget and define a set of properties to define a visual style, those properties are inherited by all child widgets that are descendants of the `Theme` widget in the widget tree. We've already briefly covered this pattern—in short, what's happening is that the app is traveling up the widget tree to find the `Theme` widget and return its `ThemeData` value:

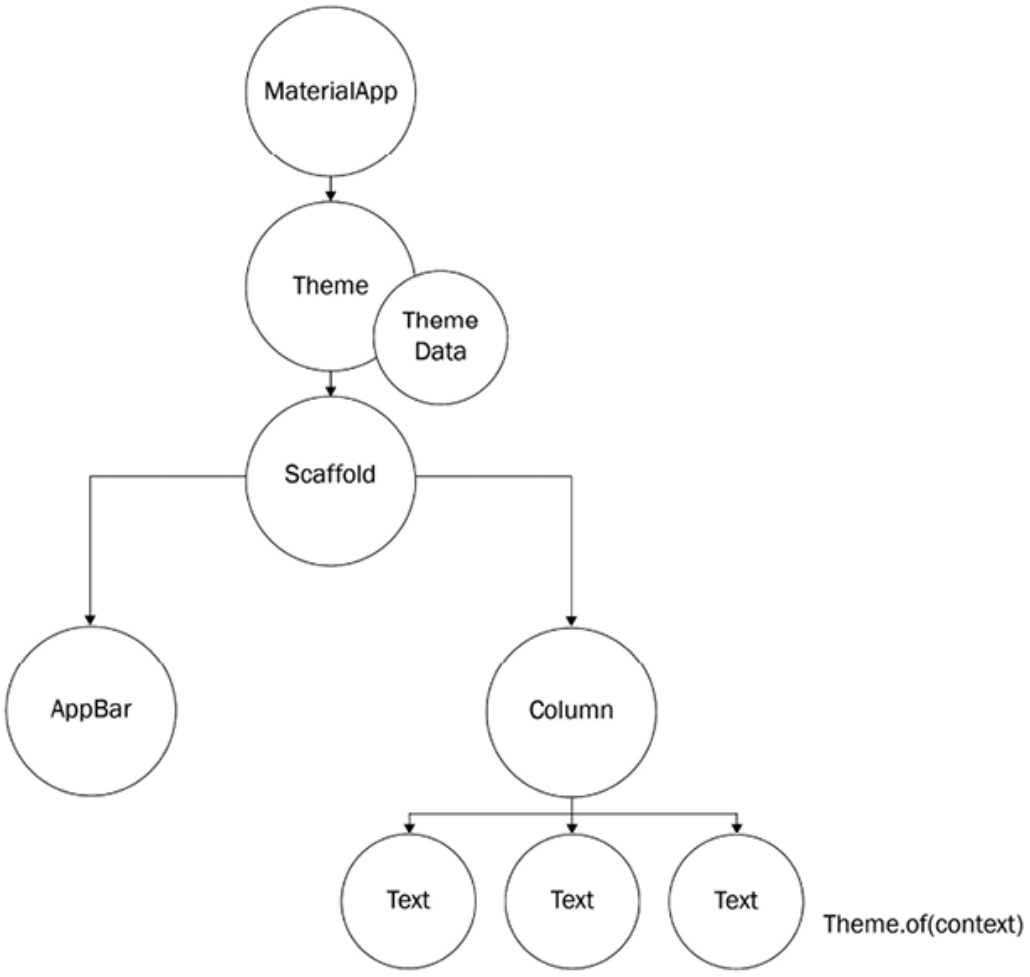


Figure 5.16: Widget tree

`BuildContext` is the key to unlocking this tree traversal. It is the only object that is aware of the underlying widget tree and the node's parent/child relationships.

Most widgets in the `Material` and `Cupertino` libraries are already theme-aware. The `AppBar` class references the theme's primary color to use for its background. The `Text` widget applies the body style of the default text theme by default. When you design your own widgets, you should strive for this same level of flexibility. It is perfectly acceptable to add properties to your widget's constructor to style your widget, but in the absence of data, fall back to the theme.

There's more...

Another interesting property when dealing with themes is the `brightness` property. Light and dark apps are now getting more common.

The `lightness` enum is how Flutter supports this feature. By toggling lightness, you have seen the background and text colors automatically become dark/light.

There is also a `darkTheme` property in `MaterialApp` where you can design the dark version of your app. These properties are platform-aware and will automatically toggle the themes based on the phone's settings.

Including this feature now in your app will future-proof your apps, as we are entering a world where both light and dark support is expected.

See also

Check out these resources to learn more about the design specifications for iOS and Android:

- Material Design spec: https://material.io/design/
- iOS *Human Interface Guidelines*: https://developer.apple.com/design/human-interface-guidelines/ios/overview/themes/

Summary

In this chapter, you have seen several ways to manage your widget trees:

- You have used `Row` and `Column` widgets to place their content in a linear way, and seen an introduction to proportional spacing between widgets with the use of `Flexible` and `Expanded` widgets.

- You have used a `CustomPaint` widget to draw custom shapes: in the code example, you have designed stars that can be used to show ratings for products or any other content.
- You have seen how to break up complex layouts into smaller, reusable widgets, and used refactoring techniques to simplify widget trees and improve maintainability and performance.
- Finally, we have covered the use of global themes to apply a consistent visual style to a Flutter app, and how to use the `Theme` widget to define properties that can be inherited by child widgets.
- One of the key takeaways from this chapter is the importance of creating reusable widgets and breaking up complex layouts into smaller, more manageable parts. This will help you write manageable user interfaces for your apps.

Adding Interactivity and Navigation to Your App

Frontend application design is often divided into two categories: **user interface** (**UI**) and **user experience** (**UX**). The user interface is made up of all the elements on the screen—images, colors, panels, text, and so on. The user experience is what happens when your users *interact* with your interfaces. It governs interactivity, navigation, and animations. If the UI is the "*what*" of app design, then the UX is the "*how*."

So far, we have covered some of the user interface components in Flutter. Now, it's time to make our widgets useful and start building interactivity. We're going to cover some of the primary widgets that are used to deal with user interactions, in particular buttons, text fields, scrolls, and dialogs. You will also see a simple way to use the Navigator class to create apps with multiple screens.

Throughout this chapter, you are going learn how to build a single app called *Stopwatch*. This will be a fully functioning stopwatch that will keep track of laps and show a full history of every completed lap.

In this chapter, we're going to cover the following recipes:

- Adding state to your app
- Interacting with buttons
- Making it scroll
- Handling large datasets with list builders
- Working with TextFields
- Navigating to the next screen
- Showing dialogs on the screen

- Presenting bottom sheets

Let's begin with dealing with state in your apps.

Adding state to your app

So far, we've only used `StatelessWidget` components to create user interfaces. These widgets are perfect for building static layouts, *but they cannot change*. Flutter has another type of widget called `StatefulWidget`. **StatefulWidgets** can keep information (state) and know how to recreate themselves whenever their `State` changes.

Compared to Stateless Widgets, StatefulWidgets have a few more moving parts. In this recipe, we're going to create a very simple stopwatch that increments its counter once a second.

Getting ready

Let's start off by creating a new project. Open your IDE (or Terminal) and create a new project called `stopwatch`. Once the project has been generated, delete all the code in `main.dart` to start with a clean app.

How to do it...

Let's start building our stopwatch with a basic counter that auto increments:

1. We need to create a new shell for the app that will host the `MaterialApp` widget. This root app will still be stateless, but things will be more mutable in its children. Add the following code for the initial shell (`StopWatch` has not been created yet, so you will see an error that we will fix shortly):

```
import 'package:flutter/material.dart';
void main() => runApp(const StopWatchApp());
class StopWatchApp extends StatelessWidget {
  const StopWatchApp({super.key});
```

```
  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: StopWatch(),
    );
  }}
```

2. In the `lib` folder, create a new file called `stopwatch.dart`. A `StatefulWidget` is divided into two classes: the widget and its state. There are IDE shortcuts that can generate this in one go, just like there are for `StatelessWidget`. However, for your first one, create the widget manually. Add the following code to create the `StatefulWidget` stopwatch:

```
import 'package:flutter/material.dart';
import 'dart:async';
class StopWatch extends StatefulWidget {
  const StopWatch({super.key});
  @override
  State<StopWatch> createState() => _StopWatchState();
}
```

3. Every `StatefulWidget` needs a `State` object that will maintain its life cycle. This is a completely separate class. StatefulWidgets and their `State` are so tightly coupled that this is one of the few scenarios where you should keep more than one class in a single file. Create the private `_StopWatchState` class right after the `StopWatch` class:

```
class _StopWatchState extends State<StopWatch> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Stopwatch'),
      ),
      body: Center(
        child: Text(
          '0 seconds',
          style:
Theme.of(context).textTheme.headlineSmall,
        ),
      ),
    );
```

```
  }
}
```

> **Note**
>
> `State` looks almost like a `StatelessWidget`, right? In `StatefulWidgets`, you put the `build` method in the `State` class, not in the widget.

4. In the `main.dart` file, add an import for the `stopwatch.dart` file:

```
import './stopwatch.dart';
```

5. Run the app. You should see a screen with text stating "*0 Seconds*" at the center of the screen. At this time, the text will never change. We can solve this by keeping track of a `seconds` property in our `State` class and using a `Timer` to increment that property every second. Each time the timer ticks, we're going to tell the stopwatch to redraw by calling `setState`.
6. Add the following code just after the `class` definition, but before the `build` method:

```
class StopWatchState extends State<StopWatch> {
  int seconds = 0;
  late Timer timer;
  @override
  void initState() {
    super.initState();
    timer = Timer.periodic(Duration(seconds: 1),
_onTick);
  }
  void _onTick(Timer time) {
        if (mounted){
    setState(() {
      ++seconds;
    });
   }
  }
```

7. Now, it's just a simple matter of updating the `build` method so that it uses the current value of the `seconds` property instead of a hardcoded value. First, let's add this helper function just after the `build` method to make sure that our text is always grammatically correct:

```
String _secondsText() => seconds == 1 ? 'second' :
'seconds';
```

8. Update the `Text` widget in the build method so that it can now use the `seconds` property:

```
Text(
  '$seconds ${_secondsText()}',
  style: Theme.of(context).textTheme.headline5,
),
```

9. Finally, we just need to make sure the timer stops ticking when we close the screen. Add the following `dispose` method at the bottom of the state class, just after the `_secondsText` method:

```
@override
void dispose() {
 timer.cancel();
 super.dispose();
}
```

10. Run the app.

You should now see a counter incrementing once a second. Well done!

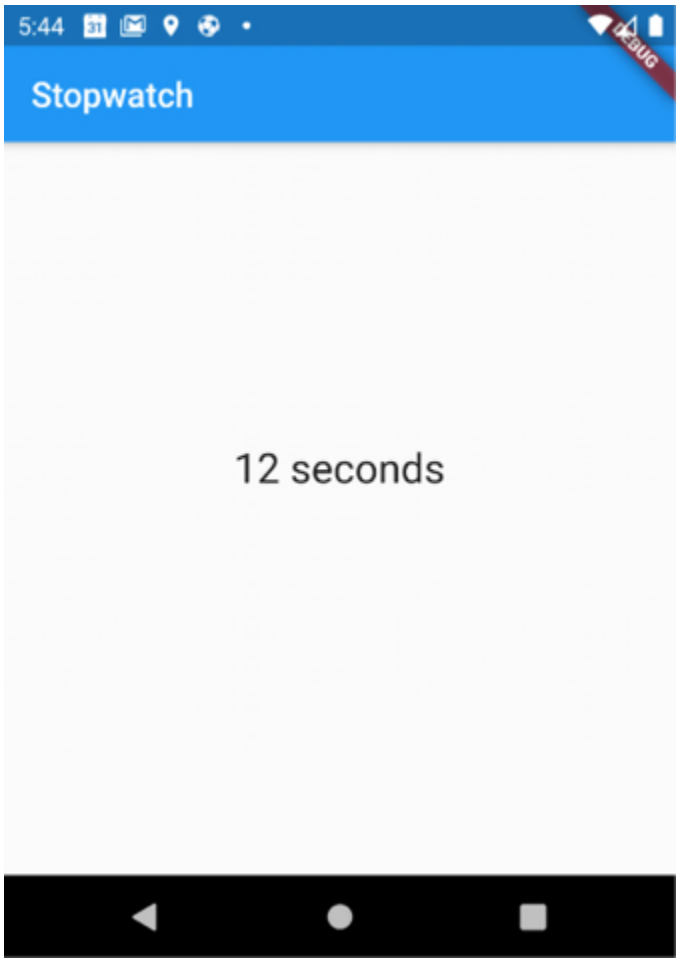


Figure 6.1: The stopwatch app user interface incrementing seconds

How it works...

`StatefulWidgets` are made up of two classes: the widget and its state. The *widget* part of `StatefulWidget` really doesn't do much. The data that changes should be placed in the State class.

Note

All widgets, whether they are stateless or stateful, are immutable. In `Stateful`Widgets, the State can change.

What doesn't have to be immutable is the `State` object. The `State` object takes over the build responsibilities from the widget. States can also be marked as dirty, which is what will cause them to repaint on the next frame. Take a close look at this line:

```
setState(() {
    ++seconds;
});
```

The `setState` function tells Flutter that a widget needs to be repainted. In this specific example, we are incrementing the `seconds` property by one, which means that when the `build` function is called again, it will replace the `Text` widget with different content.

Note

Each time you call `setState`, the widget is repainted: this means that the `build` method is called every time you call `setState`. You should never call `build` from your code: always call `setState` instead.

The following diagram summarizes how Flutter's render loop is impacted by `setState`:

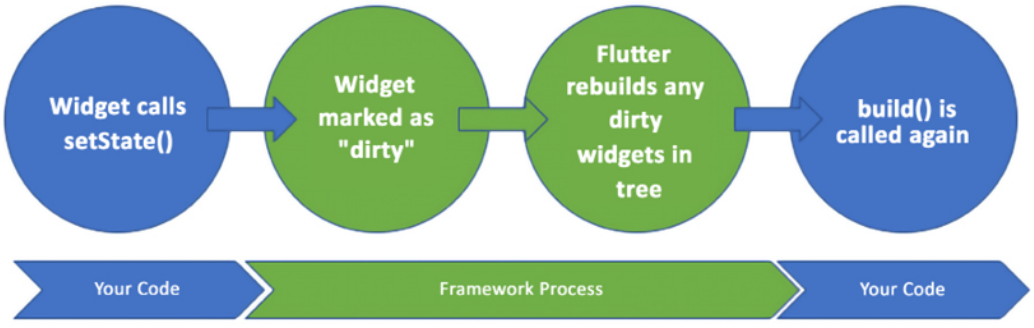


Figure 6.2: setState and build in Flutter

Please note that the closure that you use in `setState` is completely optional. It's more for code legibility purposes. We could just as easily write the following code and it would do exactly the same:

```
seconds++
setState(() {});
```

You should also avoid performing any complex operations inside the `setState` closure since that can cause performance bottlenecks. It is typically used for simple value assignments.

There's more...

The `State` class has a life cycle. Unlike `StatelessWidget`, which has nothing more than a `build` method, StatefulWidgets have life cycle methods that are called in a specific order. In this recipe, you used `initState` and `dispose`, but the full list of life cycle methods, in order, is as follows:

- `initState`
- `didChangeDependencies`
- `didUpdateWidget`
- `build` **(required)**
- `reassemble`
- `deactivate`
- `dispose`

The methods in bold are the most frequently used life cycle methods. While you could override all of them, you will mostly use the methods in bold. Let's briefly discuss these methods and their purposes:

- **initState**: This method is used to initialize any non-final value in your state class. You can think of it as performing a job similar to a constructor. In our example, we used `initState` to kick off a `Timer` that fires once a second. This method is called *after* the constructor, but *before* the widget is added to the tree.

- **didChangeDependencies**: This method is called immediately after `initState`, but unlike that method, the widget now has access to its `BuildContext`. For example, if you need to do any setup work that requires context, then this the most appropriate method for those processes.
- **build**: The `State`'s `build` method is identical to StatelessWidget's `build` method and is required. Here, you are expected to define and return this widget's tree. In other words, in the `build` method you create the UI.
- **dispose**: This clean-up method is called when the `State` object is removed from the widget tree. This is your last opportunity to clean up any resources that need to be explicitly released. In the recipe, we used the `dispose` method to stop the Timer. Otherwise, the time would just keep on ticking, even after the widget has been destroyed. Forgetting to close long-running resources can lead to memory leaks, unpredictable behavior, and in some cases, even crashes.

See also

Check out these resources for more information about StatefulWidgets:

- Video on StatefulWidgets by the Flutter team: https://www.youtube.com/watch?v=AqCMFXEmf3w
- Official state documentation by the Flutter team: https://api.flutter.dev/flutter/widgets/State-class.html

Interacting with buttons

Buttons are one of the most important pieces of UI in any app. It's almost impossible to imagine an app that doesn't have a button in some form or another. They are extremely flexible: you can customize their shape, color, and touch effects; provide haptic feedback; and more. But regardless of the styling, all buttons serve the same purpose. A button is a widget that users

can touch (or press, or click). When their finger lifts off the button, they expect the button to react. Over the years, we have interacted with so many buttons that we don't even think about this interaction anymore—it has become obvious.

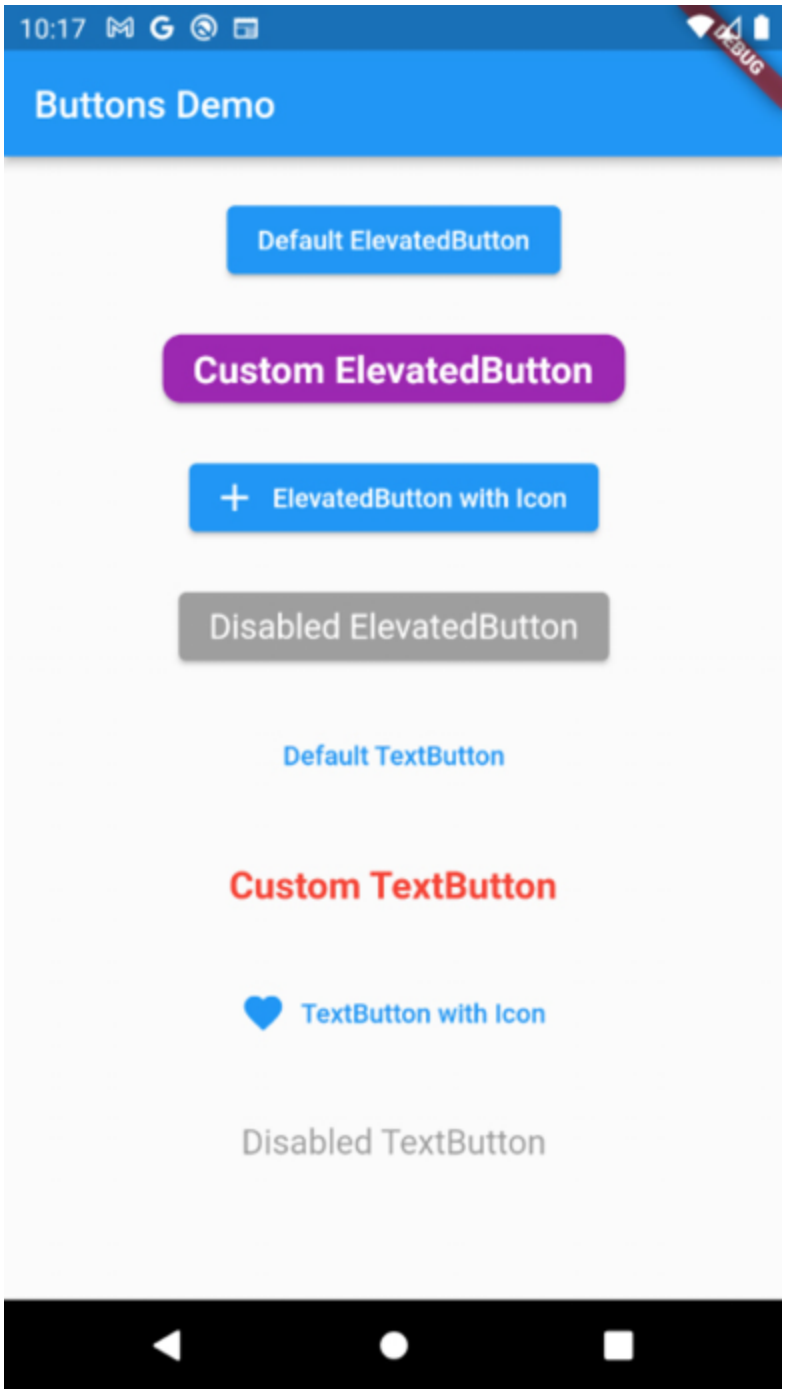


Figure 6.3: Different button styles

In this recipe, you are going to add two buttons to the stopwatch app: one to start the counter and another to stop it. You going are going to use two different styles of buttons—`ElevatedButton` and `TextButton`—for these different functions. Actually, even though they *look* different, their API is the same.

Getting ready

We're going to continue with the Stopwatch project. Make sure you have completed the previous recipe since this one builds on the existing code.

How to do it...

To complete this recipe, open `stopwatch.dart` and follow these steps:

1. Update the `build` method in `StopWatchState` to replace the `Center` widget with a `Column`. You should be able to use the quick fix dialog that we discussed in the previous chapter to quickly wrap the `Text` widget in a `Column` and then remove the `Center` widget. Then add two buttons to the screen. When you are done, the `build` method should look like this:

```
return Scaffold(
  appBar: AppBar(
        title: const Text('Stopwatch'),
      ),
      body: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Text(
            '$seconds ${_secondsText()}',
            style:
Theme.of(context).textTheme.headlineSmall,
          ),
          const SizedBox(height: 20),
          Row(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              ElevatedButton(
                style: ButtonStyle(
```

```
                  backgroundColor:
                      MaterialStateProperty.all<Color>
(Colors.green),
                  foregroundColor:
                      MaterialStateProperty.all<Color>
(Colors.white),
                ),
                onPressed: null,
                child: const Text('Start'),
              ),
              const SizedBox(width: 20),
              TextButton(
                style: ButtonStyle(
                  backgroundColor:
MaterialStateProperty.all<Color>(Colors.red),
                  foregroundColor:
                      MaterialStateProperty.all<Color>
(Colors.white),
                ),
                onPressed: null,
                child: const Text('Stop'),
              ),
            ],
          )
        ],
      ),);
```

2. If you hot reload, you'll see two buttons on the screen.
3. Press them… but nothing will happen. This is because you have to add an `onPressed` function, before the button activates.
4. Let's add a property at the top of the `State` class that will keep track of whether the timer is ticking or not and optionally add `onPressed` functions depending on that state value. Add this line at the very top of `StopWatchState`:

```
bool isTicking = false;
```

5. Add two functions that will toggle this property and cause the widget to repaint. Add these methods under the `build` method:

```
void _startTimer() {
  setState(() {
    isTicking = true;
```

```
  });
}
void _stopTimer() {
  setState(() {
    isTicking = false;
  });
}
```

6. Now, you can hook these methods into the `onPressed` property.
7. Update your buttons so that they use these ternary operators in button declarations. Note that `_startTimer` is passed as a parameter and **not called**, so make sure you omit the parenthesis. Some of the setup code has been omitted for brevity:

```
ElevatedButton(
  ...
  onPressed: isTicking ? null : _startTimer,
  ...
),
...
TextButton(
  ...
  onPressed: isTicking ? _stopTimer : null,
  ...
),
```

8. Now, it's time to add some logic to the start and stop methods to make the timer respond to our interactions.
9. Update those buttons so that they include the following code:

```
void _startTimer() {
  timer = Timer.periodic(const Duration(seconds: 1),
_onTick);
  setState(() {
    seconds = 0;
    isTicking = true;
  });
}
void _stopTimer() {
  timer.cancel();
  setState(() {
    isTicking = false;
```

```
  });
}
```

10. One more quick thing: we don't really need the `initState` method anymore now that the buttons are controlling the timer. Delete the `initState` method.

Congratulations, you should now have a fully functioning timer app!

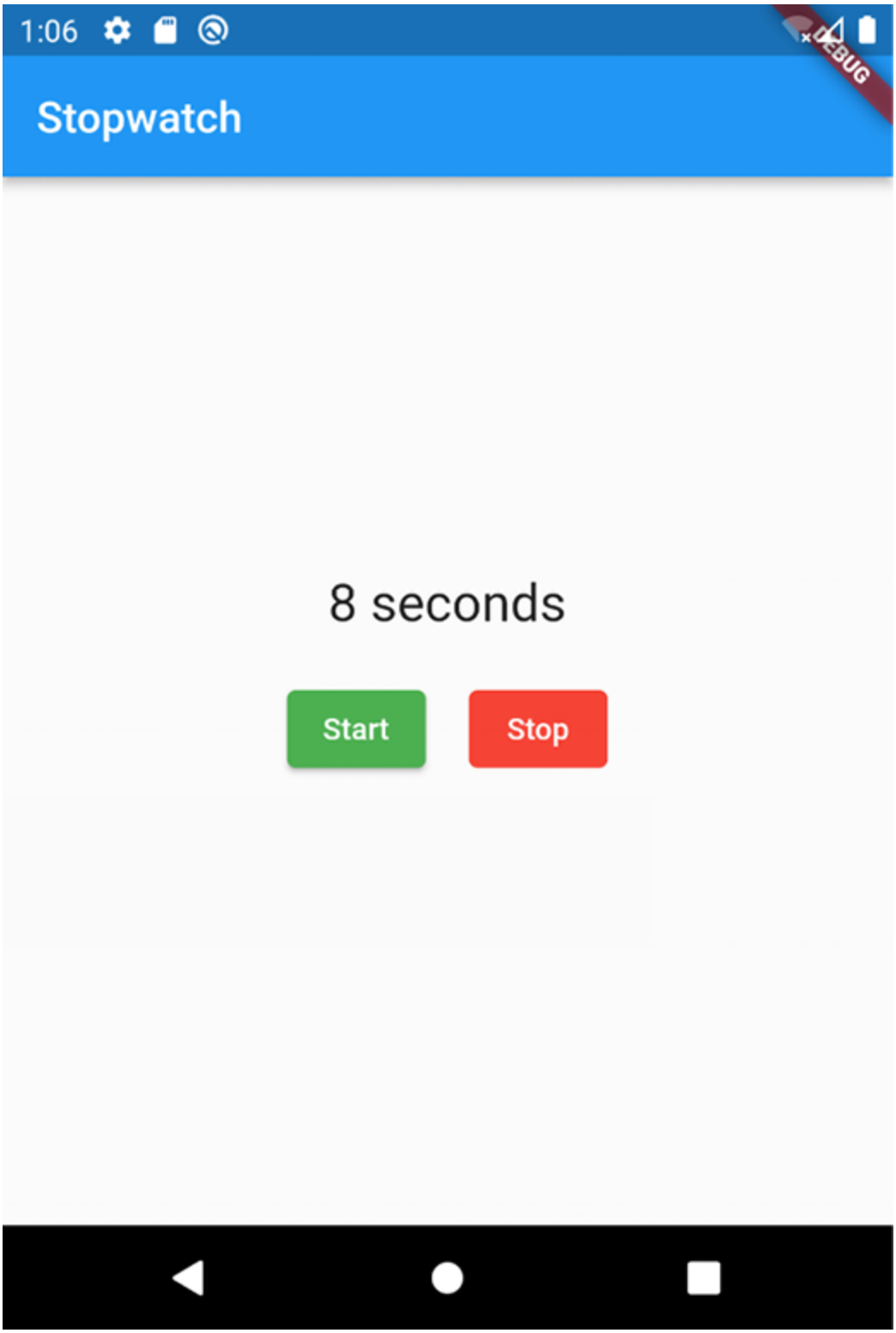


Figure 6.4: The stopwatch running with the Start and Stop buttons

How it works...

Buttons in Flutter are pretty simple—they are just widgets that accept a function. These functions are then executed when the button detects an interaction. If a `null` value is supplied to the `onPressed` property, Flutter considers the button to be disabled.

Flutter has several button types that can be used for different aesthetics, but their functionality is the same. These include:

- `ElevatedButton`
- `TextButton`
- `IconButton`
- `FloatingActionButton`
- `OutlinedButton`
- `DropDownButton`
- `CupertinoButton`

You can play around with any of these widgets until you find a button that matches your desired look.

In the example shown in this recipe, you have applied a style to the button through the style property and used the ButtonStyle class. As an alternative you can also use `ElevatedButton.stylefrom()`:

```
ElevatedButton(
  style: ElevatedButton.styleFrom(
    primary: Colors.green,
    textStyle: TextStyle(fontSize: 20, fontWeight:
FontWeight.bold),
    padding: EdgeInsets.symmetric(horizontal: 12, vertical:
8),
    shape: RoundedRectangleBorder(borderRadius:
BorderRadius.circular(10)),
  ),
)
```

In this recipe, we wrote the `onPressed` functions out as methods in the `StopWatchState` class, but it is perfectly acceptable to throw them into the functions as closures. We could have written the buttons like this:

```
ElevatedButton(
  child: Text('Start'),
  onPressed: isTicking
      ? null
      : () {
          timer = Timer.periodic(Duration(seconds: 1),
_onTick);
          setState(() {
            seconds = 0;
            isTicking = true;
          });
        },
),
```

For simple actions, this is fine, but even in our simple example, where we want to control whether the button is active or not via a ternary operator, this is already becoming harder to read.

Making it scroll

It is very rare to encounter an app that doesn't have some sort of scrolling content. Scrolling, especially vertical scrolling, is one of the most natural paradigms in mobile development. When you have a list of elements that take more than the height of a screen, you need some sort of scrollable widget.

Scrolling content is actually rather easy to accomplish in Flutter. To get started with scrolling, a great widget is `ListView`. Just like `Columns`, `ListViews` control a list of child widgets and place them one after another. However, `ListViews` will also make that content scroll automatically when their height is bigger than the height of their parent widget.

In this recipe, we're going to add laps to our stopwatch app and display those laps in a scrollable list.

Getting ready

Once again, we're going to continue with the *Stopwatch* project. You should have completed the previous recipes in this chapter before following along with this one.

How to do it...

The first thing you are going to do is make the timer a bit more precise. Seconds are not a very useful value to use for stopwatches:

1. Open up `stopwatch.dart`.
2. Rename the `seconds` property to `milliseconds`. We also need to update the `onTick`, `_startTimer`, and `_secondsText` methods:

```
int milliseconds = 0;
void _onTick(Timer time) {
  setState(() {
    milliseconds += 100;
  });
}
void _startTimer() {
  timer = Timer.periodic(Duration(milliseconds: 100),
_onTick);
  ...
}
String _secondsText(int milliseconds) {
  final seconds = milliseconds / 1000;
  return '$seconds seconds';
}
```

3. Update the `Text` widget in the `build` method, so that it shows the correct value:

```
Text(_secondsText(milliseconds),
style: Theme.of(context).textTheme.headline5,),
```

4. Now, let's add a laps list so that we can keep track of the values for each lap. We're going to add to this list each time our user taps a lap

button.

5. Add the following declaration at the top of the `StopWatchState` class, just under the declaration of the timer:

```
final laps = <int>[];
```

6. Create a new `lap` method to increment the lap count and reset the current millisecond value:

```
void _lap() {
  setState(() {
    laps.add(milliseconds);
    milliseconds = 0;
  });
}
```

7. We also need to tweak the `_startTimer` method in order to reset the lap list every time the user starts a new counter.
8. Add the following line inside the `setState` closure, in `_startTimer`:

```
laps.clear();
```

9. Now, we need to organize our widget code a bit to enable adding scrollable content. Take the existing `Column` in the `build` method and extract it into its own method, called `_buildCounter`. The refactoring tools should automatically add a `BuildContext` to that method. Change the return type of this new method from `Column` to `Widget`.
10. To make the UI a bit nicer, wrap `Column` into a `Container` and set its background to the app's primary color. Also, add one more `Text` widget above the counter to show which lap you are currently on.
11. Finally, make sure that you adjust the color of the text to white so that it's readable on a blue background. The top of the method will look like this:

```
Widget _buildCounter(BuildContext context) {
  return Container(
    color: Theme.of(context).primaryColor,
    child: Column(
```

```
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        Text(
          'Lap ${laps.length + 1}',
          style: Theme.of(context)
              .textTheme
              .headline5!
              .copyWith(color: Colors.white),
        ),
        Text(
          _secondsText(milliseconds),
          style: Theme.of(context)
              .textTheme
              .headline5!
              .copyWith(color: Colors.white),
        ),
```

12. Inside the `_buildCounter` method, extract `Row`, where the buttons are built into its own method called `_buildControls`. As always, the return type of that new method should be changed to `Widget`.
13. Add a new button between the start and stop buttons that will call the `lap` method. Put a `SizedBox` before and after this button to give it some spacing:

```
const SizedBox(width: 20),
ElevatedButton(
    style: ButtonStyle(
      backgroundColor:
MaterialStateProperty.all(Colors.yellow),
     ),
     onPressed: isTicking ? _lap : null,
     child: const Text('Lap'),
),
SizedBox(width: 20),
```

14. That's a lot of refactoring! But it was all for a good purpose. Now, you can finally add a `ListView`.
15. Add the following method before the `dispose()` method. This will create the scrollable content for the laps:

```
Widget _buildLapDisplay() {
  return ListView(
```

```
    children: [
      for (int milliseconds in laps)
        ListTile(
          title: Text(_secondsText(milliseconds)),
        )
    ],
  );
}
```

16. Update the primary `build` method to use the new scrolling content. Wrap both top-level widgets in an `Expanded` so that both the stopwatch and our `ListView` take up half the screen:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Stopwatch'),
    ),
    body: Column(
      children:[
        Expanded(child: _buildCounter(context)),
        Expanded(child: _buildLapDisplay()),
      ],
    ),
  );
}
```

17. Run the app. You should now be able to add laps to your stopwatch. After adding a few laps, you'll be able to see the laps scroll:

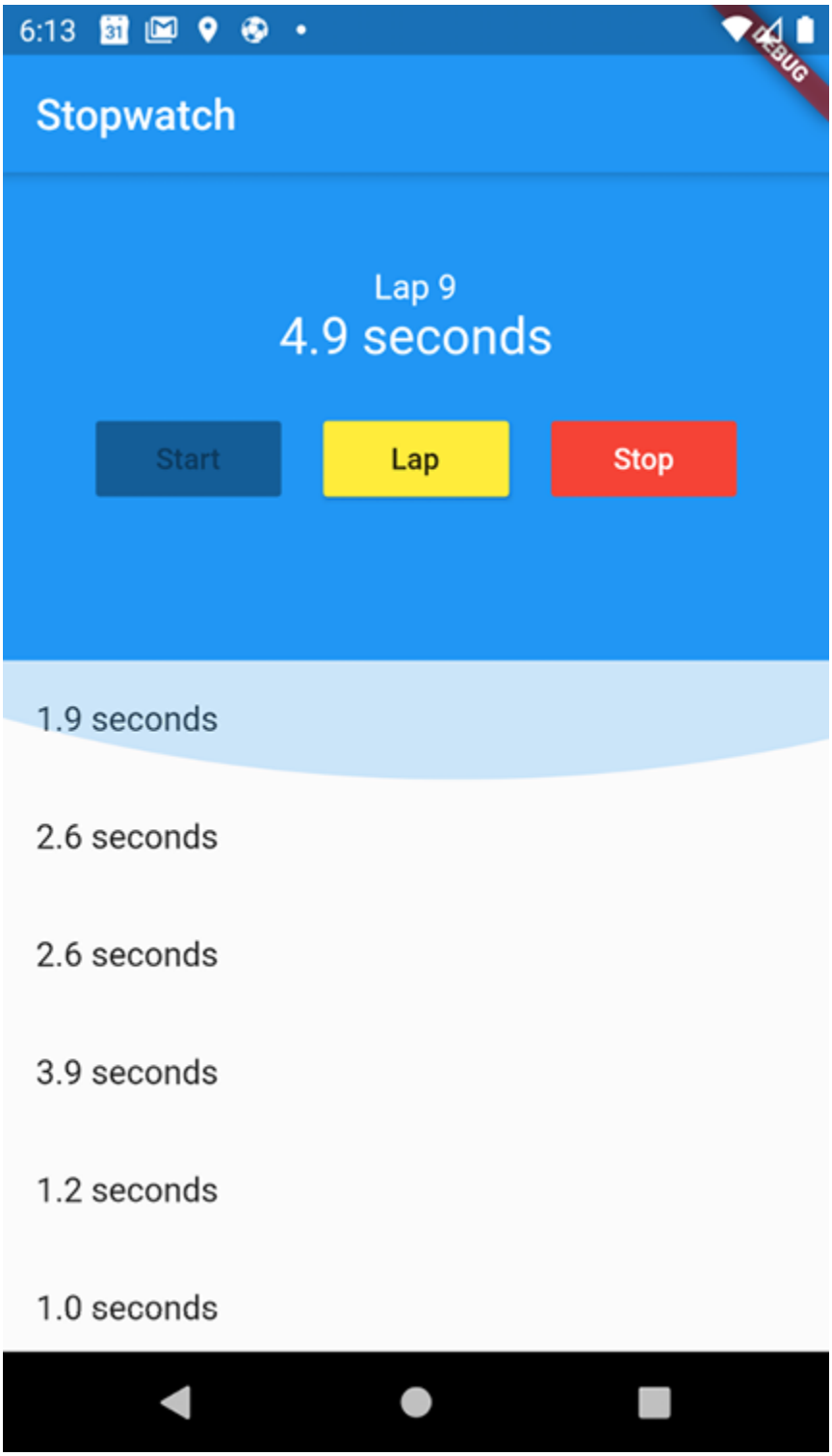


Figure 6.5: The stopwatch running with the laps list

How it works...

As you have seen in this recipe, creating scrollable widgets in Flutter is easy. There is only one method that creates a scrolling widget and that method is *tiny.* Scrolling in Flutter is just a simple matter of choosing the correct widget and passing your data. The framework takes care of the rest.

Let's break down the scrolling code that is in this recipe:

```
Widget _buildLapDisplay() {
  return ListView(
    children: [
      for (int milliseconds in laps)
        ListTile(
          title: Text(_secondsText(milliseconds)),
        )
    ],
  );
}
```

We're using a type of scrolling widget called `ListView`, which is probably one of the simplest types of scrolling widgets in Flutter. This widget functions a bit like a column, except instead of throwing errors when it runs out of space, `ListView` will enable scrolling, allowing you to use drag gestures to scroll through all the data.

In our example, we're also using the **collection-for** syntax (maybe you already know it as a **for… in** or **for… each** cycle) to create the widgets for this list. This will create one very long column as you add laps.

One other interesting thing about scrolling in Flutter is that it is platform aware. If you can, try running the app in both the Android Emulator and the iOS Simulator; you'll notice that the scroll *feels* different. You can do the same with the web and your desktop. What you are encountering is something called `ScrollPhysics`. These are objects that define how the list is supposed to scroll and what happens when you get to the end of the list. On iOS, the list is supposed to bounce, whereas, on Android, you get a glow effect when you get to the edges. The widget can pick the correct `ScrollPhysics` strategy based on the platform, but you can also override this behaviour if you want to make the app behave in a particular way, regardless of the platform, using the following code:

```
ListView(
  physics: BouncingScrollPhysics(),
  children: [...],
);
```

Note

You generally shouldn't override the platform's expected behavior, unless there is a good reason for doing so. It might confuse your users if they start adding iOS paradigms on Android and vice versa.

There's more...

One final important thing to keep in mind about scrolling widgets is that because they need to know their parent's constraints to activate scrolling, putting scroll widgets inside widgets with unbounded constraints can cause Flutter to throw errors.

In Flutter there's a rule: "Constraints go down. Sizes go up.". This means that:

- Parent widgets pass down constraints to their children. Constraints are passed in a `BoxContraints` object, that includes minimum and maximum values for the width and height of the children.
- Children widgets determine their size and then pass the information back up to the parent widget.

In our example, we placed `ListView` inside a `Column`, which is a widget that lays out its children based on their **intrinsic size** (the size a widget would be based on its content and layout rules, without the constraints dictated by parent widgets).

This works fine for widgets such as `Containers`, `Buttons`, and `Text`, but it fails for `ListViews`. To make scrolling work inside `Column`, we had to wrap it in an `Expanded` widget, which will then tell `ListView` how much space it

has to work with. Try removing `Expanded`; the whole widget will disappear and you should see an error in the Debug console:

```
I/flutter (28416): ═╡ EXCEPTION CAUGHT BY RENDERING LIBRARY ╞═══════════════════════════════
I/flutter (28416): The following assertion was thrown during performResize():
I/flutter (28416): Vertical viewport was given unbounded height.
I/flutter (28416): Viewports expand in the scrolling direction to fill their container.In this case, a vertical
I/flutter (28416): viewport was given an unlimited amount of vertical space in which to expand. This situation
I/flutter (28416): typically happens when a scrollable widget is nested inside another scrollable widget.
I/flutter (28416): If this widget is always nested in a scrollable widget there is no need to use a viewport because
I/flutter (28416): there will always be enough vertical space for the children. In this case, consider using a Column
I/flutter (28416): instead. Otherwise, consider using the "shrinkWrap" property (or a ShrinkWrappingViewport) to size
I/flutter (28416): the height of the viewport to the sum of the heights of its children.
I/flutter (28416):
I/flutter (28416): When the exception was thrown, this was the stack:
I/flutter (28416): #0      RenderViewport.performResize.<anonymous closure> (package:flutter/src/rendering/viewport.dart:1147:15)
I/flutter (28416): #1      RenderViewport.performResize (package:flutter/src/rendering/viewport.dart:1200:6)
I/flutter (28416): #2      RenderObject.layout (package:flutter/src/rendering/object.dart:1604:9)
```

Figure 6.6: Layout error in the Debug Console

These types of errors can be pretty unsettling to see and don't always immediately tell you how to fix your code. There is also a long explosion of log entries that have nothing to do with your code. When you see this error, it just means that you have an unbounded scrolling widget. If you place a scrolling widget inside a `Flex` widget (like `Row` or `Column`), which is pretty common, just don't forget to always wrap the scrolling content in `Expanded` or `Flexible` first.

Use this chart as a reference when you're designing your scrolling widget trees:

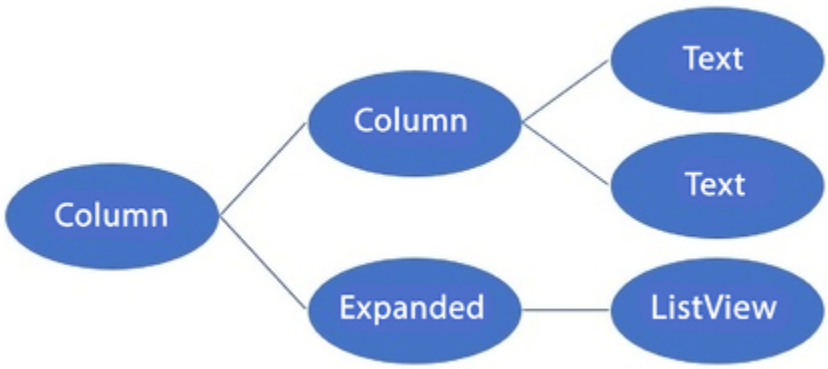


Figure 6.7: ListView in Expanded diagram

Handling large datasets with list builders

There is an interesting trick that mobile apps use when they need to render lists of data that can potentially contain more entries than your device has memory to display. This was especially critical in the early days of mobile app development, when phones were a lot less powerful than they are today. Imagine that you had to create a contacts app, where your user could potentially have hundreds and hundreds of scrollable contacts. If you put them all in a single `ListView` and asked Flutter to create all of these widgets, there would be a point where your app could run out of memory, slow down, and even crash.

Take a look at the contacts app on your phone and scroll up and down really fast. These apps don't show any *delay* while scrolling, and they certainly aren't in any danger of crashing because of the amount of data. What's the secret? If you look carefully at your app, you'll see that only so many items can fit on the screen at once, regardless of how many entries there are in your list. So, some smart engineers figured out they can *recycle* these views. When a widget moves off screen, why not reuse it with the new data? This trick has existed since the beginning of mobile development and is no different today.

In this recipe, we're going to optimize the stopwatch app from the previous recipe to employ recycling when our dataset grows beyond what our phones can handle.

How to do it...

Open the `stopwatch.dart` file and dive right into the `ListView` code:

1. Let's replace `ListView` with one of its variants, `ListView.builder`. Replace the existing implementation of `_buildLapDisplay` with this one:

```
Widget _buildLapDisplay() {
  return ListView.builder(
    itemCount: laps.length,
    itemBuilder: (context, index) {
      final milliseconds = laps[index];
      return ListTile(
```

```
        contentPadding:
EdgeInsets.symmetric(horizontal: 50),
        title: Text('Lap ${index + 1}'),
        trailing: Text(_secondsText(milliseconds)),
      );
    },
  );
}
```

2. `ScrollViews` can get too big, so it's usually a good idea to show the user their position in the list. Wrap `ListView` in a `Scrollbar` widget. There aren't any special properties to enter since this widget is entirely context aware:

```
return Scrollbar(
  child: ListView.builder(
    itemCount: laps.length,
```

3. Finally, add a quick new feature for the list to scroll to the bottom every time the lap button is tapped. Flutter makes this easy with the `ScrollController` class. At the top of `StopWatchState`, just below the laps list, add these two properties:

```
final itemHeight = 60.0;
final scrollController = ScrollController();
```

4. Now, we need to link these values with `ListView` by feeding them into the widget's constructor:

```
ListView.builder(
  controller: scrollController,
  itemExtent: itemHeight,
  itemCount: laps.length,
  itemBuilder: (context, index) {
```

5. In the `dispose()` method, let's also call `scrollController.dispose()`:

```
@override
   void dispose() {
    timer.cancel();
    scrollController.dispose();
```

```
    super.dispose();
  }
```

All we have to do now is tell `ListView` to scroll when a new lap is added.

6. At the bottom of the `_lap()` method, just after the call to `setState`, add this line:

```
scrollController.animateTo(
  itemHeight * laps.length,
  duration: const Duration(milliseconds: 500),
  curve: Curves.easeIn,
);
```

Run the app and try adding several laps. Your app can now handle virtually any number of laps without effort.

How it works...

To build an optimized `ListView` with its `builder` constructor, you need to tell Flutter how large the list is via the `itemCount` property. If you don't include it, Flutter will think that the list is infinitely long and it will never terminate. There may be a few cases where you want to use an infinite list, but they are rare. In most cases, you need to tell Flutter how long the list is; otherwise, you will get an "out of bounds" error.

The secret to scrolling performance is found in the `itemBuilder` closure. In the previous recipe, you added a list of known children to `ListView`. This forces Flutter to create and maintain the entire list of widgets. Widgets themselves are not that expensive, but the `Elements` and `RenderObjects` properties that sit underneath the widgets inside Flutter's internals are.

`itemBuilder` solves this problem by enabling **deferred rendering**. We are no longer providing Flutter with a list of widgets. Instead, we are waiting for Flutter to use what it needs and only creating widgets for a *subset* of our list.

As the user scrolls, Flutter will continuously call the `itemBuilder` function with the appropriate index. When widgets move off the screen, Flutter can remove them from the tree, freeing up precious memory. Even if our list is thousands of entries long, the size of the viewport is not going to change, and we are only going to need the same fixed number of visible entries at a time. The following diagram exemplifies this point:



Figure 6.8: Scrolling Viewport

As this viewport moves up and down the list, only seven items can fit on the screen at a time. Subsequently, nothing is gained by creating widgets for all 20 items. As the viewport moves to the left, we will probably need items 3, 2, and 1, but items 8, 9, and 10 can be dropped. The internals for how all this is executed are handled by Flutter. There is actually no API access to how Flutter optimizes your `ListView`. You just need to pay attention to the index that Flutter is requesting from `itemBuilder` and return the appropriate widget.

There's more...

We also touched on two more advanced scrolling topics in this recipe: `itemExtent` and `ScrollController`.

The `itemExtent` property is a way to supply a fixed height to all the items in `ListView`. Instead of letting the widget figure out its own height based on the content, using the `itemExtent` property will enforce a fixed height for every item. This has added performance benefits, since `ListView` now needs to do less work when laying out its children, and it also makes it easier to calculate scrolling animations.

`ScrollController` is a special object that allows to interact with a `ListView` from outside the build methods. This is a frequently used pattern in Flutter where you can optionally provide a controller object that has

methods to manipulate its widget. `ScrollController` can do many interesting things, but in this recipe, we just used it to animate `ListView` from the `_lap` method:

```
scrollController.animateTo(
  itemHeight * laps.length,
  duration: Duration(milliseconds: 500),
  curve: Curves.easeIn,
);
```

The first property of this function wants to know where in `ListView` to scroll. Since we have previously told Flutter that these items are all going to be of a fixed height, we can easily calculate the total height of the list by multiplying the number of items by the fixed height constant. The second property dictates the length of the animation, while the final property tells the animation to slow down as it reaches its destination instead of stopping abruptly. We will discuss animations in detail later in this book, in *Chapter 12*, *Adding Animations to Your Apps*.

Working with TextFields

Together with buttons, another extremely common form of user interaction is the text field. There comes a point in most apps where your users need to type something; for example, a form where users type in their username and password.

Because the text is often related to the concept of forms, Flutter also has a subclass of `TextField` called `TextFormField`, which adds functionality for multiple text fields to work together.

In this recipe, we're going to create a login form for our stopwatch app so that we know which runner we're timing.

Getting ready

Once again, we're going to continue with the `StopWatch` project. You should have completed the previous recipes in this chapter before following

along with this one.

In the `main.dart` file, in the home property of `MaterialApp`, add a call to the `LoginScreen` class. We will be creating this in this recipe:

```
home: LoginScreen(),
```

How to do it...

We're going to take a small break from the stopwatch for this recipe, and create a login screen:

1. Create a new file called `login_screen.dart` and import the `material.dart` library.
2. Generate the code snippet for a new `StatefulWidget` by typing `stful` and tapping *Enter*. Your IDE will automatically create a placeholder widget and its state class.
3. Name this class `LoginScreen`. The `State` class will automatically be named `_LoginScreenState`.
4. Our login screen needs to know whether the user is logged in to show the appropriate widget tree. We can handle this by having a boolean property called `loggedIn` and forking the widget tree accordingly.
5. Add the following code just under the class declaration of `_LoginScreenState`:

```
bool loggedIn = false;
String name = '';
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Login'),
    ),
    body: Center(
      child: loggedIn ? _buildSuccess() :
_buildLoginForm(),
    ),
```

```
  );
}
```

6. The success widget is pretty simple—it's just a checkmark and a `Text` widget to print whatever the user typed. Add the `_buildSuccess` method at the bottom of the `_LoginScreenState` class:

```
Widget _buildSuccess() {
  return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        const Icon(Icons.check, color:
Colors.orangeAccent),
        Text('Hi $name')
      ],
    );
}
```

7. Now that that's done, we can get into the main part of this recipe: the login form.
8. Add some more properties at the top of the `_LoginScreenState` class; that is, two `TextEditingControllers` to handle our `TextField` properties and `GlobalKey` to handle our `Form`:

```
final _nameController = TextEditingController();
final _emailController = TextEditingController();
final _formKey = GlobalKey<FormState>();
```

9. Override the dispose() method and dispose of the two TextEditingController widgets:

```
  @override
  void dispose() {
    _emailController.dispose();
    _nameController.dispose();
    super.dispose();
  }
```

10. We can implement the form using Flutter's `form` widget to wrap a `column`:

```
Widget _buildLoginForm() {
  return Form(
    key: _formKey,
    child: Padding(
         padding: const EdgeInsets.all(20.0),
       child: Column(
       mainAxisAlignment: MainAxisAlignment.center,
       children: [
         TextFormField(
           controller: _nameController,
           decoration: const InputDecoration(labelText:
'Runner'),
           validator: (text) =>
             text!.isEmpty ? 'Enter the runner\'s name.'
 : null,
           ),
       ],
)));}
```

11. Add the import to `login_screen.dart` in the `main.dart` file then run the app. You will see a single `TextField` floating in the center of the screen.

12. Now, let's add one more field to manage the user's email address. Add this widget inside `Column`, just after the first `TextFormField`. This widget uses a regular expression to validate its data:

```
TextFormField(
  controller: _emailController,
    keyboardType: TextInputType.emailAddress,
    decoration: const InputDecoration(labelText:
'Email'),
       validator: (text) {
          if (text!.isEmpty) {
            return 'Enter the runner\'s email.';
          }
          final regex = RegExp('[^@]+@[^.]+..+');
          if (!regex.hasMatch(text)) {
             return 'Enter a valid email';
          }
          return null;
  },
),
```

Note

Regular expressions are sequences of characters that specify a search pattern, and they are often used for input validation.

To learn more about regular expressions in Dart, go to https://api.dart.dev/stable/2.12.4/dart-core/RegExp-class.html.

13. The form items should be all set up; now, you just need a way to validate them. You can accomplish this with a button and function that calls the form's `validateAndSubmit` method. Add these two widgets inside the same `Column`, just after the second `TextField`:

```
const SizedBox(height: 20),
ElevatedButton(
  onPressed: _validate,
  child: const Text('Continue'),
),
```

14. Now, implement the `validate()` method:

```
void _validate() {
  final form = _formKey.currentState;
    if (form?.validate() ?? false) {
      return;
    }
    setState(() {
      loggedIn = true;
      name = _nameController.text;
    });
}
```

15. Perform a hot reload. As a bonus, try entering incorrect information into the form and see what happens:

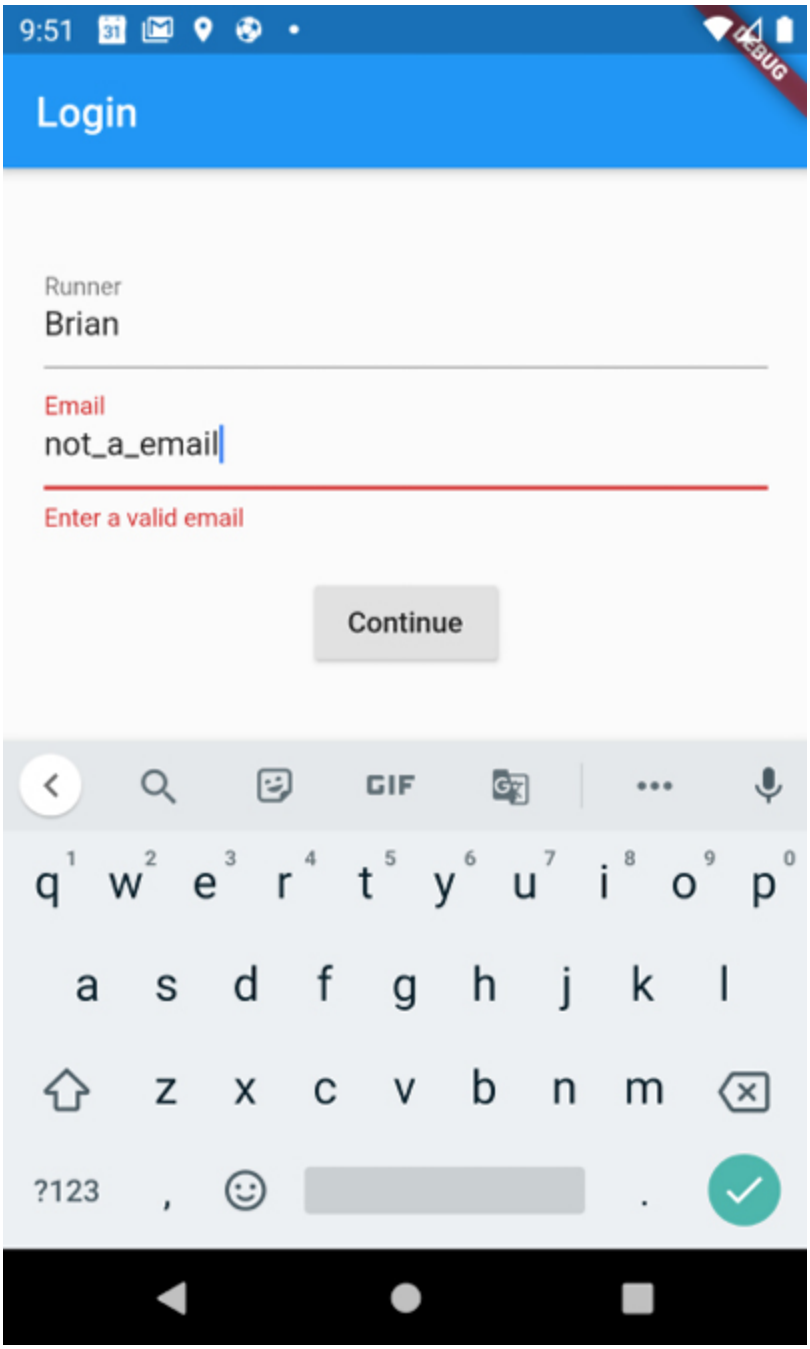


Figure 6.9: Form validation

How it works...

This recipe actually covered quite a few topics very quickly: `TextFields`, `Forms`, and `Keys`.

`TextFields` are platform-aware widgets that respect the host platform's UX paradigms. As with most things in Flutter, the look of `TextFields` is highly customizable. The default look respects the material design rules, but it can be fully customized using the `InputDecoration` property.

> **Note**
>
> TextField is a material design widget respecting the host platform UX paradigms, like selection and typography (https://docs.flutter.dev/resources/platform-adaptations).
>
> For an iOS styled text field you should use https://api.flutter.dev/flutter/cupertino/CupertinoTextField-class.html.

By now, you should be noticing some common patterns in Flutter's API. Many widgets—`Containers`, `TextFields`, `DecoratedBox`, and so on—can all accept a decoration object. It could even be argued that the consistency of the API design for these widgets has led to a sort of self-documentation. For example, can you guess what this line does to the second `TextField`?

```
keyboardType: TextInputType.emailAddress,
```

If you guessed that it lets us use the email keyboard instead of the standard keyboard, then congratulations, that's correct!

In this recipe, you used a variant of `TextField` called `TextFormField`. This subclass of `TextField` adds some extra validation callbacks that are called when submitting a form.

The first `validator` is simple – it just checks if the text is empty. Validator functions should return a string if the validation fails. If the validation is successful, then the function should return a `null`. This is one of the very few cases in the entire Flutter SDK where `null` is a good thing.

The `Form` widget that wraps the two `TextFields` is a non-rendering container widget. This widget knows how to visit any of its children that are

`FormFields` and invokes their validators. If all the validator functions return `null`, the form is considered **valid**.

You used a `GlobalKey` to get access to the form's state class from outside the `build` method. `GlobalKeys` uniquely identify elements. While `BuildContext` is an object that can find **parents** in the widget tree, `GlobalKeys` are objects that you use to retrieve a **child** widget. The topic is a bit more complex than that, but in short, with the key, you can retrieve the Form's state. The `FormState` class has a public method called `validate` that will call the validator on all its children.

> **Note**
>
> Keys go much deeper than this. However, they are an advanced topic that is outside the scope of this book. There is a link to an excellent article by Google's Emily Fortuna about keys in the *See also* section of this recipe if you want to learn more about this topic.

Finally, we have `TextEditingController`. Just like `ScrollController` in the previous recipe, `TextEditingControllers` are objects that can be used to manipulate `TextFields`. In this recipe, we only used them to extract the current value from our `TextField`, but they can also be used to programmatically set values in the widget, update text selections, and clear the fields. They are very helpful objects to keep in your toolbox.

There are also some callback functions that can be used on `TextFields` to accomplish the same thing. For instance, if you want to update the name property every time the user types a letter, you could use the `onChanged` callback that is on the core `TextField` (not `TextFormField`). In practice, having lots of callbacks and inline closures can make your functions very long and hard to read. So, while it may seem easier to use closures instead of `TextEditingControllers`, it could make your code harder to read. Clean code suggests that functions should only strive to do one thing, which means one function should handle the setup and aesthetic of your `TextView` and another function should handle the logic.

See also

Check out these resources for more information:

- An article and video about keys by Emily Fortuna — don't miss this one!: https://medium.com/flutter/keys-what-are-they-good-for-13cb51742e7d
- If you want to learn more about Forms have a look at: https://api.flutter.dev/flutter/widgets/Form-class.html

Navigating to the next screen

So far, all our examples have taken place on a single screen. In most real-world projects, you might be managing several screens, each with their own paths that can be pushed and popped onto the screen.

Flutter, and more specifically `MaterialApp`, uses a class called `Navigator` to manage your app's screens. Screens are abstracted into a concept called `Routes`, which contain both information about the widget we want to show and how we want to animate it on the screen. `Navigator` also keeps a full history of your routes so that you can return to the previous screens easily.

In this recipe, we're going to link `LoginScreen` and `StopWatch` so that `LoginScreen` actually logs you in.

How to do it...

Let's start linking the two screens in the app:

1. Start by engaging in one the most enjoyable activities for a developer —deleting code! Remove the `loggedIn` property and all the parts of the code where it's referenced. We're also no longer going to need the `buildSuccess()` method or the `ternary` method in the top `build` method.

2. Update the `build` method with the following snippet:

```
Widget build(BuildContext context) {
  return Scaffold(
      appBar: AppBar(
        title: const Text('Login'),
      ),
      body: Center(
        child: _buildLoginForm(),
      ),
    );
}
```

3. In the `_validate` method, we can kick off the navigation instead of calling `setState`:

```
void _validate() {
  final form = _formKey.currentState;
  if (form?.validate() == false) {
    return;
  }
  final name = _nameController.text;
  final email = _emailController.text;
  Navigator.of(context).push(
    MaterialPageRoute(
      builder: (_) => StopWatch(name: name, email:
email),
    ),
  );
}
```

4. The constructor for the `StopWatch` widget needs to be updated so that it can accept the name and email. Make these changes in `stopwatch.dart`:

```
class StopWatch extends StatefulWidget {
  final String name;
  final String email;
  const StopWatch({super.key, required this.name,
required this.email});
```

5. In the `build` method of the `StopWatchState` class, replace the title of `AppBar` with the runner's name, just to give the app a bit more

personality:

```
AppBar(
  title: Text(widget.name),
),
```

6. Run the app. You can now navigate back and forth between `LoginScreen` and `StopWatch`. However, isn't it a little strange that you can hit a back button to return to a login screen? In most apps, once you've logged in, that screen should no longer be accessible.
7. Use the Navigator's `pushReplacement` method to replace the current `Route` in the navigation stack:

```
Navigator.of(context).pushReplacement(
    MaterialPageRoute(
      builder: (_) => StopWatch(name: name, email:
email),
    ),
  );
```

8. Try logging in again. You will see that you no longer have the ability to return to the login screen.

How it works...

`Navigator` is a component of both `MaterialApp` and `CupertinoApp`. Accessing this object is another example of the *of-context* pattern. Internally, Navigators function as a stack. Routes can be pushed onto the stack and popped off the stack.

Normally, you would just use the standard `push()` and `pop()` methods to add and remove routes, but as we discussed in this recipe, we didn't just want to push `StopWatch` onto the screen—we also wanted to pop `LoginScreen` from the stack at the same time.

To accomplish this, we used the `pushReplacement` method:

```
Navigator.of(context).pushReplacement(
    MaterialPageRoute(
```

We also used the `MaterialPageRoute` class to represent our routes. This object will create a platform-aware transition between the two screens. On iOS, it will push onto the screen from right, while on Android, it will pop onto the screen from the bottom.

Similar to `ListView.builder`, `MaterialPageRoute` also expects a `WidgetBuilder` instead of direct child. `WidgetBuilder` is a function that provides a `BuildContext` and expects a `Widget` to be returned:

```
builder: (_) => StopWatch(name: name, email: email),
```

This allows Flutter to delay the construction of the widget until it's needed. We also didn't need the context property, so it was replaced with an underscore.

Showing dialogs on the screen

Dialogs, or popups, are used when you want to give a message to your users that needs their attention. This ranges from telling the user about some error that occurred or asking them to perform some action before continuing, or even giving them a warning.

Note

As alerts require some feedback from the user, you should use them for important information prompts or for actions that require immediate attention: in other words, only when really necessary.

The following are the default alerts for Android and iOS:

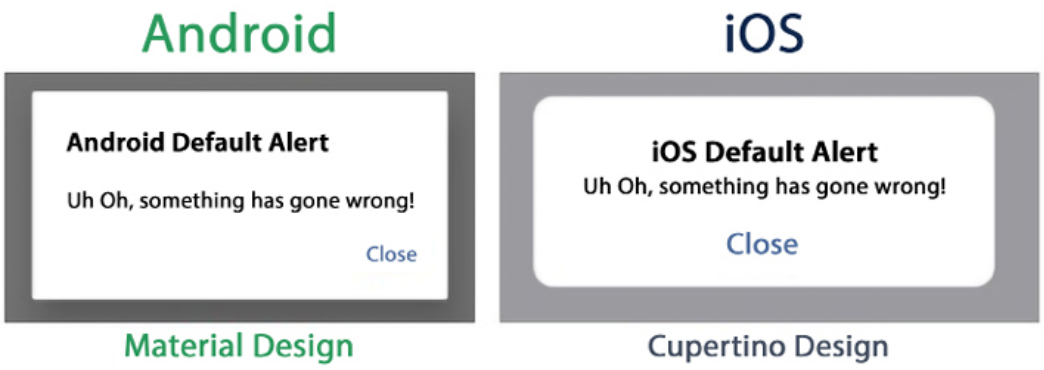


Figure 6.10: Alerts on Android and iOS

In this recipe, we're going to create a platform-aware alert and use it to show a prompt when the user stops the stopwatch.

Getting ready

Once again, we're going to continue with the `StopWatch` project. You should have completed the previous recipes in this chapter before following along with this one.

How to do it...

We will work with a new file in our project, called `platform_alert.dart`. Let's get started:

1. Open this new file and create a constructor that will accept a title and message body. This class is just going to be a simple Dart object:

```
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
class PlatformAlert {
  final String title;
  final String message;
  const PlatformAlert({required this.title, required
this.message});}
```

2. `PlatformAlert` is going to need a `show` method that will look at the app's context to determine what type of device it's running on and then show the appropriate dialog widget.
3. Add this method just after the constructor:

```
void show(BuildContext context) {
  final platform = Theme.of(context).platform;
  if (platform == TargetPlatform.iOS) {
    _buildCupertinoAlert(context);
  } else {
    _buildMaterialAlert(context);
  }
}
```

4. Showing an alert only requires invoking a global function called `showDialog`, which, just like `Navigator`, accepts a `WidgetBuilder` closure.

> **Note**
>
> The `showDialog` method returns a `Future<T>`, meaning that it can return a value that you can deal with later. In the example that follows, we do not need to listen to the user response as we are only giving some information, so the return types of the methods will just be `void`.

5. Implement the `_buildMaterialAlert` method with the following code:

```
void _buildMaterialAlert(BuildContext context) {
  showDialog(
      context: context,
      builder: (context) {
        return AlertDialog(
            title: Text(title),
            content: Text(message),
            actions: [
              TextButton(
                  child: const Text('Close'),
                  onPressed: () =>
Navigator.of(context).pop())
```

```
            ]);
      });
}
```

6. The iOS version is very similar—we just need to swap out the material components with their Cupertino counterparts. Add this method immediately after the material builder:

```
void _buildCupertinoAlert(BuildContext context) {
  showCupertinoDialog(
      context: context,
      builder: (context) {
        return CupertinoAlertDialog(
            title: Text(title),
            content: Text(message),
            actions: [
              CupertinoButton(
                  child: const Text('Close'),
                  onPressed: () =>
Navigator.of(context).pop())
            ]);
      });
}
```

7. You now have a platform-aware class that wraps both `AlertDialog` and `CupertinoAlertDialog`. You can test this out in the `stopwatch.dart` file. Show a dialog when the user stops the stopwatch that shows the total time elapsed, including all the laps. Add the following code to the bottom of the `_stopTimer` method, after the call to `setState`:

```
final totalRuntime = laps.fold(milliseconds, (total,
lap) => total + lap);
final alert = PlatformAlert(
  title: 'Run Completed!',
  message: 'Total Run Time is
${_secondsText(totalRuntime)}.',
);
alert.show(context);
```

8. Run the app and run a couple of laps. A `Dialog` will now show you the total of all the laps when you press stop. As an added bonus, try running the app on both the iOS Simulator and Android Emulator.

Notice how the UI changes to respect the platform's standards, as shown here:

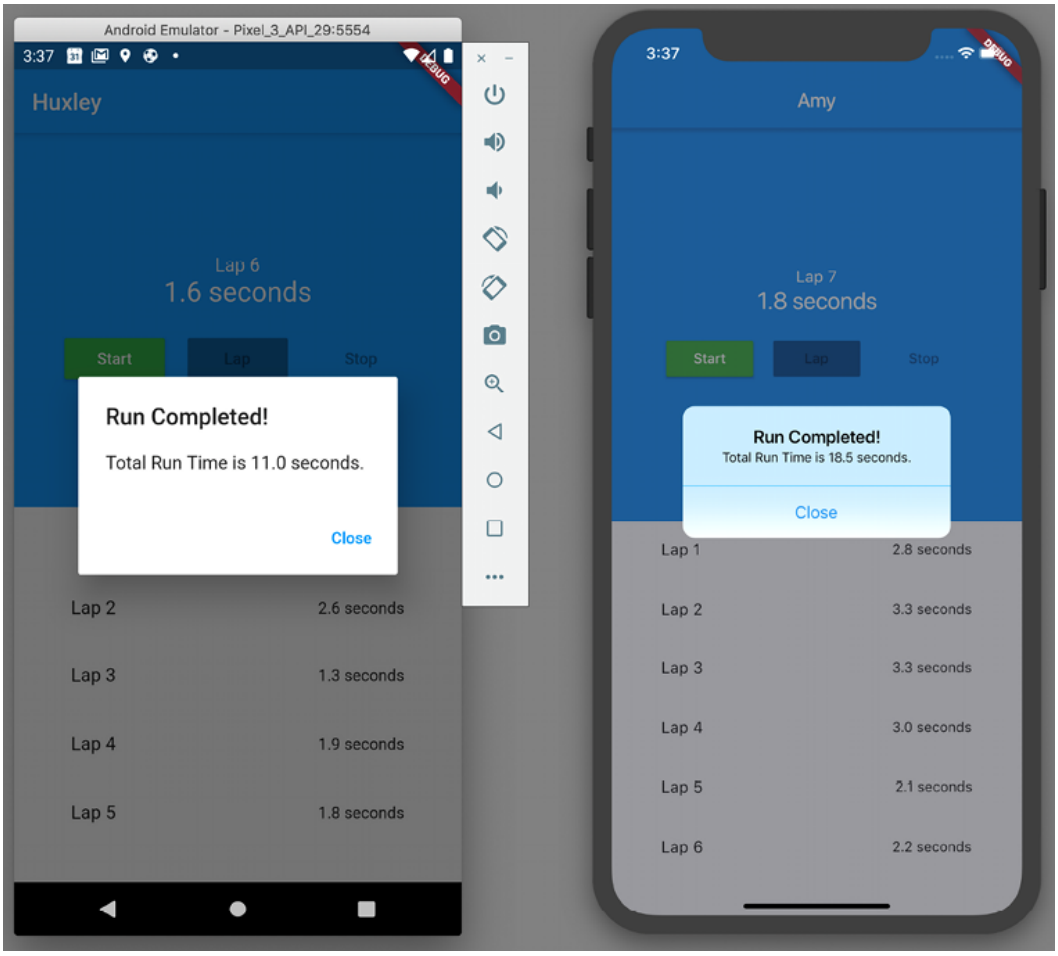


Figure 6.11: Alerts showing on a Pixel and an iPhone

How it works...

The way Flutter handles dialogs is very simple. Dialogs are just routes. The only difference between a `MaterialPageRoute` and a `Dialog` is the animation that Flutter uses to display them. Since dialogs are just routes, they use the same `Navigator` API for pushing and popping. This is accomplished by calling the `showDialog` or `showCupertinoDialog` global function. Both of these functions will look for the app's `Navigator` and push a route onto the navigation stack using the platform-appropriate animation.

An alert, whether Material or Cupertino, is made up of three components:

- `title`
- `content`
- `actions`

The `title` and `content` properties are just widgets. Typically, you would use a `Text` widget, but that's not required. If you want to put an input form and a scrolling list in a `Center` widget, you could certainly do that.

The actions are also *usually* a list of buttons, where users can perform an appropriate action. In this recipe, we used just one that will close the dialog:

```
TextButton(
  child: Text('Close'),
  onPressed: () => Navigator.of(context).pop())
```

Note that closing the dialog is just a standard call to the `Navigator` API. Since dialogs are routes, we can treat them identically. On Android, the system's back button will even pop the dialog from the stack, just as you would expect.

There's more...

In this recipe, we also used the app's theme to determine the host platform. The `ThemeData` object has an enum called `TargetPlatform` that shows the potential options where Flutter can be hosted. In this recipe, we are only dealing with mobile platforms (iOS and Android), but currently, there are several more options in this enum, including desktop platforms. This is the current implementation of `TargetPlatform`:

```
  enum TargetPlatform {
  /// Android: <https://www.android.com/>
  android,
  /// Fuchsia: <https://fuchsia.dev/fuchsia-src/concepts>
  fuchsia,
```

```
  /// iOS: <https://www.apple.com/ios/>
  iOS,
  /// Linux: <https://www.linux.org>
  linux,
  /// macOS: <https://www.apple.com/macos>
  macOS,
  /// Windows: <https://www.windows.com>
  windows,
}
```

An interesting option here is `fuchsia`. **Fuchsia** is an experimental operating system that is currently under development at Google. It has been suggested that, at some point in the future, Fuchsia might replace Android. When (or if) that happens, the primary application layer for Fuchsia will be Flutter. So, congratulations—you are already covertly a Fuchsia developer! It's still relatively early days for this operating system, but this confirms that no matter what happens, the future of Flutter looks bright.

Presenting bottom sheets

There are times where you need to present modal information, but a dialog just comes on too strong. Fortunately, there are quite a few alternative conventions for putting information on the screen that do not necessarily require action from users. Bottom sheets are one of the "gentler" alternatives to dialogs, together with `Toasts`. With a bottom sheet, information slides out from the bottom of the screen, where it can be swiped down by the user if it displeases them. Also, unlike alerts, bottom sheets do not block the main interface, allowing users to conveniently ignore these messages.

In this final recipe for the stopwatch app, we're going to replace the dialog alert with a bottom sheet and animate it away after 5 seconds.

Getting ready

You should have completed the previous recipes in this chapter before following along with this one.

How to do it...

Open `stopwatch.dart` to get started:

1. The bottom sheet API is not dramatically different from dialogs. The global function expects a `BuildContent` and a `WidgetBuilder`. Create that builder as its own function.
2. Add the following code under the `_stopTimer` method:

```
Widget _buildRunCompleteSheet(BuildContext context) {
  final totalRuntime = laps.fold(milliseconds, (total,
lap) => total + lap);
  final textTheme = Theme.of(context).textTheme;
  return SafeArea(
    child: Container(
      color: Theme.of(context).cardColor,
      width: double.infinity,
      child: Padding(
          padding: const EdgeInsets.symmetric(vertical:
30.0),
          child: Column(mainAxisSize: MainAxisSize.min,
children: [
            Text('Run Finished!', style:
textTheme.headlineSmall),
  Text('Total Run Time is
${_secondsText(totalRuntime)}.'
)
          ])),
    )
  );
}
```

3. Showing the bottom sheet should now be remarkably easy. In the `_stopTimer` method, delete the code that shows the dialog and replace it with an invocation to `showBottomSheet`:

```
void _stopTimer(BuildContext context) {
  setState(() {
    timer.cancel();
    isTicking = false;
  });
  showBottomSheet(context: context, builder:
```

```
    _buildRunCompleteSheet);
}
```

4. Try running the code now and tap the stop button to present the sheet. Did it work? I guess it didn't. Actually, you might even be seeing the following error being printed to the console:

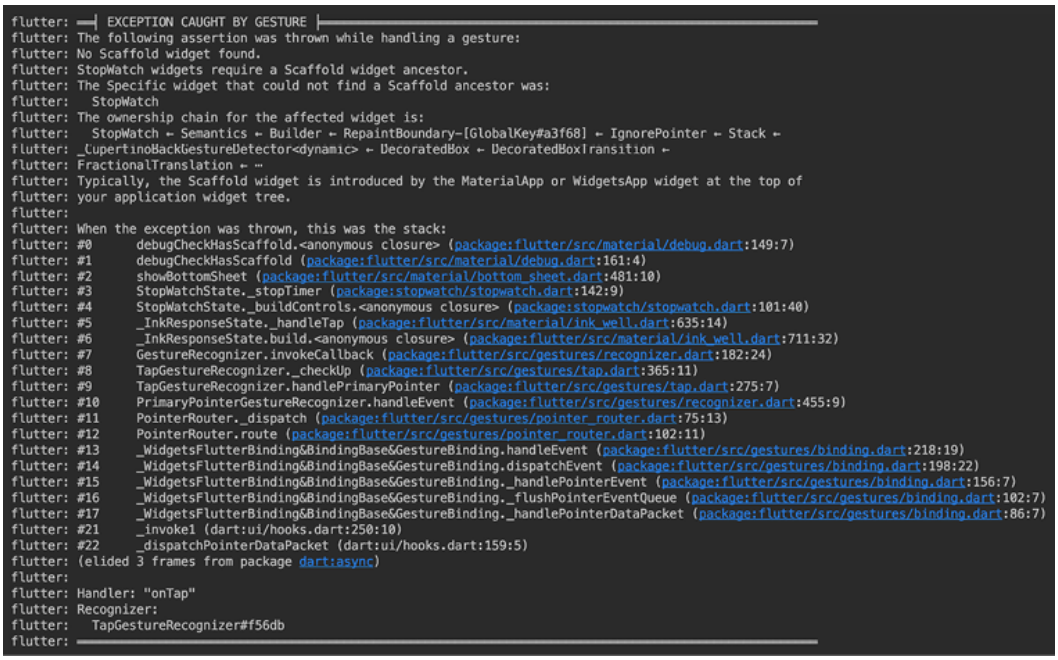


Figure 6.12: Error caused by BuildContext

5. Read this message carefully. This scary looking stack trace is saying that the context that we're using to present the bottom sheet requires a `Scaffold`, but it cannot find it. This is caused by using a `BuildContext` that is too high in the tree.
6. We can fix this by wrapping the stop button with a `Builder` and passing that new context to the `stop` method. In `_buildControls`, replace the stop button with the following code:

```
Builder(
  builder: (context) => TextButton(
    child: Text('Stop'),
    onPressed: isTicking ? () => _stopTimer(context) :
null,
  ...
```

7. We also have to update the `_stopTimer` method so that it accepts a `BuildContext` as a parameter:

```
void _stopTimer(BuildContext context) {...}
```

8. Hot reload the app and stop the timer. The bottom sheet now automatically appears after you hit the stop button. But it just stays there forever. It would be nice to have a short timer that automatically removes the bottom sheet after 5 seconds. We can accomplish this with the `Future` API. In the `_stopTimer` method, update the call so that it shows the bottom sheet, like so:

```
final controller =
    showBottomSheet(context: context, builder:
      _buildRunCompleteSheet);
Future.delayed(const Duration(seconds: 5)).then((_) {
  controller.close();
});
```

9. Hot reload the app. The bottom sheet now politely excuses itself after 5 seconds.

How it works...

The bottom sheet part of this recipe should be pretty simple to understand, but what's going on with that error? Why did showing the bottom sheet initially fail? Take a look at how we organized the widget tree for the stopwatch screen:

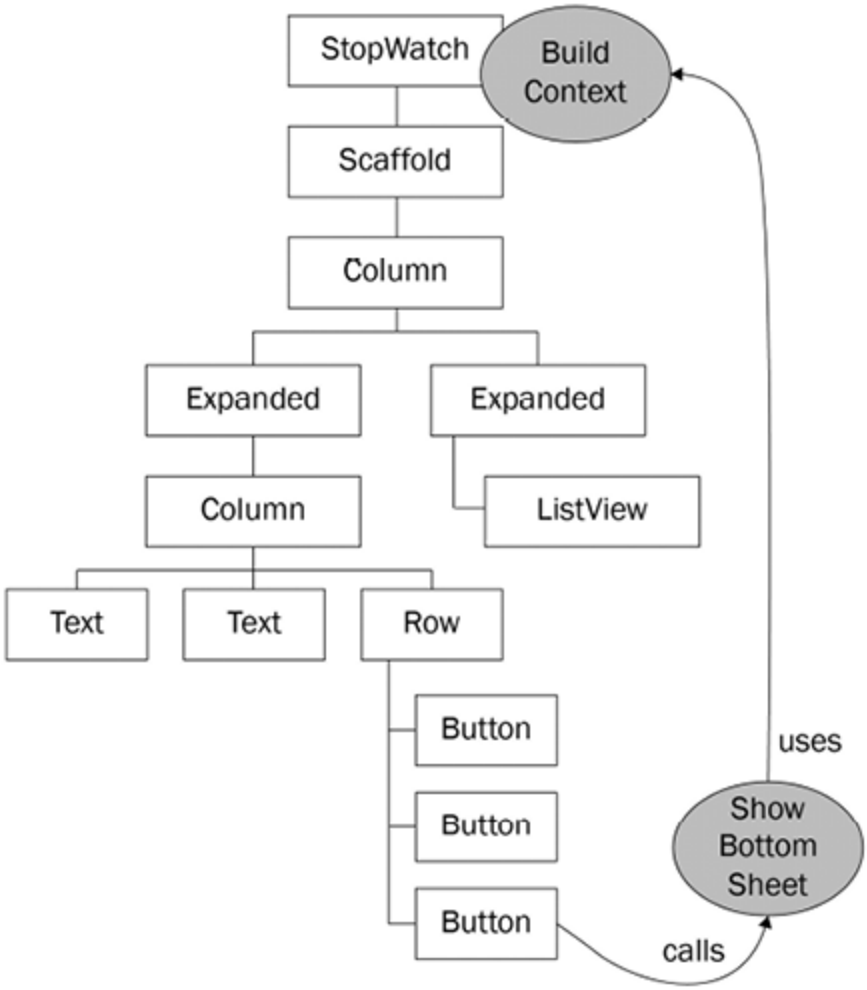


Figure 6.13: Widget tree and BuildContext

Bottom sheets are a little different from dialogs in that they are not full routes. For a bottom sheet to be presented, it attaches itself to the closest `Scaffold` in the tree using the same `of-context` pattern to find it. The problem is that the `BuildContent` class that we've been passing around and storing as a property on the `StopWatchState` class belongs to the top-level `StopWatch` widget. The `Scaffold` widget that we're using for this screen is a **child** of `StopWatch`, not a parent.

When we use `BuildContext` in the `showBottomSheet` function, it travels upward from that point to find the closest scaffold. The problem is that there aren't any scaffolds above `StopWatch`. It's only going to find a `MaterialApp` and our root widget. Consequently, the call fails.

The solution is to use a `BuildContext` that is **lower** in the tree so that it can find our `Scaffold`. This is where the `Builder` widget comes in handy. `Builder` is a widget that doesn't have a `child` or `children`, but a `WidgetBuilder`, just like routes and bottom sheets. By wrapping the button into a builder, we can grab a different `BuildContext`, one that is certainly a child of `Scaffold`, and use that to successfully show the bottom sheet:

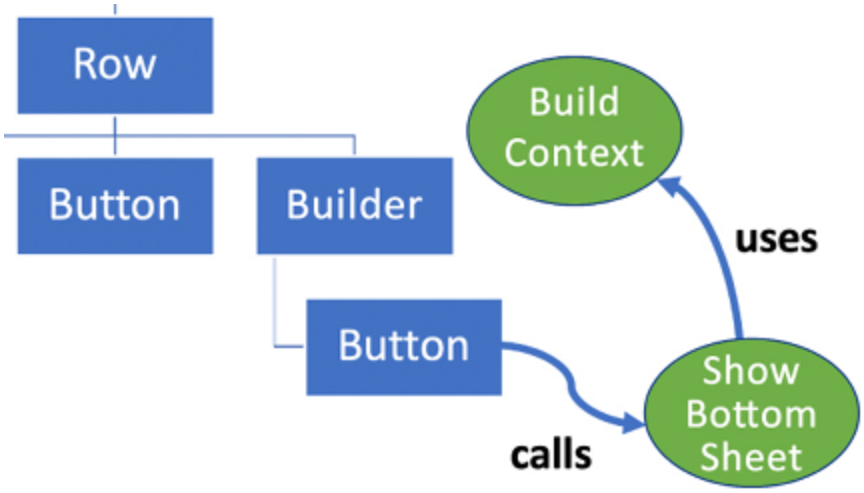


Figure 6.14: New widget tree with a Builder

This is one of the more interesting problems that you can come across when designing widget trees. It's important to keep the structure of the tree in mind when passing around the `BuildContext` class. Sometimes, the root context that you get from the widget's `build` method is not the context you are looking for.

The `showBottomSheet` method also returns a `PersistentBottomSheetController`, which is just like a `ScrollController` or `TextEditingController`. These "controller" classes are attached to widgets and have methods to manipulate them. In this recipe, we used Dart's `Future` API to call the `close` method after a 5-second delay. We will cover `Futures` later in this book in *Chapter 8*, *The Future is Now: Introduction to Asynchronous Programming*.

See also

The Google team created a short series of videos, called “Flutter Widgets 101” that explain the different types of widgets and how to use them. Have a look at it if you want to see other examples on using widgets:

- *Widgets 101*: https://www.youtube.com/playlist?list=PLOU2XLYxmsIJyiwUPCou_OVTpRIn_8UMd

Summary

In this chapter, you built a fully functioning Stopwatch app that can track laps and display a history of each completed lap.

You’ve seen an introduction to `StatefulWidget` and the basics of managing state within a Flutter app, enabling the creation of interactive widgets.

You’ve learned how to create and customize different kinds of buttons and handle button interactions using `onPressed`. You have implemented scrolling functionality using `ListView`, enabling users to smoothly scroll long lists of items.

You have explored the use of `TextField` widgets to manage the text input using `TextEditingControllers`.

You have used the `Navigator` widget and its methods, such as push, pop, and `pushReplacement`, to enable navigation between different screens.

You have seen how to display messages to your users showing alerts and bottom sheets.

This chapter showed the basics of state management and user interaction. Starting from the next chapter, you’ll see other ways to manage state in your Flutter apps.

Basic State Management

As apps grow, managing the flow of data through those apps becomes a more complex and important issue. The Flutter community has devised several solutions to deal with state management. All these solutions have one aspect they share: the separation of UI and business logic.

Before diving into any advanced state management solutions, we will lift the state of the app to a higher level.

In this chapter, you are going to build a to-do note-taking application. In the application, users will be able to create to-do lists that contain many tasks. Users will be able to add and complete their tasks.

We will cover the following recipes:

- Model-view separation
- Managing the data layer with `InheritedWidget` and `InheritedNotifier`
- Making the app state visible across multiple screens

Technical requirements

Start off by creating a brand new Flutter project called `master_plan` in your favorite IDE.

Model-view separation

Models and views are very important concepts in app architecture. **Models** are classes that deal with the data for an app, while **views** are classes that present that data on screen.

In Flutter, views are made of `widgets`. Subsequently, a model would be a basic `Dart` class that doesn't inherit from anything in the Flutter framework. Each one of these classes is responsible for one and only one job. Models are concerned with handling the data for your app. Views are concerned with drawing your interface. When you keep a clear and strict separation between your models and views, your code will become simpler and easier to read and work with.

In this recipe, we're going to build the start of our to-do app and create model classes to go along with the views.

Getting ready

Any self-respecting app architecture must ensure it has the right folder structure set up and ready to go. Create a new app and call it master_plan. Inside the `lib` folder, create both a `models` and a `views` subfolder.

Once you have created these two directories, you should see them in the `lib` folder, as shown in the following screenshot:

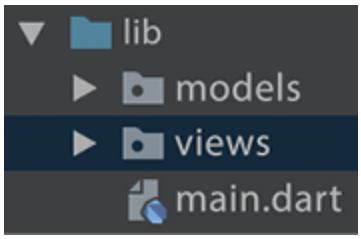


Figure 7.1: The Master Plan project lib folder

You are now ready to start working on the project.

How to do it...

To implement separation of concerns for the UI and models, follow these steps:

1. The best place to start is the data layer. This will give you a clear view of your app, without going into the details of your user interface. In the `models` folder, create a file called `task.dart` and create the `Task` class. This should have a description `String` and a complete Boolean, as well as a constructor. This class will hold the task data for our app. Add the following code:

```
class Task {
  final String description;
  final bool complete;
  const Task({
    this.complete = false,
    this.description = '',
  });
}
```

2. We also need a `plan` that will hold all our tasks. In the `models` folder, create `plan.dart` and insert this simple class:

```
import './task.dart';
class Plan {
  final String name;
  final List<Task> tasks;
  const Plan({this.name = '', this.tasks = const []});
}
```

3. We can wrap up our data layer by adding a file that will export both models. That way, our imports will not get too inflated as the app grows. Create a file called `data_layer.dart` in the `models` folder. This will only contain `export` statements, no actual code:

```
export 'plan.dart';
export 'task.dart';
```

4. Moving on to `main.dart`, we need to set up our `MaterialApp` for this project. This should hopefully be easy by now. Just delete the `MyApp` and `MyHomePage` classes and create a `StatelessWidget`, called `MasterPlanApp`, that returns a `MaterialApp` that, in its home directory, calls a widget called `PlanScreen`. We will build this shortly. The result should look like the code below:

```
import 'package:flutter/material.dart';
import './views/plan_screen.dart';
void main() => runApp(MasterPlanApp());
class MasterPlanApp extends StatelessWidget {
  const MasterPlanApp({super.key});
  @override
  Widget build(BuildContext context) {
   return MaterialApp(
      theme: ThemeData(primarySwatch: Colors.purple),
      home: PlanScreen(),
    );
  } }
```

5. With the plumbing out of the way, we can continue with the view layer. In the `views` folder, create a file called `plan_screen.dart` and use the `StatefulWidget` template to create a class called `PlanScreen`. Import the material library and build a basic `Scaffold` and `AppBar` in the `State` class. We'll also create a single plan that will be stored as a property in the `State` class:

```
import '../models/data_layer.dart';
import 'package:flutter/material.dart';
class PlanScreen extends StatefulWidget {
  const PlanScreen({super.key});
  @override
  State createState() => _PlanScreenState();
}
class _PlanScreenState extends State<PlanScreen> {
  Plan plan = const Plan();
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Master Plan')),
      body: _buildList(),
      floatingActionButton: _buildAddTaskButton(),
    );
  }
}
```

6. You'll see a few errors, as we're missing a couple of methods here. Let's start with the easier one—the **Add Task** button. This button will use the `FloatingActionButton` layout, which is a common way to add an item to a list according to the material design's specifications.

Add the following code just below the `build` method:

```
Widget _buildAddTaskButton() {
  return FloatingActionButton(
      child: const Icon(Icons.add),
      onPressed: () {
        setState(() {
          plan = Plan(
            name: plan.name,
            tasks: List<Task>.from(plan.tasks)
..add(const Task()),
          );
        });
      },
    );}
```

7. We can create a scrollable list to show all our tasks; `ListView.builder` will certainly suit our needs. Create this simple method to build our `ListView`:

```
Widget _buildList() {
  return ListView.builder(
    itemCount: plan.tasks.length,
    itemBuilder: (context, index) =>
    _buildTaskTile(plan.tasks[index], index),
  );
}
```

8. We just need to create a `ListTile` that displays the value of our task. Because we took the effort to set up a model for each task, building the view will be easy. Create a new class in the `planscreen.dart` file:

```
 Widget _buildTaskTile(Task task, int index) {
  return ListTile(
      leading: Checkbox(
          value: task.complete,
          onChanged: (selected) {
            setState(() {
              plan = Plan(
                name: plan.name,
                tasks: List<Task>.from(plan.tasks)
                  ..[index] = Task(
```

```
                    description: task.description,
                    complete: selected ?? false,
                  ),
              );
            });
        }),
      title: TextFormField(
        initialValue: task.description,
        onChanged: (text) {
          setState(() {
              plan = Plan(
                name: plan.name,
                tasks: List<Task>.from(plan.tasks)
                  ..[index] = Task(
                    description: text,
                    complete: task.complete,
                  ),
              );
            });
        },
      ),
    );
}
```

9. Run the app; you will see that everything has been fully wired up. You can add tasks, mark them as complete, and scroll through the list when it gets too long. However, there is one iOS-specific feature we need to add. Once the keyboard is open, you can't get rid of it. You can use a `ScrollController` to remove the focus from any `TextField` during a scroll event. In the `plan_screen.dart` file, add a scroll controller as a property of the `State` class, just after the `plan` property:

```
late ScrollController scrollController;
```

10. `scrollController` has to be initialized in the `initState` life cycle method. This is where you will also add the scroll listener. Add the `initState` method to the `State` class, after the `scrollController` declaration, as shown here:

```
@override
void initState() {
  super.initState();
```

```
  scrollController = ScrollController()
    ..addListener(() {
      FocusScope.of(context).requestFocus(FocusNode());
    });

}
```

11. Add the controller and the keyboard behaviour to the `ListView` in the `_buildList` method:

```
return ListView.builder(
  controller: scrollController,
keyboardDismissBehavior: Theme.of(context).platform ==
TargetPlatform.iOS
          ? ScrollViewKeyboardDismissBehavior.onDrag
          : ScrollViewKeyboardDismissBehavior.manual,
```

12. Finally, dispose of `scrollController` when the widget is removed from the tree:

```
@override
void dispose() {
  scrollController.dispose();
  super.dispose();
}
```

13. Hot restart (**not** hot reload) your app. You should see our plan coming together:

Note

Both hot reload and hot restart are faster than performing a full rebuild of the app. In general:

- Use *hot reload* for changes that update the UI, so most changes in the build method. The app state is maintained and you see your changes almost instantly.
- Use *hot restart* for changes that affect the app's state, like updating global variables, static fields, or the `main()` method. The app state is reset.

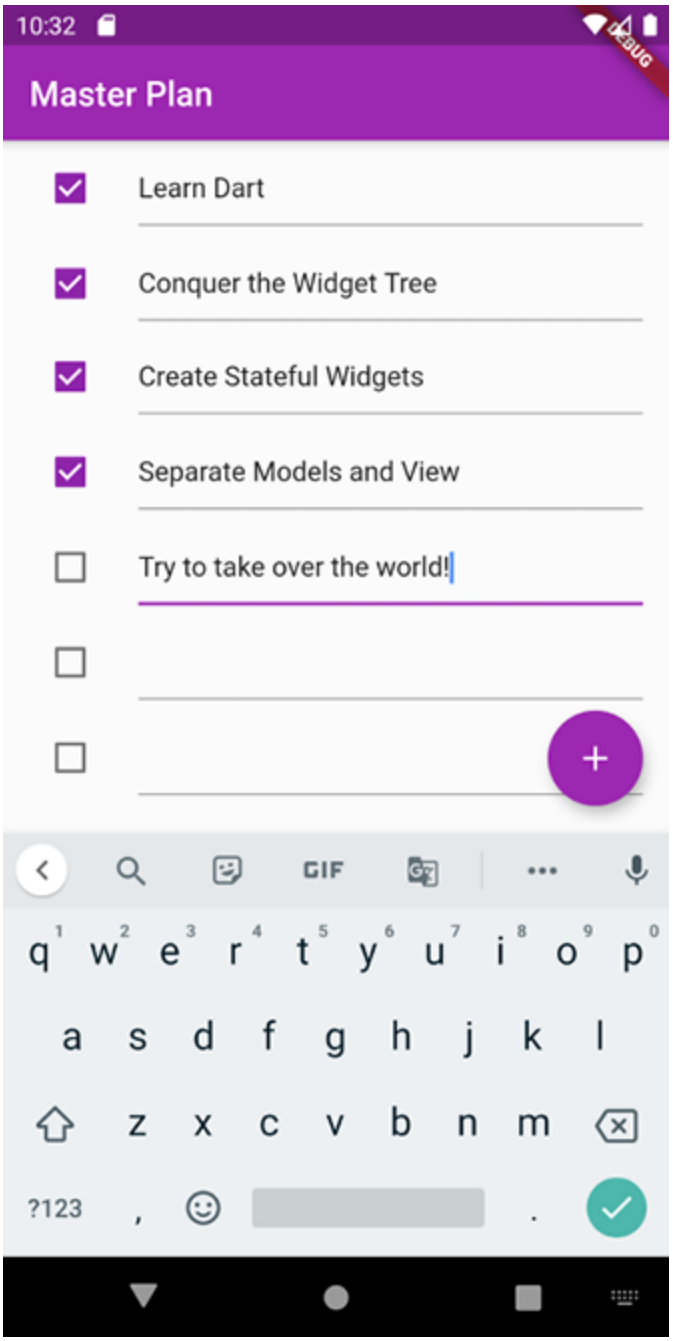


Figure 7.2: The Master Plan screen with a few tasks

How it works...

The `Task` and `Plan` objects are **immutable**: this means that once you create instances of `Task` or `Plan`, the objects cannot be changed after their creation.

You can create an immutable class using `final` and `const` fields that prevent reassignment, like in the `Task` example:

```
class Task {
  final String description;
  final bool complete;
  const Task({
    this.complete = false,
    this.description = '',
  });
}
```

In this example, `description` and `complete` are both `final`, and when you create a new `Task`, the constructor is a `const`. In other words, once you create a `Task`, you won't be able to change its properties, as it's immutable.

In an app like the one you are building in this chapter, you might want to change a task, for example, you might want to mark it complete: how can you achieve that if you cannot change its properties?

Well, **you destroy the old `Task`, and you create a new one**, like in the example below:

```
Task(
  description: task.description,
  complete: selected ?? false,
),
```

Here you are just creating a new `Task`, that will overwrite the existing one. This is more predictable than having mutable objects. In fact, immutability is considered a good pattern in Flutter, because it can improve readability and performance.

The UI in this recipe is *data-driven*. The `ListView` widget (the **`view`**) queries the `Plan` class (the **`model`**) to figure out how many items there are.

In the `itemBuilder` closure, we extract the specific `Task` that matches the item index and pass the entire model to the `buildTaskTile` method.

The tiles are also data-driven as they read the `complete` boolean value in the model to choose whether the checkbox should be checked or not.

If you look at our implementation of the `Checkbox` widget, you'll see that it takes data from the model and then returns data to the model when its state changes. This widget is truly a *view* into our data:

```
Checkbox(
  Checkbox(
          value: task.complete,
          onChanged: (selected) {
            setState(() {
              plan = Plan(
                name: plan.name,
                tasks: List<Task>.from(plan.tasks)
                  ..[index] = Task(
                    description: task.description,
                    complete: selected ?? false,
                  ),
              );
            });
          }),
```

When building the UI for each individual task, the State of these widgets is owned by the model. The UI's job is to query the model for its current state and draw itself accordingly.

In the `onChanged` callback, we take the values that are returned from the widgets/views and store them in the model. This, in turn, calls `setState`, which causes the widget to repaint with the most up-to-date data.

Note

Normally, it is not ideal to have your views directly communicate with your models. This can still lead to strong coupling with business logic leaking into the view layer. This is usually where the fancier patterns such as *BLoC* and *Redux* come in—they act as the glue between the model and the view.

There's another interesting piece of code that we've used in this recipe:

```
scrollController = ScrollController()
    ..addListener(() {
      FocusScope.of(context).requestFocus(FocusNode());
    });
```

You might wonder about the meaning of the double dots (..) before `addListener`: this is a **cascade operator**, which allows you to create a sequence of operations on the same object.

Another example is this:

```
            setState(() {
               plan = Plan(
                 name: plan.name,
                 tasks: List<Task>.from(plan.tasks)
                   ..[index] = Task(
                     description: text,
                     complete: task.complete,
                   ),
               );
             });
```

In the `setState()` method, we assign a new `Plan` object to the `plan` variable, with the same name as before, but with a modified `tasks` list. The `tasks` list is created by **copying the previous list** (this is what the `List<Task>.from(plan.tasks)` does), and replacing the element that has changed with a new `Task` object, which takes the new description from the `text` field and the same `complete` status as before.

The cascade operator makes your code shorter and easier to read: after a cascade operator you can set properties of an object or call its methods.

You may have noticed how most of this code is not too different from what we've covered in previous chapters. App architecture is an interesting beast. It can be argued that it's more of an art than a science. The fact that you created layers where different parts of your code live is not required to

make the app work. We didn't create a model for the first few chapters in this book and that didn't stop us from creating successful apps. So, why do you need to do this now?

The real reason why you would want to separate the model and view classes has little to do with functionality and more to do with productivity and testability. By separating these concepts into different components, you can compartmentalize your development process. When you are working on a model file, you don't need to think at all about the user interface.

At the data level, concepts such as buttons, colors, padding, and scrolling are a distraction. The goal of the data layer should be to focus on the data and any business rules you need to implement. On the other hand, your views do not need to think about the implementation details of the data models. In this way, you achieve what's called "separation of concerns."

See also

Check out these resources to learn more about app architecture:

- *Clean Code* by Robert Martin. If you want to write professional, maintainable code that will make everyone on your team happy, you have to read this book!: https://www.pearson.com/us/higher-education/program/Martin-Clean-Code-A-Handbook-of-Agile-Software-Craftsmanship/PGM63937.html.
- *SOLID* design principles: these are some rules that should make your designs more understandable and maintainable. Have a look at this wiki page for further information: https://en.wikipedia.org/wiki/SOLID .

Managing the data layer with InheritedWidget and InheritedNotifier

How should you access the data classes in your app?

Some of the possible options involve placing your data classes in the widget tree so they can take advantage of your application's life cycle.

The question then becomes *how can you place a model in the widget tree?* Models are not widgets, after all, and there is nothing to *build* onto the screen.

A possible solution is using `InheritedWidget`. So far, we've only been using two types of widgets: `StatelessWidget` and `StatefulWidget`. Both of these widgets are used to configure the UI on the screen; the only difference is that one can change and the other cannot. `InheritedWidget` is different. Its job is to pass data down to its children, but from a user's perspective, it's invisible. `InheritedWidget` can be used as the doorway between your **view** and **data** layers.

In this recipe, we will be updating the *Master Plan* app to move the storage of the to-do lists outside of the view classes.

Getting ready

You should complete the previous recipe, *Model-view separation*, before following along with this one.

How to do it...

Let's learn how to add `InheritedNotifier` to our project:

1. Create a new file called `plan_provider.dart` for storing our plans. Place this file in the root of the project's `lib` directory. This widget extends `InheritedNotifier`:

```
import 'package:flutter/material.dart';
import './models/data_layer.dart';
class PlanProvider extends
InheritedNotifier<ValueNotifier<Plan>> {
```

```
  const PlanProvider({super.key, required Widget child,
required ValueNotifier<Plan> notifier})
      : super(child: child, notifier: notifier);
  static ValueNotifier<Plan> of(BuildContext context) {
    return
context.dependOnInheritedWidgetOfExactType<PlanProvider>
()!.notifier!;
  }
}
```

2. Now that the provider widget is ready, it needs to be placed in the tree. In the `build` method of `MasterPlanApp`, in `main.dart`, wrap `PlanScreen` within a new `PlanProvider` class. Don't forget to fix any broken imports if needed:

```
return MaterialApp(
  theme: ThemeData(primarySwatch: Colors.purple),
  home: PlanProvider(
        notifier: ValueNotifier<Plan>(const Plan()),
        child: const PlanScreen(),
      ),
  );
```

3. Add two new methods to the `Plan` class in the `plan.dart` file. These will be used to show the progress on every plan. Call the first one `completedCount` and the second `completenessMessage`:

```
int get completedCount => tasks
  .where((task) => task.complete)
  .length;
String get completenessMessage =>
    '$completedCount out of ${tasks.length} tasks';
```

4. Tweak `PlanScreen` so that it uses the `PlanProvider`'s data instead of its own. In the `State` class, delete the `plan` property (this creates a few compile errors).

5. To fix the errors that were raised in the previous step, refactor the `_buildAddTaskButton()` method, adding `BuildContext` as its parameter, and using `PlanProvider` as the data source:

```
Widget _buildAddTaskButton(BuildContext context) {
    ValueNotifier<Plan> planNotifier =
PlanProvider.of(context);
    return FloatingActionButton(
      child: const Icon(Icons.add),
      onPressed: () {
        Plan currentPlan = planNotifier.value;
        planNotifier.value = Plan(
          name: currentPlan.name,
          tasks:
List<Task>.from(currentPlan.tasks)..add(const Task()),
        );
      },
    );
  }
```

6. Refactor the `_build TaskTile` method so that it takes the current `BuildContext`, and uses `PlanProvider` as its data source. Also change the `TextField` in `_buildTaskTile` to a `TextFormField`, to make it easier to provide initial data:

```
Widget _buildTaskTile(Task task, int index,
BuildContext context) {
  ValueNotifier<Plan> planNotifier =
PlanProvider.of(context);
  return ListTile(
    leading: Checkbox(
        value: task.complete,
        onChanged: (selected) {
          Plan currentPlan = planNotifier.value;
          planNotifier.value = Plan(
            name: currentPlan.name,
            tasks: List<Task>.from(currentPlan.tasks)
              ..[index] = Task(
                description: task.description,
                complete: selected ?? false,
              ),
          );
        }),
    title: TextFormField(
      initialValue: task.description,
      onChanged: (text) {
        Plan currentPlan = planNotifier.value;
        planNotifier.value = Plan(
          name: currentPlan.name,
```

```
            tasks: List<Task>.from(currentPlan.tasks)
              ..[index] = Task(
                description: text,
                complete: task.complete,
              ),
          );
        },
      ),
    );
}
```

7. Update the `_buildList` method so that it calls `_buildTaskTile` with the correct parameters:

```
Widget _buildList(Plan plan) {
    return ListView.builder(
      controller: scrollController,
      itemCount: plan.tasks.length,
      itemBuilder: (context, index) =>
          _buildTaskTile(plan.tasks[index], index,
context),
    );
  }
```

8. Still in the `PlanScreen` class, update the `build` method so that it shows the progress message at the bottom of the screen. Wrap the `_buildList` method in an `Expanded` widget and wrap it in a `Column` widget.
9. Finally, add a `SafeArea` widget with `completenessMessage` at the end of `Column`. The final result is shown here:

```
@override
Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Master Plan')),
      body: ValueListenableBuilder<Plan>(
        valueListenable: PlanProvider.of(context),
        builder: (context, plan, child) {
          return Column(
            children: [
              Expanded(child: _buildList(plan)),
              SafeArea(child:
Text(plan.completenessMessage))
```

```
              ],
            );
          },
        ),
        floatingActionButton:
_buildAddTaskButton(context),
      );
  }
```

Finally, build and run the app. There shouldn't be any noticeable change, but by doing this, you have created a cleaner separation of concerns between your view and the models. **Even more important, you manage state at your app's level.**

How it works...

`InheritedWidget` and `InheritedNotifier` widgets are fascinating widgets in the Flutter framework. Their job is **not** to render anything on the screen, but to pass data down to lower widgets in the tree.

Just like any other widget in Flutter, they can also have child widgets.

Let's break down the `PlanProvider` class: note the class declaration:

```
class PlanProvider extends
InheritedNotifier<ValueNotifier<Plan>>
```

This defines a class that inherits from `InheritedNotifier`, which is a class that inherits from `InheritedWidget`. It listens and updates objects that implement the `Listenable` interface.

`InheritedNotifier` widgets are generics. This specific `InheritedNotifier` works with a `ValueNotifier` that holds a `Plan` object. `ValueNotifier` is a widget that holds a single value (in this case, an instance of `Plan`), and whenever the value changes, it automatically calls `notifyListeners()`, which in turn notifies its listeners that the value has changed.

At a high level, we could highlight the three main steps required to use a `ValueNotifier`: **create, listen** and **update**:

You **create** a `ValueNotifier` with an initial value of a specified type. In our example, this is set as a `required` parameter:

```
required ValueNotifier<Plan> notifier
```

Widgets can then **listen** to the `ValueNotifier` for changes. You can use a `ValueListenableBuilder` to automatically rebuild the UI when the value changes.

In our example, this is done in `plan_screen.dart`, in the `build` method of the `_PlanScreenState` class:

```
body: ValueListenableBuilder<Plan>(…)
```

When you want to **update** the value held by a `ValueNotifier`, you simply set a new value to its `value` property. This triggers the `notifyListeners()` method, informing its listeners that the value has changed. In our example, this is also done in the `PlanScreenState` class:

```
planNotifier.value = Plan(
  name: currentPlan.name,
  tasks: List<Task>.from(currentPlan.tasks)
    ..add(const Task()),
);
```

Now, back to the `PlanProvider` class, let's have a look at the `of` method:

```
static ValueNotifier<Plan> of(BuildContext context) {
    return
context.dependOnInheritedWidgetOfExactType<PlanProvider>()!
.notifier!;
  }
```

This is a static method, and therefore it will be called on the class directly. It returns a `ValueNotifier<Plan>`, and takes an instance of `BuildContext` as

its only parameter.

The expression:

```
context.dependOnInheritedWidgetOfExactType<PlanProvider>()
```

looks for the nearest ancestor `PlanProvider` in the widget tree and returns it if it's found; otherwise, this returns null. That's why we use the non-null assertion operator (!), which asserts that the result of the previous expression is not null.

With the `.notifier` expression, you access the `PlanProvider` `notifier` property, of type `ValueNotifier<Plan>`.

In the UI, in `_PlanScreenState`, with the expression:

```
  ValueNotifier<Plan> planNotifier =
PlanProvider.of(context);
```

you access the `ValueNotifier<Plan>` directly. Then you change its value property when users make changes. For example, with the code below, you pass a new `Plan` as the value for the `planNotifier`:

```
planNotifier.value = Plan(
          name: currentPlan.name,
          tasks: List<Task>.from(currentPlan.tasks)
            ..[index] = Task(
              description: text,
              complete: task.complete,
            ),
        );
```

This will update the user interface on your screen.

To sum it up, instead of calling `setState` and updating your widget, you have moved the state for this screen higher up in the widget tree.

See also

The official documentation on `InheritedWidget` can be found at https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html.

Making the app state visible across multiple screens

One phrase that is thrown around a lot in the Flutter community is "Lift State Up." This mantra refers to the idea that `State` objects should be placed higher than the widgets that need it in the widget tree. Our `InheritedWidget`, which we created in the previous recipe, now works perfectly for a single screen, but what happens when you add a second?

In this recipe, you are going to add another screen to the *Master Plan* app so that you can create multiple plans. Accomplishing this will require our State provider to be lifted higher in the tree, closer to its root.

Getting ready

You should have completed the previous recipes in this chapter before following along with this one.

How to do it...

Let's add a second screen to the app and lift the State higher in the tree:

1. Update the `PlanProvider` class so that it can handle multiple plans. Change the type of `ValueNotifier` every time it is defined from a single `plan` to a `List` of `Plan` objects:

```
class PlanProvider extends
InheritedNotifier<ValueNotifier<List<Plan>>> {
  const PlanProvider({super.key, required Widget child,
required ValueNotifier<List<Plan>> notifier})
      : super(child: child, notifier: notifier);
  static ValueNotifier<List<Plan>> of(BuildContext
context) {
```

```
    return
context.dependOnInheritedWidgetOfExactType<PlanProvider>
()!.notifier!;
  }
}
```

2. The previous step has caused errors in `main.dart` and `plan_screen.dart`. In `main.dart`, in the `build()` method of `MasterPlanApp`, change the `build` method as shown here:

```
  @override
  Widget build(BuildContext context) {
    return PlanProvider(
      notifier: ValueNotifier<List<Plan>>(const []),
      child: MaterialApp(
        title: 'State management app',
        theme: ThemeData(
          primarySwatch: Colors.blue,
        ),
        home: const PlanScreen(),
      ),
    );
  }
```

3. Open `plan_screen.dart`. First, we need to add a `Plan` in the constructor where the specific plan can be injected and then update the build methods to read that value. Edit the `PlanScreen` widget:

```
final Plan plan;
const PlanScreen({super.key, required this.plan});
```

4. Note we have an error each time `PlanProvider.of(context)` is called. This is because this screen shows the tasks for a single plan, but now `PlanProvider` provides a list of plans.
5. First let's add a `Plan` getter at the top of `_PlanScreenState`:

```
class _PlanScreenState extends State<PlanScreen> {
  late ScrollController scrollController;
  Plan get plan => widget.plan;
```

6. In the `initState` method, set the plan to take the `Plan` that's passed to the widget:

```
  @override
  void initState() {
     super.initState();
     scrollController = ScrollController()
      ..addListener(() {

FocusScope.of(context).requestFocus(FocusNode());
      });

  }
```

7. In the `build()` method, make sure you change the references to a single `Plan` to `List<Plan>`, and update the current `plan` value:

```
  @override
  Widget build(BuildContext context) {
    ValueNotifier<List<Plan>> plansNotifier =
PlanProvider.of(context);
    return Scaffold(
      appBar: AppBar(title: Text(_plan.name)),
      body: ValueListenableBuilder<List<Plan>>(
        valueListenable: plansNotifier,
        builder: (context, plans, child) {
          Plan currentPlan = plans.firstWhere((p) =>
p.name == plan.name);
          return Column(
            children: [
              Expanded(child: _buildList(currentPlan)),
              SafeArea(child:
Text(currentPlan.completenessMessage)),
            ],);},),
      floatingActionButton:
_buildAddTaskButton(context,)
  ,);
 }
  Widget _buildAddTaskButton(BuildContext context) {
    ValueNotifier<List<Plan>> planNotifier =
PlanProvider.of(context);
    return FloatingActionButton(
      child: const Icon(Icons.add),
      onPressed: () {
        Plan currentPlan = plan;
```

```
        int planIndex =
            planNotifier.value.indexWhere((p) => p.name
== currentPlan.name);
        List<Task> updatedTasks =
List<Task>.from(currentPlan.tasks)
          ..add(const Task());
        planNotifier.value =
List<Plan>.from(planNotifier.value)
          ..[planIndex] = Plan(
            name: currentPlan.name,
            tasks: updatedTasks,
          );
        plan = Plan(
          name: currentPlan.name,
          tasks: updatedTasks,
        );},);
  }
```

8. In the `buildTaskTile()` method, make sure you use `List<Plan>` when dealing with `planNotifier`:

```
  Widget _buildTaskTile(Task task, int index,
BuildContext context) {
    ValueNotifier<List<Plan>> planNotifier =
PlanProvider.of(context);
    return ListTile(
      leading: Checkbox(
          value: task.complete,
          onChanged: (selected) {
            Plan currentPlan = plan;
            int planIndex = planNotifier.value
                .indexWhere((p) => p.name ==
currentPlan.name);
            planNotifier.value =
List<Plan>.from(planNotifier.value)
              ..[planIndex] = Plan(
                name: currentPlan.name,
                tasks:
List<Task>.from(currentPlan.tasks)
                  ..[index] = Task(
                    description: task.description,
                    complete: selected ?? false,
                  ),);
          }),
      title: TextFormField(
        initialValue: task.description,
```

```
        onChanged: (text) {
          Plan currentPlan = plan;
          int planIndex =
              planNotifier.value.indexWhere((p) =>
p.name ==
currentPlan.name);
          planNotifier.value =
List<Plan>.from(planNotifier.value)
            ..[planIndex] = Plan(
              name: currentPlan.name,
              tasks: List<Task>.from(currentPlan.tasks)
                ..[index] = Task(
                  description: text,
                  complete: task.complete,
                ),
            );
},),);}
```

9. We can now create a new screen to manage the multiple plans. This screen will depend on the `PlanProvider` to store the app's data. In the `views` folder, create a file called `plan_creator_screen.dart` and declare a new `StatefulWidget` called `PlanCreatorScreen`. Make this class the new home widget for the `MaterialApp`, replacing `PlanScreen`.

```
home: const PlanCreatorScreen(),
```

10. In the `_PlanCreatorScreenState` class, we need to add a `TextEditingController` so that we can create a simple `TextField` to add new plans. Don't forget to dispose of `textController` when the widget is unmounted:

```
final textController = TextEditingController();
@override
void dispose() {
  textController.dispose();
  super.dispose();
}
```

11. Now, let's create the build method for this screen. This screen will have a `TextField` at the top and a list of plans underneath it. Add the

following code before the `dispose` method to create a `Scaffold` for this screen:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Master Plans')),
    body: Column(children: [
      _buildListCreator(),
      Expanded(child: _buildMasterPlans())
    ]),
  );
}
```

12. The `_buildListCreator` method constructs a `TextField` and calls a function to add a plan when the user taps *Enter* on their keyboard. We're going to wrap `TextField` in a `Material` widget to make the field pop out:

```
Widget _buildListCreator() {
  return Padding(
      padding: const EdgeInsets.all(20.0),
      child: Material(
        color: Theme.of(context).cardColor,
        elevation: 10,
        child: TextField(
            controller: textController,
            decoration: const InputDecoration(
                labelText: 'Add a plan',
                contentPadding: EdgeInsets.all(20)),
            onEditingComplete: addPlan),
      ));
}
```

13. The `addPlan` method will check whether the user actually typed something into the field and will then reset the screen:

```
void addPlan() {
  final text = textController.text;
    if (text.isEmpty) {
      return;
    }
    final plan = Plan(name: text, tasks: []);
    ValueNotifier<List<Plan>> planNotifier =
```

```
PlanProvider.of(context);
    planNotifier.value =
List<Plan>.from(planNotifier.value)..add(plan);
    textController.clear();
    FocusScope.of(context).requestFocus(FocusNode());
    setState(() {});
}
```

14. We can create a `ListView` that will read the data from `PlanProvider` and print it onto the screen. This component will also be aware of its content and return the appropriate set of widgets:

```
Widget _buildMasterPlans() {
  ValueNotifier<List<Plan>> planNotifier =
PlanProvider.of(context);
    List<Plan> plans = planNotifier.value;
    if (plans.isEmpty) {
      return Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Icon(Icons.note, size: 100, color:
Colors.grey),
            Text('You do not have any plans yet.',
                style:
Theme.of(context).textTheme.headlineSmall)
          ]);
    }
    return ListView.builder(
        itemCount: plans.length,
        itemBuilder: (context, index) {
          final plan = plans[index];
          return ListTile(
              title: Text(plan.name),
              subtitle: Text(plan.completenessMessage),
              onTap: () {
                Navigator.of(context).push(
                    MaterialPageRoute(builder: (_) =>
PlanScreen(plan: plan,)));
              });
        });
}
```

When you hot restart the app, you will be able to create multiple plans with different lists on each screen, as shown in *figure 7.3*:

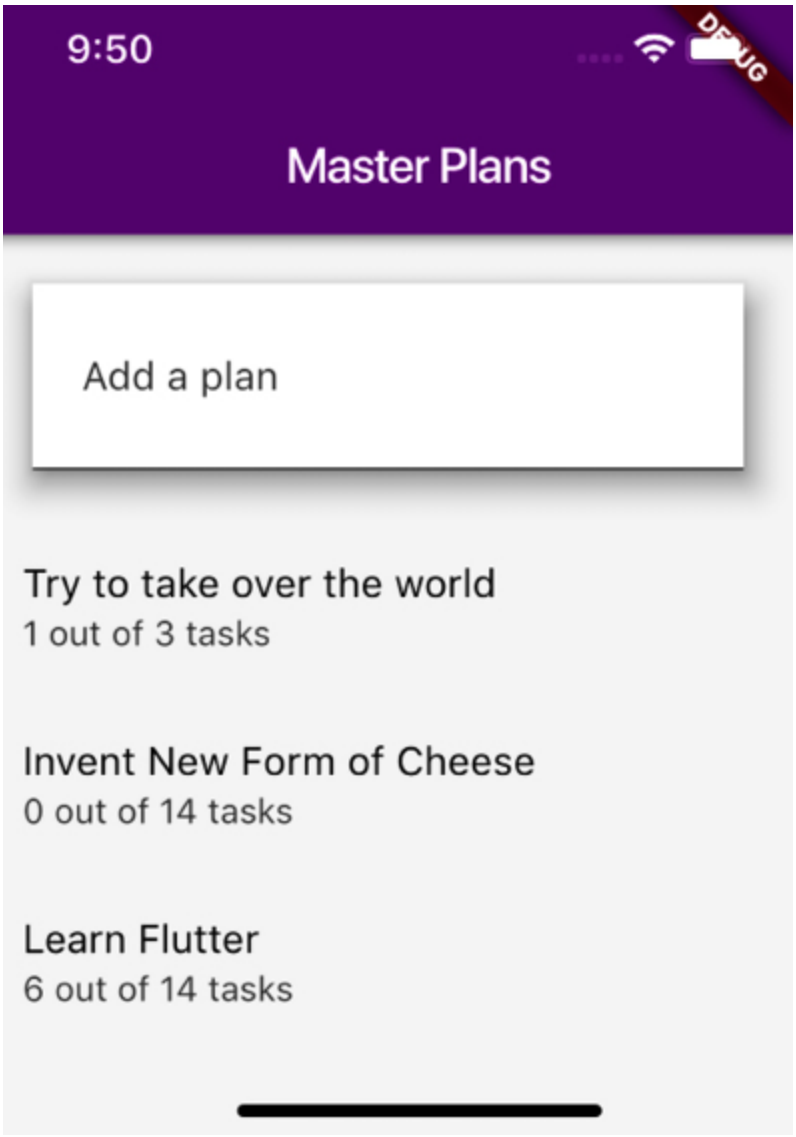


Figure 7.3: The plan creator screen

How it works...

The main takeaway from this recipe is the importance of the widget tree construction. When you push a new route onto `Navigator`, you are essentially replacing every widget that lives underneath `MaterialApp`, as shown in this diagram:

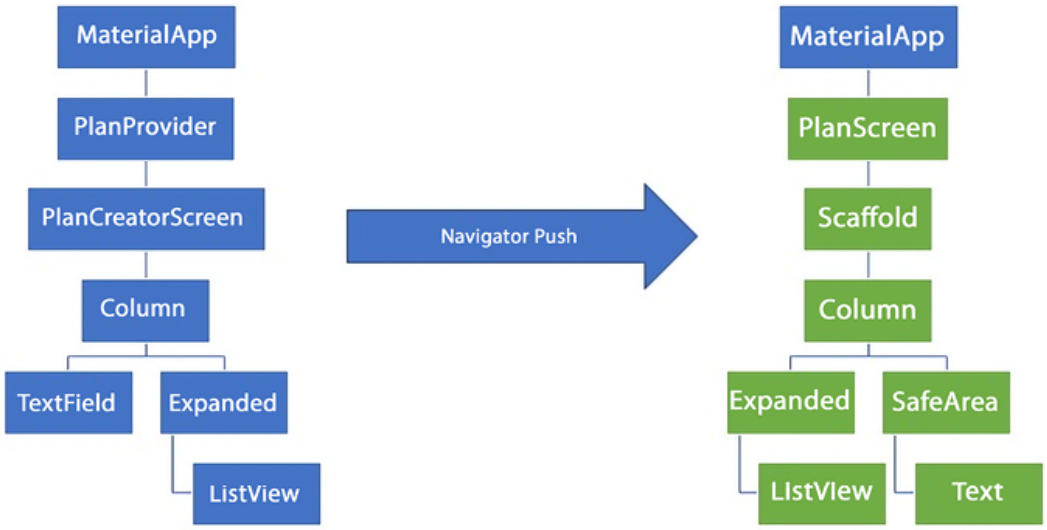


Figure 7.4: The app widget tree

If `PlanProvider` was a child of `MaterialApp`, it would be destroyed when pushing the new route, making all its data inaccessible to the next widget. If you have an `InheritedWidget` that only needs to provide data for a single screen, then placing it lower in the widget tree is optimal. However, if this same data needs to be accessed across multiple screens, it has to be placed above our `Navigator`.

Placing our global state widget at the root of the tree also has the added benefit of causing our app to update without any extra code. Try checking and unchecking a few tasks in your plans. You'll notice that the data is automatically updated, like magic. This is one of the primary benefits of maintaining a clean architecture in our apps.

See also

- While `InheritedWidget` and `InheritedNotifier` are great widgets to deal with state in Flutter, in many cases developers prefer using the `Provider` package: it's a wrapper around `InheritedWidget` that aims to make your code easier and more reusable: have a look at the package in `pub.dev`: https://pub.dev/packages/provider

- More recently, `Riverpod` was also introduced: this package solves some of the `Provider` package limitations: https://riverpod.dev/

Summary

In this chapter, you've seen some of the basic concepts of state management in Flutter, while building an app called *Master Plan*. This app allows users to create plans and add tasks to each plan, and shows how to manage the app state across multiple screens.

In particular, we've covered the following topics:

- Model-view separation: you've seen the importance of separating the application's data model from the view layer.
- By keeping the model and view separate, it's easier to maintain and scale the app as it grows in size and complexity.
- In the `MasterPlanApp`, you used the `Plan` and `Task` classes as data models, and the widgets in the app acted as the view layer.
- In the recipe *Managing the data layer with InheritedWidget and InheritedNotifier*, you added the `PlanProvider` class, which extends `InheritedNotifier`.
- You used it to manage the app's **data layer**. Using an `InheritedWidget`, you could access and modify the state of the app from anywhere in the widget tree without passing data through multiple layers of widgets or calling `setState()`.
- Finally, you saw how to manage the app state across multiple screens wrapping the `MaterialApp` widget in a `PlanProvider`. In this way, you made sure that the state is accessible throughout the app.
- You've built two screens: `PlanCreatorScreen` and `PlanScreen`. The `PlanCreatorScreen` allows users to create new plans, while the `PlanScreen` enables users to add tasks to a specific plan. By using `PlanProvider` and `InheritedNotifier`, you could propagate changes in

the state across both screens, ensuring that the app remains up-to-date on all screens.

This chapter focused on building a foundation for state management in Flutter applications. Effective state management can lead to more maintainable and scalable apps.

The Future is Now: Introduction to Asynchronous Programming

Asynchronous programming allows your app to complete time-consuming tasks, such as retrieving an image from the web, or writing some data to a web server, while running other tasks in parallel and responding to the user input. This improves the user experience and the overall quality of your software.

In Dart and Flutter, you can write asynchronous code leveraging **Futures**, and the async/await pattern: these patterns exist in most modern programming languages, but Flutter also has a very efficient way to build the user interface asynchronously using the `FutureBuilder` class.

In *Chapter 10*, *Advanced State Management with Streams*, we will also focus on streams, which are another way to deal with asynchronous programming.

> **Note**
>
> In general, you use `Futures` to get a single asynchronous operation result (like uploading data to a web server), and `Streams` to get multiple asynchronous operations results (like getting GPS coordinates or reacting to updates in a database).

By following the recipes in this chapter, you will achieve a thorough understanding of how to leverage Futures in your apps, and you will also learn how to choose and use the right tools among the several options you have in Flutter, including Futures, async/await, and `FutureBuilder`.

In this chapter, we will cover the following recipes:

- Using a Future

- Using async/await to remove callbacks
- Completing Futures
- Firing multiple Futures at the same time
- Resolving errors in asynchronous code
- Using Futures with StatefulWidgets
- Using FutureBuilder to let Flutter manage your Futures
- Turning navigation routes into asynchronous functions
- Getting the results from a dialog

Technical requirements

To follow along with the recipes in this chapter, you need your developing machine configured and connected to the internet.

Using a Future

When you write your code, you generally expect your instructions to run sequentially, one line after the other. For instance, let's say you write the following:

```
int x = 5;
int y = x * 2;
```

You expect the value of `y` to be equal to 10 because the instruction `int x = 5` completes *before* the next line. In other words, the second line waits for the first instruction to complete before being executed.

In most cases, this pattern works perfectly, but in some cases, and specifically, when you need to run instructions that take longer to complete, this is not the recommended approach, as your app would be *unresponsive* until the task is completed. That's why in almost all modern programming languages, including Dart, you can perform asynchronous operations.

Asynchronous operations do not stop the main line of execution, and therefore they allow the execution of other tasks before completing.

Consider the following diagram:

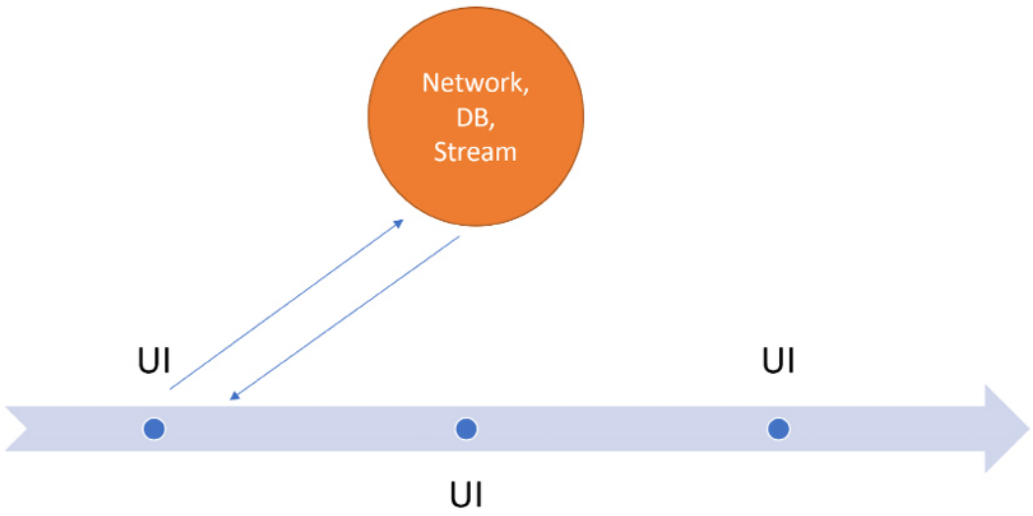


Figure 8.1: Execution flow

In the diagram, you can see how the main execution line, which deals with the user interface, may call a long-running task asynchronously without stopping to wait for the results, and when the long-running task completes, it returns to the main execution line, which can deal with it.

Dart is a single-threaded language, but despite this, you can use asynchronous programming patterns to create responsive apps.

In Dart and Flutter, you can use the `Future` class to perform asynchronous operations. Some use cases where an asynchronous approach is recommended include retrieving data from a web service, writing data to a database, finding a device's coordinates, opening a file and reading data from it. Performing these tasks asynchronously will keep your app responsive.

In this recipe, you will create an app that reads data from a web service using the `http` library. Specifically, we will read JSON data from the Google Books API.

Getting ready

In order to follow along with this recipe:

- Your device will need an internet connection to retrieve data from the web service.
- The starting code for this recipe is available at https://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_07.

How to do it...

You will create an app that connects to the Google Books API to retrieve a Flutter book's data from the web service, and you will show part of the result on the screen as shown in the screenshot:

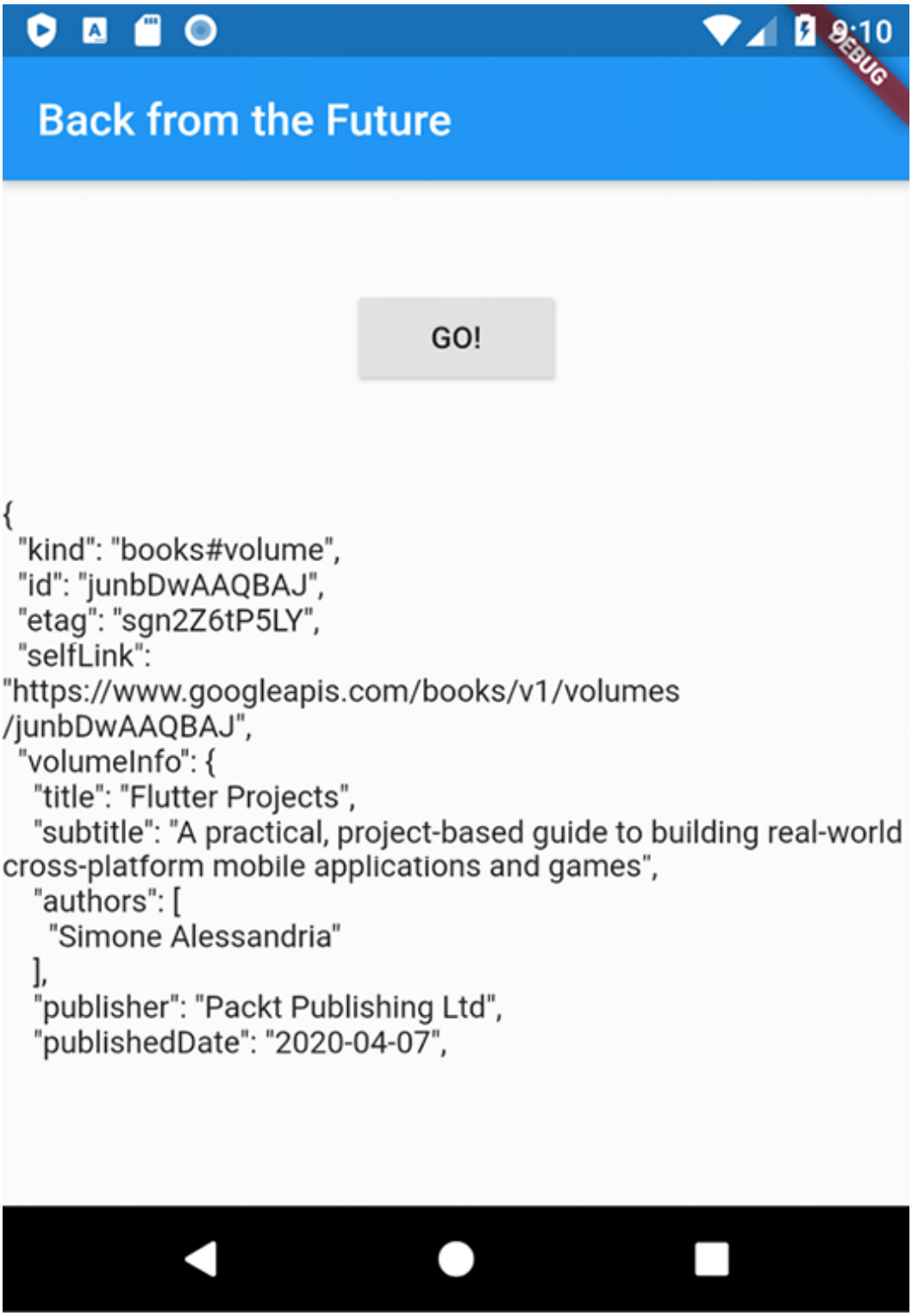


Figure 8.2: Retrieving data from Google Books

The steps required in order to retrieve and show the data using a Future are outlined here:

1. Create a new Flutter app, and call it `books`.
2. Add the `http` dependency, typing: `flutter pub add http`.
3. Check `pubspec.yaml` and make sure the dependency has been added:

```
dependencies:
  flutter:
    sdk: flutter
  http: ^0.13.5
```

4. If using macOS, you should enable networking: in `macos/Runner/DebugProfile.entitlements` and `macos/Runner/Release.entitlements` files add the following lines:

```
<key>com.apple.security.network.client</key>
<true/>
```

5. The starting code in the `main.dart` file contains an `ElevatedButton`, a `Text` containing the result, and a `CircularProgressIndicator`. You can download the beginning code at https://github.com/PacktPublishing/Flutter-Cookbook-Second-Edition/tree/main/Chapter_08. Otherwise, type the following code:

```
import 'dart:async';
import 'package:flutter/material.dart';
import 'package:http/http.dart';
import 'package:http/http.dart' as http;
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Future Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
        visualDensity:
```

```
VisualDensity.adaptivePlatformDensity,
      ),
      home: const FuturePage(),
    );
  }
}
class FuturePage extends StatefulWidget {
  const FuturePage({super.key});
  @override
  State<FuturePage> createState() => _FuturePageState();
}
class _FuturePageState extends State<FuturePage> {
  String result = '';
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Back from the Future'),
      ),
      body: Center(
        child: Column(children: [
          const Spacer(),
          ElevatedButton(
            child: const Text('GO!'),
            onPressed: () {},
          ),
          const Spacer(),
          Text(result),
          const Spacer(),
          const CircularProgressIndicator(),
          const Spacer(),
        ]),
      ),
    );
  }
}
```

Note

There's nothing special about this code. Just note that in the `Column` we have placed a `CircularProgressIndicator`: as long as the progress animation keeps moving, this means the app is responsive. When it stops, it means the user interface is waiting for a process to complete.

6. Now we need to create a method that retrieves some data from a web service: in particular, we'll use the Google Books API for this recipe. At the end of the `_FuturePageState` class, add a method called `getData()` as shown here:

```
  Future<Response> getData() async {
    const authority = 'www.googleapis.com';
    const path = '/books/v1/volumes/junbDwAAQBAJ';
    Uri url = Uri.https(authority, path);
    return http.get(url);
  }
```

7. In order to call the `getData()` method when the user taps the `ElevatedButton`, add the following code in the `onPressed` function:

```
 ElevatedButton(
            child: Text('GO!'),
            onPressed: (){
              setState(() {});
              getData()
              .then((value) {
                result =
value.body.toString().substring(0, 450);
                setState(() {});
              }).catchError((_){
                result = 'An error occurred';
                setState(() {});
        });
     },
),
```

How it works...

In the preceding code, we are calling the `getData()` method, but after that, we are adding the `then` function.

Note the following:

- The `getData()` method returns a `Future`. Futures are generics, so you have the option to specify the type of `Future` you are returning; if the

return value of a method is `Future<int>`, it means that your method will return a `Future` of an integer number. A Future does not contain the value itself, but will deliver a value some time in the future.

- It's in the `then` callback that you actually receive the value from a Future.
- In this case, specifying the type is not required, so we could also write the following:

```
Future getData() async {
```

The preceding code would work just as well.

- The `getData()` method is marked as `async`. It is considered a good practice to mark all your asynchronous methods with the `async` keyword, but it's not required in this example (it is only required when using the `await` statement, which we'll see in the next recipe in this chapter: *Using async/await to remove callbacks*).
- The `http.get` method makes a call to the `Uri` you specify as a parameter, and when the call completes, it returns an object of type `Response`.

Note

A **Uniform Resource Identifier** (**URI**) is a sequence of characters that universally identify a resource. Examples of URIs include a web address (https://www.packt.com), an email address (`mailto:name@example.com`), a barcode or ISBN book code, and even a telephone number.

- When building a `Uri` in Flutter, you pass the *authority* (which is the domain name in this example) and the *path* within the domain where your data is located. You can also optionally set more parameters, as you will see in later recipes in this chapter.

- In this example, the URL is the address of specific book data in JSON format.

Note

When a method returns a `Future`, it does not return the actual value but the **promise** of returning a value at a later time.

Imagine a takeaway restaurant: you enter the restaurant and place your order. Instead of giving you the food immediately, the teller gives you a receipt, containing the number of your order. This is a **promise** of giving you the food as soon as it's ready.

Futures in Flutter work in the same way: in our example, we will get a `Response` object at a time in the future, as soon as the `http` connection has completed, successfully or with an error.

`then` is called when the `Future` returns successfully, and `value` is the result of the call. In other words, what's happening here is this:

1. You call `getData`.
2. The execution continues.
3. At some time in the future, `getData` returns a `Response`.
4. The `then` function is called, and the `Response` value is passed to the function.
5. You update the state of the widget calling `setState` and showing the first 450 characters of the result.

After the `then()` method, you concatenate the `catchError` function. This is called if the `Future` does not return successfully: in this case, you catch the error and give some feedback to the user.

A Future may be in one of three states:

1. **Uncompleted**: You called a Future, but the response isn't available yet.
2. **Completed successfully**: The `then()` function is called.
3. **Completed with an error**: The `catchError()` function is called.

Following an asynchronous pattern in your apps using Futures is not particularly complicated in itself: the main point here is that the execution does not happen sequentially, so you should understand when the returning values of an `async` function are available (only *inside* the `then()` function) and when they are not.

See also

A great resource to understand the asynchronous pattern in Flutter is the official codelab available at https://dart.dev/codelabs/async-await.

I also recommend watching the Flutter Team video introducing Futures at http://y2u.be/OTS-ap9_aXc.

On a more theoretical basis, but extremely important to understand, is the concept of **Isolate**, which can explain how asynchronous programming works in Dart and Flutter: there's a thorough explanation in the video linked here: http://y2u.be/vl_AaCgudcY.

Using async/await to avoid callbacks

Futures, with their `then` callbacks, allow developers to deal with *asynchronous programming*. There is an alternative pattern to deal with Futures that can help make your code cleaner and easier to read and maintain: the **async/await** pattern.

Several modern languages have this alternate syntax to simplify code, and at its core, it's based on two keywords: `async` and `await`:

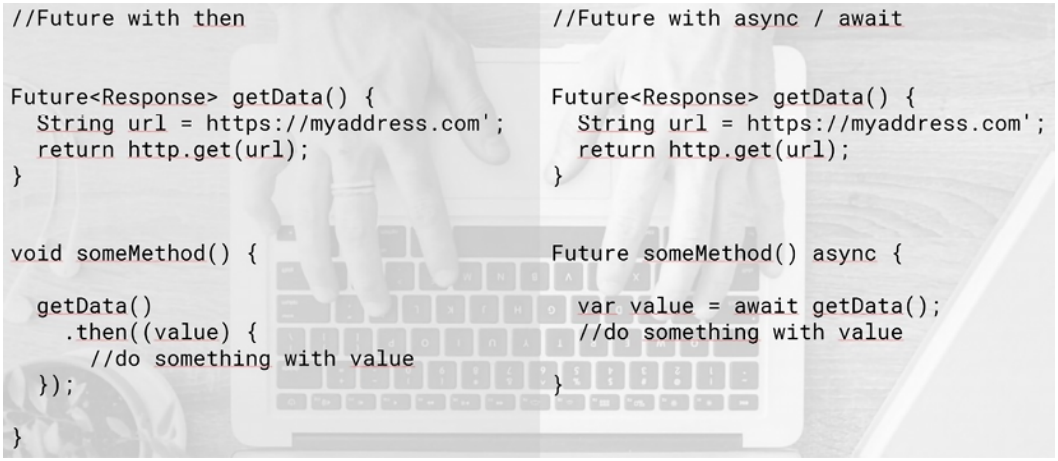
- `async` is used to mark a method as asynchronous, and it should be added before the function body.
- `await` is used to tell the framework to wait until the function has finished its execution and returns a value. While the `then` callback works in any method, `await` only works inside `async` methods.

Note

When you use `await`, the caller function **must** use the `async` modifier, and the function you call with `await` should also be marked as `async`.

What happens under the hood is that when you `await` the result of an asynchronous function, the line of execution is stopped until the `async` operation completes.

Here, you can see an example of writing the same code with the `then` callback and the `await` syntax:



```
//Future with then

Future<Response> getData() {
  String url = https://myaddress.com';
  return http.get(url);
}

void someMethod() {

  getData()
    .then((value) {
      //do something with value
  });
}
```

```
//Future with async / await

Future<Response> getData() {
  String url = https://myaddress.com';
  return http.get(url);
}

Future someMethod() async {

  var value = await getData();
  //do something with value
}
```

Figure 8.3: Then and async/await

In the code above, you can see a comparison between the two approaches of dealing with Futures. Please note that:

- The `then()` callback can be used in any method; `await` requires `async`.

- After the execution of an `await` statement, the returned value is immediately available to the following lines.
- You can append a `catchError` callback: there is no equivalent syntax with `async/await` (but you can always use a `try-catch`, as we will see in a future recipe in this chapter (see: *How to Resolve Errors in Asynchronous Code*, which deals specifically with catching errors when using asynchronous programming).

Getting ready

In order to follow along with this recipe:

- Your device will need an internet connection.
- If you followed along with the previous recipe, you'll only need to edit the existing project. Otherwise, create a new app with the starting code available at https://github.com/PacktPublishing/Flutter-Cookbook-Second-Edition/tree/main/Chapter_08.

How to do it...

In this recipe, you will see the advantages of using the async/await pattern:

1. Add the following three methods to the `main.dart` file, at the bottom of the `_FuturePageState` class:

```
Future<int> returnOneAsync() async {
  await Future.delayed(const Duration(seconds: 3));
  return 1;
}
Future<int> returnTwoAsync() async {
  await Future.delayed(const Duration(seconds: 3));
  return 2;
}
Future<int> returnThreeAsync() async {
  await Future.delayed(const Duration(seconds: 3));
```

```
    return 3;
  }
```

2. Under the three methods you just created, add the `count()` method leveraging the async/await pattern:

```
Future count() async {
    int total = 0;
    total = await returnOneAsync();
    total += await returnTwoAsync();
    total += await returnThreeAsync();
    setState(() {
      result = total.toString();
    });
  }
```

3. Call the `count()` method from the `onPressed` function of the **GO** button:

```
ElevatedButton(
   child: Text('GO!'),
      onPressed: () {
         count();
      },
}
...
```

4. Try out your app: you should see the result `6` after 9 seconds, as shown in the following screenshot (the wheel will keep turning — this is expected):

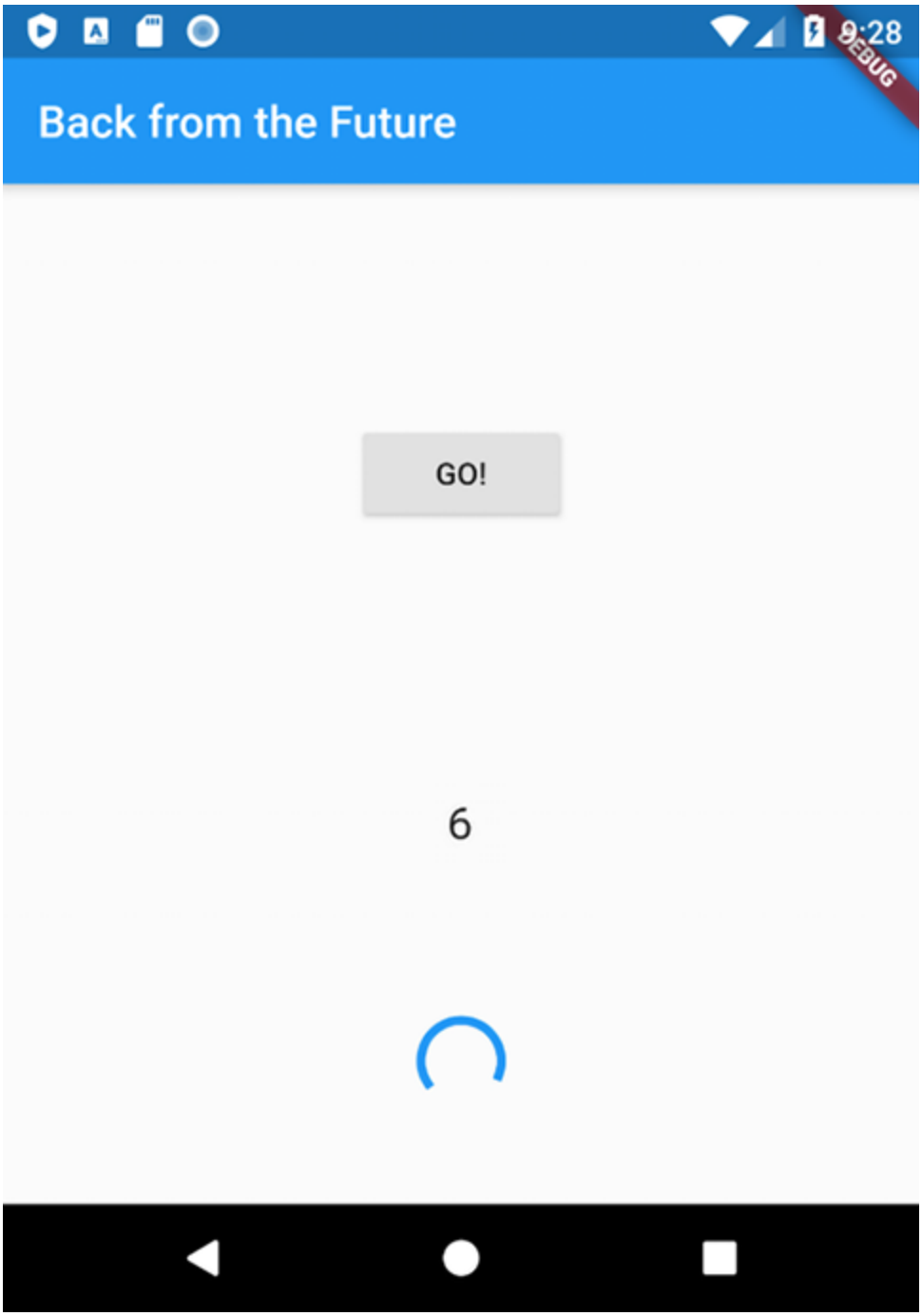


Figure 8.4: The count method returning values

How it works...

The great feature of the async/await pattern is that you can write your code like *sequential* code, with all the advantages of asynchronous programming (namely, not blocking the UI thread). After the execution of the `await` statement, the returned value is immediately available to the following lines, so the `total` variable in our example can be updated just after the execution of the `await` statement.

The three methods used in this recipe (`returnOneAsync`, `returnTwoAsync`, and `returnThreeAsync`) each wait 3 seconds and then return a number using `Future.delayed`.

`Future.delayed` creates a Future that completes after a specified amount of time. This can be useful when you want to delay the execution of some code.

If you wanted to sum the numbers returned by the three methods using the `then` callback, you would write something like this:

```
returnOneAsync().then((value) {
                  total += value;
                  returnTwoAsync().then((value) {
                    total += value;
                    returnThreeAsync().then((value) {
                      total += value;
                      setState(() {
                        result = total.toString();
                      });
                    });
                  });
                });
```

You can clearly see that, even if it works perfectly, this code would soon become hard to read and maintain: nesting several `then` callbacks one into the other is sometimes called "*callback hell*," as it easily becomes a nightmare to deal with.

The only rule to remember is that while you can place a `then` callback anywhere (for instance, in the `onPressed` method of an `ElevatedButton`), you need to create an `async` method in order to use the `await` **keyword**.

See also

A great resource to understand the asynchronous pattern in Flutter is the official codelab available at https://dart.dev/codelabs/async-await.

For the async/await pattern, in particular, I also recommend watching the official video at http://y2u.be/SmTCmDMi4BY.

Completing Futures

Using a Future with `then`, `catchError`, `async`, and `await` is probably enough for most use cases, but there is another way to deal with asynchronous programming in Dart and Flutter: the `Completer` class.

Completer creates `Future` objects that you can **complete** later with a value or an error. We will be using `Completer` in this recipe.

Getting ready

In order to follow along with this recipe, there are the following requirements:

- You can modify the project completed in the previous recipes in this chapter.
- OR, you can create a new project from the starting code available at https://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_07.

How to do it...

In this recipe, you will see how to use the `Completer` class to perform a long-running task:

1. At the top of the `main.dart` file, import `dart:async`:

```
import 'package:async/async.dart';
```

2. Add the following code in the `_FuturePageState` class:

```
late Completer completer;
Future getNumber() {
    completer = Completer<int>();
    calculate();
    return completer.future;
  }
 Future calculate() async {
    await Future.delayed(const Duration(seconds : 5));
    completer.complete(42);
  }
```

3. If you followed the previous recipes in the chapter, comment out the code in the `onPressed` function.
4. Add the following code in the `onPressed()` function:

```
getNumber().then((value) {
                setState(() {
                  result = value.toString();
                });
              });
```

If you try the app right now, you'll notice that after 5 seconds delay, the number `42` should show up on the screen, like shown in *Figure 8.5*:

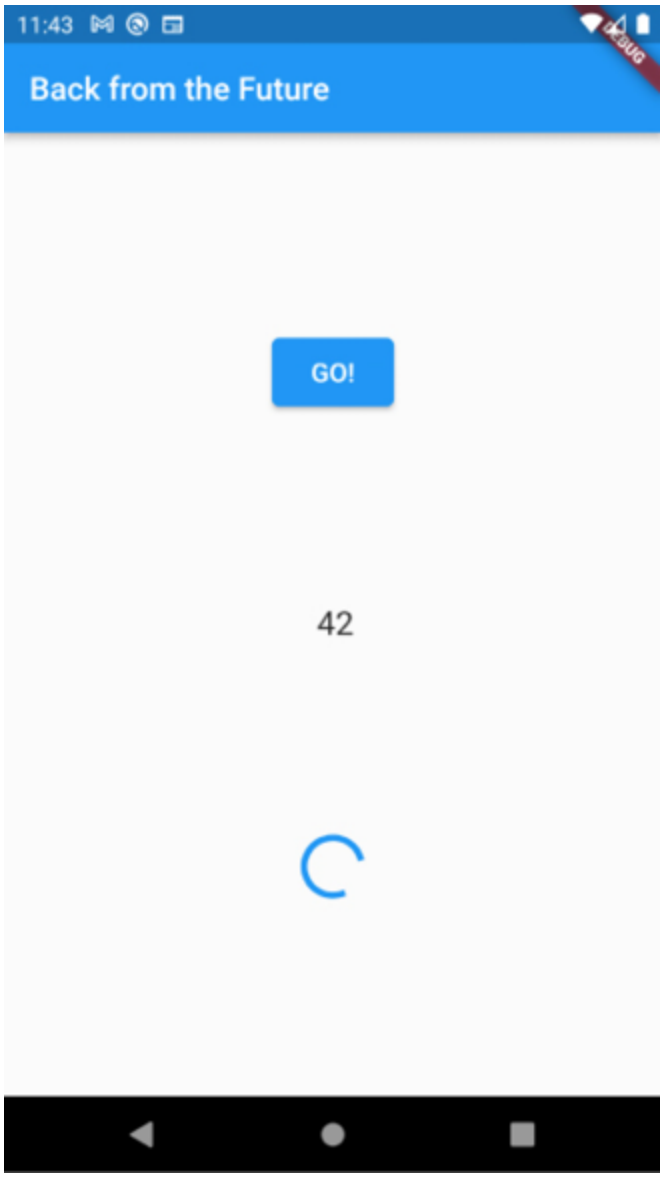


Figure 8.5: Result of a Completer function

How it works...

A `Completer` creates Future objects that can be **completed** later. In other words, it allows you to control when a Future is completed. In our example, the `Completer.future` that's set in the `getNumber` method is the Future that will be completed once `complete` is called.

When you call the `getNumber()` method, you are returning a Future, by calling the following:

```
return completer.future;
```

The `getNumber()` method also calls the `calculate()` async function, which waits 5 seconds (here, you could place any long-running task), and calls the `completer.complete` method.

`Completer.complete` changes the state of the `Completer`, so that you can get the returned value in a `then()` callback.

Completers are very useful when you call a service that does not use Futures, and you want to return a Future. It also de-couples the execution of your long-running task from the Future itself, and this can be useful when the completion of the Future depends on the outcome of some other asynchronous task.

There's more...

You can also call the `completeError` method of a `Completer` when you need to deal with an error:

1. Change the code in the `calculate()` method like this:

```
calculate() async {
    try {
      await new Future.delayed(const Duration(seconds :
5));
      completer.complete(42);
// throw Exception();
    }
    catch (_) {
      completer.completeError({});
    }
  }
```

2. In the call to `getNumber`, you could then concatenate a `catchError` to the `then` function:

```
getNumber().then((value) {
                  setState(() {
                    result = value.toString();
                  });
                }).catchError((e) {
                  result = 'An error occurred';
                });
```

See also

You can have a look at the full documentation for the `Completer` object at https://api.flutter.dev/flutter/dart-async/Completer-class.html.

Firing multiple Futures at the same time

When you need to run multiple Futures at the same time, there is a class that makes the process extremely easy: `FutureGroup`.

`FutureGroup` is available in the `async` package, which must be imported into your dart file as shown in the following code block:

```
import 'package:async/async.dart';
```

Note

Please note that `dart:async` and `async/async.dart` are different libraries: in some cases, you need both to run your asynchronous code.

FutureGroup is a collection of Futures that can be run in parallel. As all the tasks run in parallel, the time of execution is generally faster than calling each asynchronous method one after another.

When all the Futures of the collection have finished executing, a `FutureGroup` returns its values as a `List`, in the same order they were added

into the group.

You can add Futures to a `FutureGroup` using the `add()` method, and when all the Futures have been added, you call the `close()` method to signal that no more Futures will be added to the group.

Note

If any of the Futures in the group returns an error, the `FutureGroup` will return an error and will be closed.

In this recipe, you will see how to use a `FutureGroup` to perform several tasks in parallel.

Getting ready

In order to follow along with this recipe, there is the following requirement:

- You should have completed the code in the previous recipe: *Using async/await to remove callbacks*.

How to do it...

In this recipe, instead of waiting for each task to complete, you will use a `FutureGroup` to run three asynchronous tasks in parallel:

1. Add the following code in the `_FuturePageState` class, in the `main.dart` file of your project:

```
void returnFG() {
    FutureGroup<int> futureGroup = FutureGroup<int>();
    futureGroup.add(returnOneAsync());
    futureGroup.add(returnTwoAsync());
    futureGroup.add(returnThreeAsync());
    futureGroup.close();
    futureGroup.future.then((List <int> value) {
      int total = 0;
```

```
      for (var element in value) {
        total += element;
      }
      setState(() {
        result = total.toString();
      });
    });
  }
```

Note

As an alternative to a `FutureGroup`, you can also use the `Future.wait` syntax:

```
final futures = Future.wait<int>([
      returnOneAsync(),
      returnTwoAsync(),
      returnThreeAsync(),
    ]);
```

2. In order to try this code, just add the call to `returnFG()` in the `onPressed` method of the `ElevatedButton` (remove or comment out the old code if necessary):

```
onPressed: () {
    returnFG();
}
```

3. Run the code. This time you should see the result (the number 6) faster (after about 3 seconds instead of 9).

How it works...

In the `returnFG()` method, we are creating a new `FutureGroup` with this instruction:

```
FutureGroup<int> futureGroup = FutureGroup<int>();
```

A `FutureGroup` is a **generic**, so `FutureGroup<int>` means that the values returned inside the `FutureGroup` will be of type `int`.

The `add` method allows you to add several Futures in a `FutureGroup`. In our example, we added three Futures:

```
futureGroup.add(returnOneAsync());
futureGroup.add(returnTwoAsync());
futureGroup.add(returnThreeAsync());
```

Once all the Futures have been added, you always need to call the `close` method. This tells the framework that all the futures have been added, and the tasks are ready to be run:

```
futureGroup.close();
```

In order to read the values returned by the collection of Futures, you can leverage the `then()` method of the `future` property of the `FutureGroup`. The returned values are placed in a `List`, so you can use a `forEach` loop to read the values. You can also call the `setState()` method to update the UI:

```
futureGroup.future.then((List <int> value) {
     int total = 0;
     value.forEach((element) {
       total += element;
     });
     setState(() {
       result = total.toString();
     });
   });
 }
```

Resolving errors in asynchronous code

There are several ways to handle errors in your asynchronous code. In this recipe, you will see a few examples of dealing with errors, both using the

`then()` callback and the `async/await` pattern.

Getting ready

In order to follow along with this recipe:

- If you followed along with any of the previous recipes in this chapter, you'll only need to edit the existing project. Otherwise, create a new app with the starting code available at https://github.com/PacktPublishing/Flutter-Cookbook-Second-Edition/tree/main/Chapter_08.

How to do it...

We are going to divide this section into two sub-sections. In the first section, we will deal with the errors by using the `then()` callback function and in the second section, we will deal with those errors using the `async/await` pattern.

Dealing with errors using the then() callback:

The most obvious way to catch errors in a `then()` callback is using the `catchError` callback. To do so, follow these steps:

1. Add the following method to the `_FuturePageState` class in the `main.dart` file:

```
Future returnError() async {
  await Future.delayed(const Duration(seconds: 2));
  throw Exception('Something terrible happened!');
}
```

2. Whenever a method calls `returnError`, an error will be thrown. To catch this error, place the following code in the `onPressed` method of the `ElevatedButton`:

```
returnError()
    .then((value){
      setState(() {
        result = 'Success';
      });
    }).catchError((onError){
      setState(() {
        result = onError.toString();
      });
    }).whenComplete(() => print('Complete'));
```

3. Run the app without debugging and click the **GO!** button. You will see that the `catchError` callback was executed, updating the result `State` variable as shown in the following screenshot:

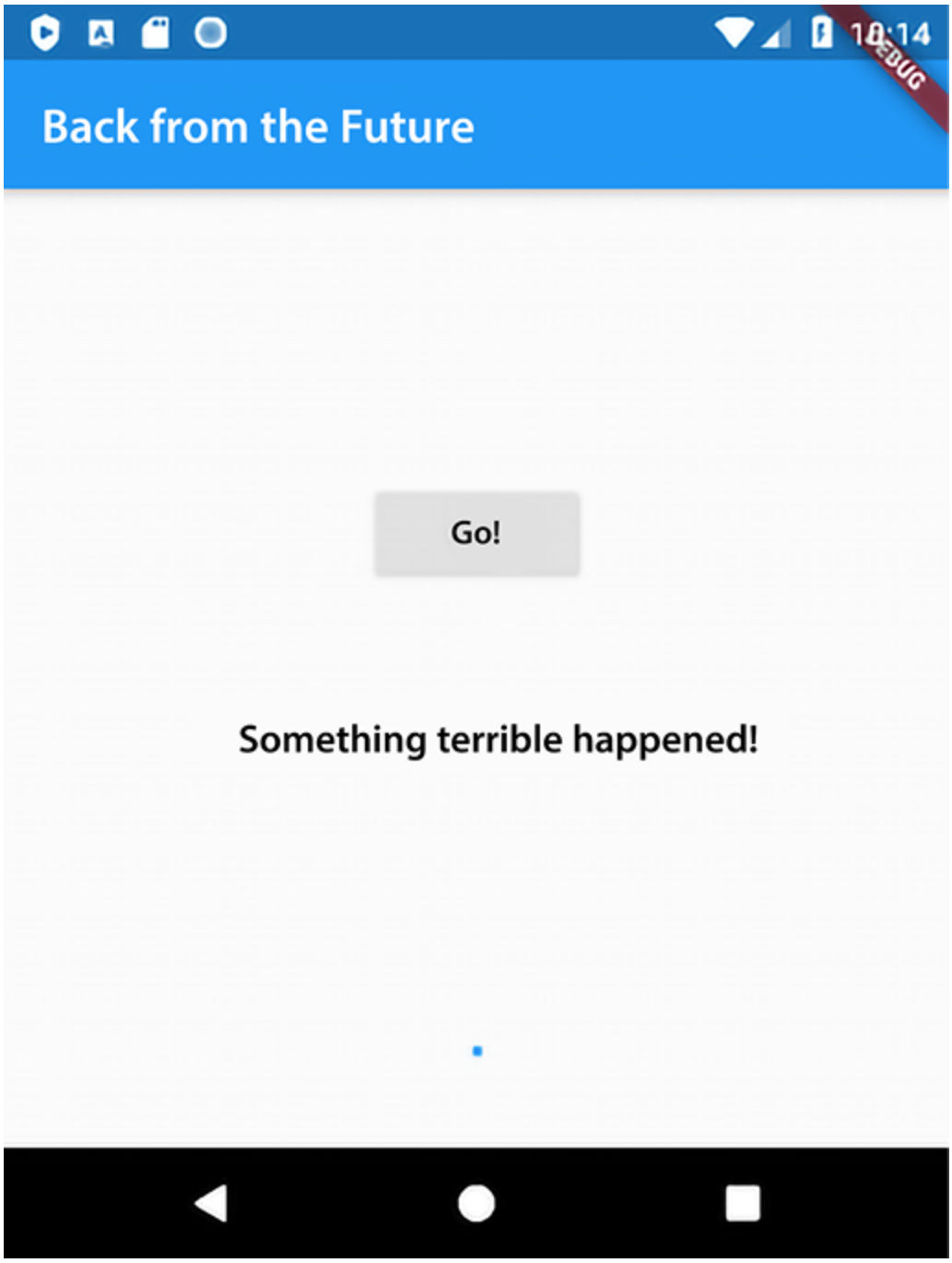


Figure 8.6: Exception message

4. You should also see that the `whenComplete` callback was called as well. In `DEBUG CONSOLE`, you should see the `Complete` string:



Figure 8.7: Debug Console message

Dealing with errors using async/await

When you use the async/await pattern, you can just handle errors with the `try… catch` syntax, exactly like you would when dealing with synchronous code.

1. Add a new method, called `handleError()` in the `_FuturePageState` class, as shown here:

```
Future handleError() async {
    try {
      await returnError();
    }
    catch (error) {
      setState(() {
        result = error.toString();
      });
    }
    finally {
      print('Complete');
    }
  }
```

2. Call `handleError()` in the `onPressed` method of the `ElevatedButton`.
3. Run the app: you should see that the `catch` was called, and the result is exactly the same as in the `catchError` callback.

How it works...

To sum it up, when an exception is raised during the execution of a `Future`, `catchError` is called; `whenComplete` is called in any case, both for

successful and error-raising Futures. When you use the async/await pattern, you can just handle errors with the `try… catch` syntax.

Again, handling errors with await/async is generally more readable than the callback equivalent.

See also

There's a very comprehensive guide on error handling in Dart. Each concept explained in this tutorial applies to Flutter as well: https://dart.dev/guides/libraries/futures-error-handling.

Using Futures with StatefulWidgets

As mentioned previously, while Stateless widgets do not keep any state information, **StatefulWidgets** can keep track of variables and properties, and in order to update the UI, you use the `setState()` method. `State` is information that can change during the life cycle of a widget.

There are four core lifecycle methods that you can leverage in order to use StatefulWidgets:

- `initState()` is only called once when the `State` is built. You should place the initial setup and starting values for your objects here. Whenever possible, you should prefer this to the `build()` method.
- `build()` gets called each time something changes. This will destroy the UI and rebuild it from scratch.
- `deactivate()` and `dispose()` are called when a widget is removed from the tree: use cases of these methods include closing a database connection or saving data before changing route.

So let's see how to deal with Futures in the context of the lifecycle of a widget.

Getting ready

In order to follow along with this recipe, you will need the following:

- If you followed any of the previous recipes in this chapter, you'll only need to edit the existing project. Otherwise, create a new app with the starting code available at https://github.com/PacktPublishing/Flutter-Cookbook-Second-Edition/tree/main/Chapter_08.
- For this recipe, we'll use the `geolocator` library, available at https://pub.dev/packages/geolocator. Add it to your `pubspec.yaml` file by typing in your Terminal:

```
flutter pub add geolocator
```

- You might need to grant Geolocation permissions to the app.
- If you are using Android, add the following lines to the `android/app/src/main/androidmanifest.xml` file:

```
<uses-permission
android:name="android.permission.ACCESS_FINE_
LOCATION"/>
<uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION"
/>
```

- If you are using iOS, add the following location permissions to `Info.plist`:

```
<key>NSLocationWhenInUseUsageDescription</key>
<string>This app needs to access your location</string>
```

How to do it...

In this example, we want to find the user location coordinates and show them on the screen as soon as they are ready. Getting the coordinates is an

asynchronous operation that returns a Future. Follow these steps:

1. Create a new file called `geolocation.dart` in the `lib` folder of your project.
2. Create a new Stateful widget, called `LocationScreen`.
3. In the `State` class of the `Geolocation` widget, add the code that shows the user their current position. The final result is shown here:

```
import 'package:flutter/material.dart';
import 'package:geolocator/geolocator.dart';
class LocationScreen extends StatefulWidget {
 const LocationScreen({super.key});
  @override
  State<LocationScreen> createState() =>
_LocationScreenState();
}
class _LocationScreenState extends
State<LocationScreen> {
  String myPosition = '';
  @override
  void initState() {
    super.initState();
    getPosition().then((Position myPos) {
      myPosition =
          'Latitude: ${myPos.latitude.toString()} -
Longitude: {myPos.longitude.toString()}';
      setState(() {
        myPosition = myPosition;
      });
    });

  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Current
Location')),
      body: Center(child: Text(myPosition)),
    );
  }
  Future<Position> getPosition() async {
    await Geolocator.requestPermission();
    await Geolocator.isLocationServiceEnabled();
    Position? position =
```

```
      await Geolocator. getCurrentPosition();
    return position;
  }}
```

4. In the `main.dart` file, in the `home` property of the `MaterialApp` in the `MyApp` class, add the call to `LocationScreen`:

```
home: LocationScreen(),
```

5. Run the app. After a few seconds, you should see your position at the center of the screen:



Figure 8.8: Geolocation in action

How it works...

`getPosition()` is an asynchronous method that returns a `Future`. It leverages a `Geolocator` class to retrieve the last known position of the user.

Now the question may arise: where should `getPosition` be called? As the position should be retrieved only once, the obvious choice should be leveraging the `initState` method, which only gets called once, when the widget is loaded.

It is recommended to keep `initState` synchronous, therefore you can use the `then` syntax to wait for the callback and update the state of the widget. `myPosition` is a state `String` variable that contains the message the user will see after the device has retrieved the coordinates, and it includes the latitude and longitude.

In the `build` method, there is just a centered `Text` containing the value of `myPosition`, which is empty at the beginning and then shows the string with the coordinates.

There's more...

There is no way to know exactly how long an asynchronous task might take, so it would be a good idea to give the user some feedback while the device is retrieving the current position, with a `CircularProgressIndicator`. What we want to achieve is to show the animation while the position is being retrieved, and as soon as the coordinates become available, hide the animation, and show the coordinates. We can achieve that with the following code in the `build()` method:

```
@override
 Widget build(BuildContext context) {
   final myWidget = myPosition == ''
     ? const CircularProgressIndicator()
     : const Text(myPosition);;
```

```
    return Scaffold(
      appBar: AppBar(title: Text('Current Location')),
      body: Center(child:myWidget),
    );
  }
```

Note

If you cannot see the animation of the `CircularProgressIndicator`, it might mean your device is too fast: try purposely adding a delay before the `Geolocator()` call, with the instruction `await Future.delayed(const Duration(seconds: 3));`.

There is also an easier way to deal with Futures and StatefulWidgets: we'll see it in the next recipe!

See also

It is extremely important to understand the widget lifecycle in Flutter. Have a look at the official documentation here for more details: https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html.

Using the FutureBuilder to let Flutter manage your Futures

The pattern of retrieving some data asynchronously and updating the user interface based on that data is quite common. So common in fact that in Flutter, there is a widget that helps you remove some of the boilerplate code you need to build the UI based on Futures: it's the **FutureBuilder** widget.

You can use a `FutureBuilder` to integrate Futures within a widget tree that automatically updates its content when the `Future` updates. As a `FutureBuilder` builds itself based on the status of a Future, you can skip the `setState` instruction, and Flutter will only rebuild the part of the user interface that needs updating.

`FutureBuilder` implements *reactive programming*, as it will take care of updating the user interface as soon as data is retrieved, and this is probably the main reason why you *should* use it in your code: it's an easy way for the UI to react to data in a `Future`.

`FutureBuilder` requires a `future` property, containing the `Future` object whose content you want to show, and a `builder`. In the `builder`, you actually build the user interface, but you can also check the status of the data: in particular, you can leverage the `connectionState` of your data so you know exactly when the `Future` has returned its data.

For this recipe, we will build the same UI that we built in the previous recipe: *Using Futures with StatefulWidgets*. We will find the user location coordinates and show them on the screen, leveraging the `Geolocator` library, available at https://pub.dev/packages/geolocator.

Getting ready

You should have completed the project in the previous recipe, *Using Futures with StatefulWidgets*, before starting this one.

How to do it...

To implement a `Futurebuilder`, follow these steps:

1. Modify the `getPosition()` method. This will wait 3 seconds and then retrieve the current device's position:

```
  Future<Position> getPosition() async {
    await Geolocator.isLocationServiceEnabled();
    await Future.delayed(const Duration(seconds: 3));
    Position position = await Geolocator.
getCurrentPosition ();
    return position;  }
```

2. Add a Future at the top of the `_LocationScreenState` class:

```
Future<Position>? position;
```

3. Add an initState method that sets the position variable:

```
@override
  void initState() {
    super.initState();
    position = getPosition();
  }
```

4. Write the following code in the `build` method of the `State` class:

```
 @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Current Location')),
      body: Center(child: FutureBuilder(
        future: position,
        builder: (BuildContext context,
AsyncSnapshot<Position>
         snapshot) {
          if (snapshot.connectionState ==
           ConnectionState.waiting) {
            return const CircularProgressIndicator();
          }
          else if (snapshot.connectionState ==
           ConnectionState.done) {
            return Text(snapshot.data.toString());
          }
          else {
            return const Text('');
          }
        },
        ),
    ));
  }
```

How it works...

In this recipe, `getPosition` is the `Future` we will pass to the `FutureBuilder`. In this case, `initState` is not required at all: `FutureBuilder` takes care of updating the user interface whenever there's a change in the data and state information.

Note that there are two properties we are setting for `FutureBuilder`: the `future`, which in this example is our `getPosition()` method, and the `builder`.

The `builder` takes the current context and an `AsyncSnapshot`, containing all the Future data and state information: the builder must return a widget.

The `connectionState` property of the `AsyncSnapshot` object makes you check the state of the `Future`. In particular, you have the following:

- `waiting` means the Future was called but has not yet completed its execution.
- `done` means that the execution completed.

There's more...

Now, we should never take for granted that a future completed the execution without any error. For exception handling, you can check whether the future has returned an error, making this class a complete solution to build your Future-based user interface. You can actually catch errors in a `FutureBuilder`, checking the `hasError` property of the `Snapshot`, as shown in the following code:

```
else if (snapshot.connectionState == ConnectionState.done) {
          if (snapshot.hasError) {
            return Text('Something terrible happened!');
          }
          return Text(snapshot.data.toString());
        }
```

As you can see, `FutureBuilder` is an efficient, clean, and reactive way to deal with Futures in your user interface.

See also

`FutureBuilder` can really enhance your code when composing a UI that depends on a Future. There is a guide and a video to dig deeper at this address: https://api.flutter.dev/flutter/widgets/FutureBuilder-class.html.

Turning navigation routes into asynchronous functions

In this recipe, you will see how to leverage Futures using `Navigator` to transform a `Route` into an `async` function: you will push a new screen in the app and then *await* the route to return some data and update the original screen.

The steps we will follow are these:

- Adding an `ElevatedButton` that will launch the second screen.
- On the second screen, we will make the user choose a color.
- Once the color is chosen, the second screen will update the background color on the first screen.

Here, you can see a screenshot of the first screen:

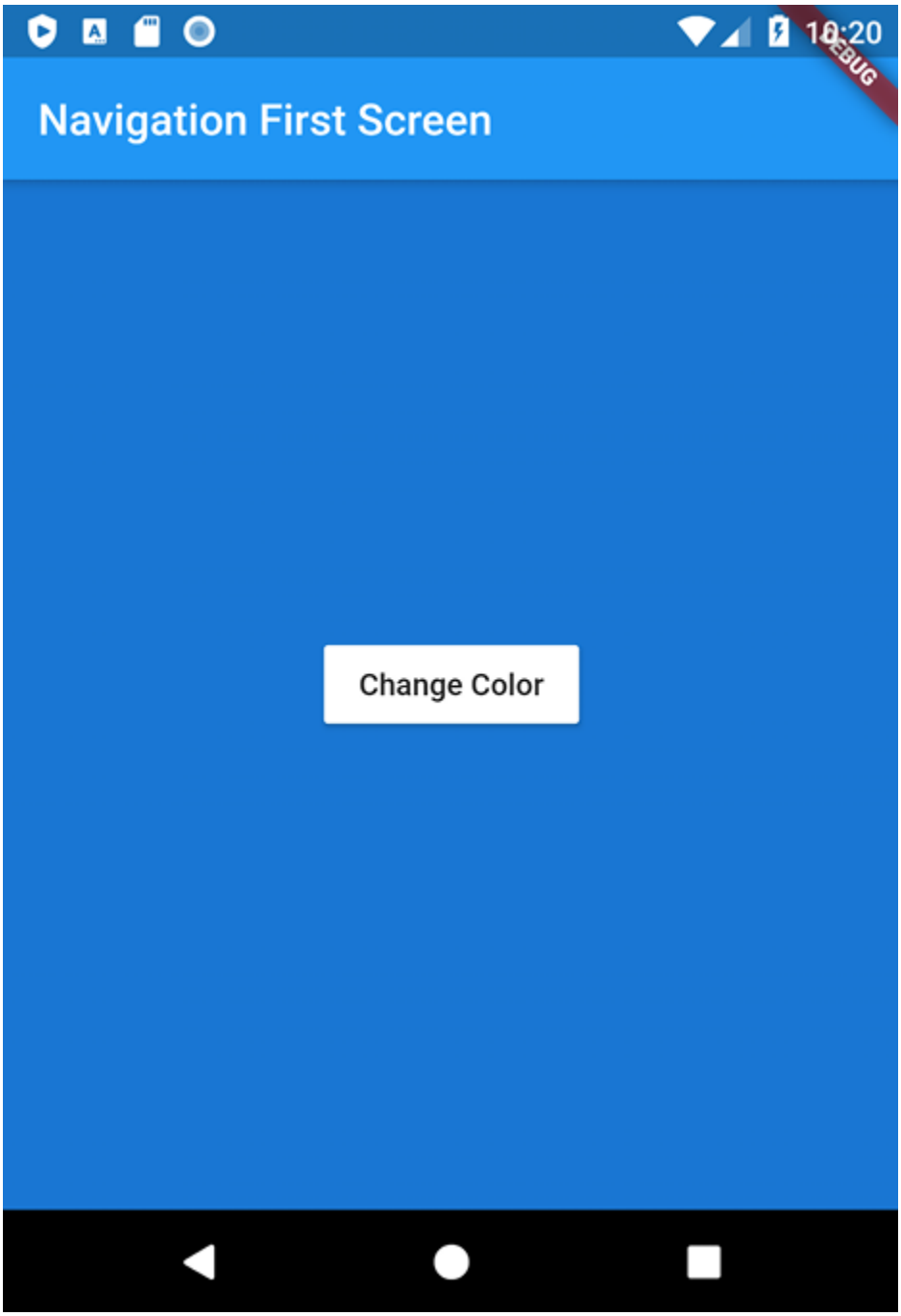


Figure 8.9: Navigation first screen

And here, the second screen, which allows choosing a color with three `ElevatedButtons`:

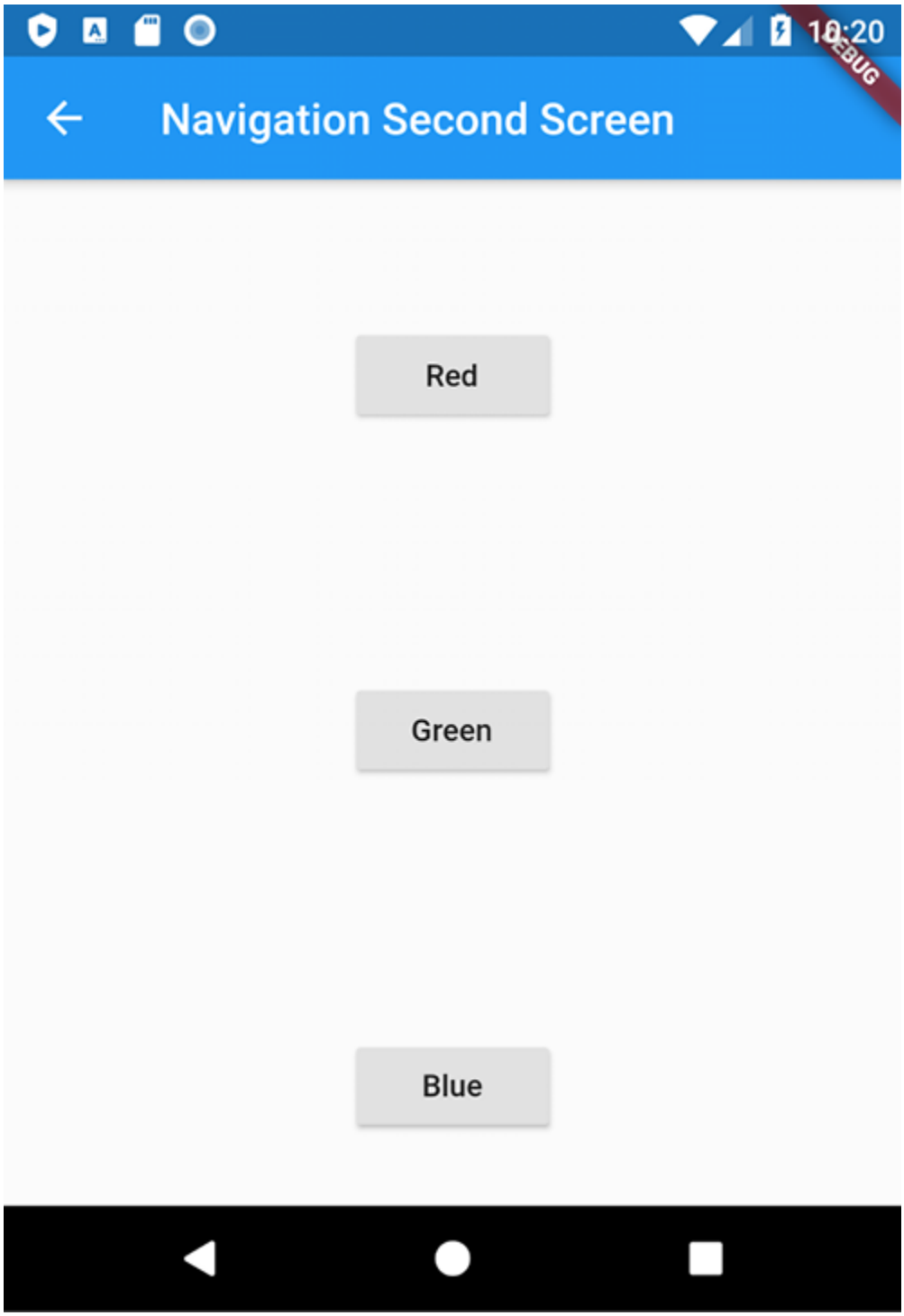


Figure 8.10: Navigation screen with buttons

Getting ready

If you followed along with any of the previous recipes in this chapter, you'll only need to edit the existing project. Otherwise, create a new app with the starting code available at https://github.com/PacktPublishing/Flutter-Cookbook-Second-Edition/tree/main/Chapter_08.

How to do it...

We will begin by creating the first screen, which is a stateful widget containing a `Scaffold` with a centered `ElevatedButton`. Follow these steps:

1. Create a new file, called `navigation_first.dart`.
2. Add the following code to the `navigation_first.dart` file (note that in the `onPressed` of the button, we are calling a method that does not exist yet, called `_navigateAndGetColor()`):

```
import 'package:flutter/material.dart';
class NavigationFirst extends StatefulWidget {
  const NavigationFirst({super.key});
  @override
  State<NavigationFirst> createState() =>
_NavigationFirstState();
}
class _NavigationFirstState extends
State<NavigationFirst> {
  Color color = Colors.blue.shade700;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: color,
      appBar: AppBar(
        title: const Text('Navigation First Screen'),
      ),
      body: Center(
        child: ElevatedButton(
            child: const Text('Change Color'),
            onPressed: () {
              _navigateAndGetColor(context);
            }),
      ),
    );
  }  }
```

3. Here comes the most interesting part of this recipe: we want to await the result of the navigation. The `_navigateAndGetColor` method will launch `NavigationSecond` and `await` the result from the `Navigator.pop()` method on the second screen. Add the following code for this method at the bottom of the `_NavigationFirstState` class:

```
Future _navigateAndGetColor(BuildContext context) async
{
    color = await Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => const
NavigationSecond()),
    ) ?? Colors.blue;
    setState(() {});
    });
  }
```

4. Create a new file called `navigation_second.dart`.
5. In the `navigation_second.dart` file, add a new stateful widget, called `NavigationSecond`. This will just contain three buttons: one for blue, one for green, and one for red.
6. Add the following code to complete the screen:

```
import 'package:flutter/material.dart';
class NavigationSecond extends StatefulWidget {
  const NavigationSecond({super.key});
  @override
  State<NavigationSecond> createState() =>
_NavigationSecondState();
}
class _NavigationSecondState extends
State<NavigationSecond> {
  @override
  Widget build(BuildContext context) {
    Color color;
    return Scaffold(
        appBar: AppBar(
          title: const Text('Navigation Second Screen'),
        ),
        body: Center(
          child: Column(
```

```
            mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
            children: [
              ElevatedButton(
                child: const Text('Red'),
                  onPressed: () {
                    color = Colors.red.shade700;
                    Navigator.pop(context, color);
                  }),
              ElevatedButton(
                  child: const Text('Green'),
                  onPressed: () {
                    color = Colors.green.shade700;
                    Navigator.pop(context, color);
                  }),
              ElevatedButton(
                  child: const Text('Blue'),
                  onPressed: () {
                    color = Colors.blue.shade700;
                    Navigator.pop(context, color);
                  }),
            ],
          ),
        ));
  }}
```

7. In the `home` property of the `MaterialApp` in the `main.dart` method, call `NavigationFirst`:

```
home: const NavigationFirst(),
```

8. Run the app and try changing the colors of the screen.

How it works...

The key for this code to work is passing data from the `Navigator.pop()` method from the second screen. The first screen is expecting a `Color`, so the `pop` method returns a `Color` to the `NavigationFirst` screen.

This pattern is an elegant solution whenever you need to await a result that comes from a different screen in your app. We can actually use this same

pattern when we want to await a result from a dialog window, which is exactly what we will do in the next recipe!

Getting the results from a dialog

This recipe shows an alternative way of await-ing some data from another screen, which was shown in the previous recipe, but this time, instead of using a full-sized page, we will use a dialog box: actually dialogs behave just like routes that can be *await*-ed.

An `AlertDialog` can be used to show pop-up screens that typically contain some text and buttons, but could also contain images or other widgets. `AlertDialogs` may contain a `title`, some `content`, and `actions`. The `actions` property is where you ask for the user's feedback (think of "save," "delete," or "accept").

There are also design properties such as elevation or background, shape or color, that help you make an `AlertDialog` well integrated into the design of your app.

In this recipe, we'll perform the same actions that we implemented in the previous recipe, *Turning navigation routes into asynchronous functions*. We will ask the user to choose a color in the dialog, and then change the background of the calling screen according to the user's answer, leveraging Futures.

The dialog will look like the one shown in the following screenshot:

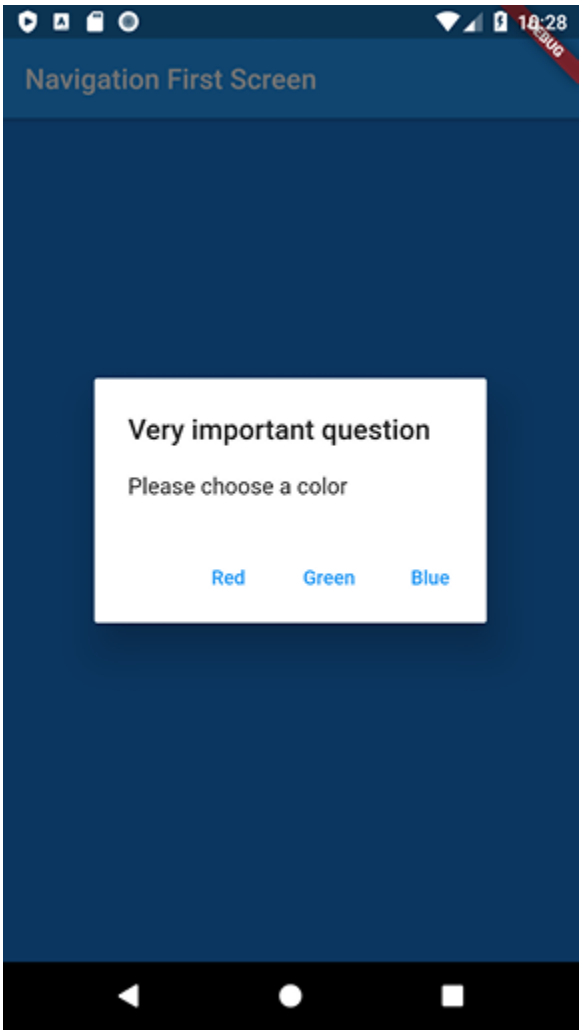


Figure 8.11: Dialog showing on the screen

Getting ready

In order to follow along with this recipe, you need the project built in any of the previous recipies, or you can create a new app and copy the code at https://github.com/PacktPublishing/Flutter-Cookbook-Second-Edition/tree/main/Chapter_08.

How to do it...

To create an asynchronous dialog for your app:

1. Add a new file to your project, calling it `navigation_dialog.dart`.
2. Add the following code to the `navigation_dialog.dart` file:

```
import 'package:flutter/material.dart';
class NavigationDialogScreen extends StatefulWidget {
  const NavigationDialogScreen ({super.key});
  @override
  State<NavigationDialogScreen> createState() =>
_NavigationDialogScreenState();
}
class _NavigationDialogScreenState extends
State<NavigationDialogScreen> {
  Color color = Colors.blue.shade700;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: color,
      appBar: AppBar(
        title: const Text('Navigation Dialog Screen'),
      ),
      body: Center(
        child:
            ElevatedButton(child: const Text('Change
Color'), onPressed: () {}),
      ),
    );
  }  }
```

3. Now create the asynchronous method that will return the chosen color, calling it `showColorDialog`, and marking it as `async`:

```
_showColorDialog(BuildContext context) async {
  await showDialog(
      barrierDismissible: false,
      context: context,
      builder: (_) {
        return AlertDialog(
          title: const Text('Very important question'),
          content: const Text('Please choose a color'),
          actions: <Widget>[
            TextButton(
                child: const Text('Red'),
                onPressed: () {
                  color = Colors.red.shade700;
                  Navigator.pop(context, color);
```

```
                }),
            TextButton(
                child: const Text('Green'),
                onPressed: () {
                  color = Colors.green.shade700;
                  Navigator.pop(context, color);
                }),
            TextButton(
                child: const Text('Blue'),
                onPressed: () {
                  color = Colors.blue.shade700;
                  Navigator.pop(context, color);
                }),
          ],
        );
      },
    );
    setState(() {});
  }
```

4. In the `build` method, in the `onPressed` property of the `ElevatedButton`, call the `_showColorDialog` that you just created:

```
onPressed: () {
  _showColorDialog(context);
}),
```

5. In the `home` property of the `MaterialApp` in the `main.dart` method, call `NavigationDialog`:

```
home: const NavigationDialog(),
```

6. Run the app and try changing the background `color` of the screen.

How it works...

In the preceding code, note that the `barrierDismissible` property tells whether the user can tap outside of the dialog box to close it (`true`) or not (`false`). The default value is `true`, but as this is a "very important question," we set it to `false`.

The way we close the alert is by using the `Navigator.pop` method, passing the color that was chosen: in this, an `Alert` works just like a `Route`.

Now we only need to call this method from the `onPressed` property of the "`Change color`" button:

```
onPressed: () {
                _showColorDialog(context);
});
```

This recipe showed the pattern of waiting asynchronously for data coming from an alert dialog.

See also

If you are interested in diving deeper into dialogs in Flutter, have a look at this link: https://api.flutter.dev/flutter/material/AlertDialog-class.html.

Summary

In this chapter, you have learned several techniques to work with asynchronous code in Flutter, using `Future` objects to handle tasks that may take some time to complete.

You have seen how to use the `async`/`await` pattern to avoid `then` callbacks, making your code more readable and easier to maintain.

You have learned how to fire multiple Futures at the same time and manage multiple asynchronous tasks, and seen how to deal with errors in your asynchronous code.

By using `FutureBuilder` objects, you have allowed Flutter to manage your Futures, providing a more efficient and responsive application.

You have turned navigation routes into asynchronous functions, optimizing the user experience.

Finally, you have seen how to get results from a dialog, ensuring a smooth interaction between different parts of your application.

Mastering asynchronous programming techniques can help you create responsive, efficient, and robust Flutter apps.

Join us on Discord

Read this book alongside other app developers.

Ask questions, participate in challenges, provide solutions to other readers, network and much more.

Scan the QR code or visit the link to join the community

https://packt.link/fluttercookbook



Data Persistence and Communicating with the Internet

Most applications, especially business applications, need to perform **CRUD** tasks: **Create, Read, Update, and Delete** data. In this chapter, we will cover recipes that explain how to perform CRUD operations in Flutter.

Data can be persisted locally or remotely. Regardless of the destination of your data, in several cases, it will need to be transformed into JSON before it can be persisted, so we will begin by talking about the JSON format in Dart and Flutter. This will be helpful for several technologies you might decide to use in your future apps, including SQLite, Sembast, and Firebase databases. These are all based on sending and retrieving JSON data.

Interacting with data generally requires an asynchronous connection, so we'll make use of Futures, which we discussed in the previous chapter.

In this chapter, we will cover the following recipes:

- Converting Dart models into JSON
- Handling JSON schemas that are incompatible with your models
- Catching common JSON errors
- Saving data simply with SharedPreferences
- Accessing the filesystem, part 1: path_provider
- Accessing the filesystem, part 2: Working with directories
- Using secure storage to store data
- Designing an HTTP client and getting data
- POST-ing data
- PUT-ting data

- DELETE-ing data

By the end of this chapter, you will know how to deal with JSON data in your apps so that you can interact with databases and web services.

Technical requirements

To follow along with the recipes in this chapter, you should have the following software installed on your Windows, Mac, Linux, or Chrome OS device:

- The Flutter SDK
- The Android SDK, when developing for Android
- macOS and Xcode, when developing for iOS
- An emulator or simulator, or a connected mobile device enabled for debugging
- An internet connection to use web services
- Your favorite code editor: Android Studio, Visual Studio Code, and IntelliJ IDEA are recommended, and all should have the Flutter/Dart extensions installed

You'll find the code for the recipes in this chapter on GitHub at https://protect-eu.mimecast.com/s/GBS3CX70XtXOvn5fDLsCZ?domain=github.com.

Converting Dart models into JSON

JSON has become the standard format for storing, transporting, and sharing data between applications: several web services and databases use JSON to receive and serve data. Later in this chapter, we will learn how to store data in a device, but in this recipe, we will learn how to serialize (or encode) and deserialize (or decode) data structures from/to JSON.

JSON stands for **JavaScript Object Notation**. Even if it largely follows JavaScript syntax, it can be used independently from Javascript. Most modern languages, including Dart, have methods to read and write JSON data.

The following screenshot shows an example of a JSON data structure:



Figure 9.1: JSON format

As you can see, **JSON** is a text-based format that describes data in **key-value** pairs: each object (a pizza, in this example) is included in curly brackets. In the object, you can specify several properties or fields. The key is generally included in quotes; then, you put a colon, and then the value. All key-value pairs and objects are separated from the next by a comma. In fact, JSON objects are very close to Dart `Map` objects.

JSON is basically a `String`. When you write an app in an object-oriented programming language such as Dart, you usually need to convert the JSON string into an object so that it works properly with the data: this process is called **de-serialization** (or **decoding**). When you want to generate a JSON string from your object, you need to perform the opposite, which is called **serialization** (or **encoding**).

In this recipe, we will perform both encoding and decoding using the built-in `dart:convert` library.

Getting ready

To follow along with this recipe, you will need some JSON data that can be copied from https://protect-eu.mimecast.com/s/GBS3CX70XtXOvn5fDLsCZ?domain=github.com.

How to do it...

For this recipe, we will create a new app that shows how to serialize and deserialize JSON Strings:

1. In your favorite editor, create a new Flutter project and call it `store_data`.
2. In the `main.dart` file, delete the existing code and add the starting code for the app. The starting code for this recipe is also available at https://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_08, in case you don't want to type it in:

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter JSON Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const MyHomePage(),
    );
  }
```

```
}
class MyHomePage extends StatefulWidget {
  const MyHomePage({super.key});
  @override
  State<MyHomePage> createState() => _MyHomePageState();
}
class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('JSON')),
      body: Container(),
    );
  }
}
```

3. Add a new folder to the root of your project called `assets`.
4. In the `assets` folder, create a new file called `pizzalist.json` and copy the content available at the link provided in the *Getting ready* section of this recipe. It contains a list of JSON objects.
5. In the `pubspec.yaml` file, add a reference to the new `assets` folder, as shown here:

```
assets:
  - assets/
```

6. In the `_MyHomePageState` class, in `main.dart`, add a state variable called `pizzaString`:

```
String pizzaString = '';
```

7. To read the content of the `pizzalist.json` file, at the bottom of the `_MyHomePageState` class in `main.dart`, add a new asynchronous method called `readJsonFile`, which will set the `pizzaString` value, as shown here:

```
Future readJsonFile() async {
    String myString = await
DefaultAssetBundle.of(context)
    .loadString('assets/pizzalist.json');
    setState(() {
```

```
    pizzaString = myString;
    });
}
```

8. In the `_MyHomePageState` class, override the `initState` method and, inside it, call the `readJsonFile` method:

```
@override
void initState() {
  super.initState();
  readJsonFile();
}
```

9. Now, we want to show the retrieved JSON in the `body` property of `Scaffold`. To do that, add a `Text` widget as a child of our `Container`:

```
body: Text(pizzaString),
```

10. Let's run the app. If everything worked as expected, you should see the content of the JSON file on the screen, as shown in *Figure 9.2*:

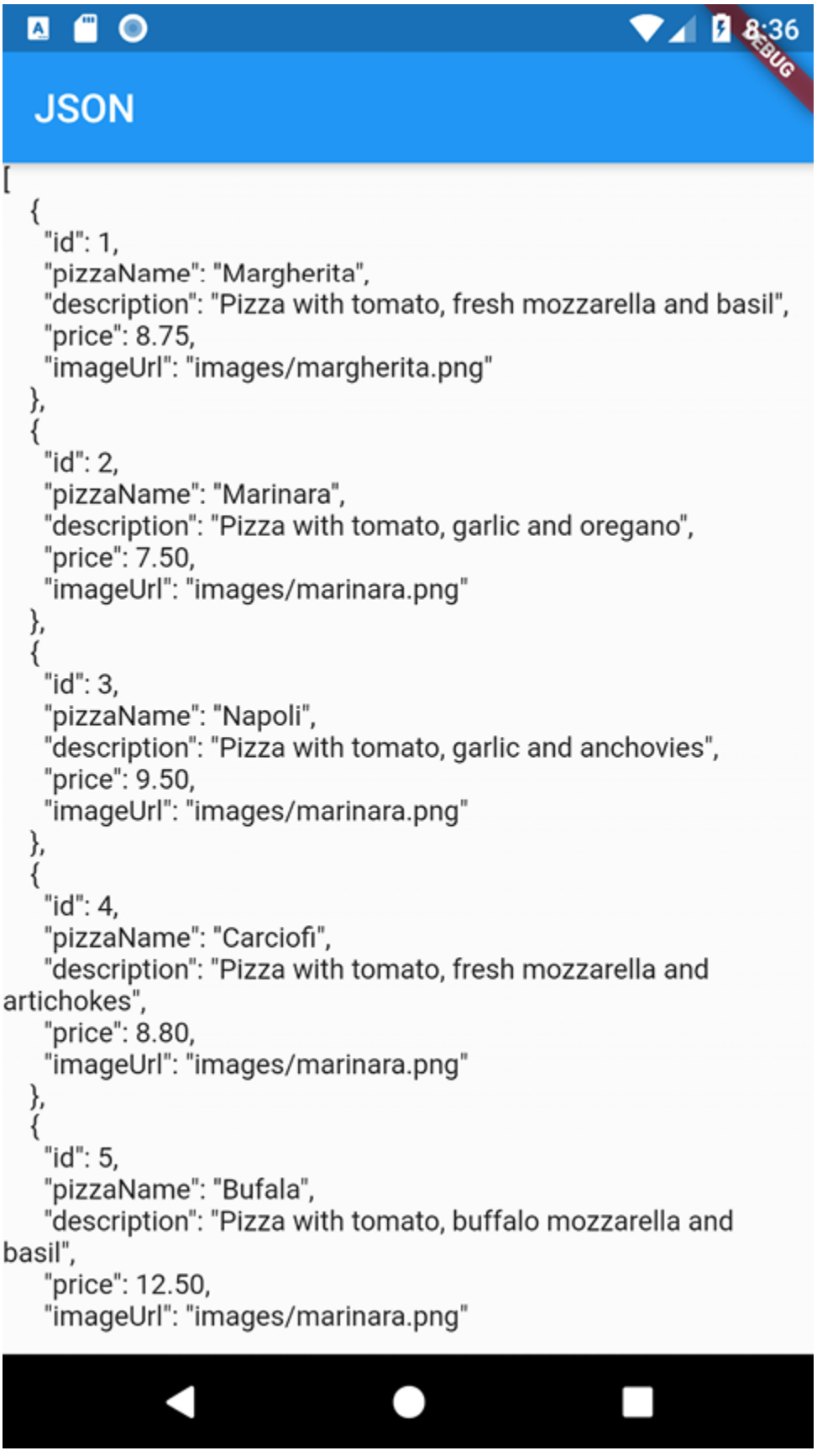

8:36
DEBUG
JSON
[
{
"id": 1,
"pizzaName": "Margherita",
"description": "Pizza with tomato, fresh mozzarella and basil",
"price": 8.75,
"imageUrl": "images/margherita.png"
},
{
"id": 2,
"pizzaName": "Marinara",
"description": "Pizza with tomato, garlic and oregano",
"price": 7.50,
"imageUrl": "images/marinara.png"
},
{
"id": 3,
"pizzaName": "Napoli",
"description": "Pizza with tomato, garlic and anchovies",
"price": 9.50,
"imageUrl": "images/marinara.png"
},
{
"id": 4,
"pizzaName": "Carciofi",
"description": "Pizza with tomato, fresh mozzarella and artichokes",
"price": 8.80,
"imageUrl": "images/marinara.png"
},
{
"id": 5,
"pizzaName": "Bufala",
"description": "Pizza with tomato, buffalo mozzarella and basil",
"price": 12.50,
"imageUrl": "images/marinara.png"

Figure 9.2: JSON content showing in the app

11. We want to transform this `String` into a `List` of objects. We'll begin by creating a new class. In the `lib` folder of our app, create a new file called `pizza.dart`.
12. Inside the file, define the properties of the `Pizza` class:

```
class Pizza {
    final int id;
    final String pizzaName;
    final String description;
    final double price;
    final String imageUrl;
}
```

13. Inside the `Pizza` class, define a named constructor called `fromJson`, which will take a `Map` as a parameter and transform `Map` into an instance of a `Pizza`:

```
Pizza.fromJson(Map<String, dynamic> json) :
    id = json['id'],
    pizzaName = json['pizzaName'],
    description = json['description'],
    price = json['price'],
    imageUrl = json['imageUrl'];
```

14. Refactor the `readJsonFile()` method in the `_MyHomePageState` class. The first step is transforming the `String` into a `Map` by calling the `jsonDecode` method. In the `readJsonFile` method, add the following highlighted code:

```
Future readJsonnFile() async {
    String myString = await
DefaultAssetBundle.of(context)
        .loadString('assets/pizzalist.json');
List pizzaMapList = jsonDecode(myString);
...
```

15. Make sure that your editor automatically added the `import` statement for the `dart:convert` library at the top of the `main.dart` file;

otherwise, just add it manually. Also, add the `import` statement for the `pizza` class:

```
import 'dart:convert';
import './pizza.dart';
```

16. The last step is to convert our JSON string into a `List` of native Dart objects. We can do this by looping through `pizzaMapList` and transforming it into `Pizza` objects. In the `readJsonFile` method, under the `jsonDecode` method, add the following code:

```
List<Pizza> myPizzas = [];
for (var pizza in pizzaMapList) {
  Pizza myPizza = Pizza.fromJson(pizza);
  myPizzas.add(myPizza);
 }
```

17. Remove or comment out the `setState` method that sets the `pizzaString` String and return the list of `Pizza` objects instead:

```
return myPizzas;
```

18. Change the signature of the method so that you show the return value explicitly:

```
Future<List<Pizza>> readJsonFile() async {
```

19. Now that we have a `List` of `Pizza` objects, we can use it. Instead of just showing our user a `Text`, we can show a `ListView` containing a set of `ListTile` widgets. At the top of the `_MyHomePageState` class, create `List<Pizza>` called `myPizzas`:

```
   List<Pizza> myPizzas = [];
```

20. In the `initState` method, make sure you set `myPizzas` with the result of the call to `readJsonFile`:

```
  @override
  void initState() {
```

```
    super.initState();
    readJsonFile().then((value) {
      setState(() {
        myPizzas = value;
      });
    });
  }
```

21. Add the following code in the body of `Scaffold`, in the `build()` method:

```
body: ListView.builder(
          itemCount: myPizzas.length,
          itemBuilder: (context, index) {
            return ListTile(
              title: Text(myPizzas[index].pizzaName),
              subtitle:
Text(myPizzas[index].description),
            );
          },
        ));
      }
```

22. Run the app. The user interface should now be much friendlier and look as shown in *Figure 9.3*:



Figure 9.3: ListView containing custom objects

How it works...

The main features of the app we have built in this recipe are:

- Reading the JSON file

- Transforming the JSON string into a list of `Map` objects
- Transforming the `Map` objects into `Pizza` objects

Let's see how we achieved each of those in this recipe.

Reading the JSON file

Generally, you get JSON data from a web service or a database. In our example, the JSON was contained in an asset file. In any case, regardless of your source for the data, reading data is generally an **asynchronous** task. That's why the `readJsonFile` method was set to `async` from the very beginning.

When you read from a file that's been loaded into the assets, you can use the `DefaultAssetBundle.of(context)` object, as we did with the following instruction:

```
String myString = await
DefaultAssetBundle.of(context).loadString('assets/pizzalist.j
son');
```

Note

An asset is a file that you can deploy with your app and access at runtime. Examples of assets include configuration files, images, icons, text files, and, as in this example, some data in JSON format.

When you add assets in Flutter, you need to specify their position in the `pubspec.yaml` file, in the `assets` key:

```
assets:
- assets/
```

Transforming the JSON string into a list of Map objects

The `jsonDecode` method, available in the `dart:convert` library, decodes JSON. In this case, it takes a JSON string and transforms it into a `List` of `Map` objects:

```
List myMap = jsonDecode(myString);
```

A `Map` is a key-value pair set. The key is a String: `id`, `pizzaName`, `description`, and `price` are all keys and strings. The types of the values can change based on the key: `price` is a number, while `description` is a string.

After executing the `jsonDecode` method, we have a list of `Map` objects that show exactly how a `Map` is composed:

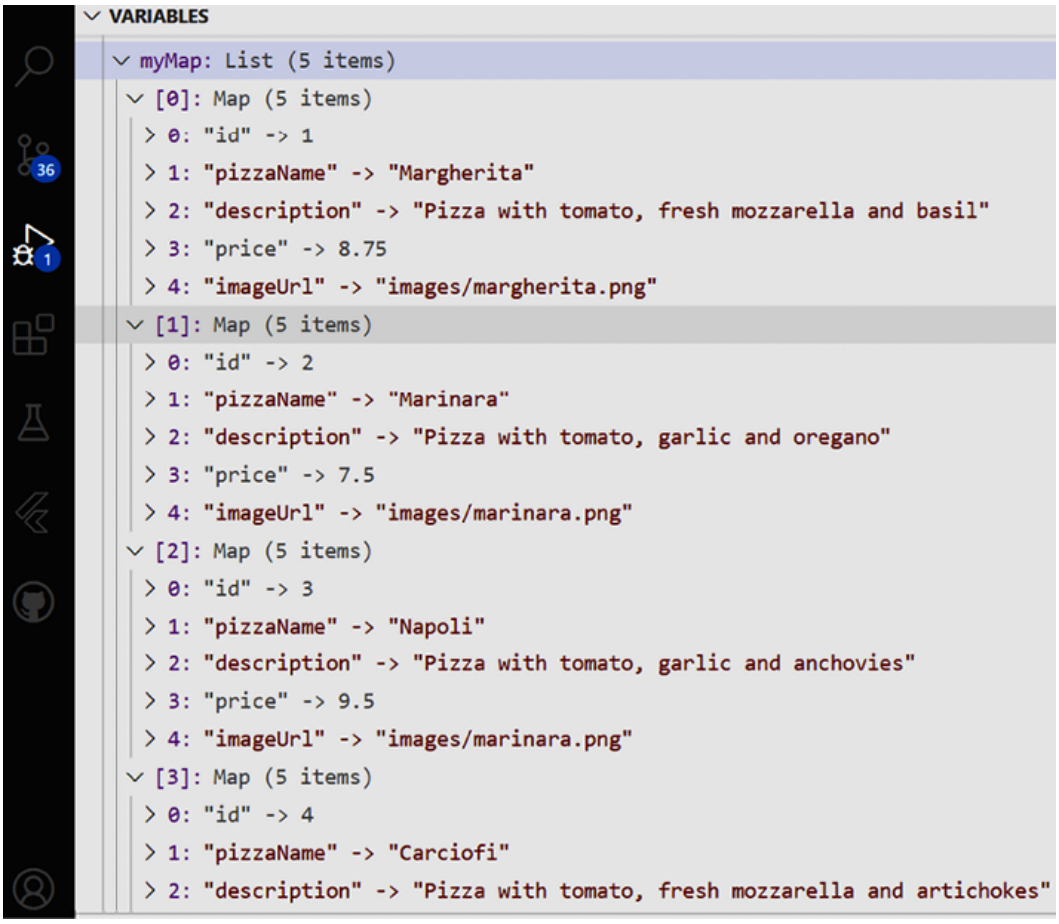


Figure 9.4: Maps showing in VS Code Variables pane

Transforming the Map objects into Pizza objects

In this recipe, we transformed the `Map` objects into `Pizza` objects. To do this, we created a constructor in the `Pizza` class that takes `Map<String, dynamic>` and, from that, creates a new `Pizza` object:

```
Pizza.fromJson(Map<String, dynamic> json) {
    id = json['id'];
    pizzaName = json['pizzaName'];
    description = json['description'];
    price = json['price'];
    imageUrl = json['imageUrl'];
}
```

The interesting part of this method is how you access the values of the map—that is, using square brackets and the name of the key (for example, `json['pizzaName']` takes the name of `Pizza`).

When this named constructor has finished executing, you get a `Pizza` object that's ready to be used in your app.

There's more...

While deserializing JSON is an extremely common task, there are cases where you also need to **serialize** some JSON. It's very important to learn how to transform a Dart class into a JSON string. Follow these steps:

1. Add a new method to the `Pizza` class, in the `pizza.dart` file, called `toJson`. This will actually return a `Map<String, dynamic>` from the object:

```
Map<String, dynamic> toJson() {
  return {
    'id': id,
    'pizzaName': pizzaName,
    'description': description,
    'price': price,
    'imageUrl': imageUrl,
  };
}
```

2. Once you have a `Map`, you can serialize it back into a JSON string. Add a new method at the bottom of the `_MyHomePageState` class, in the `main.dart` file, called `convertToJSON`:

```
String convertToJSON(List<Pizza> pizzas) {
  return jsonEncode(pizzas.map((pizza) =>
jsonEncode(pizza)).toList());
}
```

3. This method transforms our `List` of `Pizza` objects back into a Json string by calling the `jsonEncode` method again in the `dart_convert`

library.

4. Finally, let's call the method and print the JSON string in the Debug Console. Add the following code to the `readJsonFile` method, just before returning the `myPizzas` List:

```
...
String json = convertToJSON(myPizzas);
print(json);
return myPizzas;
```

Run the app. You should see the JSON string printed out, as shown in the following screenshot:



Figure 9.5: Deserialized JSON in the Debug Console

Now, you know how to serialize and deserialize JSON data in your app. This is the first step in dealing with web services and several databases in Flutter.

See also

In this recipe, we manually serialized and deserialized JSON content. A great resource for understanding this process is the official Flutter guide, available at https://flutter.dev/docs/development/data-and-backend/json.

Handling JSON schemas that are incompatible with your models

In a perfect world, JSON data would always be 100% compatible with your Dart classes, so the code you completed in the previous recipe would be ready to go to production and never raise errors. But as you are fully aware, this world is far from perfect, and so is the data you are likely to deal with in your apps. That's why, in this recipe, you will learn how to transform your code and make it more solid and resilient, and, hopefully, production ready.

Getting ready

To follow along with this recipe, you need to do the following:

- Complete the previous recipe.
- Download a more real-world `pizzalist_broken.json` file at https://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_08.
- Rename the file to `pizzalist.json`.

How to do it...

For this recipe, using the project we built in the previous recipe, we will deal with a real-world JSON file, available at https://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_08. Instead of being perfectly compatible with our class, it will have some inconsistencies and missing fields. This will force us to refactor our code, which will make it more resilient and less error-prone. To do this, follow these steps:

1. Make sure `pizzalist.json` has the content of `pizzalist_broken.json` at https://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_08, as detailed in the *Getting ready* section for this recipe.
2. Run the app. You should get the following error:

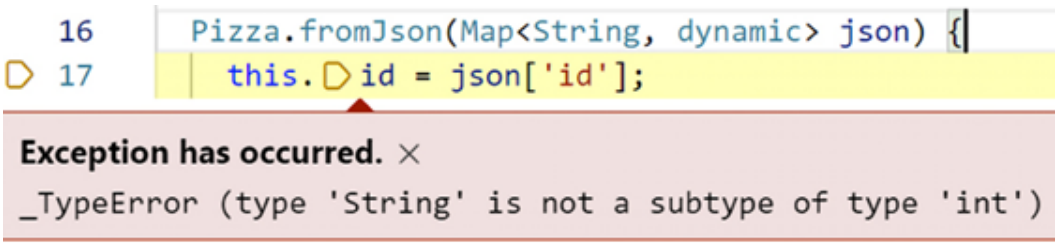


Figure 9.6: JSON error

3. Edit the `Pizza.fromJson` constructor method in the `Pizza.dart` file, and try transforming the `json['id']` value into an integer:

```
int.tryParse(json['id'].toString()) ?? 0;
```

4. Restart the app, and note the error has now changed:

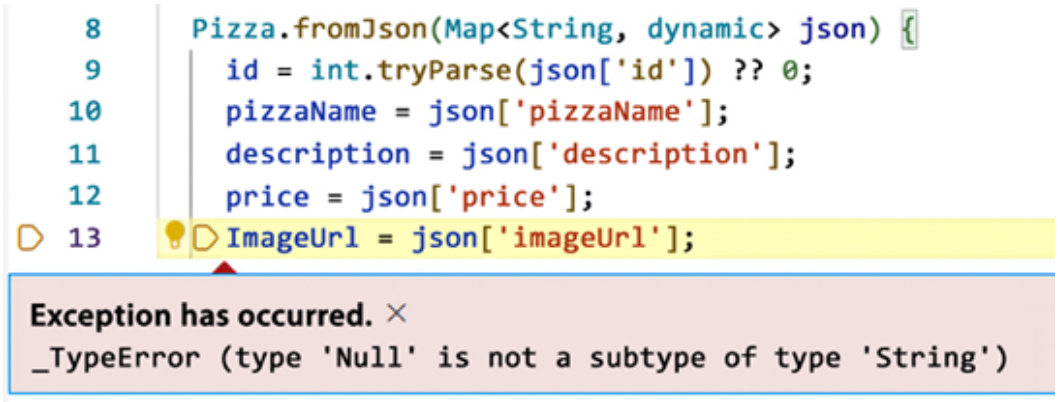


Figure 9.7: JSON Null error on imageUrl

5. Add a null coalescing operator to the `imageUrl` value, and restart the app:

```
imageUrl = json['imageUrl'] ?? '';
```

The error has changed again:

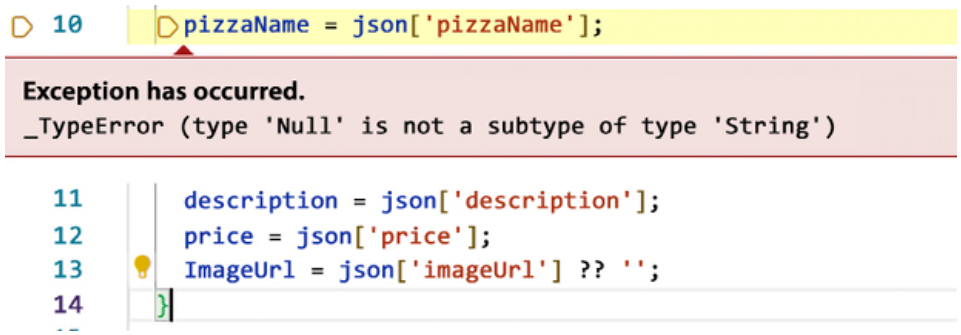


Figure 9.8: JSON Null error on pizzaName

6. Edit the `Pizza.fromJson` constructor method in the `Pizza.dart` file by adding a `toString()` method to the `String` values in the method:

```
Pizza.fromJson(Map<String, dynamic> json) {
    id = json['id'];
    pizzaName = json['pizzaName'].toString();
    description = json['description'].toString();
    price = json['price'];
    imageUrl = json['imageUrl'] ?? '';
}
```

7. Run the app again. The error you can see should have changed, as shown here:

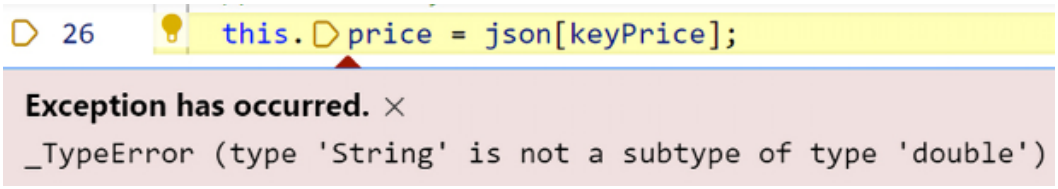


Figure 9.9: Conversion error

8. Edit the `Pizza.fromJson` constructor in the `Pizza.dart` file—this time, adding the `tryParse` method to the numeric values:

```
Pizza.fromJson(Map<String, dynamic> json) {
    id = int.tryParse(json['id'].toString());
    pizzaName = json['pizzaName'].toString();
    description = json['description'].toString();
    price = double.tryParse(json['price'].toString())
?? 0;
```

```
    imageUrl = json['imageUrl'].toString();
}
```

9. Run the app again. This time, the app runs, but you have several `null` values that you do not want to show to your users:

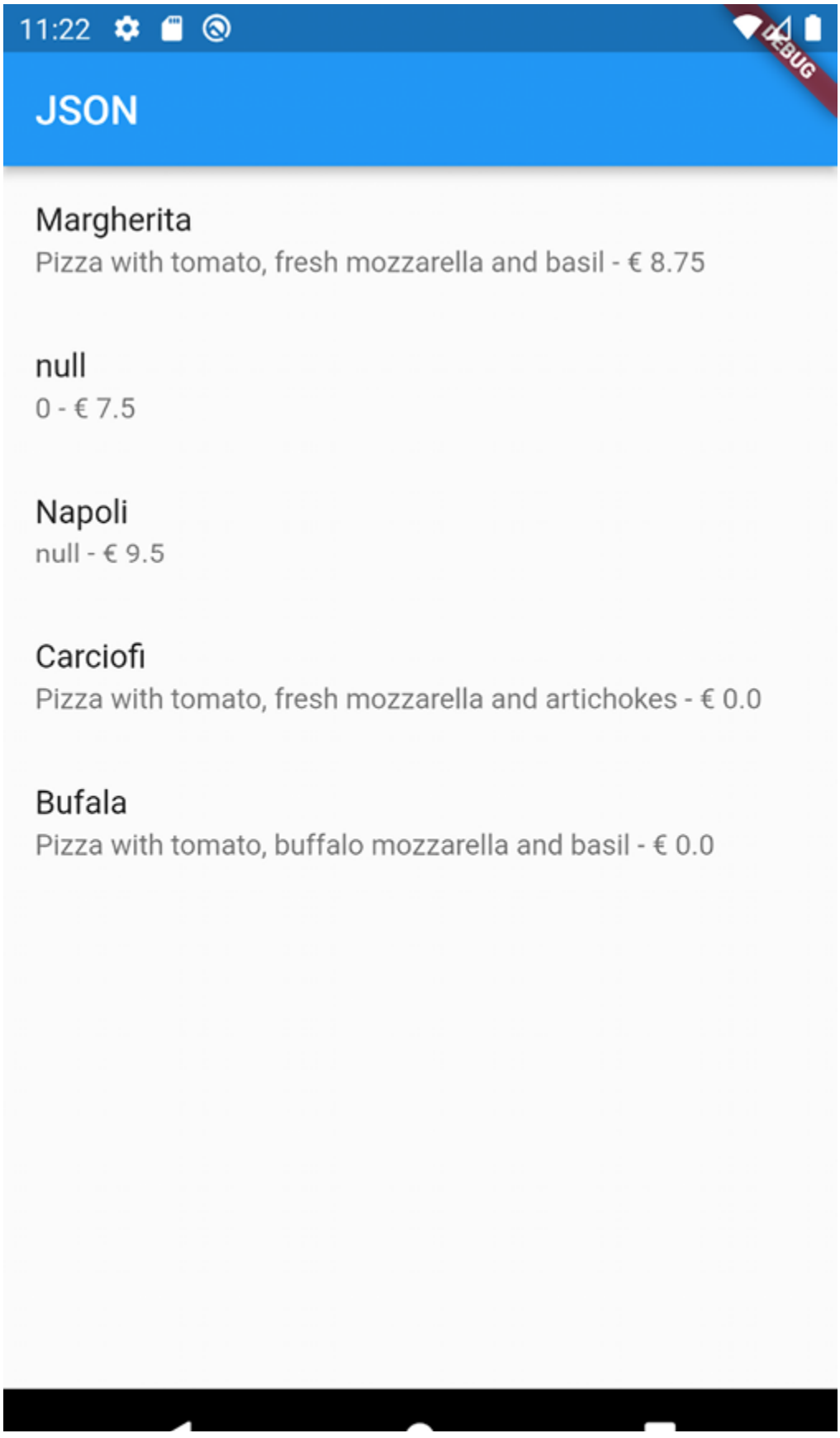

11:22
DEBUG
JSON
Margherita
Pizza with tomato, fresh mozzarella and basil - € 8.75
null
0 - € 7.5
Napoli
null - € 9.5
Carciofi
Pizza with tomato, fresh mozzarella and artichokes - € 0.0
Bufala
Pizza with tomato, buffalo mozzarella and basil - € 0.0



Figure 9.10: Null appearing on the screen

10. Edit the `Pizza.fromJson` constructor one last time by adding ternary operators, as shown here:

```
Pizza.fromJson(Map<String, dynamic> json) {
    id = int.tryParse(json['id'].toString()) ?? 0;
    pizzaName = json['pizzaName'] != null ?
json['pizzaName'].toString() : 'No name';
    description =
        (json['description'] != null) ?
json['description'].toString() : '';
    price = double.tryParse(json['price'].toString())
?? 0;
    imageUrl = json['imageUrl'] ?? '';}
```

11. Finally, run the app once again. This time, everything works as expected, and null data is not shown to the user.

How it works...

The new `pizzalist.json` file contains several inconsistencies for our class: some fields are missing, while others have the wrong data type. These are common occurrences when dealing with data. It's a good idea to always "massage" your data so that it becomes compatible with your classes.

The first error we received after adding the new JSON file was a data type error: the ID of the first `pizza` object in the JSON file has a String instead of a number:

```
{
      "id": "1",
      …
    },
```

That's why we received `_TypeError: type 'String' is not a subtype of type 'int'`, which basically means, "I was expecting an integer, but I

found a String. Not good."

It's possible to easily transform (cast) types on the run; that's why modifying the instruction solves the issue:

```
id = int.tryParse(json['id'].toString()) ?? 0;
```

Here, several things happen at the same time: the `toString` method, applied over any other type, transforms it into a string, which is the expected data of the `tryParse` method. `Int.tryparse` converts a string into an integer, whenever possible. When the conversion fails, it returns null. That's why we're also adding the null coalescing operator here: if the conversion fails, we just set the ID to take 0 (`?? 0`).

The second `pizza` object in the file contains a description key, with a number instead of a String:

```
"description": 0,
```

That's why we received another `TypeError: type 'int' is not a subtype of type 'String'`—exactly the opposite of the previous error.

Again, it's possible to easily transform almost any type into a String by adding the `toString()` method to the data you want to transform. That's why modifying the instruction like so solves the issue:

```
this.description = json['description'].toString();
```

To avoid other similar issues, we also added the `toString()` method to the other strings in our class.

Another type error we received was for the price. In the pizza with an ID of 4, there was the following key-value pair:

```
"price": "N/A",
```

Here, the price is a `String`, but in our class, the price is obviously a `double`. The solution here was adding a `double.tryParse` to the value, which behaves like the `int.tryParse` method, except it returns a double instead of an integer:

```
double.tryParse(json['price'].toString()) ?? 0;
```

When you want to transform some kind of data into another, you can use the `parse` method (without `try`). But in this case, when the conversion cannot happen ("N/A" **cannot** be transformed into a number), the conversion triggers an error. The `tryParse` method behaves differently: when the conversion cannot happen, instead of causing an error, it just returns null.

Because `double.tryParse` takes a `String` as a parameter, we made sure we always passed a String by adding `toString()` to the price as well.

In this way, we could solve the errors on the app, but the `null` value was showing in the user interface, and this is something you generally want to avoid. That's why the last refactoring we applied to our code was using a ternary operator for our fields:

```
pizzaName = (json['pizzaName'] != null) ?
json['pizzaName'].toString() : '';
```

For the strings, if the value is NOT null (`json['pizzaName'] != null`), we return the value transformed into a String:

```
json['pizzaName'].toString() otherwise 'No Name'
```

After applying those checks, we can use problematic JSON code and apply it successfully to our `Pizza` class.

See also

In this recipe, we have dealt with some of the most common errors you may encounter when dealing with JSON data. Most of them depend on data types. For a more general discussion on type safety and Dart, have a look at the following article: https://dart.dev/guides/language/sound-problems.

Catching common JSON errors

As you saw in the previous recipe, you need to deal with JSON data that may be incompatible with your own data. But there is another source of errors that comes from within your code.

When dealing with JSON, it often happens that the key names are repeated several times in your code. Since we are not perfect, we might create an error by typing in something that's incorrect. It's rather difficult to handle these kinds of errors as they only occur at runtime.

A common way to prevent these kinds of typos is to use **constants** instead of typing the key names each time you need to reference them.

Getting ready

To follow along with this recipe, you will need to have completed the previous recipe.

How to do it...

In this recipe, we will use **constants** instead of typing in the keys of each field of the JSON data. The starting code is at the end of the previous recipe:

1. At the top of the `pizza.dart` file (before the `Pizza` class), add the constants that we will need later within the `Pizza` class:

```
const keyId = 'id';
const keyName = 'pizzaName';
const keyDescription = 'description';
```

```
const keyPrice = 'price';
const keyImage = 'imageUrl';
```

2. In the `Pizza.fromJson` constructor, remove the strings for the JSON object and add the constants instead:

```
Pizza.fromJson(Map<String, dynamic> json) {
    id = int.tryParse(json[keyId].toString()) ?? 0;
    pizzaName =
        json[keyName] != null ?
json[keyName].toString() : 'No name';
    description =
        (json[keyDescription] != null) ?
json[keyDescription].toString() : '';
    price = double.tryParse(json[keyPrice].toString())
?? 0;
    imageUrl = json[keyImage] ?? '';}
```

3. Finally, in the `toJson` method, repeat the same process:

```
Map<String, dynamic> toJson() {
    return {
      keyId: id,
      keyName: pizzaName,
      keyDescription: description,
      keyPrice: price,
      keyImage: imageUrl,    };
}
```

How it works...

There's not much to explain here: we simply used constants instead of repeating the name of the keys for our JSON data.

If you come from Android development, you'll find this pattern rather familiar. Using constants instead of hard typing in the field names is usually a great idea and helps prevent errors that are difficult to debug.

One note about the style: in other languages, such as Java, you usually name your constants in `SCREAMING_CAPS`. In our code, for example, the first

constant would be called `KEY_ID`. In Dart and Flutter, the recommendation is to use `lowerCamelCase` for constants.

See also

In the first three recipes of this chapter, we manually created the methods that encode and decode JSON data. As data structures become more complex, you can use libraries that create those methods for you. See, for example, `json_serializable` at https://pub.dev/packages/json_serializableand `built_value` at https://pub.dev/packages/built_value for two of the most commonly used libraries.

Saving data simply with SharedPreferences

Among the several ways we can save data with Flutter, arguably one of the simplest is using `SharedPreferences`: it works for Android, iOS, the web, and desktop, and it's great when you need to store simple data within your device.

You shouldn't use `SharedPreferences` for critical data as data stored there is *not* encrypted, and writes are not always guaranteed. On the other hand, it's also rather reliable and suitable for most use cases.

At its core, `SharedPreferences` stores key-value pairs on disk. More specifically, only primitive data can be saved: numbers, Booleans, Strings, and `stringLists`. All data is saved within the app.

In this recipe, you will create a very simple app that keeps track of the number of times the user opened the app and allows the user to delete the record.

Getting ready

To follow along with this recipe, you will need the starting code for this recipe at https://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_08.

How to do it...

In this recipe, we will create an app that stores the number of times the app itself has been opened. While the logic of the app is extremely simple, this is useful when you need to keep track of the usage of your app or need to send targeted messages to specific users. In doing so, you will learn how to use the `shared_preferences` library. Follow these steps:

1. First, add a dependency to `shared_preferences`. From your Terminal, type the following command:

```
flutter pub add shared_preferences
```

2. To update the dependencies in your project, run the `flutter pub get` command from your Terminal window.
3. At the top of the `main.dart` file, import `shared_preferences`:

```
import
'package:shared_preferences/shared_preferences.dart';
```

4. At the top of the `_MyHomePageState` class, create a new integer state variable called `appCounter`:

```
int appCounter = 0;
```

5. In the `_MyHomePageState` class, create a new asynchronous method called `readAndWritePreferences()`:

```
Future readAndWritePreference() async {}
```

6. Inside the `readAndWritePreference` method, create an instance of `SharedPreferences`:

```
SharedPreferences prefs = await
SharedPreferences.getInstance();
```

7. After creating the `prefs` instance, try reading the value of the `appCounter` key. If its value is null, set it to `0`; then increment it:

```
appCounter = prefs.getInt('appCounter') ?? 0;
appCounter++;
```

8. After updating `appCounter`, set the value of the `appCounter` key in `prefs` to the new value:

```
await prefs.setInt('appCounter', appCounter);
```

9. Update the `appCounter` state value:

```
setState(() {
    appCounter = appCounter;
});
```

10. In the `initState` method in the `_MyHomePageState` class, call the `readAndWritePreference()` method:

```
@override
void initState() {
super.initState();
   readAndWritePreference();
}
```

11. In the `build` method, add the following code inside the `Container` widget:

```
child: Center(
    child: Column(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    children: [
    Text(
    'You have opened the app $appCounter times.'),
    ElevatedButton(
    onPressed: () {},
```

```
    child: Text('Reset counter'),
)],)),
```

12. Run the app. The first time you open it, you should see a screen similar to the following:

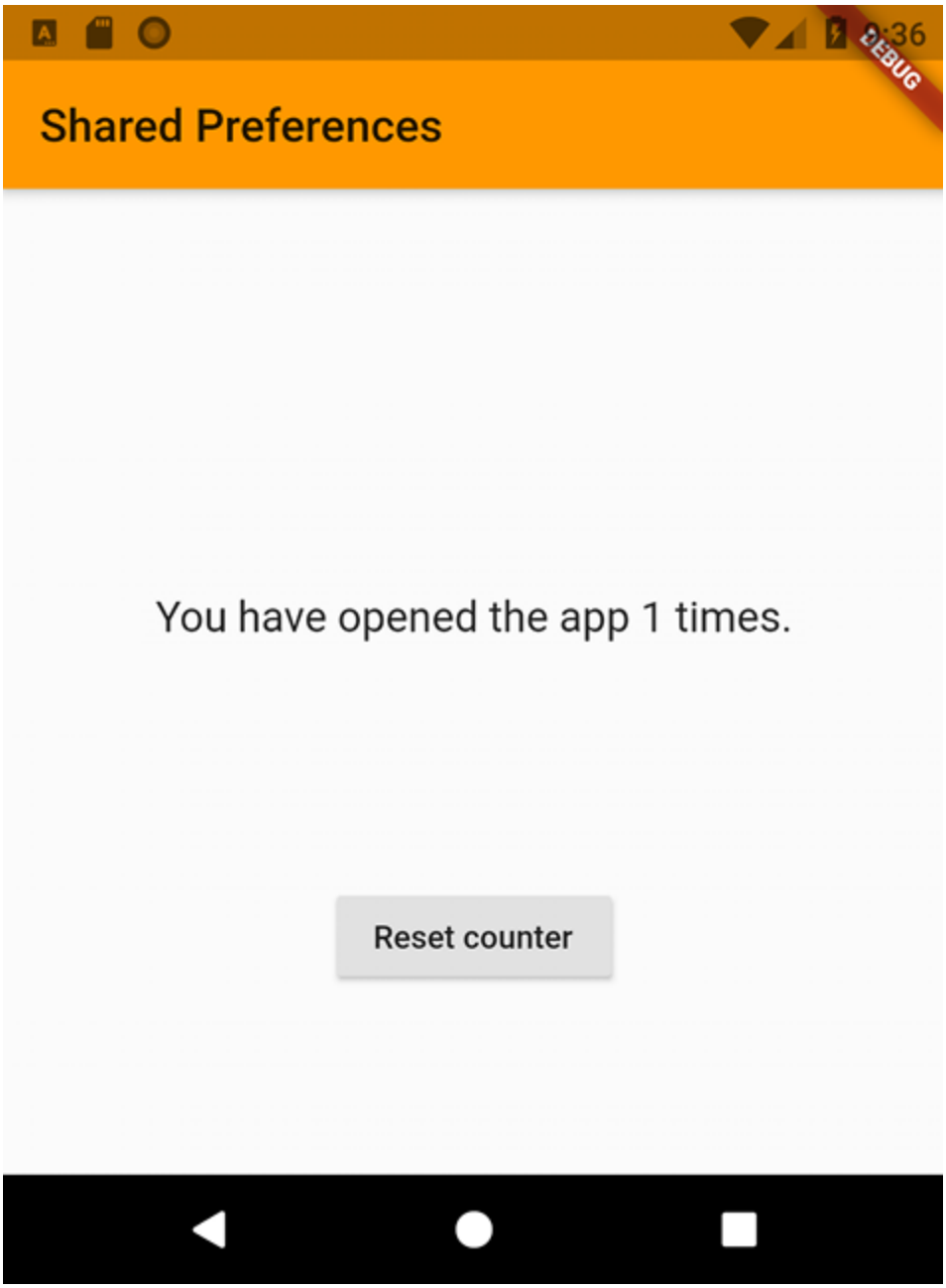


Figure 9.11: Using SharedPreferences data

13. Add a new method to the `_MyHomePageState` class called `deletePreference()`, which will delete the saved value:

```
Future deletePreference() async {
    SharedPreferences prefs = await
SharedPreferences.getInstance();
    await prefs.clear();
    setState(() {
    appCounter = 0;
});
```

14. From the `onPressed` property of the `ElevatedButton` widget in the `build()` method, call `deletePreference()`:

```
ElevatedButton(
    onPressed: () {
      deletePreference();
    },
    child: Text('Reset counter'),
)
```

15. Run the app again. Now, when you press the **Reset counter** button, the `appCounter` value will be deleted.

How it works...

In the code shown in the previous section, we used an instance of `SharedPreferences` to read, write, and delete values from our device.

When using `SharedPreferences`, the first step is adding the reference to the `pubspec.yaml` file, since this is an external library. Then, you need to import the library at the top of the file where you need to use it.

The next step is getting an instance of `SharedPreferences`. You can do that with the following instruction:

```
SharedPreferences prefs = await
SharedPreferences.getInstance();
```

The `getInstance()` method returns a `Future<SharedPreferences>`, so you should always treat this **asynchronously**; for example, using the `await` keyword in an `async` method.

Once you have the instance of `SharedPreferences`, you can read and write values to your device. To read and write values, you use a **key**.

For example, let's take a look at the following instruction:

```
await prefs.setInt('appCounter', 42);
```

This instruction writes in a key named `appCounter` with an integer value of `42`. If the key does not exist, it gets created; otherwise, you just overwrite its value.

`setInt` writes an `int` value. Quite predictably, when you need to write a String, you use the `setString` method.

When you need to **read** a value, you still use the key, but you use the `getInt` method for integers and the `getString` method for Strings.

Here is a table you may find useful for using the read/write actions over `SharedPreferences`:

Type	Read (get)	Write (set)
`int`	`getInt(key)`	`setInt(key, value)`
`double`	`getDouble(key)`	`setDouble(key, value)`
`bool`	`getBool(key)`	`setBool(key, value)`
`String`	`getString(key)`	`setString(key, value)`
`stringList`	`getStringList(key)`	`setStringList(key, listOfvalues)`

Table 8.1: SharedPreferences methods

Note

All **writes** to `SharedPreferences` and the `getInstance` method are asynchronous.

The last feature of this recipe was deleting the value from the device. We achieved this by using the following instruction:

```
await prefs.clear();
```

This deletes all the keys and values for the app.

See also

The `shared_preferences` plugin is one of the most mature and well-documented Flutter libraries. For another example of using it, have a look at https://flutter.dev/docs/cookbook/persistence/key-value.

Accessing the filesystem, part 1: path_provider

The first step whenever you need to write files to your device is knowing where to save those files. In this recipe, we will create an app that shows the current system's temporary and document directories.

`path_provider` is a library that allows you to find common paths in your device, regardless of the operating system your app is running on. For example, on iOS and Android, the path of the document folder is different. By leveraging `path_provider`, you don't need to write different methods based on the OS you are using; you just get the path by using the library's methods.

The `path_provider` library currently supports Android, iOS, Windows, macOS, and Linux. Check out https://pub.dev/packages/path_provider for the updated OS support list.

Getting ready

To follow along with this recipe, you must create a new app or update the code from the previous recipe.

How to do it...

To use `path_provider` in your app, follow these steps:

1. As usual, the first step of using a library is adding the relevant dependency to the `pubspec.yaml` file. Add `path_provider` by typing this command from your Terminal:

```
flutter pub add path_provider
```

2. At the top of the `main.dart` file, import `path_provider`:

```
import 'package:path_provider/path_provider.dart';
```

3. At the top of the `_MyHomePageState` class, add the State variables that we will use to update the user interface:

```
String documentsPath='';
String tempPath='';
```

4. Still in the `_MyHomePageState` class, add the method for retrieving the temporary and documents directories:

```
Future getPaths() async {
    final docDir = await
getApplicationDocumentsDirectory();
    final tempDir = await getTemporaryDirectory();
        setState(() {
            documentsPath = docDir.path;
            tempPath = tempDir.path;
        });
}
```

5. In the `initState` method of the `_MyHomePageState` class, call the `getPaths` method:

```
@override
void initState() {
    super.initState();
    getPaths();
}
```

6. In the `build` method of `_MyHomePageState`, create the UI with two `Text` widgets that show the retrieved paths:

```
@override
Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Path
Provider')),
      body: Column(
        mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
        children: [
```

```
            Text('Doc path: $documentsPath'),
            Text('Temp path $tempPath'),
          ],
        ),
      );
}
```

7. Run the app. You should see a screen that looks similar to the following:

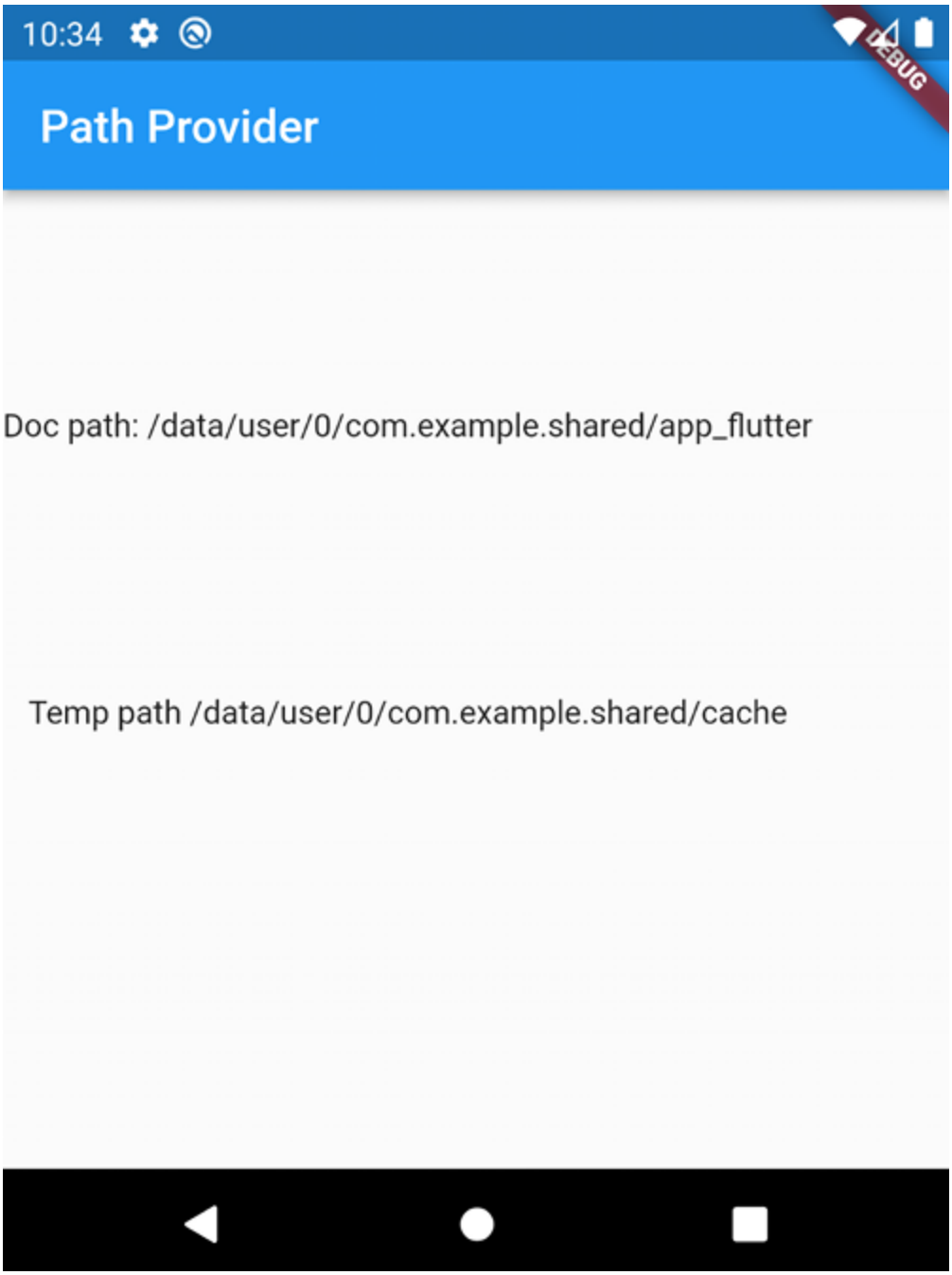


Figure 9.12: Documents and Temp paths in an Android device

This completes this recipe.

How it works...

There are two directories you can get with `path_provider`: the **documents directory** and the **temporary directory**.

Since the temporary directory can be cleared by the system at any time, you should use the documents directory whenever you need to store data that you need to save, and use the temporary directory as a sort of cache or session storage for your app.

The two methods that retrieve the directories when you use the `path_provider` library are `GetApplicationDocumentsDirectory` and `getTemporaryDirectory`.

Both are asynchronous and return a `Directory` object. The following is an example of a directory, with all its properties, as it looks on a Windows PC with an Android simulator attached to it:

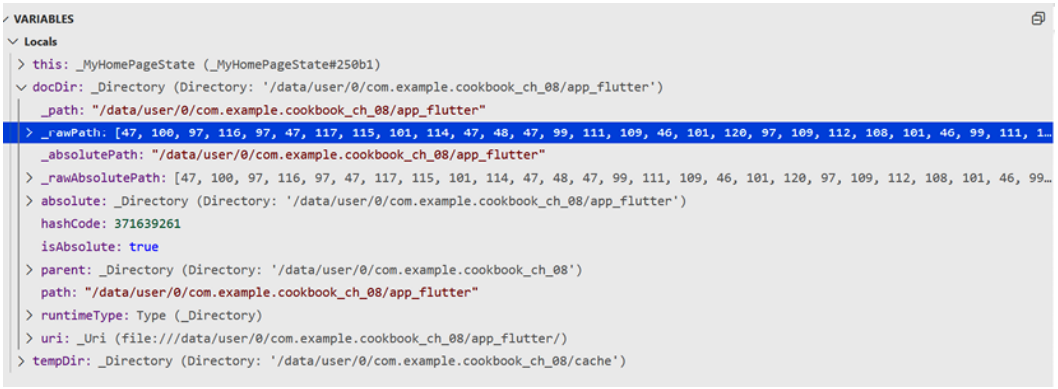


Figure 9.13: A Directory object

As you can see, there is a `path` string property that contains the absolute path of the directory that was retrieved. That's why we used the following instructions:

```
setState(() {
    documentsPath = docDir.path;
    tempPath = tempDir.path;
});
```

That's how we updated the state variables in our app to show the absolute path of the temporary and document directories in our user interface.

See also

To learn about reading and writing files in iOS and Android with Flutter, have a look at https://flutter.dev/docs/cookbook/persistence/reading-writing-files.

Accessing the filesystem, part 2: Working with directories

In this section, we'll build upon the previous recipe so that you can read and write to files using Dart's `Directory` class and the `dart:io` library.

Getting ready

To follow along with this recipe, you need to have completed the previous recipe.

How to do it...

In this recipe, we will:

- Create a new file inside our device or simulator/emulator.
- Write some text.
- Read the content in the file and show it on the screen.

The methods for reading and writing to files are included in the `dart.io` library. Follow these steps:

1. At the top of the `main.dart` file, import the `dart:io` library:

```
import 'dart:io';
```

2. At the top of the `_MyHomePageState` class, in the `main.dart` file, create two new State variables for the file and its content:

```
late File myFile;
String fileText='';
```

3. Still in the `MyHomePageState` class, create a new method called `writeFile` and use the `File` class of the `dart:io` library to create a new file:

```
Future<bool> writeFile() async {
    try {
        await myFile.writeAsString('Margherita,
Capricciosa, Napoli');
        return true;
    } catch (e) {
        return false;
    }
}
```

4. In the `initState` method, after calling the `getPaths` method, in the `then` method, create a file and call the `writeFile` method:

```
@override
void initState() {
    getPaths().then((_) {
        myFile = File('$documentsPath/pizzas.txt');
        writeFile();
    });
    super.initState();
}
```

5. Create a method to read the file:

```
Future<bool> readFile() async {
    try {
        // Read the file.
        String fileContent = await
myFile.readAsString();
        setState(() {
            fileText = fileContent;
        });
        return true;
    } catch (e) {
        // On error, return false.
        return false;
    }
}
```

6. In the `build` method, in the `Column` widget, update the user interface with an `ElevatedButton`. When the user presses the button, it will attempt to read the file's content and show it on the screen:

```
children: [
    Text('Doc path: ' + documentsPath),
    Text('Temp path' + tempPath),
    ElevatedButton(
        child: const Text('Read File'),
        onPressed: () => readFile(),
    ),
    Text(fileText),
],
```

7. Run the app and press the **Read File** button. Under the button, you should see the text **Margherita, Capricciosa, Napoli**, as shown in the following screenshot:

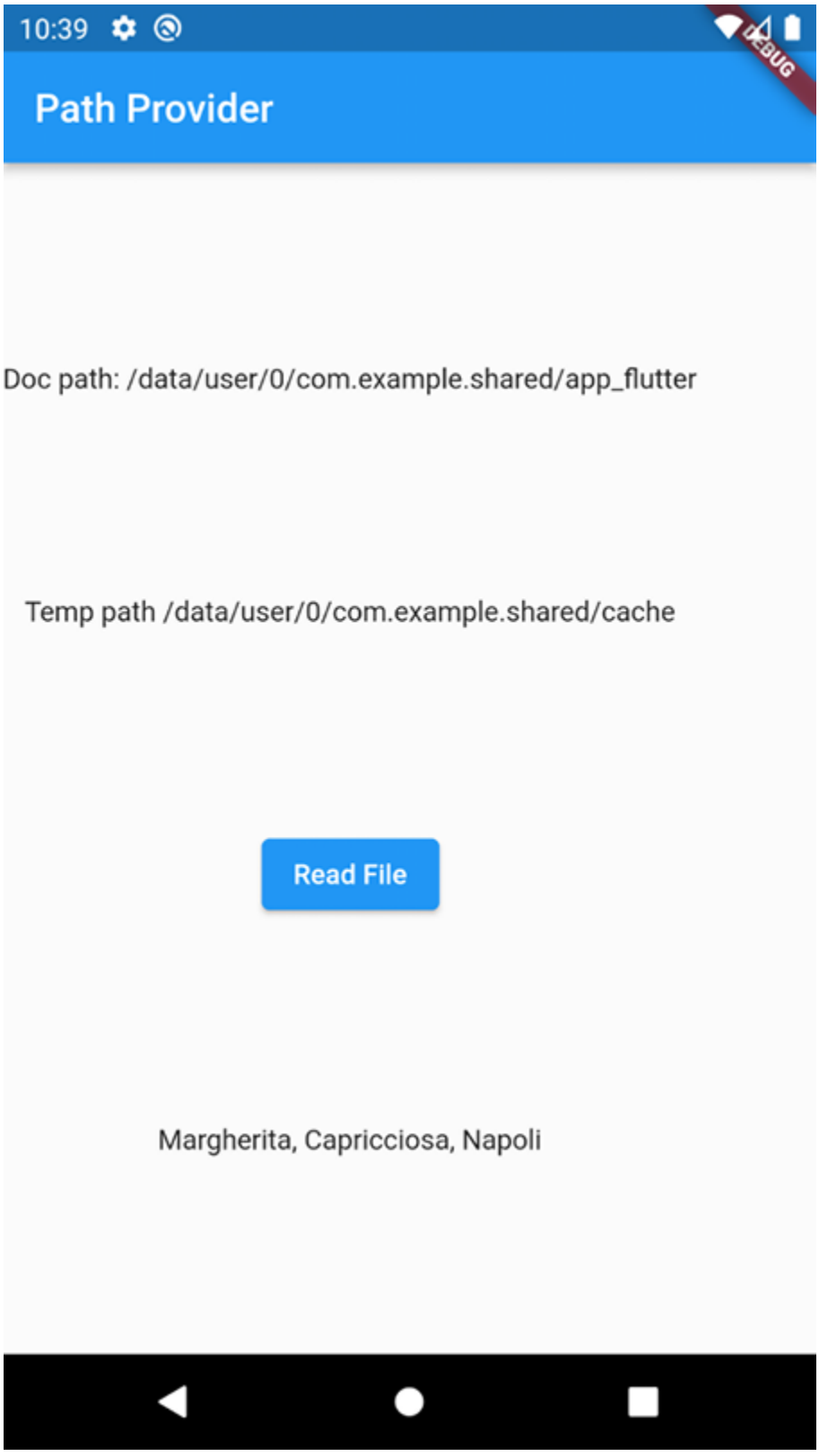


Figure 9.14: On-screen result after reading a file

How it works...

In the preceding code, we are combining the methods contained in two libraries: we're using `path_provider` to retrieve the documents folder in the device and the `dart:io` library to create a new file, write content, and read its content.

Now, you might be wondering, why do you need to use `path_provider` to get the documents folder, and not just write anywhere?

Local drives in iOS and Android are mostly inaccessible to apps as a security measure. Apps can only write to select folders, and those include the `temp` and `documents` folders.

When dealing with files, you need to do the following:

1. Get a reference to the file.
2. Write some content.
3. Read the file's content.

We performed those steps in this recipe. Note the following instruction:

```
myFile = File('$documentsPath/pizzas.txt');
```

This creates a `File` object, whose path is specified as a parameter.

Then, we have the following instruction:

```
await myFile.writeAsString('Margherita, Capricciosa,
Napoli');
```

This writes the `String` contained as a parameter into the file.

The `writeAsString` method is asynchronous, but there is also a synchronous version of it called `writeAsStringSync()`. Unless you have a

very good reason to do otherwise, always prefer the *asynchronous* version of the method.

In general, when dealing with tasks that might take some time to complete, you should always use asynchronous methods: use cases include network requests, file input/output, database operations, and heavy computations.

To read the file, we used the following instruction:

```
String fileContent = await myFile.readAsString();
```

After executing the preceding instruction, the `fileContent` String contained the content of the file.

See also

You are not limited to reading and writing strings, of course; there are several other methods you can leverage to access files in Flutter. For a comprehensive guide to accessing files, have a look at https://api.flutter.dev/flutter/dart-io/File-class.html and https://flutter.dev/docs/cookbook/persistence/reading-writing-files.

Using secure storage to store data

A very common task for apps is storing user credentials, or other sensitive data, within the app itself. `SharedPreferences` is not an ideal tool to perform this task as data cannot be encrypted. In this recipe, you will learn how to store encrypted data using `flutter_secure_storage`, which provides an easy and secure way to store data.

Getting ready

To follow along with this recipe, you will need to do the following:

- If you are using Android, you need to set `compileSdkVersion` in your app's `build.gradle` file to `33`. The file is located in your project folder, in `/android/app/build.gradle`. The `compileSdkVersion` key must be set in the `defaultConfig` node, like this:

```
compileSdkVersion 33
```

- Also, make sure the `minSdkVersion` is at least `18`:

```
minSdkVersion 18
```

- Download the starting code for this app, which is available at https://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_08.

How to do it...

For this recipe, we will create an app that takes a `String` from a text field and stores it securely using the `flutter_secure_storage` library. Follow these steps:

1. Add `flutter_secure_storage` to your project, by typing:

```
flutter pub add flutter_secure_storage
```

2. In the `main.dart` file, copy the code available at https://bit.ly/flutter_secure_storage.
3. At the top of the `main.dart` file, add the required `import`:

```
import
'package:flutter_secure_storage/flutter_secure_storage.d
art';
```

4. At the top of the `_myHomePageState` class, create secure storage:

```
final storage = const FlutterSecureStorage();
final myKey = 'myPass';
```

5. In the `_myHomePageState` class, add the method for writing data to the secure storage:

```
Future writeToSecureStorage() async {
    await storage.write(key: myKey, value:
pwdController.text);
}
```

6. In the `build()` method of the `_myHomePageState` class, add the code that will write to the store when the user presses the **Save Value** button:

```
ElevatedButton(
    child: const Text('Save Value'),
    onPressed: () {
        writeToSecureStorage();
    }
),
```

7. In the `_myHomePageState` class, add the method for **reading** data from the secure storage:

```
Future<String> readFromSecureStorage() async {
    String secret = await storage.read(key: myKey) ??
'';          return secret;
}
```

8. In the `build()` method of the `_myHomePageState` class, add the code for reading from the store when the user presses the **Read Value** button and updates the `myPass` State variable:

```
ElevatedButton(
    child: Text('Read Value'),
    onPressed: () {
      readFromSecureStorage().then((value) {
      setState(() {
        myPass = value;
      });
    });
}),
```

9. Run the app and write some text of your choice in the text field. Then, press the **Save Value** button. After that, press the **Read Value** button. You should see the text that you typed into the text field, as shown in the following screenshot:

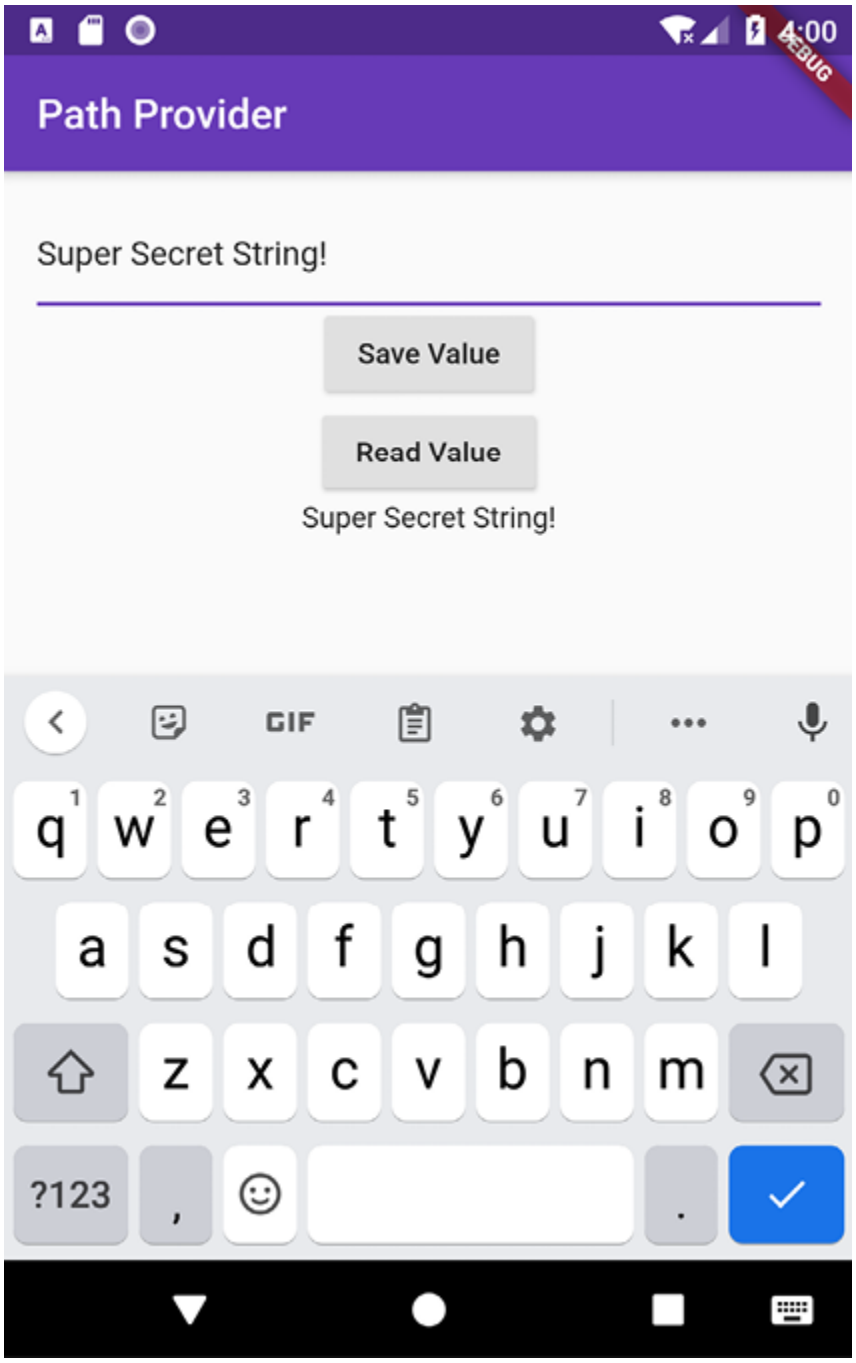


Figure 9.15: Using secure storage

How it works...

The beauty of `flutter_secure_storage` is that, while it's extremely easy to use, secure encryption happens under the hood. Specifically, if you are using iOS, data gets encrypted using `Keychain`. On Android, `flutter_secure_storage` uses `AES` encryption, which, in turn, is encrypted with `RSA`, whose key is stored in `KeyStore`.

To use it, we must get an instance of the `FlutterSecureStorage` class, which we can obtain with the following instruction:

```
final storage = const FlutterSecureStorage();
```

Like `shared_preferences`, it uses key-value pairs to store data. You can write a value into a key with the following instruction:

```
await storage.write(key: myKey, value: pwdController.text);
```

You can read the value from a key with the following instruction:

```
String secret = await storage.read(key: myKey) ?? '';
```

As you may have noticed, the `read()` and `write()` methods are both **asynchronous**.

See also

There are several libraries in Flutter that make encrypting data an easy process. Just to name two of the most popular ones, have a look at `encrypt` at https://pub.dev/packages/encrypt and `crypto` at https://pub.dev/packages/crypto.

Designing an HTTP client and getting data

Most mobile apps rely on data that comes from an external source. Think of apps for reading books, watching movies, sharing pictures with your friends, reading the news, or writing emails: all these apps use data taken from an external source. When an app consumes external data, usually, there is a **backend service** that provides that data for the app: a **web service** or **web API**.

What happens is that your app (**frontend** or **client**) connects to a web service over HTTP and requests some data. The backend service then responds by sending the data to the app, usually in JSON or XML format.

For this recipe, we will create an app that reads and writes data from a web service. As creating a web API is beyond the scope of this book, we will use a mock service, called **Wire Mock Cloud**, which will simulate the behavior of a real web service, but will be extremely easy to set up and use. In a later chapter, you will also see another way of creating a real-world backend with `Firebase`.

Getting ready

To follow along with this recipe, your device will need an internet connection to retrieve data from the web service. The starting code for this recipe is available at https://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_08.

How to do it...

In this recipe, we will simulate a web service using Wire Mock Cloud and create an app that reads data from the mock service. We'll begin by setting up a new service:

1. Sign up for the Mock Lab service at https://app.wiremock.cloud/ and register to the site, choosing a username and password.
2. Log in to the service, go to the example mock API, and click on the `Stubs` section of the example API. Then, click on the first entry—that is, **Get a JSON resource**. You should see a screen similar to the following:

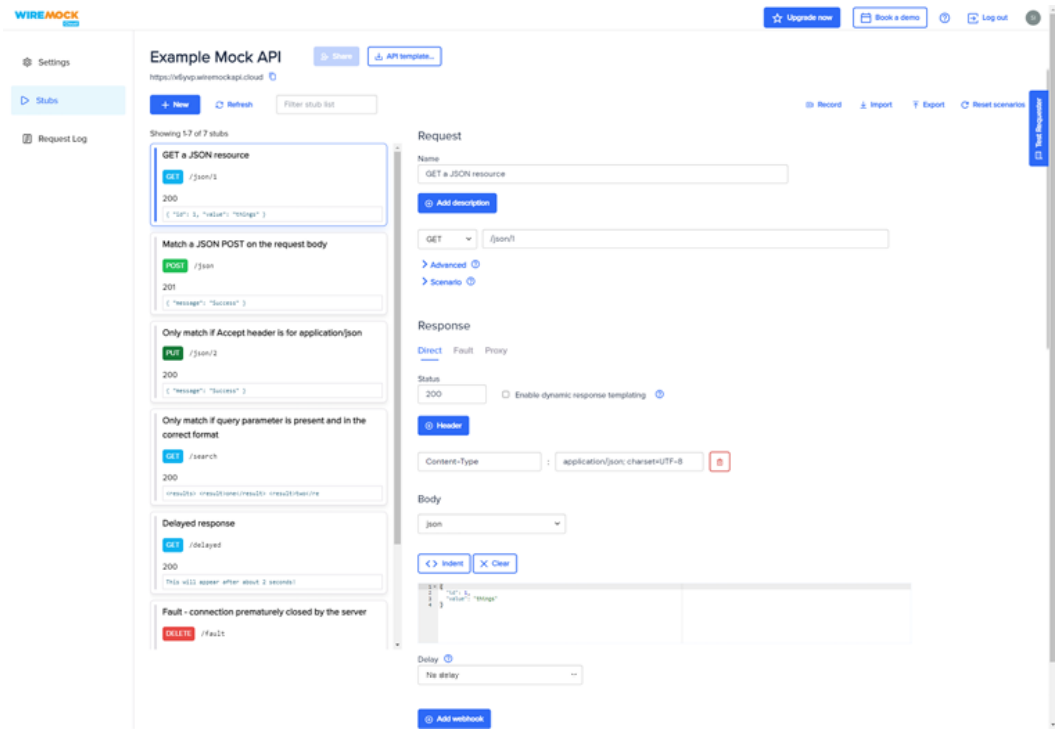


Figure 9.16: WireMock Stubs

3. Click on the **New** button. For its **Name**, type `Pizza List`, leave **GET** as a verb, and in the text box near the GET verb, type `/pizzalist`. Then, in the **Response** section, for the 200 status, select **JSON** as the format and paste the JSON content available at https://bit.ly/pizzalist. The final result is shown here:

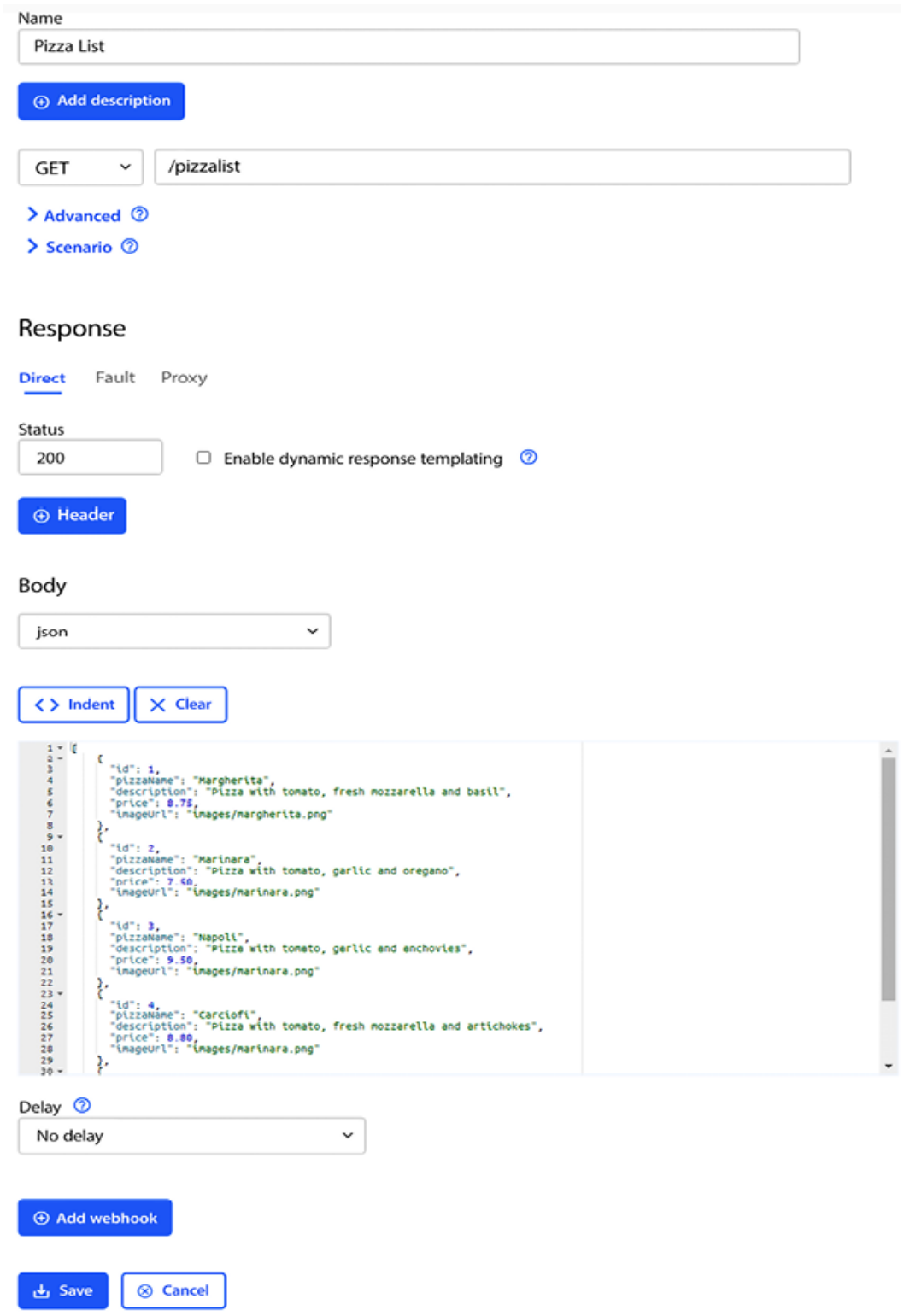


Figure 9.17: WireMock pizzalist get stub

4. Press the **Save** button at the bottom of the page to save the stub. This completes the setup for the backend mock service.
5. Back to the Flutter project, add the `http` dependency by typing in your Terminal:

```
flutter pub add http
```

6. In the `lib` folder within your project, add a new file called `httphelper.dart`.
7. In the `httphelper.dart` file, add the following code:

```
import 'dart:io';
import 'package:http/http.dart' as http;
import 'dart:convert';
import 'pizza.dart';
class HttpHelper {
 final String authority = '02z2g.mocklab.io';
 final String path = 'pizzalist';
 Future<List<Pizza>> getPizzaList() async {
   final Uri url = Uri.https(authority, path);
   final http.Response result = await http.get(url);
    if (result.statusCode == HttpStatus.ok) {
      final jsonResponse = json.decode(result.body);
      //provide a type argument to the map method to
avoid type
      error
      List<Pizza> pizzas =
          jsonResponse.map<Pizza>((i) =>
            Pizza.fromJson(i)).toList();
      return pizzas;
    } else {
      return [];
    }
  }
```

8. In the `main.dart` file, in the `_MyHomePageState` class, add a method named `callPizzas`. This returns a `Future` of a `List` of `Pizza` objects by calling the `getPizzaList` method of the `HttpHelper` class, as follows:

```
  Future<List<Pizza>> callPizzas() async {
    HttpHelper helper = HttpHelper();
```

```
    List<Pizza> pizzas = await helper.getPizzaList();
    return pizzas;
  }
```

9. In the `build` method of the `_MyHomePageState` class, in the body of `Scaffold`, add a `FutureBuilder` that builds a `ListView` of `ListTile` widgets containing the `Pizza` objects:

```
Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('JSON')),
      body: FutureBuilder(
          future: callPizzas(),
          builder: (BuildContext context,
AsyncSnapshot<List<Pizza>> snapshot) {
          if (snapshot.hasError) {
            return const Text('Something went wrong');
          }
          if (!snapshot.hasData) {
            return const CircularProgressIndicator();
          }
            return ListView.builder(
                itemCount: (snapshot.data == null) ? 0
: snapshot.data!.length,
                itemBuilder: (BuildContext context, int
position) {
                  return ListTile(
                    title: Text(snapshot.data!
[position].pizzaName),
                    subtitle: Text(snapshot.data!
[position].description +
                        ' - € ' +
                        snapshot.data!
[position].price.toString()),
                  );
                });
          }),
    );
}
```

10. Run the app. You should see a screen similar to the following:

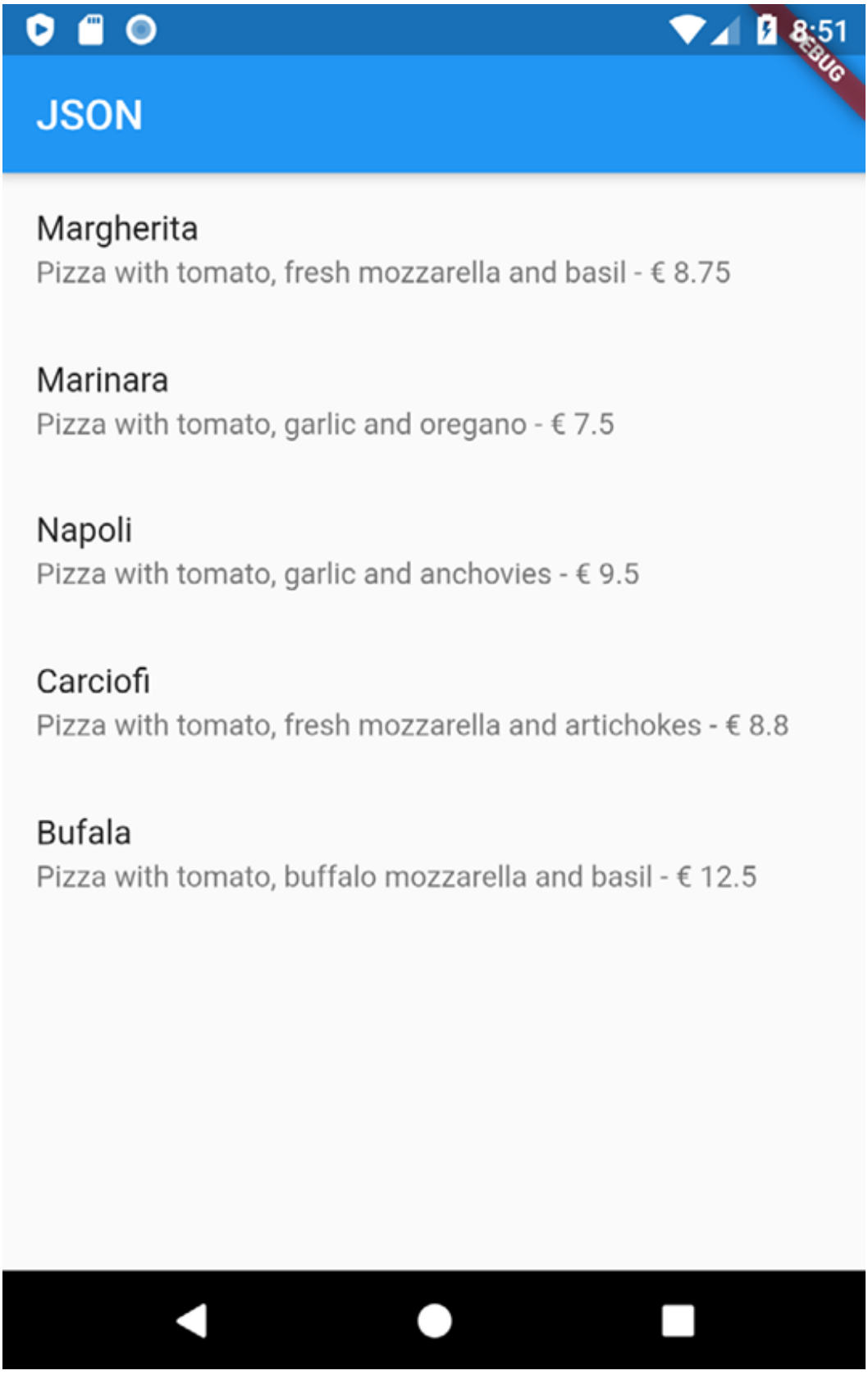

8:51
DEBUG
JSON
Margherita
Pizza with tomato, fresh mozzarella and basil - € 8.75
Marinara
Pizza with tomato, garlic and oregano - € 7.5
Napoli
Pizza with tomato, garlic and anchovies - € 9.5
Carciofi
Pizza with tomato, fresh mozzarella and artichokes - € 8.8
Bufala
Pizza with tomato, buffalo mozzarella and basil - € 12.5

Figure 9.18: List of items retrieved through HTTP

How it works...

In this recipe, we leveraged a service that mocks a web API. It's an easy way to create client code without having to build a real web service. **Wiremock** works with "**stubs**": a stub is a URL that contains a **verb** (such as GET) and a **response**.

Basically, we created an address that, when called, would return a JSON response containing an array of `Pizza` objects.

The `http` library allows the app to make requests to web services.

Note the following instruction:

```
import 'package:http/http.dart' as http;
```

With the `as http` command, you are **naming** the library so that you can use the functions and classes of the `http` library through the HTTP name, such as `http.get()`.

The `get()` method of HTTP returns a `Future` containing a `Response` object. When dealing with web services, you use "verbs." Verbs are actions you perform on the web service. A GET action retrieves data from the web service. You will see other verbs such as POST, PUT, and DELETE in the remaining recipes of this chapter.

When a `get` call is successful, the `http.Response` class contains all the data that has been received from the web service.

To check whether the call has been successful, you can use the `HttpStatus` class (this requires you to import the `dart:io` library), as shown in the following code:

```
if (result.statusCode == HttpStatus.ok) {
      final jsonResponse = json.decode(result.body);
```

A `Response` (`result`, in our example) contains a `statusCode` and a `body`. `statusCode` can be a successful status response, such as `HttpStatus.ok`, or an error, such as `HttpStatus.notFound`.

In the preceding code, when our `Response` contains a valid `statusCode`, you call the `json.decode` method over the body of the `Response`. Then, you can use the following instruction:

```
List<Pizza> pizzas = jsonResponse.map<Pizza>((i) =>
Pizza.fromJson(i)).toList();
```

This transforms the response into a `List` of the `Pizza` type.

When calling the `map` method, you can specify a type argument. This can help you avoid a type error as `map` generally returns a `List` of `dynamic`.

Then, in `main.dart`, in the `callPizzas()` method, you must create an instance of the `HttpHelper` class. From there, you call the `getPizzaList` method to return a `List` of `Pizza` objects:

```
Future<List<Pizza>> callPizzas() async {
    HttpHelper helper = HttpHelper();
    List<Pizza> pizzas = await helper.getPizzaList();
    return pizzas;
  }
```

From there, it's easy: you leverage a `FutureBuilder` to show a `ListView` containing `ListTile` widgets that contain the `Pizzas` information.

There's more...

We now have only one method that uses our `HttpHelper` class. As the application grows, we may need to call `HttpHelper` several times in different parts of the app, and it would be a waste of resources to create many instances of the class each time we need to use a method in the class.

One way to avoid this is by using the `factory` constructor and the **singleton pattern**: this makes sure you only instantiate your class once. It is useful

whenever only one object is needed in your app and when you need to access a resource that you want to share in the entire app.

> **Note**
>
> There are several patterns in Dart and Flutter that allow you to share services and business logic in your app, and the *singleton* pattern is only one of those. Other choices include **Dependency injection**, **Inherited Widgets**, and **Provider** and **Service Locators**. There is an interesting article on the different choices that are available in Flutter at http://bit.ly/flutter_DI.

In the `httphelper.dart` file, add the following code to the `HttpHelper` class, just under the declarations:

```
static final HttpHelper _httpHelper = HttpHelper._internal();
HttpHelper._internal();
factory HttpHelper() {
    return _httpHelper;
}
```

In our example, this means that the first time you call the factory constructor, it will return a new instance of `HttpHelper`. Once `HttpHelper` has been instantiated, the constructor will not build a new instance of `HttpHelper`; it will only return the existing one.

See also

If you want to learn more about the Wiremock service, have a look at https://wiremock.org/docs/getting-started/.

POST-ing data

In this recipe, you will learn how to perform a POST action on a web service. This is useful whenever you are connecting to web services that not only provide data but also allow you to change the information stored on

the server side. Usually, you will have to provide some form of authentication to the service, but for this recipe, as we are using a mock service, this will not be necessary.

Getting ready

To follow along with this recipe, you need to have completed the code in the previous recipe.

How to do it...

To perform a POST action on a web service, follow these steps:

1. Log in to the Mock Lab service at https://app.wiremock.cloud/ and click on the `Stubs` section of the example API. Then, create a new stub.
2. Complete the request, as follows:

    - Name: `Post Pizza`
    - Verb: `POST`
    - Address: `/pizza`
    - Status: `201`
    - Body type: `json`
    - Body: `{"message": "The pizza was posted"}`

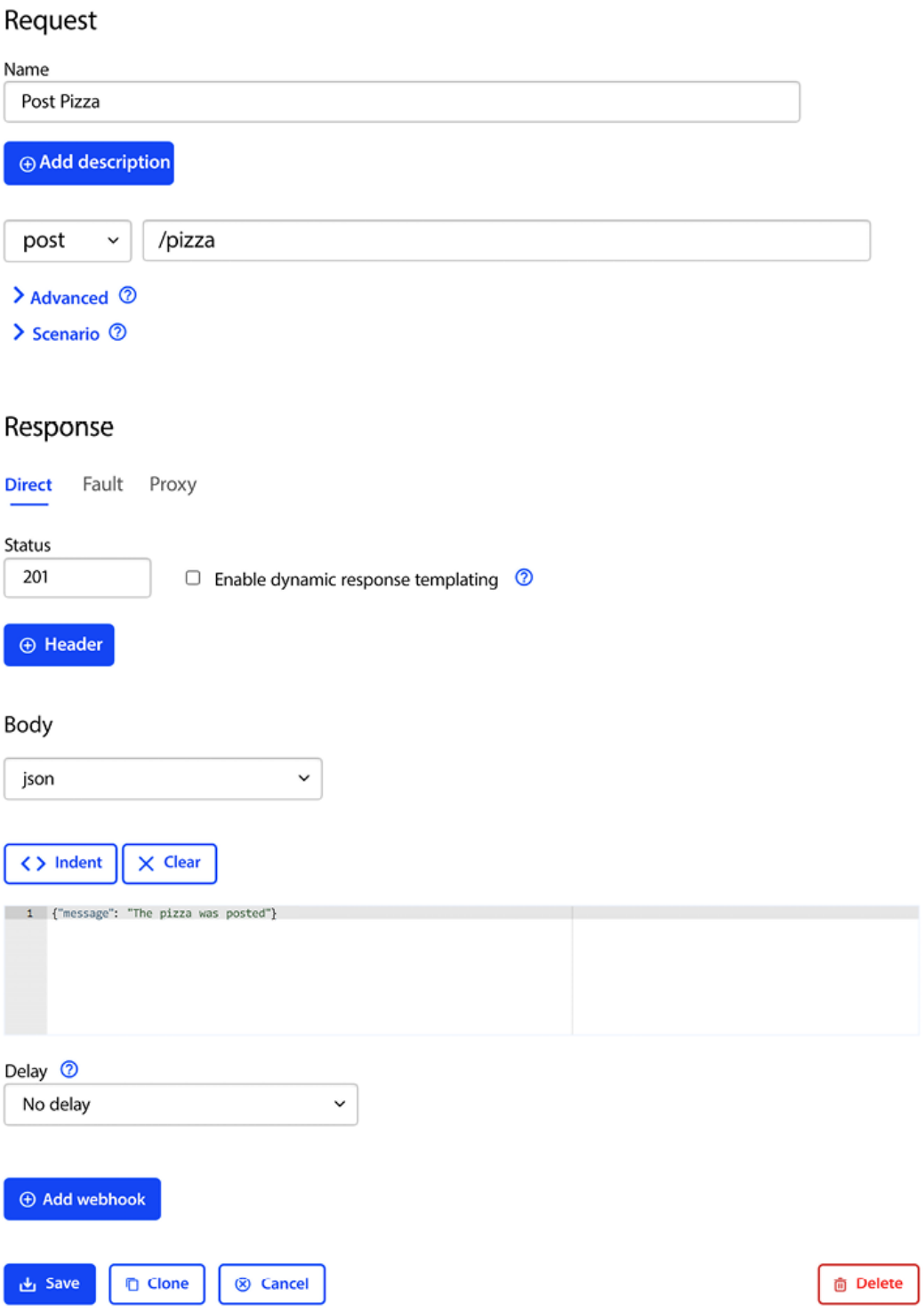


Figure 9.19: WireMock Post Pizza stub

3. Press the **Save** button.

4. In the Flutter project, in the `httpHelper.dart` file, in the `HttpHelper` class, create a new method called `postPizza`, as follows:

```
  Future<String> postPizza(Pizza pizza) async {
    const postPath = '/pizza';
    String post = json.encode(pizza.toJson());
    Uri url = Uri.https(authority, postPath);
    http.Response r = await http.post(
      url,
      body: post,
    );
    return r.body;
  }
```

5. In the project, create a new file called `pizza_detail.dart`.
6. At the top of the new file, add the required imports:

```
import 'package:flutter/material.dart';
import 'pizza.dart';
import 'httphelper.dart';
```

7. Create a `StatefulWidget` called `PizzaDetailScreen`:

```
class PizzaDetailScreen extends StatefulWidget {
  const PizzaDetailScreen({super.key});
  @override
  State<PizzaDetailScreen> createState() =>
_PizzaDetailScreenState();
}
class _PizzaDetailScreenState extends
State<PizzaDetailScreen> {
  @override
  Widget build(BuildContext context) {
    return Placeholder();
  }
}
```

8. At the top of the `_PizzaDetailScreenState` class, add five `TextEditingController` widgets. These will contain the data for the `Pizza` object that will be posted later. Also, add a `String` that will contain the result of the POST request:

```
  final TextEditingController txtId =
TextEditingController();
  final TextEditingController txtName =
TextEditingController();
  final TextEditingController txtDescription =
    TextEditingController();
  final TextEditingController txtPrice =
TextEditingController();
  final TextEditingController txtImageUrl =
    TextEditingController();
  String operationResult = '';
```

9. Override the `dispose()` method to dispose of the controllers:

```
@override
  void dispose() {
    txtId.dispose();
    txtName.dispose();
    txtDescription.dispose();
    txtPrice.dispose();
    txtImageUrl.dispose();
    super.dispose();
  }
```

10. In the `build()` method of the class, return a `Scaffold`, whose `AppBar` contains `Text` stating "`Pizza Detail`" and whose `body` contains a `Padding` and a `SingleChildScrollView` containing a `Column`:

```
      return Scaffold(
        appBar: AppBar(
          title: const Text('Pizza Detail'),
        ),
        body: Padding(
            padding: const EdgeInsets.all(12),
            child: SingleChildScrollView(
              child: Column(children: []),
            )));
```

11. For the `children` property of `Column`, add some `Text` that will contain the result of the post, five `TextFields`, each bound to their own `TextEditingController`, and an `ElevatedButton` to actually complete the POST action (the `postPizza` method will be created next). Also, add a `SizedBox` to distance the widgets on the screen:

```
Text(
  operationResult,
  style: TextStyle(
      backgroundColor: Colors.green[200],
      color: Colors.black),
),
const SizedBox(
  height: 24,
),
TextField(
  controller: txtId,
  decoration: const InputDecoration(hintText: 'Insert
ID'),
),
const SizedBox(
  height: 24,
),
TextField(
  controller: txtName,
  decoration: const InputDecoration(hintText: 'Insert
Pizza Name'),
),
const SizedBox(
  height: 24,
),
TextField(
  controller: txtDescription,
  decoration: const InputDecoration(hintText: 'Insert
Description'),
),
const SizedBox(
  height: 24,
),
TextField(
  controller: txtPrice,
  decoration: const InputDecoration(hintText: 'Insert
Price'),
),
const SizedBox(
  height: 24,
),
TextField(
  controller: txtImageUrl,
  decoration: const InputDecoration(hintText: 'Insert
Image Url'),
),
const SizedBox(
```

```
  height: 48,
),
ElevatedButton(
    child: const Text('Send Post'),
    onPressed: () {
      postPizza();
    })
],
```

12. At the bottom of the `_PizzaDetailState` class, add the `postPizza` method:

```
Future postPizza() async {
    HttpHelper helper = HttpHelper();
    Pizza pizza = Pizza();
    pizza.id = int.tryParse(txtId.text);
    pizza.pizzaName = txtName.text;
    pizza.description = txtDescription.text;
    pizza.price = double.tryParse(txtPrice.text);
    pizza.imageUrl = txtImageUrl.text;
    String result = await helper.postPizza(pizza);
    setState(() {
      operationResult = result;
    });
  }
```

13. In the `main.dart` file, import the `pizza_detail.dart` file:

```
import 'pizza_detail.dart';
```

14. In `Scaffold` of the `build()` method of the `_MyHomePageState` class, add a `FloatingActionButton` that will navigate to the `PizzaDetail` route:

```
floatingActionButton: FloatingActionButton(
          child: const Icon(Icons.add),
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(
                  builder: (context) => const
PizzaDetailScreen()),
            );
          }),
```

15. Run the app. On the main screen, press our `FloatingActionButton` to navigate to the `PizzaDetail` route.
16. Add pizza details in the text fields and press the **Send Post** button. You should see a successful result, as shown in *Figure 9.20*:

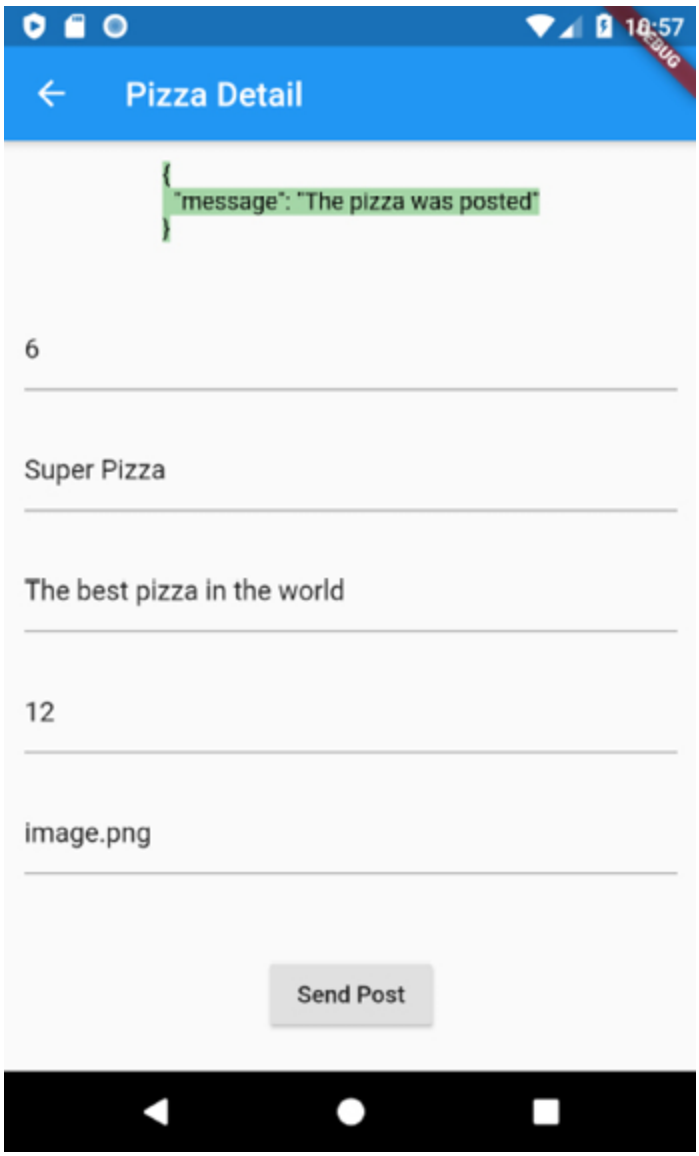


Figure 9.20: Post success message

How it works...

Web services (specifically, **RESTful** web services) work with verbs. There are four main actions (or verbs) you generally use when data is involved: `GET`, `POST`, `PUT`, and `DELETE`.

In this recipe, we used `POST`, which is the verb that's conventionally used when an app asks the web server to insert a new piece of data.

This is why we had to instruct our mock web service to accept a POST at the `/pizza` address first—so that we could try sending some data to it and have it respond with a success message.

> **Note**
>
> When working with web APIs, understanding exactly what data you are sending to the server may be a huge time saver.
>
> One of the most commonly used tools for sending web requests is **Postman**, which you can find at https://www.postman.com/.
>
> Postman can even work with requests from an emulator or simulator. Take a look at https://blog.postman.com/using-postman-proxy-to-capture-and-inspect-api-calls-from-ios-or-android-devices/ for more information.

After creating the POST stub in Wiremock, we created the `postPizza` method, to actually make the call to the server. As this takes a JSON string, we used the `json.encode` method to transform our `Map` into JSON:

```
String post = json.encode(pizza.toJson());
```

Then, we called the `http.post` method to actually send the `post` String to the web server. If this were a real project, the server would probably add a new record to a database: in this case, this won't happen, but we will be able to verify that the call is made and we receive a success message in response.

As usual with HTTP actions, POST is asynchronous, so in the code, we used the `await` keyword to wait for the call to complete. `http.post` takes both the URL (unnamed parameter) and the body, which is the content you send to the server, also called the **payload**:

```
Uri url = Uri.https(authority, postPath);
    http.Response r = await http.post(
      url,
      body: post,
    );
```

As the data to be sent in our example is a new `Pizza`, we need to give the user the ability to specify the details for `Pizza`. This is why we added a new screen to our project: so that our user could specify the `id`, `name`, `description`, `price`, and `imageUrl` details of the new object.

For the user interface of the new screen, we used two widgets that are extremely useful. The first was called `SingleChildScrollView`. This allows its child to scroll when the container is too small. In `Column`, in order to create some space between our `TextFields`, we used a `SizedBox`. In this case, the cleaner `mainAxisAlignment` would not work as it's included in `SingleChildScrollView`, which doesn't have a fixed height. Of course, we could also have simply used a `ListView`.

Finally, we created the method that calls the `HttpHelper.postPizza` method. As this method requires an instance of `Pizza`, we had to read the values in the `TextFields`, using their `TextEditingControllers`, and then create a new `Pizza` object and pass it to the `postPizza` method.

After receiving the response, we wrote it in the `operationResult` String, which updates the screen with a success message.

PUT-ting data

In this recipe, you will learn how to perform a PUT action on a web service. This is useful when your apps need to edit existing data in a web service.

Getting ready

To follow along with this recipe, you need to have completed the code in the previous recipe.

How to do it...

To perform a PUT action on a web service, follow these steps:

1. Log in to the Wiremock service at https://app.wiremock.cloud and click on the `Stubs` section of the example API. Then, create a new stub.
2. Complete the request, as follows:

    - Name: `Put Pizza`
    - Verb: `PUT`
    - Address: `/pizza`
    - Status: `200`
    - Body Type: `json`
    - Body: `{"message": "Pizza was updated"}`

Request

Name

Post Pizza

Add description

PUT /pizza

Advanced

Scenario

Response

Direct Fault Proxy

Status

200

Enable dynamic response templating

Header

Body

json

Indent Clear

```
1
2 {"message": "Pizza was updated"}
3
4
```

Delay

No delay

Add webhook

Save Cancel

Figure 9.21: WireMock Put Pizza stub

3. Click the **Save** button.
4. In the Flutter project, add a `putPizza` method to the `HttpHelper` class in the `http_helper.dart` file:

```
  Future<String> putPizza(Pizza pizza) async {
    const putPath = '/pizza';
    String put = json.encode(pizza.toJson());
    Uri url = Uri.https(authority, putPath);
    http.Response r = await http.put(
      url,
      body: put,
    );
    return r.body;
  }
  }
```

5. In the `PizzaDetailScreen` class in the `pizza_detail.dart` file, add two properties, a `Pizza` and a `boolean`, and in the constructor, set the two properties:

```
  final Pizza pizza;
  final bool isNew;
  const PizzaDetailScreen(
      {super.key, required this.pizza, required
this.isNew});
```

6. In the `PizzaDetailScreenState` class, override the `initState` method. When the `isNew` property of the `PizzaDetail` class is not new, it sets the content of the `TextFields` with the values of the `Pizza` object that was passed:

```
@override
  void initState() {
    if (!widget.isNew) {
      txtId.text = widget.pizza.id.toString();
      txtName.text = widget.pizza.pizzaName;
      txtDescription.text = widget.pizza.description;
      txtPrice.text = widget.pizza.price.toString();
      txtImageUrl.text = widget.pizza.imageUrl;
    }
```

```
    super.initState();
  }
```

7. Edit the `savePizza` method so that it calls the `helper.postPizza` method when the `isNew` property is `true`, and `helper.putPizza` when it's `false`:

```
Future savePizza() async {
...
    final result = await (widget.isNew
  ? helper.postPizza(pizza)
  : helper.putPizza(pizza));
  setState(() {
      operationResult = result;
    });
  }
```

8. In the `main.dart` file, in the `build` method of `_MyHomePageState`, add the `onTap` property to `ListTile` so that when a user taps on it, the app will change route and show the `PizzaDetail` screen, passing the current `pizza` and `false` for the `isNew` parameter:

```
return ListTile(
    title: Text(pizzas.data![position].pizzaName),
    subtitle: Text(pizzas.data![position].description +
                  ' - € ' +
                 pizzas.data!
[position].price.toString()),
    onTap: () {
       Navigator.push(
          context,
          MaterialPageRoute(
             builder: (context) => PizzaDetailScreen(
                pizza: pizzas.data![position], isNew:
false)),
    );
```

9. In the `floatingActionButton`, pass a new `Pizza` and `true` for the `isNew` parameter to the `PizzaDetail` route:

```
floatingActionButton: FloatingActionButton(
          child: Icon(Icons.add),
          onPressed: () {
```

```
            Navigator.push(
              context,
              MaterialPageRoute(
                  builder: (context) =>
PizzaDetailScreen(
                        pizza: Pizza(),
                        isNew: true,
                      )),
                );
          }),
    );
```

10. Run the app. On the main screen, tap on any `Pizza` to navigate to the `PizzaDetail` route.
11. Edit the pizza details in the text fields and press the **Save** button. You should see a message denoting that the pizza details were updated.

How it works...

This recipe is quite similar to the previous one, but here we used `PUT`, a verb that's conventionally used when an app asks the web server to update an existing piece of data.

This is why we had to instruct our mock web service to accept a PUT at the `/pizza` address—so that we could try sending some data to it.

Note that the `/pizza` address is exactly the same one we set for POST—the only thing that changes is the verb. In other words, you can perform a different action at the same URL, depending on the verb you use.

After creating the PUT stub in Wiremock, we created the `putPizza` method, which works just like the `postPizza` method, except for its verb.

To make the user update an existing `Pizza`, we used the same screen, `PizzaDetail`, but we also passed the `Pizza` object we wanted to update and a Boolean value that tells us whether the `Pizza` object is a new object (so it should use POST) or an existing one (so it should use PUT).

DELETE-ing data

In this recipe, you will learn how to perform a DELETE action on a web service. This is useful when your apps need to delete existing data from a web service.

Getting ready

To follow along with this recipe, you must have completed the code in the previous recipe.

How to do it...

To perform a DELETE action on a web service, follow these steps:

1. Log in to the Wiremock service at https://app.wiremock.cloud and click on the `Stubs` section of the example API. Then, create a new stub.
2. Complete the request, with the following data:

    - Name: `Delete Pizza`
    - Verb: `DELETE`
    - Address: `/pizza`
    - Status: `200`
    - Body Type: `json`
    - Body: `{"message": "Pizza was deleted"}`

3. Save the new stub.
4. In the Flutter project, add a `deletePizza` method to the `HttpHelper` class in the `http_helper.dart` file:

```
Future<String> deletePizza(int id) async {
    const deletePath = '/pizza';
    Uri url = Uri.https(authority, deletePath);
    http.Response r = await http.delete(
      url,
    );
    return r.body;
  }
```

5. In the `main.dart` file, in the `build` method of the `_MyHomePageState` class, refactor `itemBuilder` of `ListView.builder` so that `ListTile` is contained in a `Dismissible` widget, as follows:

```
return ListView.builder(
    itemCount: (pizzas.data == null) ? 0 :
pizzas.data.length,
    itemBuilder: (BuildContext context, int position) {
        return Dismissible(
                    key: Key(position.toString()),
                    onDismissed: (item) {
                      HttpHelper helper = HttpHelper();
                      pizzas.data!.removeWhere(
                          (element) => element.id ==
pizzas.data![position].id);
                      helper.deletePizza(pizzas.data!
[position].id!);
                    },
                    child: ListTile(...
```

6. Run the app. When you swipe any element from the list of pizzas, `ListTile` disappears.

How it works...

This recipe is quite similar to the previous two, but here, we used `DELETE`, a verb that's conventionally used when an app asks the web server to delete an existing piece of data.

This is why we had to instruct our mock web service to accept a `DELETE` at the `/pizza` address—so that we could try sending some data to it and it would respond with a success message.

After creating the `DELETE` stub in Wiremock, we created the `deletePizza` method, which works just like the `postPizza` and `putPizza` methods; it just needs the ID of the pizza to delete it.

To make the user delete an existing `Pizza`, we used the `Dismissible` widget, which is used when you want to swipe an element (left or right) to remove it from the screen.

Summary

In this chapter, you have seen several ways to manage data in Flutter applications and read and write data to web services.

First, you have seen the process of converting Dart models into JSON format, a lightweight and universally accepted data format.

You have also seen how to serialize and deserialize Dart objects to and from JSON and learned how to deal with JSON data that is incompatible with your application's models, with a few strategies to adapt and overcome this challenge.

You've used `SharedPreferences`, a simple key-value storage system that allows saving small amounts of data. You have then seen how to use `SecureStorage` as a method for storing sensitive data like passwords.

You learned how to access the filesystem leveraging the `path_provider` package, which enables developers to retrieve locations on the device's filesystem. You have seen how to create, read, and write files on your device.

Shifting the focus to communication with web services, you've learned about the design of an HTTP client and seen how to retrieve data from a web service and send data to the server using the POST, PUT, and DELETE verbs.

In summary, in this chapter, you got an in-depth look at JSON handling, data persistence, secure storage, and communication with the internet. You

are now equipped with the knowledge and skills necessary to effectively manage data and communicate with remote servers in your Flutter apps.

Advanced State Management with Streams

In Dart and Flutter, futures and streams are the main tools to deal with asynchronous programming.

While `Future` represents a single value that will be delivered at a later time in the future, `Stream` is a set (sequence) of values (0 or more) that can be delivered asynchronously, at any time. Basically, it is a flow of continuous data.

In order to get data from a stream, you subscribe (or *listen*) to it. Each time some data is emitted, you can receive and manipulate it as required by the logic of your app.

> **Note**
>
> By default, each stream allows only a single subscription, but you will also see how to enable multiple subscriptions.

This chapter focuses on several use cases of streams in a Flutter app. You will see streams used in different scenarios, and how to read and write data to streams, build user interfaces based on streams, and use the BLoC state management pattern.

Like futures, streams can generate *data* or *errors*, so you will also see how to deal with those in our recipes.

In this chapter, we will cover the following topics:

- How to use Dart streams
- Using stream controllers and sinks

- Injecting data transforms into streams
- Subscribing to stream events
- Allowing multiple stream subscriptions
- Using StreamBuilder to create reactive user interfaces
- Using the BLoC pattern

By the end of this chapter, you will be able to understand and use streams in your Flutter apps.

Technical requirements

To follow along with the recipes in this chapter, you should have the following software installed on your Windows, Mac, Linux, or ChromeOS device:

- The Flutter SDK
- The Android SDK when developing for Android
- macOS and Xcode when developing for iOS
- An emulator or simulator, or a connected mobile device enabled for debugging
- Your favorite code editor: Android Studio, Visual Studio Code, and IntelliJ IDEA are recommended; all should have the Flutter/Dart extensions installed

You'll find the code for the recipes in this chapter on GitHub at https://github.com/PacktPublishing/Flutter-Cookbook-Second-Edition/tree/main/Chapter_10.

How to use Dart streams

In this recipe's demo, you will change the background color of a screen each second. You will create a list of five colors, and every second you will change the background color of a `Container` widget that fills the whole screen.

The color information will be *emitted* from a `Stream` of data. The main screen will need to *listen* to the `Stream` to get the current `Color`, and update the background accordingly.

While changing a color isn't exactly something that strictly requires a `Stream`, the principles explained here may apply to more complex scenarios, including getting flows of data from a web service. For instance, you could write a chat app that updates the content based on what users write in real time, or an app that shows stock prices in real time.

Getting ready

In order to follow along with this recipe, you can download the starting code for the project at https://github.com/PacktPublishing/Flutter-Cookbook-Second-Edition/tree/main/Chapter_10.

How to do it...

For this recipe, we will build a simple example of using streams:

1. Create a new app and name it `stream_demo`.
2. Update the content of the `main.dart` file as follows:

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Stream',
```

```
      theme: ThemeData(
        primarySwatch: Colors.deepPurple,
      ),
      home: const StreamHomePage(),
    );
  }
}
class StreamHomePage extends StatefulWidget {
  const StreamHomePage({super.key});
  @override
  State<StreamHomePage> createState() =>
_StreamHomePageState();
}
class _StreamHomePageState extends
State<StreamHomePage> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

3. Create a new file in the `lib` folder of your project, called `stream.dart`.
4. In the `stream.dart` file, import the `material.dart` library and create a new class, called `ColorStream`, similar to what is shown here:

```
import 'package:flutter/material.dart';
class ColorStream {
}
```

5. At the top of the `ColorStream` class, create a list of colors called `colors`, containing five colors (or any other colors of your choice):

```
    final List<Color> colors = [
      Colors.blueGrey,
      Colors.amber,
      Colors.deepPurple,
      Colors.lightBlue,
      Colors.teal
    ];
```

6. In the `ColorStream` class, a method called `getColors` returns a stream of the `Color` type and is marked as `async*` (note the asterisk at

the end of `async`, used for `Streams`):

```
Stream<Color> getColors() async* {
}
```

7. After the `colors` declaration, add the `yield*` command, as shown here. This completes the `ColorStream` class:

```
    yield* Stream.periodic(
      const Duration(seconds: 1), (int t) {
      int index = t % colors.length;
      return colors[index];
    });
```

8. In the `main.dart` file, import the `stream.dart` file:

```
import 'stream.dart';
```

9. At the top of the `_StreamHomePageState` class, add two properties, `Color` and `ColorStream`, as shown here:

```
  Color bgColor = Colors.blueGrey;
  late ColorStream colorStream;
```

10. In the `main.dart` file, at the bottom of the `_StreamHomePageState` class, add an asynchronous method called `changeColor`, which listens to the `colorStream.getColors` stream and updates the value of the `bgColor` state variable:

```
void changeColor() async {
     await for (var eventColor in
colorStream.getColors()) {
      setState(() {
        bgColor = eventColor;
      });
    }
  }
```

11. Override the `initState` method in the `_StreamHomePageState` class. There, initialize `colorStream` and call the `changeColor` method:

```
@override
 void initState() {
   super.initState();
   colorStream = ColorStream();
   changeColor();

 }
```

12. Finally, in the `build` method, return a scaffold. In the `body` of `Scaffold`, add a container with a `decoration`, whose `BoxDecoration` will read the `bgColor` value to update the background color of `Container`:

```
@override
 Widget build(BuildContext context) {
   return Scaffold(
       appBar: AppBar(
         title: const Text('Stream'),
       ),
       body: Container(
         decoration: BoxDecoration(color: bgColor),
       ));
 }
```

13. Run the app and you should see the screen changing color every second on your emulator:

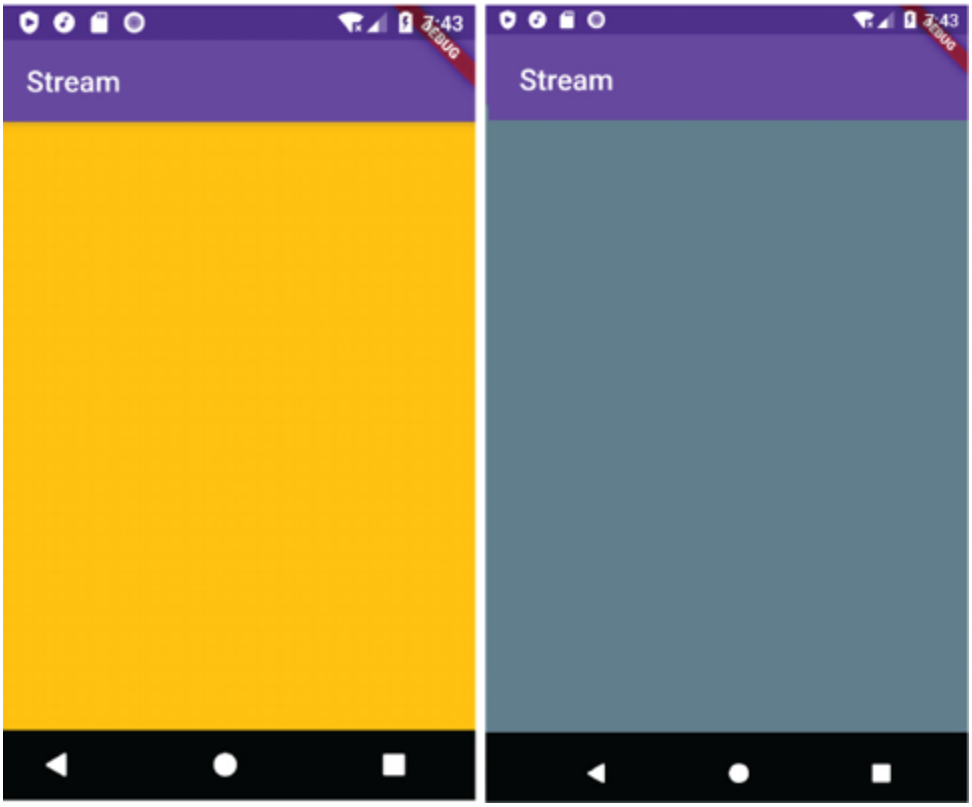


Figure 10.1: Colors changing on the screen through a Stream of data

Pro Tip

To see the full-color version of Figure 10.1, and all the other images in this book, download the PDF file here: https://packt.link/WE615

How it works...

The two core parts of the app that you have implemented in this recipe create a stream of data, and listen (or subscribe) to the stream.

You have created a stream of data in the `stream.dart` file. Here, you added a method that returns a `Stream` of `Color`, and you marked the method as `async*`:

```
Stream<Color> getColors() async* {
```

In previous chapters, we have always marked a function as `async` (**without** the asterisk * symbol). In Dart and Flutter, you use `async` for futures and `async*` (**with** the asterisk * sign) for streams.As mentioned before, the main difference between a stream and a future is the number of events that are returned: just 1 for `Future`, and 0 to many for `Stream`. When you mark a function as `async*`, you create a specific type of function called a **generator function** because it generates a sequence of values (a stream).

Note the following code snippet:

```
yield* Stream.periodic(const Duration(seconds: 1), (int t) {
      int index = t % colors.length;
      return colors[index];
    });
```

In order to return a stream in an `async*` method, you use the `yield*` statement. Simplifying this slightly, you can think of `yield*` as a return statement, with a huge difference: `yield*` **does not** end the function.

`Stream.periodic()` is a constructor that creates a `Stream`. The stream **emits events** at the intervals that you specify in the value you pass as a parameter. In our code, the stream will emit a value each second.

In the method inside the `Stream.periodic` constructor, we use the modulus operator to choose which color to show, based on the number of seconds that have passed since the beginning of the call to the method, and we return the appropriate color.

This creates a stream of data. In the `main.dart` file, you added the code to listen to the stream with the `changeColor` method:

```
changeColor() async {
     await for (var eventColor in colorStream.getColors()) {
      setState(() {
        bgColor = eventColor;
      });
    }
  }
```

The core of the method is the `await for` command. This is an asynchronous `for` loop that iterates over the events of a stream. Basically, it's like a `for` (or `for each`) loop, but instead of iterating over a set of data (like a list), it asynchronously keeps listening to each event in a stream. From there, we call the `setState` method to update the `bgColor` property.

There's more...

Instead of using an asynchronous `for` loop, you can also leverage the `listen` method over a stream. To do that, perform the following steps:

1. Remove or comment out the content of the `changeColor` method in the `main.dart` file.
2. Add the following code to the `changeColor` method:

```
colorStream.getColors().listen((eventColor) {
 setState(() {
   bgColor = eventColor;
 });
});
```

3. Run the app. You'll notice that our app behaves just like before, changing the color of the screen each second.

The main difference between `listen` and `await for` is that when there is some code after the loop, `listen` will allow the execution to continue, while `await for` stops the execution until the stream is completed.

In this particular app, we never stop listening to the stream, but you should always close a stream when it has completed its tasks. In order to do that, you can use the `close()` method, as shown in the next recipe.

Streams in Flutter are a powerful way to handle asynchronous data and can be used in several real-world scenarios, like real-time messaging, file uploads and downloads, user location tracking, dealing with data (like sensor data from a device), and several more.

See also

A great resource to obtain information about streams is the official tutorial available at https://dart.dev/tutorials/language/streams.

Using stream controllers and sinks

`StreamControllers` create a linked `Stream` and `Sink`. While streams contain data emitted sequentially that can be received by any subscriber, `sinks` are used to insert events.

`StreamControllers` simplify stream management, automatically creating a stream and a sink, and methods to control their events and features.

In this recipe, you will create a stream controller to listen to and insert new events. This will show you how to fully control a stream.

Getting ready

In order to follow along with this recipe, you should have completed the code in the previous recipe, *How to use Dart streams*.

How to do it...

For this recipe, we will make our app display a random number on the screen, leveraging `StreamControllers` and their `sink` property:

1. In the `stream.dart` file, import the `dart:async` library:

```
import 'dart:async';
```

2. At the bottom of the `stream.dart` file, add a new class, called `NumberStream`:

```
class NumberStream {
}
```

3. In the `NumberStream` class, add a stream controller of the `int` type, called `controller`:

```
final StreamController<int> controller =
StreamController<int>();
```

4. Still in the `NumberStream` class, add a method called `addNumberToSink`, with the code shown here:

```
  void addNumberToSink(int newNumber) {
    controller.sink.add(newNumber);
  }
```

5. Next, add a `close` method at the bottom of the class:

```
  close() {
    controller.close();
  }
```

6. In the `main.dart` file, add the imports to `dart:async` and `dart:math`:

```
import 'dart:async';
import 'dart:math';
```

7. At the top of the `_StreamHomePageState` class, declare three variables: `int`, `StreamController`, and `NumberStream`:

```
  int lastNumber = 0;
  late StreamController numberStreamController;
  late NumberStream numberStream;
```

8. Edit the `initState` method so that it contains the code shown here:

```
 @override
  void initState() {
    numberStream = NumberStream();
    numberStreamController = numberStream.controller;
    Stream stream = numberStreamController.stream;
```

```
    stream.listen((event) {
      setState(() {
        lastNumber = event;
      });
    });
    super.initState();
  }
```

9. Override the `dispose` method, and call the `close` method on `numberStream`:

```
@override
  void dispose() {
    numberStreamController.close();
    super.dispose();
  }
```

10. At the bottom of the `_StreamHomePageState` class, add a method called `addRandomNumber`, with the code shown here:

```
void addRandomNumber() {
    Random random = Random();
    int myNum = random.nextInt(10);
    numberStream.addNumberToSink(myNum);
}
```

11. In the `build` method, edit the body of the `Scaffold` so that it contains a `SizedBox` with a `Column`, which in turn contains a `Text` and an `ElevatedButton`:

```
body: SizedBox (
          width: double.infinity,
          child: Column(
            mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
            crossAxisAlignment:
CrossAxisAlignment.center,
            children: [
              Text(lastNumber.toString()),
              ElevatedButton(
                onPressed: () => addRandomNumber(),
                child: Text('New Random Number'),
              )
            ],
```

```
            ),
        )
```

12. Run the app. You'll notice that each time you press the button, a random number appears on the screen, as shown in the following screenshot:



Figure 10.2: Random number through a StreamController

How it works...

You can think of a stream as a one-way pipe, with two ends. One end of the pipe only allows you to insert data, and the other end is where data gets out.

In Flutter:

- You can use a `StreamController` to control a stream.

- A `StreamController` has a `sink` property to insert new data.
- The `stream` property of `StreamController` is a way out of `StreamController`.

You can see a diagram of this concept in the following figure:

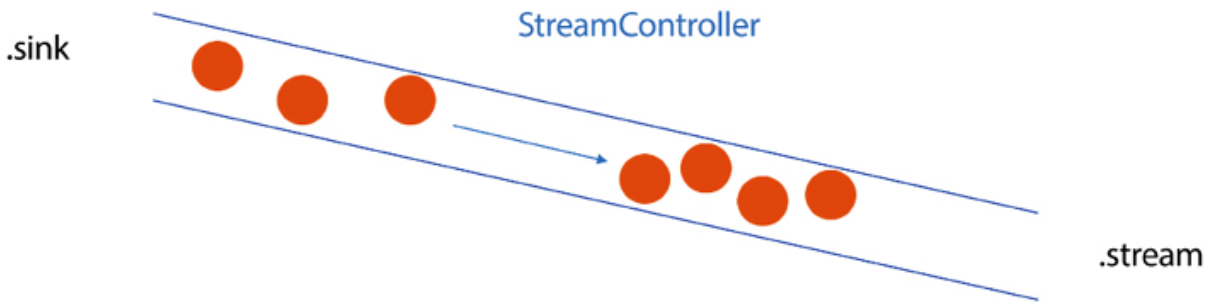


Figure 10.3: Flutter StreamController

In the app you have built in this recipe, the first step involved the creation of a `StreamController`, and you did that in the `NumberStream` class, with the help of the following command:

```
StreamController<int> controller = StreamController<int>();
```

As you can see, a stream controller is generic, and you can choose its type (in this case, `int`) depending on the needs of your app.

The next step was adding data to the stream controller, leveraging its `sink` property, and we did that with the following command:

```
controller.sink.add(newNumber);
```

Specifically, a sink is an instance of the `StreamSink` class, which is the "way in" for a stream.

The `stream` property of `StreamController` contains an instance of a stream, and we can listen to its `stream`, leveraging the `listen` method over it. In our code, we did this with the following commands:

```
Stream stream = numberStreamController.stream;
    stream.listen((event) {
      setState(() {
        lastNumber = event;
      });
    })
```

As a side note, in this code:

```
body: SizedBox (
          width: double.infinity,
```

by setting the width of the `SizedBox` to `double.infinity`, we are telling the widget to take all the available horizontal space in its parent.

The effect of `double.infinity` depends on the parent's own constraints: if the parent widget allows for infinite width, then the child will fill all the available horizontal space. If the parent has width constraints, the child will be limited by that.

So in this simple example, you used `StreamController`, `Stream`, and `StreamSink`. But there is another important feature that we should mention in this recipe: dealing with errors.

There's more...

`StreamController` also helps you when dealing with errors. To enable error handling, perform the following steps:

1. In the `stream.dart` file, add a new method, called `addError`, with the following code:

```
addError() {
    controller.sink.addError('error');
  }
```

2. In the `main.dart` file, ppend the `StreamHomePageState` of the `onError` method to the listen method in the `initState` function, as

shown here:

```
stream.listen((event) {
      setState(() {
        lastNumber = event;
      });
    }).onError((error) {
      setState(() {
        lastNumber = -1;
      });
    });
```

3. Finally, in the `addRandomNumber` method, comment out the call to `addNumberToSink` and call `addError` on the `numberStream` instance instead:

```
  void addRandomNumber() {
    Random random = Random();
    //int myNum = random.nextInt(10);
    //numberStream.addNumberToSink(myNum);
    numberStream.addError();
  }
```

4. Run the app and press the New Random Number button. Note the error caused by the `addError` method.
5. Remove the comments on the number generator lines, and comment out the `addError` method beforehand so that you can complete the following recipes.

As you can see, another great feature of a stream controller is the ability to catch errors, and you do that with the `onError` function. You can raise errors by calling the `addError` method over `StreamSink`.

See also

There is a great article on creating streams and using `StreamController` in Dart available at the following address: https://dart.dev/articles/libraries/creating-streams.

Injecting data transforms into streams

A rather common scenario is the need to manipulate and transform data emitted from a stream before it gets to its final destination.

This is extremely useful when you want to *filter* data based on any type of condition, validate it, modify it before showing it to your users, or process it to generate some new output.

Examples include converting a number into a string, making a calculation, or omitting data repetitions.

In this recipe, you will inject `StreamTransformers` into `Streams` in order to map and filter data.

Getting ready

In order to follow along with this recipe, you should have completed the code in the previous recipe, *Using stream controllers and sinks*.

How to do it...

For this recipe, you will edit the random number on the screen, leveraging a `StreamTransformer`, starting from the code that you completed in the previous recipe, *Using stream controllers and sinks*:

1. At the top of the `_StreamHomePageState` class, in the `main.dart` file, add a declaration of `StreamTransformer`:

```
  late StreamTransformer transformer;
```

2. In the `initState` method, just after the declarations, create an instance of `StreamTransformer`, calling the `fromHandlers`

constructor:

```
transformer = StreamTransformer<int, int>.fromHandlers(
        handleData: (value, sink) {
          sink.add(value * 10);
        },
        handleError: (error, trace, sink) {
          sink.add(-1);
        },
        handleDone: (sink) => sink.close());
```

3. Still in the `initState` method, edit the `listen` method over the stream, so that you call `transform` over it, passing the `transformer` as a parameter:

```
stream.transform(transformer).listen((event) {
      setState(() {
        lastNumber = event;
      });
    }).onError((error) {
      setState(() {
        lastNumber = -1;
      });
    });
    super.initState();
```

4. Run the app. Now you will see that the numbers go from 0 to 90, instead of from 0 to 9, as shown in the following screenshot:



Figure 10.4: Using a StreamTransformer

How it works...

It is often useful to manipulate values when listening to a stream. In Dart, `StreamTransformer` is an object that performs data transformations on a stream, so that the listeners of the stream then receive the transformed data.

In the code in this recipe, you transformed the random number emitted by the stream by multiplying it by 10.

The first step was to create a stream transformer, using the `fromHandlers` constructor:

```
transformer = StreamTransformer<int, dynamic>.fromHandlers(
        handleData: (value, sink) {
          sink.add(value * 10);
        },
        handleError: (error, trace, sink) {
          sink.add(-1);
        },
        handleDone: (sink) => sink.close());
```

With the `StreamTransformer.fromHandlers` constructor, you specify callback functions with three named parameters: `handleData`, `handleError`, and `handleDone`.

`handleData` receives data events emitted from the stream. This is where you apply the transformation you need to perform. The function you specify in `handledata` receives as parameters the data emitted by the stream and the `EventSink` instance of the current stream.

Here, you used the `add` method to send the transformed data to the stream listener. `handleError` responds to error events emitted by the stream. The arguments here contain the error, a stack trace, and the `EventSink` instance.

`handleDone` is called when there is no more data, when the `close()` method of the stream's sink is called.

See also

There are other ways to transform data into a stream. One of those is using the `map` method, which creates a stream that converts each element in to a new value. For more information, have a look at https://api.dart.dev/stable/2.18.6/dart-async/Stream/map.htmland https://dart.dev/articles/libraries/creating-streams.

Subscribing to stream events

In the previous recipes of this chapter, we used the `listen` method to get values from a stream. This generates a `Subscription`. Subscriptions contain methods that allow you to listen to events from streams in a structured way.

In this recipe, we will use `Subscription` to gracefully handle events and errors, and close the subscription.

Getting ready

In order to follow along with this recipe, you should have completed the code in the previous recipe, *Injecting data transforms into streams*.

How to do it...

For this recipe, we will use a `StreamSubscription` with its methods. We will also add a button to close the stream. Perform the following steps:

1. At the top of the `_StreamHomePageState` class, declare a `StreamSubscription` called `subscription`:

```
late StreamSubscription subscription;
```

2. In the `initState` method of the `_StreamHomePageState` class, remove `StreamTransformer` and set the subscription. The final result is shown here:

```
@override
  void initState() {
    numberStream = NumberStream();
    numberStreamController = numberStream.controller;
    Stream stream = numberStreamController.stream;
    subscription = stream.listen((event) {
      setState(() {
        lastNumber = event;
      });
```

```
    });
   super.initState();
  }
```

3. Still in the `initState` method, after setting `subscription`, set the optional `onError` property of `subscription`:

```
 subscription.onError((error) {
      setState(() {
        lastNumber = -1;
      });
    });
```

4. After the `onError` property, set the `onDone` property:

```
subscription.onDone(() {
      print('OnDone was called');
});
```

5. At the bottom of the `_StreamHomePageState` class, add a new method, called `stopStream`. It calls the `close` method of `StreamController`:

```
void stopStream() {
    numberStreamController.close();
}
```

6. In the `dispose` method, cancel the subscription:

```
subscription.cancel();
```

7. In the `build` method, at the end of the `Column` widget, add a second `ElevatedButton`, which calls the `stopStream` method:

```
ElevatedButton(
    onPressed: () => stopStream(),
    child: const Text('Stop Subscription'),
)
```

8. Edit `addRandomNumber` so that it checks the `isClosed` value of `StreamController` before adding a number to the sink. If the

`isClosed` property is true, call the `setState` method to set `lastNumber` to `-1`:

```
void addRandomNumber() {
   Random random = Random();
   int myNum = random.nextInt(10);
   if (!numberStreamController.isClosed) {
     numberStream.addNumberToSink(myNum);
   } else {
     setState(() {
       lastNumber = -1;
     });
   }
 }
```

9. Run the app. You should now see two buttons on the screen, as shown in the following screenshot:

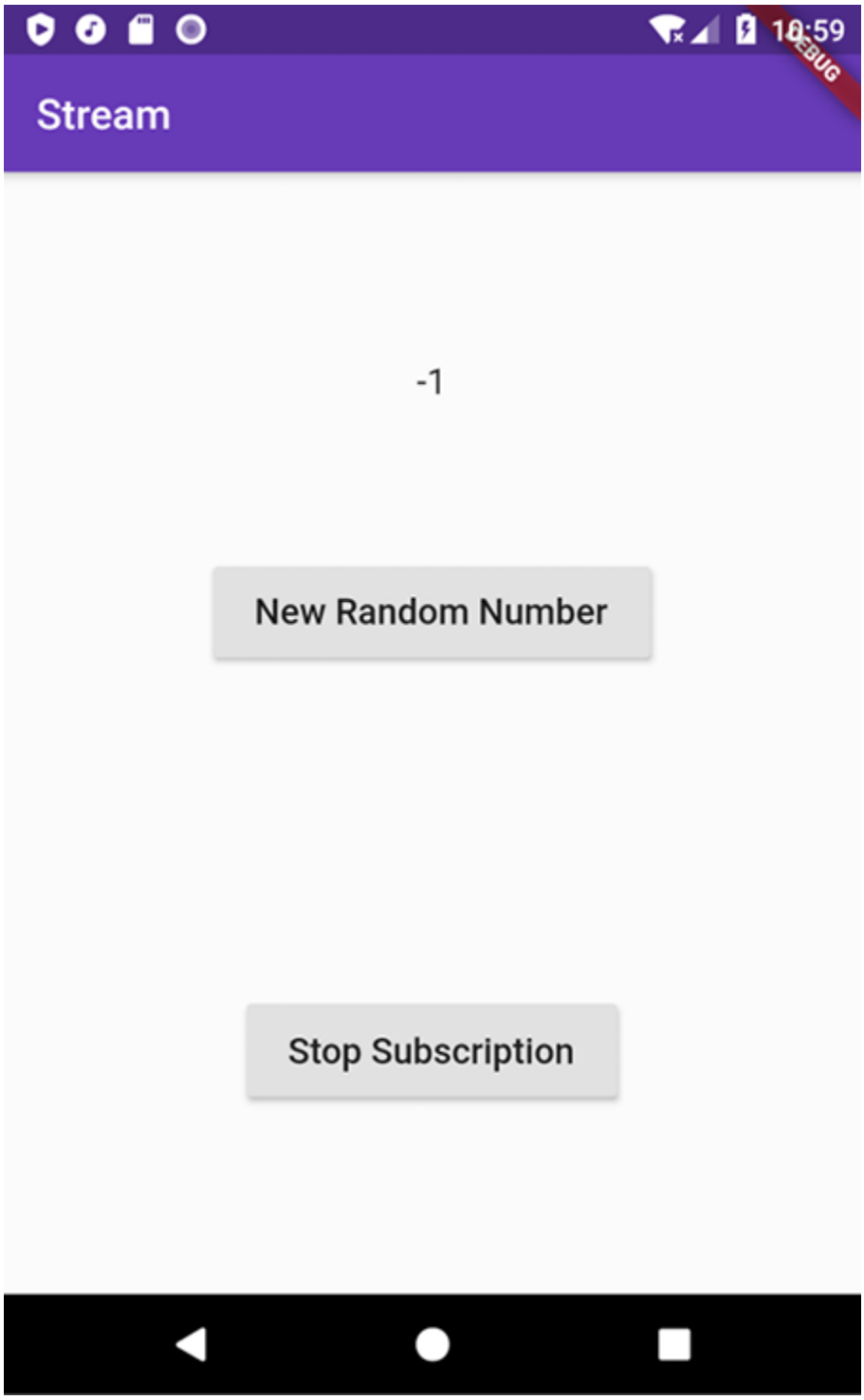


Figure 10.5: Cancelling a StreamSubscription

10. Press the **Stop Subscription** button, and then the **New Random Number** button. In the debug console, you should see a -1 on the

screen and the message `OnDone was called` in the **Debug Console**, as shown in the following screenshot:

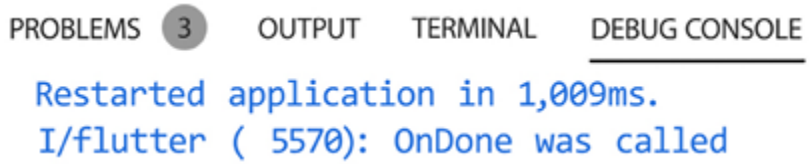


Figure 10.6: The subscription onDone callback

How it works...

When you call the `listen` method over a stream, you get a `StreamSubscription`. When you create a `StreamSubscription`, you may set four parameters, one required and three optional:

Parameter	Optional/Required	Type
`onListen`	Required – positional as the first parameter	Function
`onDone`	Optional	Function
`onError`	Optional	Function
`cancelOnError`	Optional	Bool

Table 10.1: `StreamSubscription` parameters

In the example in this recipe, we set the first parameter (`onListen`) when we create the `StreamSubscription`:

```
subscription = stream.listen((event) {
      setState(() {
        lastNumber = event;
      });
    });
```

As you also saw in previous recipes, this callback is triggered whenever some data is emitted by the stream. For the optional parameters, we set them later through `subscription` properties.

In particular, we set the `onError` property with the help of the following command:

```
subscription.onError((error) {
 setState(() {
 lastNumber = -1;
 });
 });
```

`onError` gets called whenever the stream emits an error. In this case, we want to show a -1 on the screen, so we set the state value of `lastNumber` to `-1`.

Another useful callback is `onDone`, which we set with the following command:

```
subscription.onDone(() {
      print('OnDone was called');
});
```

This is called whenever there is no more data from `StreamSubsription`, usually because it has been closed. In order to try the `onDone` callback, we had to explicitly close `StreamController`, with its `close` method:

```
numberStreamController.close();
```

Once you close `StreamController`, the subscription's `onDone` callback is executed, and therefore, the `OnDone was called` message appears in the Debug console.

We did not set `cancelOnError` (which is `false` by default). When `cancelOnError` is `true`, the subscription is automatically canceled whenever an error is raised.

Now, if the user presses the **New Random Number** button, and the app tries to add a new number to the sink, an error will be raised, as `StreamController` has been closed. That's why it is always a good idea to check whether the subscription has been closed with the following command:

```
if (!numberStreamController.isClosed) { ...
```

This allows you to make sure your `StreamController` is still alive before performing operations on it.

See also

For the complete properties and methods of `StreamSubscription`, have a look at https://api.flutter.dev/flutter/dart-async/StreamController-class.html.

Allowing multiple stream subscriptions

By default, each stream allows only a single subscription. If you try to listen to the same subscription more than once, you get an error. Flutter also has a specific kind of stream, called a **broadcast** stream, which allows multiple listeners. In this recipe, you will see a simple example of using a broadcast stream.

Getting ready

In order to follow along with this recipe, you should have completed the code in the previous recipe, *Subscribing to stream events*.

How to do it...

For this recipe, we'll add a second listener to the stream that we built in the previous recipes in this chapter:

1. At the top of the `_StreamHomePageState` class, in the `main.dart` file, declare a second `StreamSubscription`, called `subscription2`, and a string, called `values`:

```
late StreamSubscription subscription2;
String values = '';
```

2. In the `initState` method, edit the first subscription and listen to the stream with the second subscription:

```
subscription = stream.listen((event) {
      setState(() {
        values += '$event - ';
      });
    });

 subscription2 = stream.listen((event) {
      setState(() {
        values += '$event - ';
      });
    });
```

3. Run the app. You should see an error, as shown in the following screenshot:

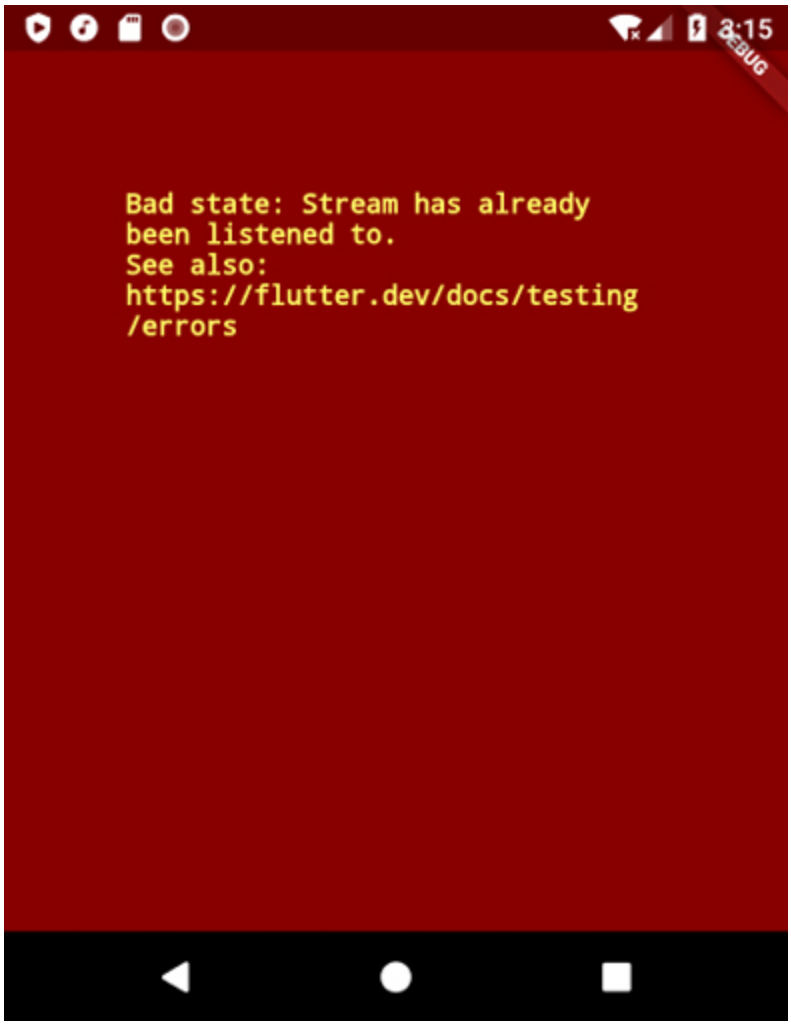


Figure 10.7: Error on multiple listeners on a Stream

4. Still in the `initState` method, set the stream as a broadcast stream:

```
 void initState() {
    numberStream = NumberStream();
    numberStreamController = numberStream.controller;
    Stream stream =
numberStreamController.stream.asBroadcastStream();
...
```

5. In the `build` method, edit the text in the column so that it prints the `values` string:

```
child: Column(
            mainAxisAlignment:
```

```
MainAxisAlignment.spaceEvenly,
            crossAxisAlignment:
CrossAxisAlignment.center,
            children: [
              Text(values),
```

6. Run the app and press the **New Random Number** button a few times. Each time you press the button, the value is appended to the string twice, as shown in the following screenshot:

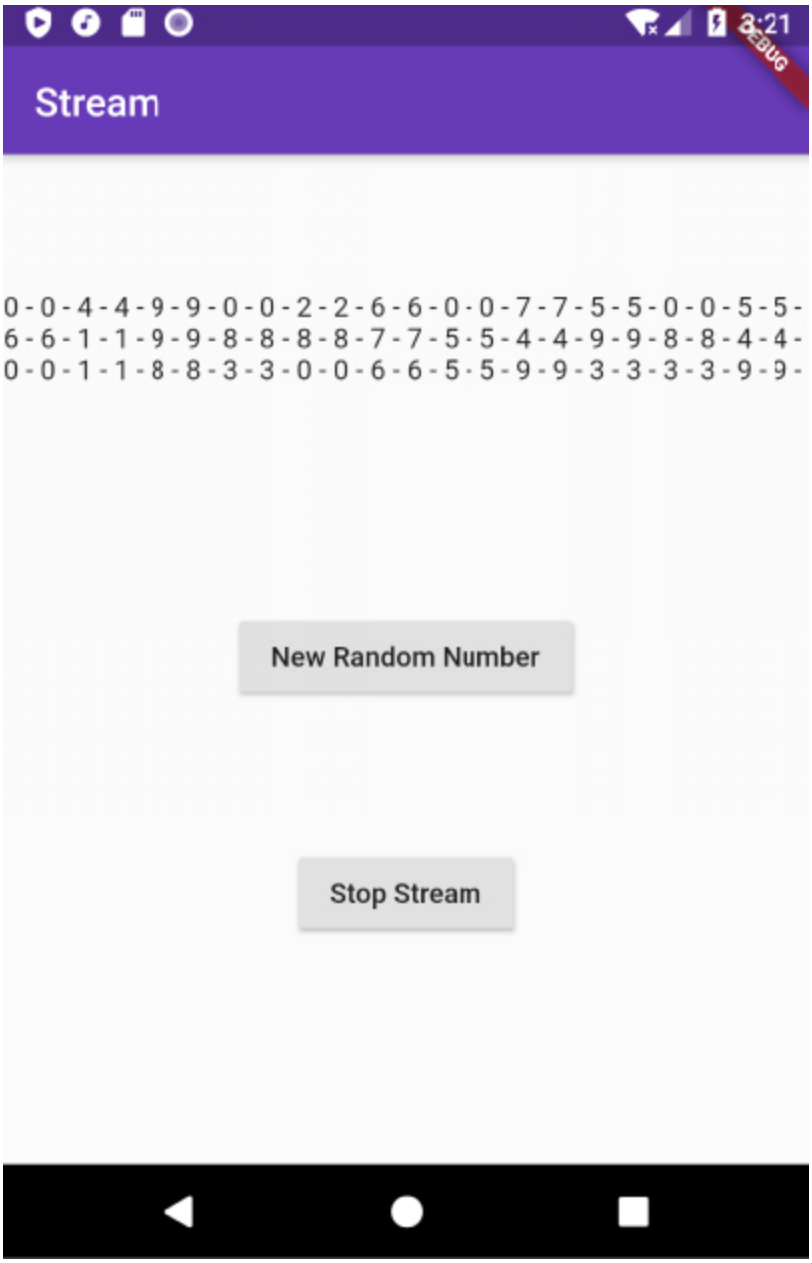


Figure 10.8: Multiple subscriptions on a BroadcastStream

How it works...

In the first part of this recipe, we tried to create two `StreamSubscriptions`, listening to the same stream, and this generated the "*Stream has already been listened to*" error.

The `stream.asBroadcastStream()` method returns a multi-subscription (broadcast) stream. We created our broadcast stream with the following command:

```
Stream stream =
numberStreamController.stream.asBroadcastStream();
```

Each subscriber receives the same data, so in our code, each value was repeated twice.

See also

There is an interesting discussion on the difference between single and broadcast subscriptions in Dart, available at the following link: https://www.dartcn.com/articles/libraries/broadcast-streams.

Using StreamBuilder to create reactive user interfaces

`Streambuilder` is a widget that listens to events emitted by a stream, and whenever an event is emitted, it rebuilds its descendants. Like the `FutureBuilder` widget, which we saw in *Chapter 7*, *The Future Is Now: Introduction to Asynchronous Programming*, `StreamBuilder` makes it extremely easy to build reactive user interfaces that update every time new data is available.

In this recipe, we will update the text on the screen with `StreamBuilder`. This is very efficient compared to the update that happens with a `setState` method and its build call, as **only the widgets contained in** `StreamBuilder` **are actually redrawn**.

Getting ready

In order to follow along with this recipe, you will build an app from scratch. Call this new app `streambuilder_app`.

How to do it...

For this recipe, we will create a stream and use `StreamBuilder` to update the user interface. Perform the following steps:

1. In your new app, in the `lib` folder of your project, create a new file, called `stream.dart`.
2. In the `stream.dart` file, create a class called `NumberStream`:

```
class NumberStream {}
```

3. Inside the `NumberStream` class, add a method that returns a stream of the `int` type and returns a new random number each second:

```
import 'dart:math';
class NumberStream {
  Stream<int> getNumbers() async* {
    yield* Stream.periodic(const Duration(seconds: 1),
(int t) {
      Random random = Random();
      int myNum = random.nextInt(10);
      return myNum;
    });
  }
}
```

4. In the `main.dart` file, edit the existing code of the sample app so that it looks like the following:

```
import 'package:flutter/material.dart';
import 'stream.dart';
import 'dart:async';
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Stream',
      theme: ThemeData(
        primarySwatch: Colors.deepPurple,
      ),
      home: const StreamHomePage(),
    );
  }
}
class StreamHomePage extends StatefulWidget {
  const StreamHomePage({super.key});
  @override
  State<StreamHomePage> createState() =>
_StreamHomePageState();
}
class _StreamHomePageState extends
State<StreamHomePage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: const Text('Stream'),
        ),
        body: Container(
          ),
        );
  }
}
```

5. At the top of the `_StreamHomePageState` class, declare a stream of the `int` type called `numberStream`:

```
  late Stream<int> numberStream;
```

6. In the `_StreamHomePageState` class, override the `initState` method, and in there, call the `getNumbers` function from a new `NumberStream` instance:

```
@override
 void initState() {
   numberStream = NumberStream().getNumbers();
   super.initState();
 }
```

7. In the `build` method, in the container in the body of `Scaffold`, as a child, add a `StreamBuilder` that has `numberStream` in its `stream` property and, in the builder, will return a centered `Text` containing the snapshot data:

```
body: StreamBuilder(
        stream: numberStream,
        initialData: 0,
        builder: (context, snapshot) {
          if (snapshot.hasError) {
            print('Error!');
          }
          if (snapshot.hasData) {
            return Center(
                child: Text(
              snapshot.data.toString(),
              style: const TextStyle(fontSize: 96),
            ));
          } else {
            return const SizedBox.shrink();
          }
        },
      ),
),
```

8. Run the app. Now, every second, you should see a new number in the center of the screen, as shown in the following screenshot:



Figure 10.9: Streambuilder in action

How it works...

The first step when using `StreamBuilder` is setting its `stream` property, and we did that with the following command:

```
StreamBuilder(
          stream: numberStream,
```

With the `initialdata` property, you can specify which data to show when the screen loads and before the first event is emitted:

```
 initialData: 0,
```

Then you write a `builder`. This is a function that takes the current context and a snapshot, which contains the data emitted by the stream, in the `data` property. Hence, this is triggered automatically each time `Stream` emits a new event, and new data is available. In our example, we check whether the snapshot contains some data with the help of the following command:

```
 if (snapshot.hasData) {...
```

If there is data in the snapshot, we show it in a `Text`:

```
return Center(
    child: Text(
        snapshot.data.toString(),
        style: const TextStyle(fontSize: 96),
));
```

The snapshot `hasError` property allows you to check whether errors were returned. As usual, this is extremely useful, and you should always include it in your code to avoid unhandled exceptions:

```
if (snapshot.hasError) {
    print('Error!');
}
```

Note that in this last recipe, we never called a `setState` method. This way, we have made the first step into separating the logic and state of the app from the user interface. We will complete this transition in the next recipe, where we will see how to deal with state using the BLoC pattern.

See also

The documentation relating to `StreamBuilder` is complete and well laid out. Have a look at the official guide at https://api.flutter.dev/flutter/widgets/StreamBuilder-class.html.

Using the BLoC pattern

When using the BLoC pattern, everything is a stream of events. A **BLoC** (which stands for **Business Logic Component**) is a layer between any source of data and the user interface that will consume the data.

Examples of sources include HTTP data retrieved from a web service, or JSON received from a database.

The BLoC receives streams of data from the source, processes it as required by your business logic, and returns streams of data to its subscribers.

A simple diagram of the role of a BLoC is shown in *Figure 11.10*:

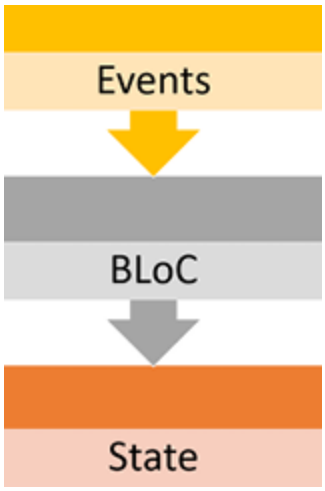


Figure 10.10: The BLoC pattern

The main reason for using BLoCs is to separate the concern of the business logic of your app from the presentation that occurs with your widgets, and it is especially useful when your apps become more complex and need to share state in several different places. This makes it easier to reuse your code in different parts of your application, or even in different apps. Also, as BLoCs are completely independent of the UI, they can be easily tested in isolation.

The example we will build in this recipe is very simple, but it can be scaled as needed for bigger apps.

Getting ready

To follow along with this recipe, create a new Flutter app, and call it `bloc_random`.

How to do it...

We will create a very simple app that generates a random number each time the user clicks a button, using a BLoC:

1. Create a new file in the `lib` folder of your project and call it `random_bloc.dart`.
2. In the `random_bloc.dart` file, import the `dart:async` and `dart:math` libraries:

```
import 'dart:async';
import 'dart:math;
```

3. Still in the `random_bloc.dart` file, create a new class, called `RandomNumberBloc`:

```
class RandomNumberBloc {}
```

4. At the top of the `RandomNumberBloc` class, create a `StreamController` to handle input events, another to handle output events, an input `Sink`, and an output `Stream`:

```
  // StreamController for input events
  final _generateRandomController =
StreamController<void>();
  // StreamController for output
  final _randomNumberController = StreamController<int>
();
  // Input Sink
  Sink<void> get generateRandom =>
_generateRandomController.sink;
  // Output Stream.
  Stream<int> get randomNumber =>
_randomNumberController.stream;
_secondsStreamController.sink;
```

5. Create an unnamed constructor. In it set up a listener on the `_generateRandomController` stream, and generate a random number between 0 and 9:

```
RandomNumberBloc() {
    _generateRandomController.stream.listen((_) {
      final random = Random().nextInt(10);
      _randomNumberController.sink.add(random);
    });
  }
```

6. Create a `dispose` method to clean up the resources:

```
  void dispose() {
    _generateRandomController.close();
    _randomNumberController.close();
  }
```

7. In the `main.dart` file, edit the existing code of the sample app so that it looks like the following:

```
import 'package:flutter/material.dart';
import 'random_screen.dart';
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const RandomScreen(),
    );
  }
}
```

8. Create a new file in the `lib` folder of your project and call it `random_screen.dart`.

9. At the top of the new file, import `material.dart` and the `random_bloc.dart` file:

```
import 'package:flutter/material.dart';
import 'random_bloc.dart';
```

10. Create a Stateful widget, calling it `RandomScreen`.
11. At the top of the `_RandomScreenState` class, declare a `RandomNumberBloc`:

```
final _bloc = RandomNumberBloc();
```

12. In the `_StreamHomePageState` class, override the `dispose` method to clean up the `_bloc`:

```
  @override
  void dispose() {
    _bloc.dispose();
    super.dispose();
  }
```

13. In the `build` method, type the code below:

```
 @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Random
Number')),
      body: Center(
        child: StreamBuilder<int>(
          stream: _bloc.randomNumber,
          initialData: 0,
          builder: (context, snapshot) {
            return Text(
              'Random Number: ${snapshot.data}',
              style: const TextStyle(fontSize: 24),
            );
          },
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () => _bloc.generateRandom.add(null),
```

```
        child: const Icon(Icons.refresh),
      ),
    );
  },
```

Run the app. You should see the random numbers between 0 and 9 each time you click the FloatingActionButton.

How it works...

When you want to use a BLoC as a state management pattern, a number of steps need to be performed, as follows:

1. Create a class that will serve as the BLoC.
2. In the class, declare the data that needs to be updated in the app.
3. Set `StreamControllers`.
4. Create the getters for streams and sinks.
5. Add the logic of the BLoC.
6. Add a constructor in which you'll set the data.
7. Listen to changes.
8. Set a `dispose` method.
9. From the UI, create an instance of the BLoC.
10. Use `StreamBuilder` to build the widgets that will use the BLoC data.
11. Add events to the sink for any changes to the data (if required).

Most of the code in this recipe is similar to the previous one, *Using StreamBuilder to create reactive user interfaces*. The main difference is that we moved the logic of the app (the random number in this case) to the BLoC class so that the user interface has almost no logic at all, which is the purpose of using a BLoC.

Note the code in the `RandumNumberBloc` constructor:

```
RandomNumberBloc() {
    _generateRandomController.stream.listen((_) {
      final random = Random().nextInt(10);
      _randomNumberController.sink.add(random);
    });
  }
}
```

Here, you set up a listener on the `_generateRandomController` stream. When an event is added to the input sink, this listener will be called.

Here, you generate a random number between 0 and 9, and then you add it to the output stream's sink using `_randomNumberController.sink.add(random)`.

Basically, the `RandomNumberBloc` class shows the BLoC pattern: it listens for input events using a stream, generates random numbers, and emits the results through another stream.

See also

Implementing a BLoC manually requires some boilerplate code. While this is useful to understand the main moving parts of a BLoC, you should be aware of the `flutter_bloc` package, available at https://pub.dev/packages/flutter_bloc. This package makes the integration of BLoCs in Flutter easier with the `Bloc` class. Even easier to manage are Cubits, also available with the same package.

The BLoC pattern is one of the possible state management patterns in Flutter. For a complete overview of your options, have a look at https://flutter.dev/docs/development/data-and-backend/state-mgmt/options.

This whole chapter has dealt with streams. For a recap of the main concepts of using streams in Dart, refer to the tutorial at https://dart.dev/tutorials/language/streams.

Summary

In this chapter, you saw how to use a `Stream` in a Flutter app, read and write data to streams, build reactive user interfaces based on `Streams`, and implement the BLoC state management pattern.

You used `StreamControllers` and their `Streams` and `Sinks`. You learned how to subscribe to stream events, allow multiple stream subscriptions, and use the `StreamBuilder` widget to create reactive user interfaces.

You saw how to handle data and errors generated by streams, ensuring your app can manage different outcomes gracefully.

Now you can use `Streams` and BLoCs in your Flutter applications; this can help you create more robust, maintainable, and scalable apps that respond seamlessly to changing data and user interactions.

Using Flutter Packages

Packages are certainly one of the greatest features that Flutter offers. Both the Flutter team and third-party developers add and maintain packages in the Flutter ecosystem daily. This makes building apps much faster and more reliable, as you can focus on the specific features of your app while leveraging classes and functions that have been created and tested by other developers.

Packages are published at https://pub.dev. This is the hub where you can go to search for packages and verify their platform compatibility (iOS, Android, web, and desktop), popularity, versions, and use cases. Chances are that you've already used `pub.dev` several times before reading this chapter.

Here is the current home of the `pub.dev` repository, with the search box at the center of the screen:



Figure 11.1: pub.dev home page

In this chapter, we will cover the following topics:

- Importing packages and dependencies

- Using dev packages
- Creating your own package (part 1)
- Creating your own package (part 2)
- Creating your own package (part 3)
- Adding Google Maps to your app
- Using location services
- Adding markers to a map

By the end of this chapter, you will be able to use packages, create and publish your own package, and integrate the Google Maps plugin into your app.

Technical requirements

To follow along with the recipes in this chapter, you should have the following software installed on your Windows, Mac, Linux, or ChromeOS device:

- The Flutter SDK
- The Android SDK when developing for Android
- macOS and Xcode when developing for iOS
- An emulator or simulator, or a connected mobile device enabled for debugging
- Your favorite code editor, with the Flutter/Dart extensions installed

You'll find the code for the recipes in this chapter on GitHub at https://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_11.

Importing packages and dependencies

This recipe shows you how to get packages and plugins from https://pub.dev and integrate them into your app's `pubspec.yaml` file.

Specifically, you will check the version and package name from `pub.dev`, import a package into the `pubspec.yaml` file in your project, download it, and use it in your classes.

By the end of this recipe, you will know how to import any package available in the `pub.dev` hub into your apps.

Getting ready

In this recipe, you will create a new project. There are no prerequisites to follow along.

How to do it...

When you install a package from `pub.dev`, the process is very simple. For this example, we will install the `http` package and connect to a Git repository:

1. Create a new Flutter project, called `plugins`.
2. Go to https://pub.dev.
3. In the search box, type `http`.
4. Click on the `http` package on the results page.
5. From the `http` package's home page, click on the **Installing** button. You should see a page similar to the picture below:

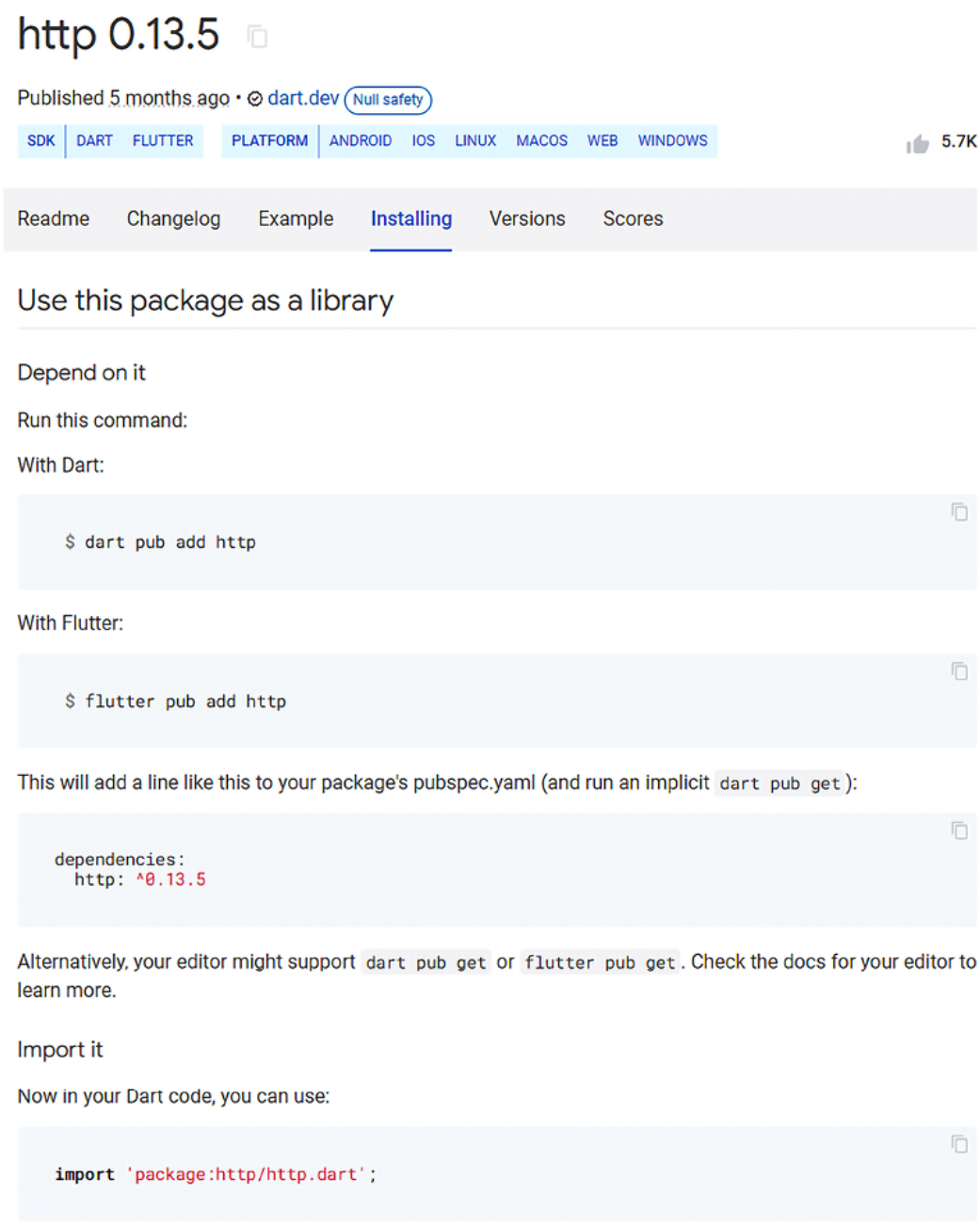


Figure 11.2: A package installing page

There are two ways to add a package to your apps: one is by using the terminal, and the other is by adding the dependency to your `pubspec.yaml` file:

1. If you want to use the Terminal, just type the Flutter command in the app's folder:

```
flutter pub add http
```

2. The advantage of using the Terminal is that you do not need to know the current package version; the latest version of the package will be automatically added to your project. As the command is always:

```
flutter pub add [package_name]
```

3. you only need to remember the name of the package in order to use this command.
4. If you want to add a package manually, copy the dependencies value: at this time it's:

```
http: ^0.13.5
```

5. Open the `pubspec.yaml` file in your project.
6. Paste the `http` dependency into the `dependencies` section of your `pubspec.yaml` file. Your dependencies should look like the following code (the `http` version number may be different by the time you read this, as updates are normal). Make sure that the alignment is precisely as shown here, where `http` is on the exact same level as `flutter`:

```
dependencies:
  flutter:
    sdk: flutter
  http: ^0.13.5
```

7. In most cases, the package will be automatically downloaded for you. To download the package manually, the actions depend on your system. In the **Terminal**, type `flutter pub get`. This will initiate downloading the package.
8. In **Visual Studio Code**, press the **Get Packages** button in the top-right corner of the screen, or, from the command palette, execute the `Pub: Get Packages` command, as shown here (you can also just save the `pubspec.yaml` file and wait a few seconds. The download should start automatically):

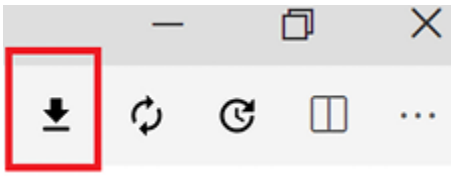


Figure 11.3: The Get Packages button in VS Code

9. In **Android Studio/IntelliJ IDEA**, click the **Packages get** button at the top right of the window, as shown in the following screenshot:

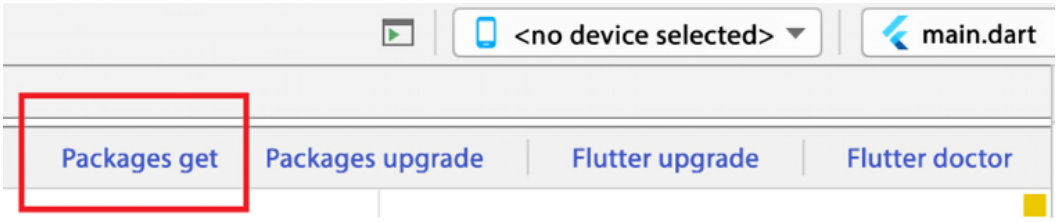


Figure 11.4: The Packages get button in Android Studio

How it works...

`pub.dev` is the main resource where packages and plugins are published. You can look for packages or publish them for other users. A few very important points to note are the following:

- All packages must include a `LICENCE` file
- Once published, packages cannot ever be removed
- All packages can only depend on other published packages (this means that if a package depends on other packages, those packages must be published on pub.dev)

All these rules benefit the end user. If you use a package from `pub.dev`, you always know that your packages will remain available.

The `pubspec.yaml` file is written in **YAML**, which is a language mainly used for configurations. It is a superset of **JSON**, with key-value pairs. The most important feature to understand in YAML is that it uses indentation for

nesting, so you need to pay attention to how you use indentation and spacing, as it may raise unexpected errors when used inappropriately.

The following command imports the `http` package into the project. It uses the caret (`^`) syntax for the version number:

```
http: ^0.13.5
```

Once you import the package, you can use its properties and methods anywhere in your project. The `http` package, for instance, allows you to connect to web services from your app, using the `http` or `https` protocols. Whenever you want to use a package that you have added to your project, you also need to import it into the file where you use the package with an `import` statement:

```
import 'package:http/http.dart';
```

In this case `http.dart` in the `http` package is the barrel file, which is a single file that re-exports parts of other files or entire files. You can consider this as the entry point for packages.

The three numbers of the version are denoted as **MAJOR.MINOR.PATCH**, where breaking changes happen only in major releases, after version 1.0.0. For versions before 1.0.0, breaking changes can happen at every minor release (but please take into account that breaking changes may happen unexpectedly at any release).

So, when you write `^0.13.5`, this means any version equal to or bigger than 0.13.5 and lower than 0.14.0.

When you write `^1.0.0`, this means any version equal to or bigger than 1.0.0 and lower than 2.0.0.

In some cases `BUILD` is added to **MAJOR.MINOR.PATCH**, following a + sign; for example, package 1.2.3+build.4.a5b6c7d. This may be useful to differentiate versions.

Pro Tip

Tip: When you import packages, you usually add the latest version of the package in the `pub.dev` repository. You should also update and solve dependency issues from time to time. There are tools that may help you save a lot of time in adding and updating dependencies. If you use Visual Studio Code, you can install the *Pubspec Assist* `plugin`, available at the following link: https://marketplace.visualstudio.com/items?itemName=jeroen-meijer.pubspec-assist.

If you use Android Studio or IntelliJ IDEA, you can add the *Flutter Enhancement* tools, available at https://plugins.jetbrains.com/plugin/12693-flutter-enhancement-suite. Both tools allow your dependencies to be added and updated easily, without leaving your editor.

After adding a package to the `dependencies` section of the `pubspec.yaml` file, you can download them manually from the terminal, with the `flutter pub get` command, or by pressing your editor's **Get** button. This step might not be necessary for VS Code and Android Studio, as both can be configured to automatically get the packages when you update the `pubspec.yaml` file.

See also

Choosing the best packages and plugins for your apps may not always be simple. That's why the Flutter team created the **Flutter Favorite** program, to help developers identify the packages and plugins that should be considered first when creating an app. For details about the program, refer to the following link: https://flutter.dev/docs/development/packages-and-plugins/favorites.

When using GitHub, a must-have tool for dependency management is called Dependabot; you'll find it at https://github.com/dependabot.

Using dev dependencies

In Flutter, `dev` dependencies are packages that you only need during development, and not in production. In this recipe you will see how to add and use a common Flutter linting package, unsurprisingly called `lint`.

Getting ready

In this recipe, you will use the project created in the previous one: *Importing packages and dependencies*.

How to do it...

In this recipe you will add the `lint` package to your Flutter project and configure it to use linting rules:

1. Add the latest version of the `lint` package as a `dev` dependency in your `pubspec.yaml`, and remove any other linting packages you may find there:

```
dev_dependencies:
  flutter_test:
    sdk: flutter
  lint: ^2.0.0
```

2. In the console, run `flutter pub get` to download the package.
3. If required, create a new file, called `analysis_options.yaml`, in the `root` of your project.
4. Alternatively, open `analysis_options.yaml`, and edit its content so that it only contains the instruction:

```
include: package:lint/strict.yaml
```

5. In the `main.dart` file, in the `build` method, declare a variable that will contain the screen for the home page of `MaterialApp`:

```
var screen = PackageScreen();
return MaterialApp(
```

```
    title: 'Packages App',
    theme: ThemeData(
      primarySwatch: Colors.blue,
    ),
    home: screen,
  );
```

6. In the **PROBLEMS** window in your editor, note the errors that show up, like the ones shown in this figure:

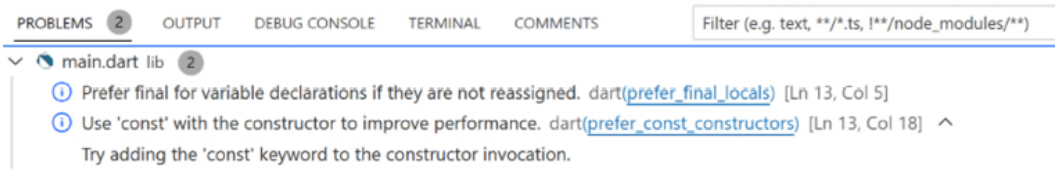


Figure 11.5: The PROBLEMS window in VS Code

7. Fix the screen declaration and note that the problems disappear:

```
const screen = PackageScreen();
```

How it works...

While dependencies are packages that are available both during development and at runtime, dev dependencies are only available during development and testing, and they can't be imported and used in the production app code.

In your `pubspec.yaml`, you declare your dev dependencies under `dev_dependencies`:

```
dev_dependencies:
  lint: ^2.0.0
```

You can use dev dependencies for tasks like linting, testing, or code generation. While regular dependencies are included in the final production app, adding to the app's size, dev dependencies are not included. This

allows you to use packages that help with development and testing without impacting the app's performance or size.

See also

The package we used in this recipe is a very common linting library; for further information on its use have a look at https://pub.dev/packages/lint.

Creating your own package (part 1)

While using packages made by other developers can really boost your app creation speed, sometimes you need to create your own packages. Some of the main reasons for creating a new package are as follows:

- Modularity
- Code reuse
- Encapsulation
- Low-level interaction with a specific environment

Packages help you write modular code, as you can include several files and dependencies in a single package, and use it in your app. At the same time, reusing code is made extremely simple, as packages can be shared among different apps. Also, when you make changes to a package, you only need to make them in one place, and they will automatically cascade to all the apps that point to that package.

There is a special type of package, called a **plugin**, that contains platform-specific implementations for iOS, Android, web, or desktop. You generally create a plugin when you need to interact with specific low-level features of a system. Examples include hardware, such as the camera, or software, such as the contacts in a smartphone.

This is the first recipe in a series of three that will show you how to create and publish a package on GitHub and `pub.dev`.

Getting ready

Create a new Flutter project with your favorite editor or use the project created in the previous recipe, *Importing packages and dependencies*.

How to do it...

In this recipe, you will create a simple Dart package that finds the area of a rectangle or a triangle:

1. In the root folder of your project, create a new folder, called `packages`.
2. Open a Terminal window, pointing to your project's root folder.
3. In the terminal, type `cd .\packages\`.
4. Type `flutter create --template=package area`.
5. In the `pubspec.yaml` file in the `package` folder of your app, add the dependency to the latest version of the `intl` package:

```
dependencies:
  flutter:
    sdk: flutter
  intl: ^0.18.0
```

6. In the `area.dart` file, in the `package/lib` folder of your app, delete the existing code, import the `intl` package, and write a method to calculate the area of a rectangle, as shown here:

```
library area;

import 'package:intl/intl.dart';

String calculateAreaRect(double width, double height) {
    double result = width * height;
    final formatter = NumberFormat('#.####');
    return formatter.format(result);
}
```

7. Under the `calculateAreaRect` method, write another method to calculate the area of a triangle, as shown here:

```
String calculateAreaTriangle(double width, double
height) {
    double result = width * height / 2;
```

```
    final formatter = NumberFormat('#.####');
    return formatter.format(result);
}
```

8. Remove the `test` folder in the `area package` folder.
9. In the `pubspec.yaml` file of the container project (not the package, the top-level project), add the dependency to the package you have just created (please make sure it's at the same level as the other dependencies):

```
area:
    path: packages/area
```

10. At the top of the `main.dart` file of the container project, import the `area` package:

```
import 'package:area/area.dart';
```

11. Remove the `MyHomePage` class created in the sample app.
12. Refactor the `MyApp` class, as shown here:

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const PackageScreen(),
    );
  }
}
```

13. Under the `MyApp` class, create a new stateful widget, called `PackageScreen`. You can use the `stful` shortcut to save some time. The final result is shown here:

```
class PackageScreen extends StatefulWidget {
  const PackageScreen({super.key});
  @override
  State<PackageScreen> createState() =>
_PackageScreenState();
}
class _PackageScreenState extends State<PackageScreen> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

14. In the `_PackageScreenState` class, create two `TextEditingController` widgets, for the width and height of the shape, and a `String` for the result:

```
final TextEditingController txtHeight =
TextEditingController();
final TextEditingController txtWidth =
TextEditingController();
String result = '';
```

15. Still in the `_PackageScreenState` class, override the `dispose` method:

```
@override
  void dispose() {
    txtHeight.dispose();
    txtWidth.dispose();
    super.dispose();
  }
```

16. At the bottom of the `main.dart` file, create a stateless widget that will take a `TextEditingController`, and a `String`, which will return a `TextField` with some padding around it that we will use for the user input:

```
class AppTextField extends StatelessWidget {
  final TextEditingController controller;
  final String label;
  const AppTextField(this.controller, this.label,
```

```
{super.key});
  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: const EdgeInsets.all(24),
      child: TextField(
        controller: controller,
        decoration: InputDecoration(hintText: label),
      ),
    );
  }
}
```

17. In the `build` method of the `_PackageScreenState` class, remove the existing code and return a `Scaffold`, containing an `appBar` and a `body`, as shown here:

```
return Scaffold(
    appBar: AppBar(
        title: const Text('Package App'),
      ),
      body: Column(
        children: [
          AppTextField(txtWidth, 'Width'),
          AppTextField(txtHeight, 'Height'),
          const Padding(
            padding: EdgeInsets.all(24),
          ),
          ElevatedButton(child: const Text('Calculate
Area'), onPressed: () {}),
          const Padding(
            padding: EdgeInsets.all(24),
          ),
          Text(result),
        ],
      ),
);
```

18. In the `onPressed` function in the `ElevatedButton`, add the code to call the `calculateAreaRect` method in the `area` package:

```
double width = double.tryParse(txtWidth.text) ?? 0;
double height = double.tryParse(txtHeight.text) ?? 0;
String res = calculateAreaRect(width, height);
setState(() {
```

```
    result = res;
});
```

19. Run the app.
20. Insert two valid numbers into the text fields on the screen and press the **Calculate Area** button. You should see the result in the text widget, as shown in the following screenshot:

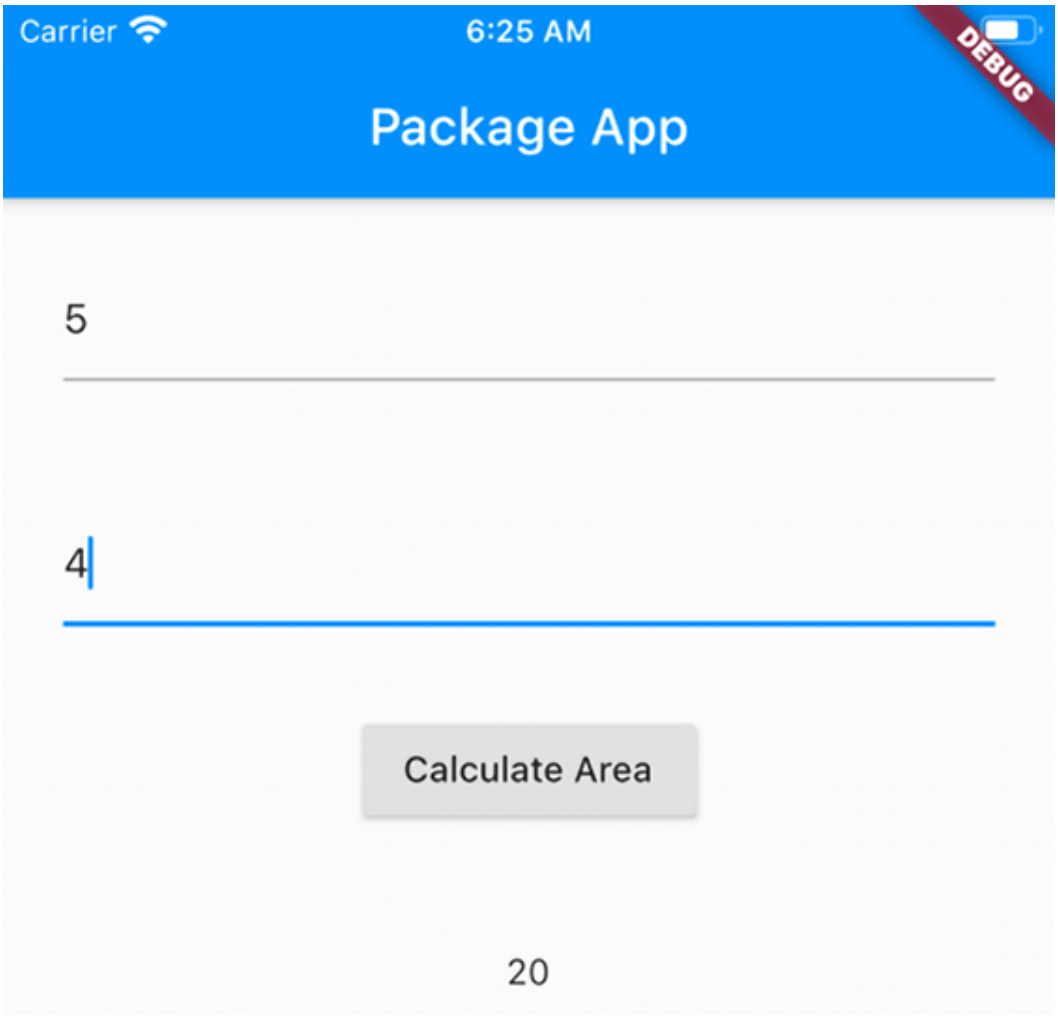


Figure 11.6: App using a custom package

How it works...

Packages enable the creation of modular code that can be shared easily. The simplest package should at least contain the following:

- A `pubspec.yaml` file, with the package name, version, and other metadata
- A `lib` folder with package code, when a package contains some code other than tests or just a `bin` folder

A package can contain more than a single file in the `lib` folder, but it should contain at least a single `dart` file with the name of the package — in our example, `area.dart`.

The `area.dart` file contains two methods, one to calculate the area of a rectangle, and the second to calculate the area of a triangle.

In the `pubspec.yaml` file, we import the `intl` package:

```
intl: ^0.18.0
```

One of the great features of packages is that when you add a dependency, and you depend on a package, you don't need to depend on that package from the main app. That means you do not have to import `intl` into the main package. This is called a transitive dependency.

A transitive dependency is a dependency that's not directly specified in your project's `pubspec.yaml` file but is required by another dependency in your project — in other words, a dependency of a dependency.

At the root of your project you can find the `pubspec.lock` file: this is automatically created and updated by the package manager. It manages your project's dependencies and keeps track of the versions of each package, including their transitive dependencies.

Please note the following command:

```
final formatter = NumberFormat('#.####');
```

This creates a `NumberFormat` instance. This will create a `String` from a number, with a limit of four decimals for the result. In the `pubspec.yaml` file of the main project, we add the dependency to the `area` package with the following lines:

```
area:
    path: packages/area
```

This is allowed when your package is stored locally, in the `packages` directory. Other common options include Git repositories and `pub.dev`.

In the `main.dart` file of the project, before using the package, you need to import it at the top of the file, as you would do for any other third-party package:

```
import 'package:area/area.dart';
```

The user interface is rather simple. It contains a `Column` with two text fields. In order to manage the content of the fields, we use two `TextEditingController` widgets, which are declared with the following commands:

```
final TextEditingController txtHeight =
TextEditingController();
final TextEditingController txtWidth =
TextEditingController();
```

Once the `TextEditingController` widgets are declared, we associate them with their `TextField` in the `AppTextField StatelessWidget`, passing the controller parameter with the commands highlighted here:

```
    @override
    Widget build(BuildContext context) {
        return Padding(
            padding: EdgeInsets.all(24),
            child: TextField(
            controller: controller,
            decoration: InputDecoration(hintText: label),
            ),
```

```
        );
    }
```

Finally, we use the package by calling the `calculateAreaRect` method, which is immediately available in the main project after importing the `area.dart` package:

```
String res = calculateAreaRect(width, height);
```

See also

For more information regarding the difference between packages and plugins, and a full guide to creating both, refer to https://flutter.dev/docs/development/packages-and-plugins/developing-packages.

Creating your own package (part 2)

The previous recipe works when your package is contained within your project. In this second part of the *Creating your own package* recipe, you will see how to create a package made of multiple files and depend on a Git repository in the main project.

Getting ready

You should have completed the previous recipe, *Creating your own package (part 1),* before following this one.

How to do it...

For this recipe, first, we will separate the functions we created in the `area.dart` file into two separate files, using the `part` and `part of` keywords. Then, for the dependency, we will use a Git repository instead of a package inside the project's folder:

1. In your **package**'s `lib` folder, create a new file, called `rectangle.dart`.
2. Still there, create another file, called `triangle.dart`.
3. In the `rectangle.dart` file, at the top of the file, specify that this is part of the `area` package:

```
part of area;
```

4. Under the `part of` statement, paste the method to calculate the area of the rectangle and remove it from `area.dart`. You will see an error on the `NumberFormat` method; this is expected, and we will solve this shortly:

```
String calculateAreaRect(double width, double height) {
    double result = width * height;
    final formatter = NumberFormat('#.####');
    return formatter.format(result);
}
```

5. Repeat the process for the `triangle.dart` file. The code for the `triangle.dart` file is shown here:

```
part of area;

String calculateAreaTriangle(double width, double
height) {
    double result = width * height / 2;
    final formatter = NumberFormat('#.####');
    return formatter.format(result);
}
```

6. In the `area.dart` file, after removing the existing methods, add two `part` statements. This will solve the errors in `triangle.dart` and `rectangle.dart`. The full code for the `area.dart` file is shown here:

```
library area;

import 'package:intl/intl.dart';

part 'rectangle.dart';
part 'triangle.dart';
```

7. The package can now be uploaded to any Git repository.
8. Once the package is published, you can update the `pubspec.yaml` file of the main project with the Git address. Remove the following dependency from the `pubspec.yaml` file of the main project:

```
area:
    path: packages/area
```

9. Add your own Git URL, or use the following Git address in the dependencies:

```
area:
    git: https://github.com/simoales/area_ns.git
```

How it works...

In this recipe, you have seen how to create a package made of multiple files and depend on it using a Git repository.

The `part` and `part of` keywords allow a library to be split into several files. In the main file (`area.dart` in this example), you specify all the other files that make the library, using the `part` statement. Note the commands:

```
part 'rectangle.dart';
part 'triangle.dart';
```

The preceding commands mean that the `triangle.dart` and `rectangle.dart` files are parts of the `area` library. This is also where you put all the `import` statements, which are visible in each linked file. In the linked files, you also add the `part of` statement:

```
part of area;
```

This means that each file is a part of the `area` package.

> **Note**
>
> It's now generally recommended that you prefer small libraries (also called mini-packages) that avoid using the `part/part of` commands, when possible. Still, knowing that you can separate complex code into several files can be useful in certain circumstances.

The other key part of this recipe was the dependency in the `pubspec.yaml` file:

```
area:
    git: https://github.com/simoales/area_ns.git
```

This syntax allows packages to be added from a Git repository. This is useful for packages that you want to keep private within your team, or if you want to depend on a package before it gets published to `pub.dev`. Being able to depend on a package available in a Git repository allows you to simply share packages between projects and teams, without necessarily making them public.

To specify a specific branch or commit of a Git repository, you can use the `ref` property. This makes the dependency less error-prone, as it ensures you are using a stable version of the dependency rather than the latest commit on the default branch.

In our example, you can add a `ref` property to the area dependency this way:

```
area:
  git:
    url: https://github.com/simoales/area_ns.git
    ref: some_commit_sha_or_tag_or_branch
```

See also

There is a list of guidelines when creating libraries comprising multiple files in Dart. It is available at the following link: https://dart.dev/guides/language/effective-dart/usage.

Another option to use packages without making them public is using hosted packages; see this link for further information: https://dart.dev/tools/pub/custom-package-repositories.

Creating your own package (part 3)

If you want to contribute to the Flutter community, you can share your packages in the `pub.dev` repository. In this recipe, you will see the steps required in order to achieve this.

Getting ready

You should have completed the previous recipes, *Creating your own package (part 1)* and *Creating your own package (part 2)*.

How to do it...

Let's look at the steps to publish a package to `pub.dev`:

1. In a terminal window, move to the `area` directory:

```
cd packages/area
```

2. Run the `flutter pub publish --dry-run` command. This will give you some information about the changes required before publishing.
3. Copy the BSD license, available at the following link: https://opensource.org/licenses/BSD-3-Clause.

> **Note**
>
> BSD licenses are open source licenses that allow almost any legitimate use of the software, cover the author from any liability, and only add minimal restrictions on the use and distribution of the software, both for private and commercial reasons.

4. Open the `LICENSE` file in the `area` directory.

5. Paste the BSD license into the `LICENCE` file.
6. In the first line of the license, add the current year and your name, or the name of your entity:

```
Copyright 2023 Your Name Here
```

7. Open the `README.md` file.
8. In the `README.md` file, remove the existing code and add the following:

```
# area
A package to calculate the area of a rectangle or a
triangle
## Getting Started
This project is a sample to show how to create and
publish packages
from a local directory, then a git repo, and finally
pub.dev
You can view some documentation at the link:
[online documentation](https://youraddress.com), that
contains samples and a getting started guide.
Open the CHANGELOG.md file, and add the content below:
## [0.0.1]
* First published version
```

9. In the `pubspec.yaml` file in your project, remove the author key (if available) and add the home page of the package. This may also be the address of the Git repository. The final result of the first four lines in the `pubspec.yaml` file should look similar to the following example:

```
name: area
 description: The area Flutter package.
 version: 0.0.1
 homepage: https://github.com/simoales/area_ns
```

10. Run the `flutter pub publish --dry-run` command again and check that there are no more warnings.

> **Note**
>
> Before running the command that follows, make sure your package adds real value to the Flutter community. *This specific package is definitely not a good candidate!*

11. Run the `flutter pub publish` command to upload your package to the `pub.dev` public repository.

How it works...

The `flutter pub publish --dry-run` command does not publish the package. It just tells you which files will be published and whether there are warnings or errors. This is a good starting point when you decide to publish a package to `pub.dev`.

A package published in `pub.dev` must contain a license. The Flutter team recommends the BSD license, which basically grants all kinds of use without attribution and releases the author from any liability. It's so short that you can actually read it (which is a miracle in itself).

The `LICENSE` file in the package project is where the text of the license is placed.

Another extremely important file for your packages is the `README.md` file. This is the main content that users will see on your package home page. It uses the **Markdown** format, which is a markup language that you can use to format plain text documents.

In the example shown above, we used three formatting options:

- `# area`: The single # is a level 1 heading (the `H1` in HTML).
- `## Getting Started`: The double ## is a level 2 heading (the `H2` in HTML).

- `[online documentation](https://youraddress.com)`: This creates a link to the URL in parentheses (*youraddress.com*) and shows the text in square brackets.

The `CHANGELOG.md` file is not required but highly recommended. It is also a Markdown file, which should contain all the changes you make when updating your package. If you add it to the package, it will show as a tab on your package's page on the `pub.dev` site.

Before finally publishing it, you should also upgrade the `pubspec.yaml` file, which should include the package name and description, a version number, and the home page of the package, which often is the GitHub repository:

```
name: area
description: The area Flutter package.
version: 0.0.1
homepage: https://github.com/simoales/area
```

The `flutter pub publish` terminal command publishes your package to `pub.dev`. Please note that once published, other developers can depend on it, so you will be unable to remove your package from the repository. Of course, you will be able to upload updates.

See also

Publishing a package is not just about running the `flutter pub publish` command. There are several rules, a scoring system, and recommendations to create a high-quality, successful package. For more information, see https://pub.dev/help/publishing.

Adding Google Maps to your app

Most packages and plugins don't require any configuration other than typing `flutter pub add` in a Terminal window. But a few plugins require additional steps, like getting API keys or adding platform-specific

configurations. One of these plugins is also one of the most useful: the Google Maps plugin. This recipe shows you how to add it to a Flutter app.

Specifically, you will see how to get a Google Maps API key, how to add Google Maps to your Android and iOS project, and how to show a map on the screen.

By the end of this recipe, you'll know how to integrate Google Maps into your projects.

Getting ready

In this recipe, you will create a new project.

How to do it...

In this recipe, you will add Google Maps to your app. Follow the instructions below:

1. Create a new Flutter app, and call it `map_recipe`.
2. Add the Google Maps package dependency to the project's `pubspec.yaml` file. The name of the package is `google_maps_flutter`. So just type the following,

   ```
   flutter pub add google_maps_flutter
   ```

   and the package will be added to your `pubspec.yaml`:

   ```
   dependencies:
       google_maps_flutter: ^2.2.3
   ```

3. In order to use Google Maps, you need to obtain an **API key**. You can get one from the **Google Cloud Platform** (**GCP**) console at the following link: https://cloud.google.com/maps-platform/.

4. Once you enter the console with a Google account, you should see a screen similar to the one shown here:

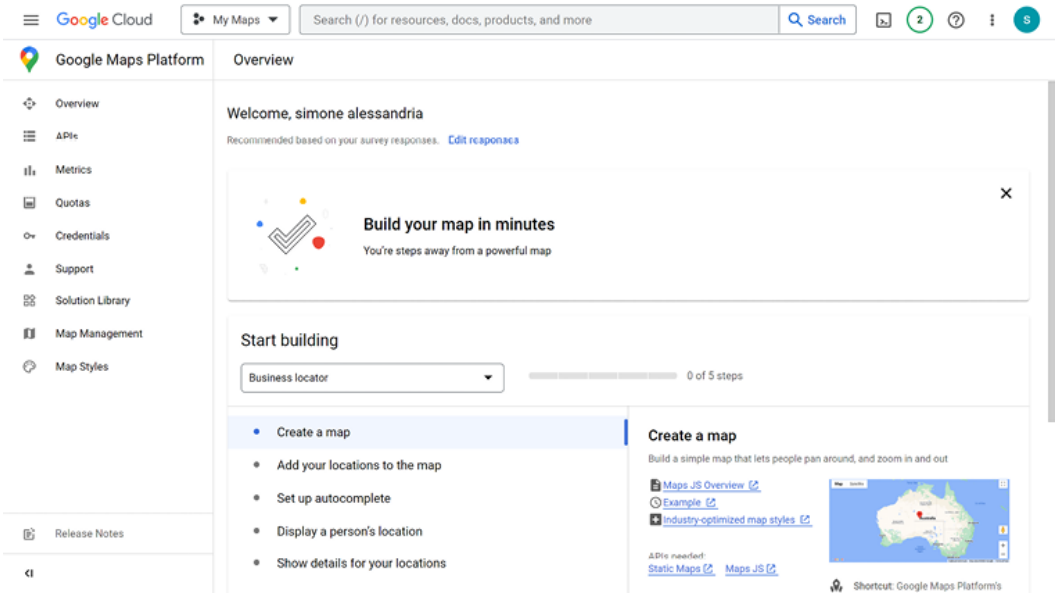


Figure 11.7: Google Maps overview page

5. Every API key belongs to a project. Create a new project called `maps-recipe`, leaving `No Organization` for your location, and then click the **Create** button.
6. On the credentials page, you can create new credentials. Click on the **Create credentials** button and choose **API Key**. You will generally want to restrict the use of your keys, but this won't be necessary for this test project.
7. Once your key has been created, copy the key to the clipboard. You can retrieve your key from the `Credentials` page later on.
8. On iOS and Android, you should also enable the Maps SDK for your target system, from the API page. The end result is shown in the following screenshot:

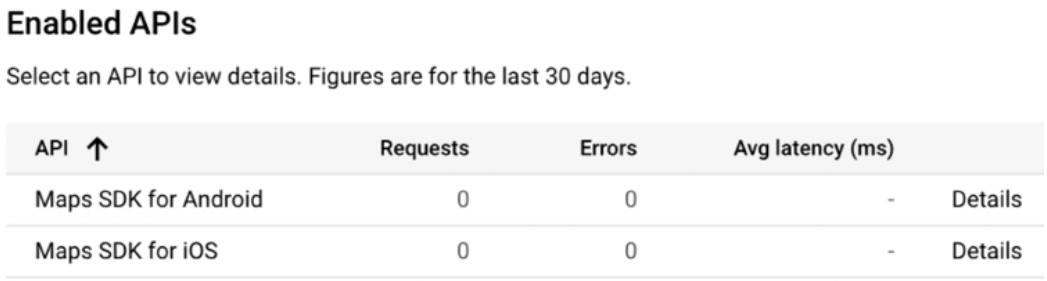


Figure 11.8: Maps enabled APIs

The following steps vary based on the platform you are using.

Adding Google Maps on Android

1. Open the `android/app/src/main/AndroidManifest.xml` file in your project.
2. Add the following line under the icon launcher icon, in the application node:

```
android:icon="@mipmap/ic_launcher">
 <meta-data
android:name="com.google.android.geo.API_KEY"
 android:value="[PUT YOUR KEY HERE]"/>
```

3. Set the `minSdkVersion` in the `android/app/build.gradle` file:

```
android {
   defaultConfig {
       minSdkVersion 20
   }
}
```

Adding Google Maps on iOS

1. Open the `AppDelegate` file, which you will find at `ios/Runner/AppDelegate.swift`.

2. At the top of the `AppDelegate.swift` file, import `GoogleMaps`, as follows:

```
import UIKit
import Flutter
import GoogleMaps
```

3. Add the API key to the `AppDelegate` class, as follows:

```
@objc class AppDelegate: FlutterAppDelegate {
override func application(
_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?
) -> Bool {
GMSServices.provideAPIKey("YOUR API KEY HERE")
GeneratedPluginRegistrant.register(with: self)
return super.application(application,
didFinishLaunchingWithOptions: launchOptions)
}
```

To show a map on the screen, perform the following steps:

1. At the top of the `main.dart` file, import the **Google Maps for Flutter** package:

```
import
'package:google_maps_flutter/google_maps_flutter.dart';
```

2. Remove the `MyHomePage` class from the file.
3. Create a new stateful widget using the `stful` shortcut, and call the class `MyMap`:

```
class MyMap extends StatefulWidget {
  const MyMap({super.key});
  @override
  State<MyMap> createState() => _MyMapState();
}
class _MyMapState extends State<MyMap> {
  @override
  Widget build(BuildContext context) {
    return Container();
```

```
  }
}
```

4. In the `MyApp` class, remove the comments and change the title and the `home` of `MaterialApp` as follows:

```
class MyApp extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            title: 'Map Demo',
            theme: ThemeData(
                primarySwatch: Colors.blue,
            ),
            home: const MyMap(),
        );
    }
}
```

5. In the `body` of a new `Scaffold` in the `_MyMapState` class, add a `GoogleMap` object, passing the `initialCameraPosition` parameter as follows:

```
class _MyMapState extends State<MyMap> {
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(title: Text('Google
Maps'),),
            body: GoogleMap(
                initialCameraPosition: CameraPosition(
                target: LatLng(51.5285582, -0.24167),
                zoom: 12)
            ,),
        );
    }
}
```

6. Run the app. You should see a map covering the screen, showing part of London:


9:04
DEBUG
Google Maps
CANONS PARK
EALD
BURNT OAK
M1
FINCHLE
COLINDALE
KINGSBURY
WEST HENDON
A406
HAMPST
GARDEN S
BRENT CROSS
CHILDS HILL
NEASDEN
Wembley Stadium
CRICKLEWOOD
BRENT PARK
Wembley
WILLESDEN
lorsenden
ALPERTON
HARLESDEN
Perivale
A40
A40
PARK ROYAL
MAID
NORTH ACTON
Westway
NOTTING HILL
EALING
WHITE CITY
ACTON
SHEPHERD'S
BUSH
SOUTH ACTON
A3220
M4
HAMMERSMITH
CHISWICK
Brentford
Royal Botanic
Gardens, Kew
FULHAM
eworth
MORTLAKE
A316
A205
Richmond
PUTNEY

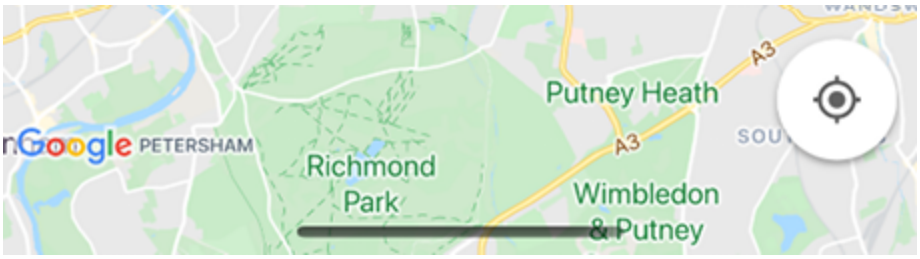


Figure 11.9: Maps showing on a mobile device

How it works...

In this recipe, you have added Google Maps to your app. This requires three steps:

1. Getting the Google Maps credentials and enabling them
2. Configuring your app with the credentials
3. Using Google Maps in your project

The credentials are free, and you can use Google Maps for free as well, up to a certain threshold. This should be enough for all development purposes and more, but if you want to publish your app, you should take into account the price to use the API.

> **Note**
>
> For more details about pricing and thresholds in Google Maps, visit the following link: https://cloud.google.com/maps-platform/pricing/.

Configuring the app depends on the system you are using. For Android, you need to add the Google Maps information to the Android app manifest. This file contains essential information about your app, including the package name, the permissions that the app needs, and the system requirements.

On iOS, you use the `AppDelegate.swift` file, the root object of any iOS app, which manages the app's shared behaviors. For iOS projects, you also

need to opt into the preview of the embedded view, which can be done in the app's `Info.plist` file.

As you saw, showing a map is rather simple. The object you use is `GoogleMap`. The only required parameter is `initialCameraPosition`, which takes a `CameraPosition` object. This is the center of the map and requires a target, which in turn takes a `LatLng` object to express the position on the map. This includes two coordinates expressed in decimal numbers, one for latitude and one for longitude. Optionally, you can also specify a zoom level: the bigger the number, the higher the scale of the map. Google uses latitude and longitude to position a map and to place markers on it.

When the map is shown on the screen, users can zoom in and out, and move the center of the map in the four cardinal directions.

In the next recipe, you will see how to position the map dynamically, based on the position of the user.

See also

While Google Maps is certainly an awesome product, you might want to use other map services, such as Bing or Apple. You can actually choose your favorite map provider with Flutter. One of the platform-independent plugins currently available is the `flutter_map` plugin, available at https://pub.dev/packages/flutter_map.

Using location services

In the previous recipe, you learned how to show and position a map using Google Maps, with fixed coordinates. In this recipe, you will find the current position of a user so that the map will change based on the user's position.

Specifically, you will add the `location` package to your project, retrieve the coordinates of the device's position, and set the map's position to the coordinates you have retrieved.

By the end of this recipe, you will understand how to leverage the user's location in your apps.

Getting ready

You should have completed the previous recipe, *Adding Google Maps to your app*, before following this one.

How to do it...

There's a Flutter package called `location` that you can use to access the platform-specific location services:

1. Add the latest version of the `location` package in the dependencies to the `pubspec.yaml` file:

```
location: ^4.4.0
```

2. On Android, add permission to access the user's location. In the Android manifest file, which you can find at `android/app/src/main/AndroidManifest.xml`, add the following node to the `Manifest` node (before the `application` node):

```
<uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION"
/>
```

3. In the `main.dart` file, import the `location` package:

```
import 'package:location/location.dart';
```

4. In the `_MyMapState` class, add a `LatLng` variable at the top of the class:

```
late LatLng userPosition;
```

5. Still in the `_MyMapState` class, add a new method at the bottom of the class, containing the code to retrieve the current user's location:

```
Future<LatLng> findUserLocation() async {
    Location location = Location();
    LocationData userLocation;
    PermissionStatus hasPermission = await
location.hasPermission();
    bool active = await location.serviceEnabled();
    if (hasPermission == PermissionStatus.granted &&
active) {
      userLocation = await location.getLocation();
      userPosition = LatLng(userLocation.latitude!,
userLocation.longitude!);
    } else {
      userPosition = const LatLng(51.5285582, -0.24167);
    }
    return userPosition;}
```

6. At the top of the `_MyMapState` class, declare a `Future` for the user position:

```
Future<LatLng> _userLocationFuture;
```

7. In the `initState` method of the `_MyMapState` class, call the `findUserLocation()` method:

```
_userLocationFuture = findUserLocation();
```

8. In the `build` method of the `_MyMapState` class, enclose `GoogleMap` in a `FutureBuilder` object whose `future` property calls the `findUserLocation` method:

```
body: FutureBuilder(
    future: _userLocationFuture,
    builder: (BuildContext context, AsyncSnapshot
snapshot) {
        if (snapshot.hasData) {
            return GoogleMap(
             initialCameraPosition:
 CameraPosition(target: snapshot.data, zoom: 12),
                );
              } else {
```

```
                    return Container();
                  }
            },
),
```

9. Run the app. Now you should see your position on the map, or an emulator, and the position set on the emulator itself.

How it works...

In order to find the current location of our user, you created an `async` method called `findUserLocation`. This method leverages the device's GPS to find the latitude and longitude of the user's current location (if available) and returns it to the caller. This method is then used to set the `future` property of the `FutureBuilder` object in the user interface.

Before trying to retrieve the user's location, there are two important steps. You should always check whether location services are activated and that the user has been granted permission to retrieve their location. In the example in this recipe, you used the following command:

```
PermissionStatus hasPermission = await
location.hasPermission();
```

Later, you added the following:

```
bool active = await location.serviceEnabled();
```

The `hasPermission` method returns a `PermissionStatus` value, which includes the state of the location permission. The `serviceEnabled` method returns a Boolean, which is `true` when location services are enabled. Both are prerequisites before trying to ascertain the device's location.

The `getLocation` method returns a `LocationData` object. This not only contains `latitude` and `longitude` but also `altitude` and `speed`, which we didn't use in this recipe but might be useful for other apps.

In the `build` method, we use a `FutureBuilder` object to automatically set `initialPosition` when it becomes available.

In the next recipe, we will also add markers to our map.

See also

As you saw in this recipe, you need specific permissions to use location services. In order to deal better with all the permission requirements of an app, there is a package called `permission_handler`. See https://pub.dev/packages/permission_handler to learn more.

Adding markers to a map

In this recipe, you will see how to make a query to the Google Maps Places service and add markers to the map in the app. Specifically, you will search for all restaurants near the user's location within a radius of 1,000 meters.

By the end of this recipe, you will know how to query the huge *Google Places* archive and point to any place in your maps with a marker.

Getting ready

You should have completed two previous recipes from this chapter, *Adding Google Maps to your app* and *Using location services*, before following this one.

How to do it...

To add markers to the map in your project, perform the following steps:

1. Navigate to the Google Maps API console and enable the *Places* API for your app. Make sure that your `Flutter Maps` project is selected, and then click the **Enable** button.

2. Import the `http` package to your `pubspec.yaml`.
3. At the top of the `main.dart` file, add two new imports, one for `http` and another for the `dart:convert` package, as shown here:

```
import 'package:http/http.dart' as http;
import 'dart:convert';
```

4. At the top of the `_MyMapState` class, add a new `List` of `Marker` objects:

```
class _MyMapState extends State<MyMap> {
late LatLng userPosition;
List<Marker> markers = [];
```

5. In the `AppBar` contained in the `build` method of the `_MyMapState` class, add the `actions` property, containing an `IconButton`, which, when pressed, calls a `findPlaces` method that we will create in the next steps:

```
@override
Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
            title: Text('Google Maps'),
            actions: [
                IconButton(
                    icon: const Icon(Icons.map),
                    onPressed: () => findPlaces(),
                )
            ],
        ),
```

6. In the `GoogleMap` object, add the `markers` argument, which will take a `Set` of `Marker` objects taken from the `markers` list:

```
return GoogleMap(
    initialCameraPosition:
        CameraPosition(target: snapshot.data, zoom:
12),
    markers: Set<Marker>.of(markers),),),
);
```

7. At the bottom of the `_MyMapState` class, create a new asynchronous method, called `findPlaces`:

```
Future findPlaces() async {}
```

8. Inside the `findPlaces` method, add your Google Maps key and the base URL of the query:

```
const key = '[Your Key Here]';
const placesUrl =
'https://maps.googleapis.com/maps/api/place/nearbysearch
/json?';
```

9. Under the declarations, add the dynamic part of the URL:

```
String url = placesUrl +
'key=$key&type=restaurant&location=${userPosition.latitu
de},${userPosition.longitude}' + '&radius=1000';
```

10. Next, make an `http get` call to the generated URL. If the response is valid, call a `showMarkers` method, which we will create next, passing the retrieved data; otherwise, throw an `Exception`. The code is shown here:

```
final response = await http.get(Uri.parse(url));
    if (response.statusCode == HttpStatus.OK) {
      final data = json.decode(response.body);
      showMarkers(data);
    } else {
      throw Exception('Unable to retrieve places');
    }
}
```

11. Create a new method, called `showMarkers`, that takes a `data` parameter:

```
showMarkers(data) {}
```

12. Inside the method, create a `List`, called `places`, that reads the `results` node of the `data` object that was passed, and clears the

`markers` `List`:

```
List places = data['results'];
markers.clear();
```

13. Create a `for … in` loop over the result list, which adds a new marker for each item of the list. For each marker, set `markerId`, `position`, and `infoWindow`, as shown here:

```
for (var place in places) {
      markers.add(Marker(
          markerId: MarkerId(place['reference']),
          position: LatLng(place['geometry']['location']
['lat'],
              place['geometry']['location']['lng']),
          infoWindow:
              InfoWindow(title: place['name'], snippet:
place['vicinity'])));
};
```

14. Finally, update the `State` setting the markers:

```
setState(() {
    markers = markers;
});
```

15. Run the app.
16. Press the markers button in the **AppBar**. You should see a list of markers on the map, with all the restaurants near your location.
17. Tap on one of the markers; an info window should appear, containing the name and address of the restaurant.

How it works...

The Google Places API contains over 150 million points of interest that you can add to your maps. Once you activate the service, you can then make search queries using `get` calls with the `http` class. The base address to make queries based on your position is the following:

https://maps.googleapis.com/maps/api/place/nearbysearch/json

In the last part of the address, you can specify `json` or `xml`, based on the format you wish to receive. Before that, note that `nearbysearch` makes queries for places near a specified location. When using the nearby search, there are three required parameters and several optional ones. You must separate each parameter with the ampersand (&) character.

The required parameters are as follows:

- `key`: Your API key.
- `location`: The position around which you want to get the places. This requires latitude and longitude.
- `radius`: The radius, expressed in meters, within which you want to retrieve the results.

In our example, we have also used an optional parameter, called `type`.

`type` filters the results so that only places matching the specified type are returned. In this recipe, we used `restaurant`. Other types include `café`, `church`, `mosque`, `museum`, and `school`. For a full list of supported types, have a look at https://developers.google.com/places/web-service/supported_types.

An example of the final address of the URL is as follows:

```
https://maps.googleapis.com/maps/api/place/nearbysearch/json?
key=[YOUR KEY
HERE]&type=restaurant&location=41.8999983,12.49639830000001&rad
ius=10000
```

Once you've built the query, you need to call the web service with the `http.get` method. If the call is successful, it returns a response containing the JSON information of the places that were found. A partial selection of the contents is shown here:

```
"results": [
  {
    "geometry" : {
      "location" : {
        "lat" : 41.8998425,
        "lng" : 12.499711
      },
  },
  "name" : "UNAHOTELS Decò",
  "place_id" : "ChIJk6d0a6RhLxMRVH_wYTNrTDQ",
  "reference" : "ChIJk6d0a6RhLxMRVH_wYTNrTDQ",
  "types" : [ "lodging", "restaurant", "food",
"point_of_interest",
  "establishment" ],
  "vicinity" : "Via Giovanni Amendola, 57, Roma"
},
```

You can use markers to pin places on a map. Markers have a `markerId`, which uniquely identifies the place; a `position`, which takes a `LatLng` object; and an optional `infoWindow`, which shows some information about the place when users tap or click on `Marker`. In our example, we've shown the name of the place and its address (called `vicinity` in the API).

In this recipe, the `showMarkers` method adds a new `Marker` for each of the retrieved places, using the `forEach` method over the `places List`.

In the `GoogleMaps` object, the `markers` parameter is used to add the markers to the map.

There's more...

For more information about searching places with Google Maps, have a look at https://developers.google.com/places/web-service/search.

Summary

In this chapter, you've learned how to import packages into your project, leveraging community-developed tools and saving you time and effort.

You've also delved into the specifics of using development packages, which allow you to access features that are useful during development but not required in production.

Throughout the chapter, you've seen how to create your own package. First, you set up the basic structure of your package and learned about the necessary files and directories. Then you implemented the core functionality of your package, and finally, you saw how to share it with others.

You saw how to integrate Google Maps into your apps, which allows you to display interactive maps. You used the location services and learned how to add markers to a map, giving your users the ability to visualize and interact with specific points of interest.

Adding Animations to Your App

Animations are important: they can significantly improve your app's user experience by providing a changing user interface. This can range from adding and removing items from a list to fading in elements when you need to draw a user's attention to them. The Animation API in Flutter is powerful, and in this chapter, you'll learn how to create effective animations and add them to your projects. This will allow you to create visually pleasing experiences for your users.

In this chapter, we will cover the following recipes:

- Creating basic container animations
- Designing animations part 1 — Using the `AnimationController`
- Designing animations part 2 — Adding multiple animations
- Designing animations part 3 — Using curves
- Optimizing animations
- Using Hero animations
- Using premade animation transitions
- Using the `AnimatedList` widget
- Implementing swiping with the `Dismissible` widget
- Using the Flutter `animations` package

By the end of this chapter, you will be able to insert several different types of animations into your apps.

Creating basic container animations

In this recipe, you will place a square in the middle of the screen. When you click the `IconButton` in the `AppBar`, three animations will take place at the same time. The square will change color, size, and its top margin:



Figure 12.1: Example of AnimatedContainer

After following this recipe, you will understand how to work with the `AnimatedContainer` widget, one of the **implicit animations**, which allows the creation of simple animations with a `Container`.

Getting ready

In order to follow this recipe, create a new Flutter app, and call it `my_animations`.

How to do it...

In the following steps, you will create a new screen that shows an animation with an `AnimatedContainer` widget:

1. Remove the `MyHomePage` class created in the sample app.
2. Refactor the `MyApp` class as shown here:

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
        return MaterialApp(
      title: 'Animations Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
     home: MyAnimation(),
   );
  }
}
```

3. Create a file called `my_animation.dart` and add a new stateful widget called `MyAnimation`, using the `stful` shortcut. The result is shown here:

```
import 'package:flutter/material.dart';
class MyAnimation extends StatefulWidget {
  const MyAnimation({super.key});
  @override
  State<MyAnimation> createState() =>
_MyAnimationState();
}
class _MyAnimationState extends State<MyAnimation> {
```

```
  @override
  Widget build(BuildContext context) {
    return Placeholder();
  }
}
```

4. Remove the container in the `_MyAnimationState` class and insert a `Scaffold` instead. In the `appBar` parameter of the `Scaffold`, add an `AppBar`, with a title of `Animated Container`, and empty `actions`. In the body of the `Scaffold`, add a `Center` widget. The result is shown here:

```
Widget build(BuildContext context) {
  return Scaffold(
     appBar: AppBar(
       title: const Text('Animated Container'),
       actions: []
     ),
    body: Center());
```

5. At the top of the `_MyAnimationState` class, add a `List` of colors:

```
final List<Color> colors = const [
  Colors.red,
  Colors.green,
  Colors.yellow,
  Colors.blue,
  Colors.orange
];
```

6. Under the `List`, add an integer state variable called `iteration`, which will start with a value of `0`:

```
int iteration = 0;
```

7. In the body of the `Scaffold`, in the `Center` widget, add an `AnimatedContainer`. The `AnimatedContainer` has a `width` and `height` of 100, a `Duration` of 1 second, and a `color` with a value of `colors[iteration]`:

```
body: Center(
  child: AnimatedContainer(
```

```
    width: 100,
    height: 100,
    duration: const Duration(seconds: 1),
    color: colors[iteration],
)));
```

8. In the `actions` property of the `AppBar`, add an `IconButton`, whose icon is the `run_circle` icon.
9. In the `onPressed` parameter of the `IconButton`, if the value of `iteration` is less than the length of the `colors` list, increment `iteration` by 1; otherwise, reset it to 0.
10. Then, call the `setState` method, setting the new `iteration` value:

```
actions: [
  IconButton(
    icon: const Icon(Icons.run_circle),
    onPressed: () {
      setState(() {
        iteration = (iteration + 1) % colors.length;
    });
   },
)],
```

11. Make sure you add the import to `my_animation.dart` in `main.dart` and then run the app.
12. Click the icon a few times and you should see that the square changes color each time. The change is automatically interpolated so that the transition is smooth.
13. Add two new lists to the top of the `_MyAnimationState` class:

```
final List<double> sizes = [100, 125, 150, 175, 200];
final List<double> tops = [0, 50, 100, 150, 200];
```

14. Edit the `AnimatedController` widget, with the code shown here:

```
child: AnimatedContainer(
  duration: Duration(seconds: 1),
  color: colors[iteration],
  width: sizes[iteration],
 height: sizes[iteration],
```

```
 margin: EdgeInsets.only(top: tops[iteration]),
)
```

15. Run the app again. Click the `IconButton` a few times and view the results, and notice the changes to the color, size, and margin properties of the square.

How it works...

An `AnimatedContainer` is an animated version of a `container` widget that changes its properties over a specific period of time. In this recipe, you created a transition animation that changes the color, width, height, and margin of a widget.

When you use an `AnimatedContainer`, the `duration` is required. Here is the instruction:

```
AnimatedContainer(
  duration: const Duration(seconds: 1),
```

This specifies that each time a property of the `AnimatedContainer` changes, the transition between the old value and the new one will take 1 second.

Next, you set the property or properties of the `AnimatedContainer` that should change over your specified duration.

> **Note**
>
> Not all the properties of an `AnimatedContainer` must change, but if they change, they will change together during the time specified in the `duration` property.

In our project, we used four properties, `color`, `width`, `height`, and `margin`, with the following code:

```
color: colors[iteration],
width: sizes[iteration],
```

```
height: sizes[iteration],
margin: EdgeInsets.only(top: tops[iteration])
```

The animation runs when the user clicks the `IconButton` in the `AppBar`, in the `onPressed` callback. In order to animate the container, you only need to change the state of the widget. You performed this action by changing the value of `iteration`:

```
onPressed: () {
  iteration < 4 ? iteration++ : iteration = 0;
  setState(() {
    iteration = iteration;
  });
},
```

See also

The official documentation for the `AnimatedContainer` class contains a demo video and several tips on how to use the `AnimatedContainer`. You will find it at https://api.flutter.dev/flutter/widgets/AnimatedContainer-class.html.

Designing animations part 1 — Using the AnimationController

In this recipe, you will perform the first step of making your widgets animatable, by conforming to a ticker `Mixin` and initializing an `AnimationController`. You will also add the appropriate listeners to make sure the `build` function reruns at every tick.

You will build an animation that moves a ball diagonally starting from the top of the screen, then stopping at an ending position as shown in *Figure 12.2*:

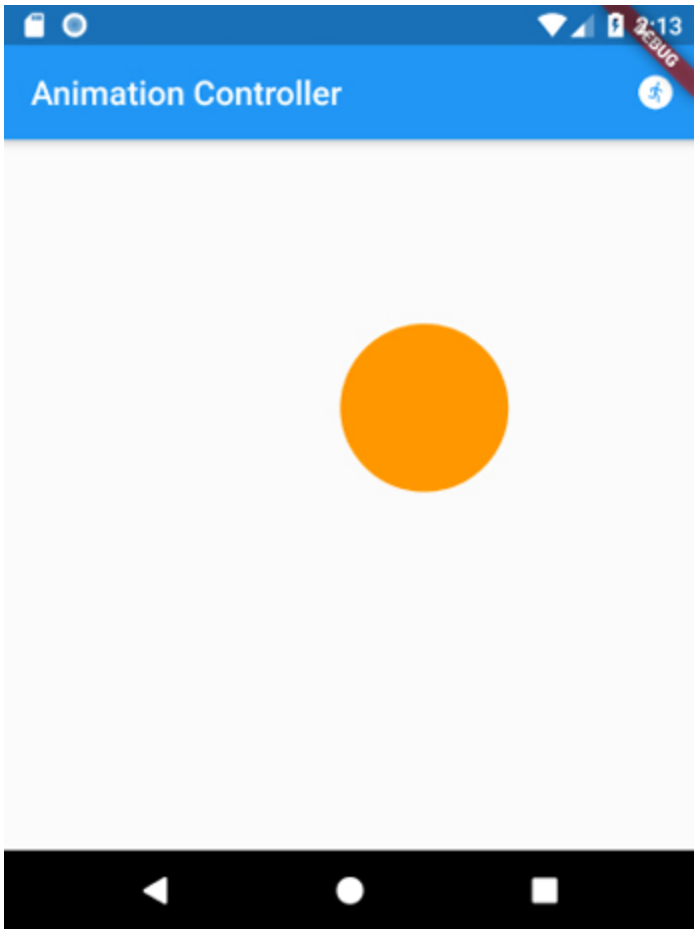


Figure 12.2: Example of AnimationController

Getting ready

To follow along with this recipe, you can create a new Flutter project, or you can use the app created in the previous recipe, *Creating basic container animations*.

How to do it...

In this recipe, you will build a widget that moves across the screen:

1. In the `lib` folder, create a new file called `shape_animation.dart`.
2. At the top of the file, import `material.dart`:

```
import 'package:flutter/material.dart';
```

3. Create a new stateful widget using the `stful` shortcut, and call it `ShapeAnimation`. The result is shown here:

```
class ShapeAnimation extends StatefulWidget {
  const ShapeAnimation({super.key});
  @override
  State<ShapeAnimation> createState() =>
_ShapeAnimationState();
}
class _ShapeAnimationState extends
State<ShapeAnimation> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

4. At the bottom of the `shape_animation.dart` file, create a stateless widget, using the `stless` shortcut, called `Ball`:

```
class Ball extends StatelessWidget {
  const Ball({super.key});
  @override
  Widget build(BuildContext context) {
```

```
    return Container();
  }
}
```

5. In the `build` method of the `Ball` class, set the `Container` so that it has a `width` and `height` of `100`. Then, in its `decoration`, set the `color` to orange and the `shape` to be a circle:

```
return Container(
  width: 100,
  height: 100,
  decoration: const BoxDecoration(color: Colors.orange,
shape:
   BoxShape.circle),
);
```

6. At the top of the `_ShapeAnimationState` class, declare an `AnimationController` called `controller`:

```
late AnimationController controller;
```

7. Override the `dispose` method, and here dispose of the controller:

```
@override
  void dispose() {
    controller.dispose();
    super.dispose();
  }
```

8. In the `_ShapeAnimationState` class, override the `initState` method:

```
@override
void initState() {
    super.initState();
}
```

9. In the `initState` method, set the controller to a new `AnimationController` with a `duration` of 3 seconds, and a `vsync` of `this`. This will generate an error in the `vsync` parameter, which we will solve in the next step:

```
controller = AnimationController(
  duration: const Duration(seconds: 3),
  vsync: this,
);
```

10. In the `_ShapeAnimationState` declaration, add the `with SingleTickerProviderStateMixin` instruction:

```
class _ShapeAnimationState extends State<ShapeAnimation>
 with SingleTickerProviderStateMixin {
```

11. At the top of the `_ShapeAnimationState` class, add the declaration of an `Animation` of type `double`:

```
late Animation<double> animation;
```

12. Under the `Animation` declaration, declare a `double` called `pos`:

```
double pos = 0;
```

13. In the `build` method, add a `Scaffold`, with an `AppBar` that contains a `Text` with a value of `Animation Controller`, and a `body` that contains a `Stack`. In the `Stack`, put a `Positioned` widget, whose child will be an instance of `Ball`. Then, set the `left` and `top` properties to `pos`, as shown here:

```
return Scaffold(
      appBar: AppBar(
        title: const Text('Animation Controller'),
      ),
      body: Stack(
        children: [
          Positioned(left: pos, top: pos, child: const
Ball()),
        ],
      ),
    );}
```

14. At the bottom of the `_ShapeAnimationState` class, create a new method, called `moveBall`, which will change the `pos` state value to the

`value` property of `animation`:

```
void moveBall() {
  setState(() {
    pos = animation.value;
  });
}
```

15. In the `initState` method, set the `Animation` with a `Tween` of type `double`, a `begin` value of `0`, and an `end` value of `200`. Append an `animate` method, passing the controller, and add a listener for the animation. The function inside the listener will simply call the `moveBall` method, as shown here:

```
animation = Tween<double>(begin: 0, end:
200).animate(controller)
..addListener(() {
  moveBall();
});
```

16. In the `AppBar` in the `build` method, add the `actions` parameter. This will contain an `IconButton` with the `run_circle` icon, and when pressed, it will reset the controller and call its `forward` method:

```
actions: [
  IconButton(
    onPressed: () {
      controller.reset();
      controller.forward();
    },
    icon: const Icon(Icons.run_circle),
)],
```

17. At the top of the `main.dart` file, import `shape_animation.dart`:

```
import './shape_animation.dart';
```

18. In the `build` method of the `MyApp` class, set the `home` of the `MaterialApp` to `ShapeAnimation`:

```
home: const ShapeAnimation(),
```

19. Run the app, and observe the movement of the ball on the screen: the ball should move from the top-left corner to a position 200 pixels from both the top and the left margins.

How it works...

This recipe contains the preparation for the next two recipes that follow, and you added a few important features that make it possible to animate widgets in Flutter. In particular, there are three important classes that you used:

- `Animation`
- `Tween`
- `AnimationController`

You have declared an `Animation` at the top of the `_ShapeAnimationState` class, with the following declaration:

```
late Animation<double> animation;
```

`The Animation class takes some values and transforms them into animations`: you use it to interpolate the values used for your animation.

Interpolation is the process of generating intermediate frames between the starting value and the end value of an animation, to create the illusion of a smooth motion. These intermediate frames are also called “in-between” frames.

`Animation<double>` means that the values that will be interpolated are of type `double`.

An instance of `Animation` is not bound to other widgets on the screen: it’s only aware of the state of the animation itself during each frame change.

A `Tween` (short for “in-be**tween**") contains the values of the property or properties that change during the animation. Consider this instruction:

```
animation = Tween<double>(begin: 0, end:
200).animate(controller);
```

This will interpolate the numbers from 0 to 200 in the time specified in the `AnimationController`.

An `AnimationController` controls one or more `Animation` objects. You can use it for several tasks: starting animations, specifying a duration, resetting them, and repeating them. It generates a new value whenever the hardware is ready for a new frame: specifically, it generates numbers from 0 to 1 for a duration that you specify. It also needs to be disposed of when you finish using it.

In the `initState` method, you put the following instruction:

```
controller = AnimationController(
  duration: const Duration(seconds: 3),
  vsync: this,
);
```

The `duration` property contains a `Duration` object, where you can specify the time length of the animations associated with the controller in seconds. You can also choose other time measures, such as milliseconds or minutes.

The `vsync` property requires a `TickerProvider`: a **Ticker** is a class that sends a signal at a regular interval, which ideally is 60 times per second when the device allows that frame rate. In our example, in the declaration of the `State` class, we used the following:

```
class _AnimatedSquareState extends State<AnimatedSquare> with
 SingleTickerProviderStateMixin {
```

The `with` keyword means we are using a `Mixin`, which is a class containing methods that can be used by other classes without inheriting from those other classes. Basically, we are including the class, not using it as a parent class. Mixins are a very effective way of reusing the same class code in multiple hierarchies.

The `with SingleTickerProviderStateMixin` means we are using a ticker provider that delivers a single ticker and is only suitable when you have a single `AnimationController`. When using multiple `AnimationController` objects, you should use a `TickerProviderStateMixin` instead.

The `addListener` method, which you can append to an `Animation`, is called each time the value of the animation changes.

Note the instructions:

```
addListener(() {
  moveBall();
});
```

In the preceding code, you call the `moveBall` method, which updates the position of the ball on the screen at each frame change. The `moveBall` method itself shows the change to the user calling the `setState` method:

```
void moveBall() {
  setState(() {
    pos = animation.value;
});}
```

The `pos` variable contains the `left` and `top` values for the `Ball`: this creates a linear movement that goes toward the bottom and right of the screen.

See also

The Flutter team has invested a lot of resources into creating and documenting animations. The first place to start when understanding the internals of how animations work is the animations tutorial available at https://flutter.dev/docs/development/ui/animations/tutorial.

Designing animations part 2 — Adding multiple animations

We will now wire up a series of **tweens** that describe what values the animation is supposed to change and then link the tweens to the `AnimationController`. This will allow us to move the ball on the screen at different speeds.

In this recipe, you will learn how to perform several animations with the same `AnimationController`, which will give you the flexibility to perform more interesting custom animations in your app.

Getting ready

To follow along in this recipe, you need the app built in the previous recipe, *Designing animations part 1 — VSync and the AnimationController*.

How to do it…

In the next few steps, you will see how to perform two `Tween` animations at the same time using a single `AnimationController`:

1. At the top of the `_ShapeAnimationState` class, remove the `pos` double, and add two new values, one for the top position and one for the left position of the ball. Also, remove the `animation` variable, and add two animations, one for the top and one for the left values, as shown here:

```
double posTop = 0;
double posLeft = 0;
late Animation<double> animationTop;
late Animation<double> animationLeft;
```

2. In the `initState` method, remove the `animation` object and set `animationLeft` and `animationTop`. Only add the listener to

`animationTop`:

```
animationLeft = Tween<double>(begin: 0, end:
150).animate(controller);
animationTop = Tween<double>(begin: 0, end:
300).animate(controller)
..addListener(() {
  moveBall();
});
```

3. In the `moveBall` method, in the `setState` call, set `posTop` to the value of the `animationTop` animation, and do the same for `posLeft` with `animationLeft`:

```
void moveBall() {
  setState(() {
    posTop = animationTop.value;
    posLeft = animationLeft.value;
  });
}
```

4. In the `Positioned` widget in the `build` method, set the `left` and `top` parameters to take `posLeft` and `posTop`:

```
Positioned(left: posLeft, top: posTop, child: Ball()),
```

5. Run the app and observe the movement of the ball: now, the ball moves to the bottom twice as fast as to the right of the screen: the end position of the ball is shown in *Figure 12.3*:

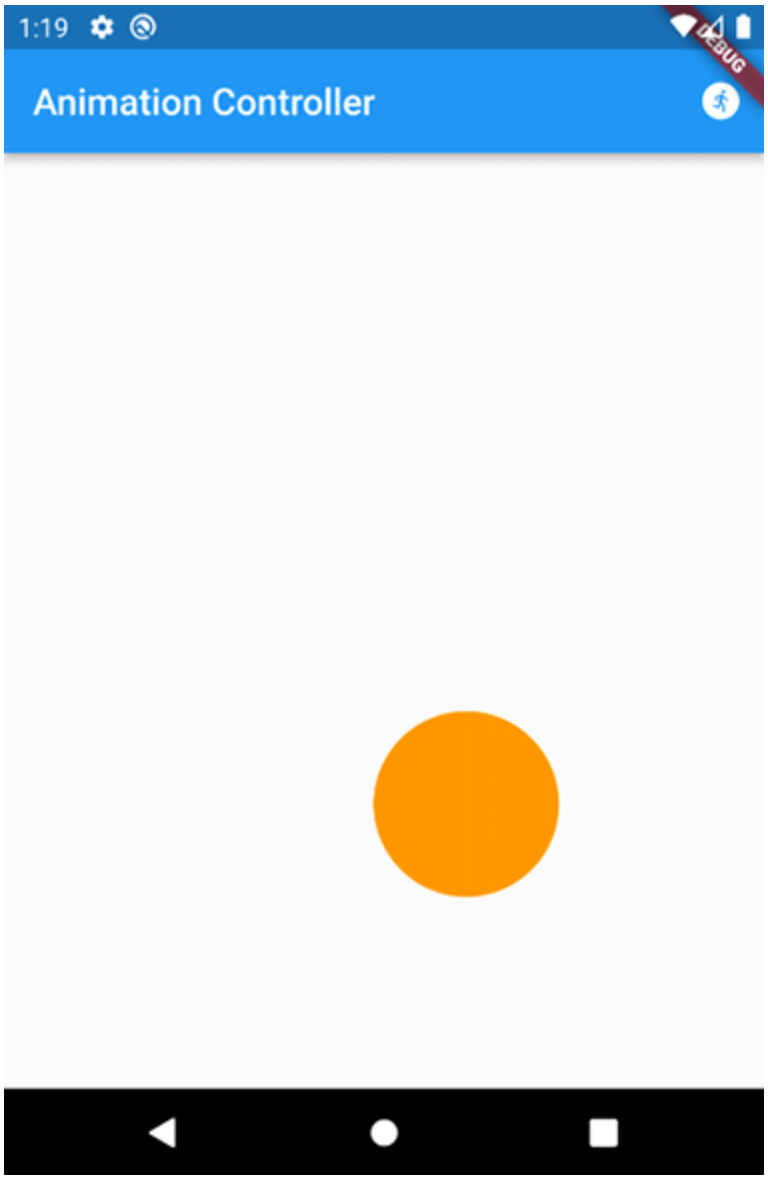


Figure 12.3: Finishing position after two different animations

How it works...

An `AnimationController` controls one or more animations. In this recipe, you saw how to add a second animation to the same controller. This is extremely important as it allows you to change different properties of an object, with different values, which is what you will probably need in your apps most of the time.

The goal of this recipe was to separate the values for the left and top coordinates of the `Ball` object.

Consider the instructions:

```
animationLeft = Tween<double>(begin: 0, end:
200).animate(controller);
animationTop = Tween<double>(begin: 0, end:
400).animate(controller)
..addListener(() {
  moveBall();
});
```

The two `Tween` animations, `animationLeft` and `animationTop`, interpolate the numbers from 0 to 200 and from 0 to 400, in the time specified in the `AnimationController`. This means that the vertical movement will be faster than the horizontal one, as during the same time frame the ball will cover twice the space vertically.

Designing animations part 3 — Using curves

Linear movement exists only in theoretical physics. Why not make the movement a bit more realistic with *ease-in* and *ease-out* curves?

In general, `CurvedAnimations` provide a more realistic and visually appealing movement, and a variety of pre-built curves to choose from, like ease-in, ease-out, and bounce.

In this recipe, you will add a curve to the animation you built in parts 1 and 2 of *Designing animations*, so that the ball will start moving slowly, then increase its speed, and then slow down again before stopping.

By the end of this recipe, you will understand how to add curves to your animations, making them more realistic and appealing.

Getting ready

To follow along with this recipe, you need the app built in the previous two recipes: *Designing animations part 1 — VSync and the AnimationController* and *Designing animations part 2 — Adding multiple animations*.

How to do it...

In this recipe, you will refactor the animation, adding a curve and changing the movement so that it takes up all the available space:

1. At the top of the `_ShapeAnimationState` class, add two new `double` values for the maximum value for the top coordinate and for the left coordinate, a new `Animation` of type `double` (the animation may already exist if you didn't delete it in the previous recipe; otherwise, just add it again), and a `final int` containing the size of the ball:

```
double maxTop = 0;
double maxLeft = 0;
late Animation<double> animation;
final int ballSize = 100;
```

2. In the `build` method, enclose the `Stack` widget in a `SafeArea` with a `LayoutBuilder`. In the builder, set the `maxLeft` and `maxTop` values to be the constraints' max values, minus the size of the ball (100). The left and top values of the `Positioned` widget will take `posLeft` and `posTop`. The full body of the `Scaffold` is shown here:

```
body: SafeArea(child: LayoutBuilder(
  builder: (BuildContext context, BoxConstraints
constraints) {
    maxLeft = constraints.maxWidth - ballSize;
    maxTop = constraints.maxHeight - ballSize;
    return Stack(
      children: [
        Positioned(left: posLeft, top: posTop, child:
const Ball()),
      ],
   );
  },
)));
```

3. In the `initState` method, remove or comment out the `animationLeft` and `animationTop` settings, and add a new `CurvedAnimation`:

```
animation = CurvedAnimation(
  parent: controller,
  curve: Curves.easeInOut,
);
```

4. Under the `animation` setting, add the listener, as shown here:

```
animation.addListener(() {
      moveBall();
});
```

5. In the `moveBall` method, set `posTop` and `posLeft` as shown here:

```
void moveBall() {
  setState(() {
    posTop = animation.value * maxTop;
    posLeft = animation.value * maxLeft;
  });
}
```

6. Run the app and observe the movement of the ball, going from the top left corner, accelerating, then slowing down, and reaching the bottom-right corner.

How it works...

This recipe added two features to the animation you completed here: finding the size of the space where the `Ball` object could move, and adding a curve to the movement: both are very useful tools when designing animations for your apps.

In order to find the available space, we used a `SafeArea` widget containing a `LayoutBuilder`.

`SafeArea` is a widget that adds some padding to its child in order to avoid the operating system intruding, like the status bar at the top of the screen or the notch that you find on some phones. This is useful when you want to only use the available space for your app.

A `LayoutBuilder` allows measuring the space available in the current context, as it provides the parent's constraints: in our example, the constraints of the `SafeArea` widget. A `LayoutBuilder` widget requires a `builder` in its constructor. This takes a function with the current context and the parent's constraints.

Consider the following instructions:

```
body: SafeArea(child: LayoutBuilder(
builder: (BuildContext context, BoxConstraints constraints) {
maxLeft = constraints.maxWidth - ballSize;
maxTop = constraints.maxHeight - ballSize;
```

Here, we used the constraints of the `SafeArea` widget (`BoxConstraints constraints`) and subtracted the size of the ball (100) in order to stop the movement when the ball reaches the safe area borders.

The other important part of this recipe was using a curve, which we achieved with this instruction:

```
animation = CurvedAnimation(
  parent: controller,
  curve: Curves.easeInOut,
);
```

You use a `CurvedAnimation` when you want to apply a non-linear curve to an animation. There are several prebuilt curves available: in this recipe, you used the `easeInOut` curve, which starts slowly, then speeds up, and finally slows down again. The values returned by `CurvedAnimation` objects go from `0.0` to `1.0` but with different speeds throughout the duration of the animation. This explains the way we modified the `moveBall` method:

```
void moveBall() {
  setState(() {
    posTop = animation.value * maxTop;
    posLeft = animation.value * maxLeft;
  });
}
```

As the value of the animation starts at 0, the position of the ball will start at 0, then reach its limit at the `maxLeft` and `maxTop` coordinates, with a value of 1 when the animation completes.

When you run the app, you should see the ball movement starting slowly, then accelerating, and then slowing down again.

See also

For a full list of the many available curves in Flutter, have a look at https://api.flutter.dev/flutter/animation/Curves-class.html.

Optimizing animations

In this recipe, you will use `AnimatedBuilder` widgets, which simplify the process of writing animations and provide some important performance optimizations.

In particular, you will design a "yo-yo-like" animation. By using an `AnimatedBuilder`, your animations will only need to redraw the descendants of the widget. In this way, you will both optimize the animations and simplify the process of designing them.

Getting ready

To follow along with this recipe, you need the app built in the previous three recipes: *Designing animations part 1 — VSync and the AnimationController*, *Designing animations part 2 — Adding multiple animations,* and *Designing animations part 3 — Using curves*.

How to do it...

You will now add an `AnimatedBuilder` widget to your app to optimize the movement of the ball on the screen:

1. In the `build` method of the `_ShapeAnimationState` class, remove the `actions` parameter from the `AppBar`.
2. In the body of the `Scaffold`, include the `Positioned` widget in an `AnimatedBuilder`, and in the `builder` parameter, add a call to the `moveBall` method:

```
return Stack(children: [
            AnimatedBuilder(
                animation: controller,
                child: const Ball(),
                builder: (context, child) {
                  return Positioned(
                    left: animation.value * maxLeft,
```

```
                    top: animation.value * maxTop,
                    child: child!,
                  );
                })
            ]);
```

3. In the `moveBall` method, remove the `setState` instructions, so that it looks as shown here:

```
void moveBall() {
  posTop = animation.value * maxTop;
  posLeft = animation.value * maxLeft;
}
```

4. In the `initState` method, when setting the controller, `AnimationController`, append a `repeat` method, as shown here:

```
controller = AnimationController(
  duration: const Duration(seconds: 3),
  vsync: this,
)..repeat();
```

5. Run the app and notice how the animation repeats.
6. Add to the `repeat()` method the parameter `reverse: true`, as shown here:

```
controller = AnimationController(
  duration: const Duration(seconds: 3),
  vsync: this,
)..repeat(reverse: true);
```

7. Run the app and notice how the movement has changed.

How it works...

The Flutter animation framework gives us many choices when building animations. One of the most flexible ones is the `AnimatedBuilder`: this widget describes animations as part of a `build` method for another widget. It takes an `animation`, a `child`, and a `builder`. The optional `child` exists

independently of the animation. An `AnimatedBuilder` listens to the notifications from an Animation object and calls its `builder` for each value provided by an Animation, only rebuilding its descendants: this is an efficient way of dealing with animations.

In the `moveBall` method, there is no need to call `setState`, as redrawing the ball is a task that the `AnimatedBuilder` performs automatically.

The `repeat` method, appended to the `AnimationController`, runs this animation from start to end and restarts the animation as soon as it completes.

If you set the `reverse` parameter to `true`, when the animation completes, instead of always restarting from its beginning value (`min`) it starts from its ending value (`max`) and decreases it, thus creating a "yo-yo-like" animation. This addition to the Flutter framework makes it very easy to create this kind of animation.

See also

One of the challenges when designing animations is choosing the most appropriate one for your projects: there's a very interesting article on Medium that explains when it's worth using `AnimatedBuilder` and `AnimatedWidget`, available at https://medium.com/flutter/when-should-i-useanimatedbuilder-or-animatedwidget-57ecae0959e8.

Using Hero animations

In Flutter, a **Hero** is an animation that makes a widget "fly" from one screen to another. During the flight, the `Hero` may change position, size, and shape. The Flutter framework deals with the transition automatically.

In this recipe, you will see how to implement a `Hero` transition in your apps, from an icon in a `ListTile` to a bigger icon on a detail screen, as shown in the following screenshots. The first screen contains a `List`:

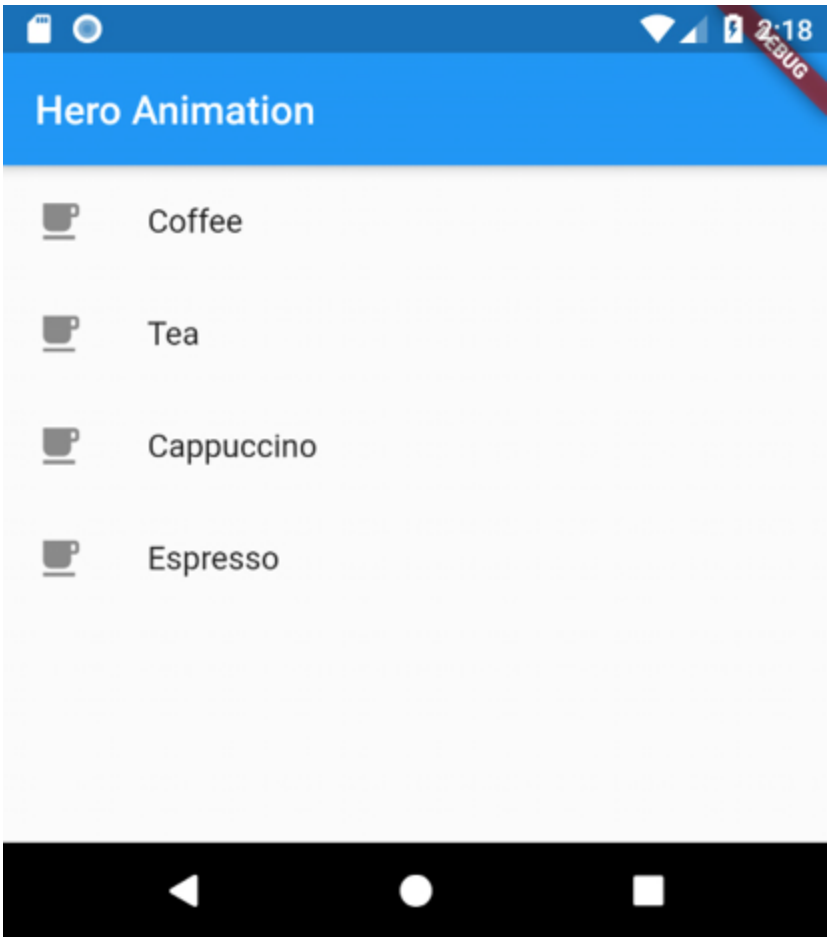


Figure 12.4: A List in a Hero animation

When the user clicks on one of the items in the List, the second screen appears with an animation:

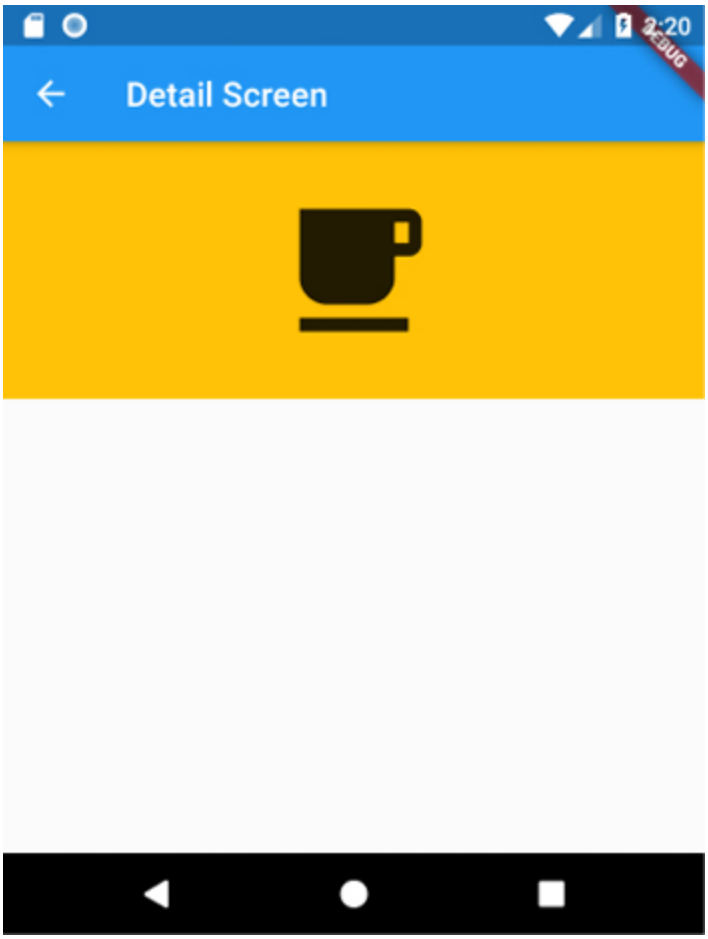


Figure 12.5: Detail screen after the Hero animation

Getting ready

To follow along with this recipe, you need any existing Flutter project already created, or you can use the app created in any of the previous recipes.

How to do it...

In this recipe, you will create a `Hero` animation by transforming an icon, and changing its position and size:

1. Create a new file in the `lib` folder in your project called `list_screen.dart`.

2. At the top of the new file, import `material.dart`:

```
import 'package:flutter/material.dart';
```

3. Create a stateless widget using the `stless` shortcut, calling the widget `ListScreen`:

```
class ListScreen extends StatelessWidget {
ListScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

4. At the top of the `ListScreen` widget, add a new `List` of strings, called `drinks` (make sure you remove `const` from the constructor method):

```
final List<String> drinks= ['Coffee', 'Tea',
'Cappuccino', 'Espresso'];
```

5. In the `build` method, return a `Scaffold`. In its `appBar` parameter, add an `AppBar` with a title of `'Hero Animation'`:

```
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Hero Animation'),
    ),
    body: Container()
);}
```

6. In the body of the `Scaffold`, add a `ListView` with its `builder` constructor. Set the `itemCount` parameter to take the length of the `drinks` list, and in the `itemBuilder`, return a `ListTile`:

```
body: ListView.builder(
  itemCount: drinks.length,
  itemBuilder: (BuildContext context, int index) {
     return ListTile();
```

```
  }
)
```

7. In the `ListTile`, set the `leading` parameter to take a `Hero` widget. The `tag` property takes a string with the concatenation of `cup` and the `index`. The child takes the `free_breakfast` icon:

```
return ListTile(
  leading: Hero(tag: 'cup$index', child:
   const Icon(Icons.free_breakfast)),
  title: Text(drinks[index]),
);
```

8. At the bottom of the `ListTile`, add the `onTap` parameter, which will navigate to another screen called `DetailScreen`. This will give an error, as we still need to create the `DetailScreen` class:

```
onTap: () {
  Navigator.push(context, MaterialPageRoute(
  builder: (context) => DetailScreen(index)
  ));
},
```

9. Add a new file in the `lib` folder of the project called `detail_screen.dart`.
10. At the top of the new file, import `material.dart`:

```
import 'package:flutter/material.dart';
```

11. Create a stateless widget using the `stless` shortcut, calling the widget `DetailScreen`:

```
class DetailScreen extends StatelessWidget {
const DetailScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

12. At the top of the `DetailScreen` widget, add a `final int` called `index`, and in the constructor, add a parameter that allows setting the index:

```
final int index;
const DetailScreen(this.index, {super.key});
```

13. In the `build` method of `DetailScreen`, return a `Scaffold`. The `appBar` will contain a `title` containing a `Text` with `'Detail Screen'`, and in the `body`, set a `Column`:

```
return Scaffold(
  appBar: AppBar(
    title: const Text('Detail Screen'),
  ),
  body: Column(children: []));
```

14. In the `children` of the `Column`, add two `Expanded` widgets, one with a `flex` of `1`, and the second with a `flex` of `3`. The `child` of the first `Expanded` widget will contain a `Hero`, with the same tag that was set in the `ListScreen`, and a `free_breakfast` icon with a `size` of `96`. The second `Expanded` widget will just contain an empty container:

```
children: [
  Expanded(
    flex: 1,
    child: Container(
      width: double.infinity,
      decoration: const BoxDecoration(color:
Colors.amber),
      child: Hero(
        tag: 'cup$index',
        child: const Icon(Icons.free_breakfast, size:
96,),
     ),
    ),
  ),
  const Spacer(
    flex: 3,
  )
],
```

15. At the top of the `listscreen.dart` file, import `detailscreen.dart`:

```
import './detail_screen.dart';
```

16. In the `main.dart` file, in the `home` of the `MaterialApp`, import and call `ListScreen`:

```
home: ListScreen(),
```

17. Run the app, tap on any item in the `ListView`, and observe the animation to get to the `DetailScreen`. Then, get back to the List screen, and have a look at the animation again.

How it works...

In order to create a `Hero` animation, you need to create two Hero widgets: one for the **source**, and one for the **destination**.

In order to implement a `Hero` animation, do the following:

- Create the **source Hero**. A `Hero` requires a `child`, which defines how the Hero looks (an image or an `Icon` like in the example of this recipe, or any other relevant widget), and a `tag`. You use the `tag` to uniquely identify the widget, and both source and destination Hero widgets must share the same tag. In the example in this recipe, we created the source `Hero` with this instruction:

```
Hero(tag: 'cup$index', child: const
Icon(Icons.free_breakfast)),
```

By concatenating the index to the `cup` string, we achieved a unique tag.

Note

Each `Hero` requires a *unique* hero `tag`: this is used to identify which widget will be animated. In a `ListView`, for example, you cannot repeat the same tag for the items, even if they have the same child; otherwise, you will get an error.

- Create the **destination Hero**. This must contain the same tag as the source `Hero`. Its child should be similar to the destination but can change some properties, such as its size and position.

We created the **`destination`** `Hero` with this instruction:

```
Hero(
  tag: 'cup$index',
  child: Icon(Icons.free_breakfast, size: 96,),
),
```

Use the `Navigator` object to get to the `Route` containing the destination `Hero`. You can use `push` or `pop`, as both will trigger the animation for each `Hero` that shares the same tag in the source and destination routes.

In this recipe, in order to trigger the animation, you used this instruction:

```
Navigator.push(context, MaterialPageRoute(
  builder: (context) => DetailScreen(index)
));
```

See also

Hero animations are beautiful and easy to implement. For a full list of the features of heroes in Flutter, see https://flutter.dev/docs/development/ui/animations/hero-animations.

Using premade animation transitions

You can use transition widgets to create animations in an easier way than using traditional animations. The Flutter framework contains several pre-

made transitions, which makes animating objects extremely straightforward. These include:

- `DecoratedBoxTransition`
- `FadeTransition`
- `PositionedTransition`
- `RotationTransition`
- `ScaleTransition`
- `SizeTransition`
- `SlideTransition`

In this recipe, you will use the `FadeTransition` widget, but the same animation rules that you will see for `FadeTransition` apply to the other transitions in the Flutter framework.

In particular, you will make a square appear slowly on the screen, over a specified duration of time:

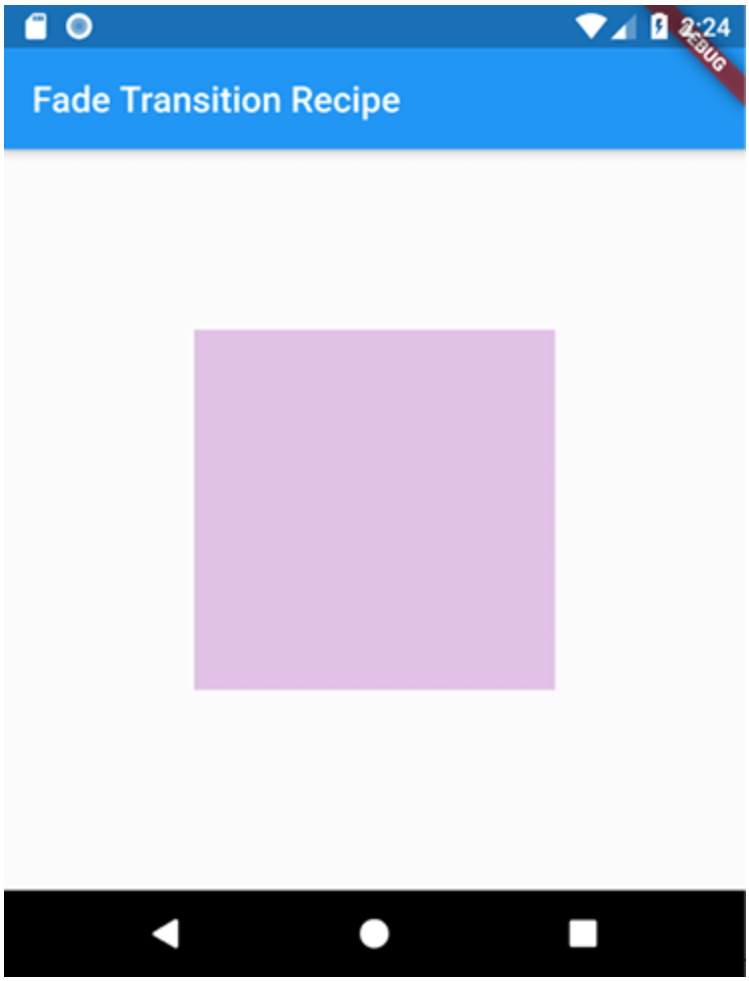


Figure 12.6: A square appearing with FadeTransition

Getting ready

To follow along with this recipe, you need any existing Flutter project already created, or you can use the app created in any of the previous recipes.

How to do it...

You will use the `FadeTransition` widget in your app to make a square smoothly appear on the screen:

1. Create a new file in the `lib` folder of your project called `fade_transition.dart`.
2. At the top of the `fade_transition.dart` file, import the `material.dart` library:

```
import 'package:flutter/material.dart';
```

3. In the `fade_transition` file, create a new stateful widget, using the `stful` shortcut, and call the new widget `FadeTransitionScreen`:

```
class FadeTransitionScreen extends StatefulWidget {
    const FadeTransitionScreen({super.key});
  @override
  State<FadeTransitionScreen> createState() =>
_FadeTransitionScreenState();
}
class _FadeTransitionScreenState extends
State<FadeTransitionScreen> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

4. Add the `with SingleTickerProviderStateMixin` instruction to the `_FadeTransitionScreenState` class:

```
class _FadeTransitionScreenState extends
State<FadeTransitionScreen> with
SingleTickerProviderStateMixin
```

5. At the top of the `_FadeTransitionScreenState` class, declare two variables — an `AnimationController` and an `Animation`:

```
late AnimationController controller;
late Animation <double> animation;
```

6. Override the `initState` method. In the function, set the `AnimationController` so that it takes `this` in its `vsync` parameter, and set its duration to 3 seconds:

```
@override
void initState() {
    super.initState();
   controller = AnimationController(vsync: this,
duration:
   const Duration(seconds: 3));
}
```

7. Still in the `initState` method, under the `controller` configuration, set the `Animation` to be a `Tween` with a beginning value of `0.0` and an end value of `1.0`:

```
animation = Tween(begin: 0.0, end:
1.0).animate(controller);
```

8. At the bottom of the `initState` method, call the `forward` method on the `AnimationController`:

```
controller.forward();
```

9. In the `build` method, instead of returning the default `container`, return a `Scaffold`, with an `AppBar` and a `body`. In the `body`, return a `Center` widget, and there add a `FadeTransition` widget, setting the `opacity` so that it takes the animation. As a child, add a simple `Container`, with purple as its color, as shown here:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
      appBar: AppBar(
        title: const Text('Fade Transition Recipe'),
      ),
      body: Center(
        child: FadeTransition(
          opacity: animation,
          child: Container(
            width: 200,
            height: 200,
            color: Colors.purple,
          ),
        ),
      ),
```

```
    );
}
```

10. At the bottom of the class, override the `dispose` method, and there call the `controller.dispose()` method:

```
@override
void dispose() {
  controller.dispose();
  super.dispose();
}
```

11. In the `main.dart` file, in the `home` property of `MaterialApp`, import and call the `FadeTransitionScreen` widget:

```
home: const FadeTransitionScreen(),
```

12. Run the app, and notice how when the screen loads, a purple square fades in, taking 3 seconds.

How it works...

A `FadeTransition` widget animates the opacity of its child. This is the perfect animation when you want to show or hide a widget over a specified duration of time.

When using `FadeTransition`, you need to pass two parameters:

- `opacity`: This requires an `Animation`, which controls the transition of the child widget.
- `child`: The widget that fades in or out with the animation specified in the `opacity`.

In this recipe, you set the `FadeTransition` with these instructions:

```
FadeTransition(
  opacity: animation,
```

```
  child: Container(
    width: 200,
    height: 200,
    color: Colors.purple,
),),
```

In this case, the child is a `Container` with a width and height of 200 device-independent pixels and a purple background. Of course, you could specify any other widget instead, including an image or an icon.

`Animation` widgets require an `AnimationController`. You set the controller with this instruction:

```
controller = AnimationController(vsync: this, duration: const
 Duration(seconds: 3));
```

This specified the `Duration` (3 seconds) and the `vsync` parameter. In order to set `vsync` to `this`, you need to add the `with SingleTickerProviderStateMixin` instruction.

> **Note**
>
> For a refresher on mixins, see the first recipe in this chapter: *Designing animations part 1 — Using the AnimationController*.

For the animation, we used a `Tween` in this case. There is no need to use more complex animations, as for fading in and out, you can generally use a linear animation. Take the following instruction:

```
animation = Tween(begin: 0.0, end: 1.0).animate(controller);
```

It creates a `Tween` that linearly goes from 0 to 1, using the `AnimationController`.

The last step to complete this recipe was adding the code to dispose of the controller inside the `dispose` method.

```
@override
void dispose() {
  controller.dispose();
  super.dispose();
}
```

Using pre-made transitions, such as `FadeTransition`, makes it easy to add nice-looking animations to your projects.

See also

`FadeTransition` is only one of the possible transition widgets you can use in Flutter. For a full list of the animated widgets in Flutter, have a look at https://flutter.dev/docs/development/ui/widgets/animation.

Using the AnimatedList widget

`ListView` widgets are arguably one of the most important widgets for showing lists of data in Flutter. When using a `ListView`, its contents can be added, removed, or changed. A problem that may happen is that users may miss the changes in the list. That's when another widget comes to help. It's the `AnimatedList` widget. It works just like a `ListView`, but when you **`add`** or **`remove`** an item in the list, you can show an animation so that you help the user be aware of the change.

In this recipe, you will make items appear and disappear slowly in an animated list:

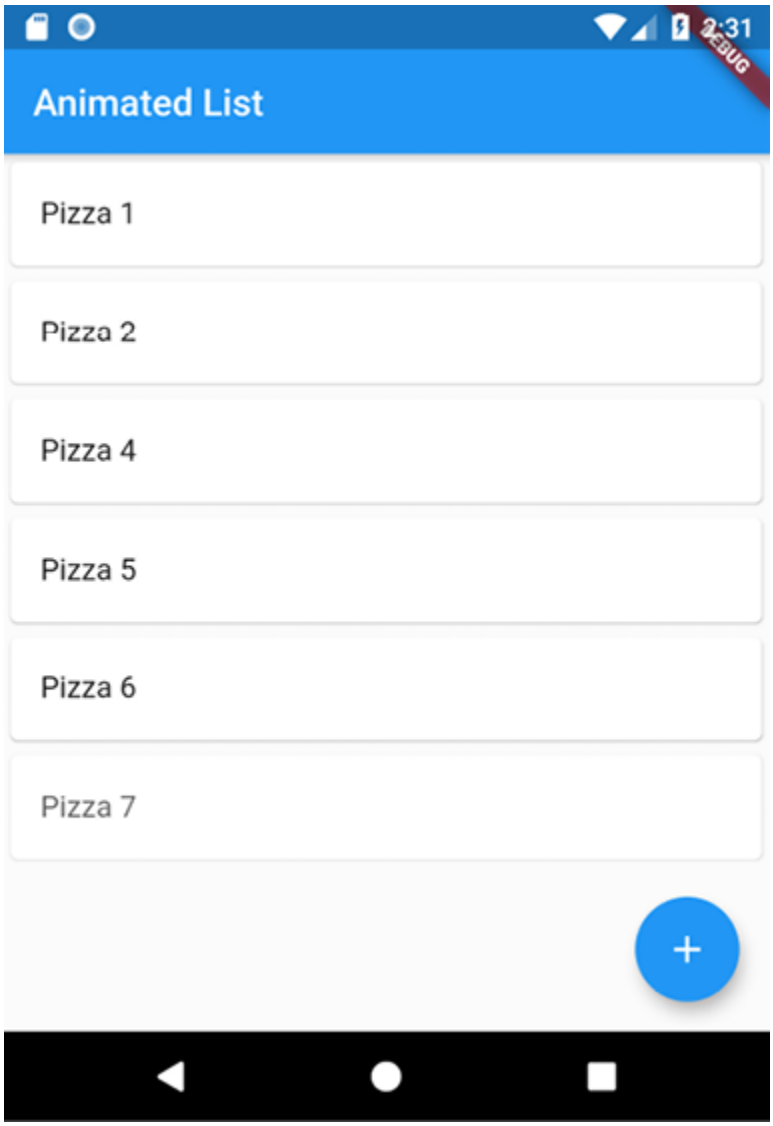


Figure 12.7: An AnimatedList example

Getting ready

To follow along with this recipe, you need to create a new Flutter project, or you can use the app created in any of the previous recipes.

How to do it...

In this recipe, we will use a `FadeTransition` into an `AnimatedList`:

1. Create a new file in the `lib` folder called `animated_list.dart`.
2. At the top of the file, import `material.dart`:

```
import 'package:flutter/material.dart';
```

3. Create a stateful widget, calling it `AnimatedListScreen`:

```
class AnimatedListScreen extends StatefulWidget {
  const AnimatedListScreen({super.key});
  @override
  State<AnimatedListScreen> createState() =>
_AnimatedListScreenState();
}
class _AnimatedListScreenState extends
State<AnimatedListScreen> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }}
```

4. At the top of the `_AnimatedListScreenState` class, declare a `GlobalKey`, which you will use to access the `AnimatedList` from anywhere within the class:

```
final GlobalKey<AnimatedListState> listKey =
GlobalKey<AnimatedListState>();
```

5. Under the `GlobalKey`, declare a `List` of `int` values, and set the list to contain numbers from 1 to 5 and a `counter` integer:

```
final List<int> _items = [1, 2, 3, 4, 5];
int counter = 5;
```

6. In the `_AnimatedListScreenState` class, add a new method called `fadeListTile`, which will take the current context, an integer, and an `Animation`. Inside the method, retrieve the current item in the list through the index that was passed. Then, return a `FadeTransition`, which will animate the opacity of its child with a `Tween`, from 0 to 1.
7. For the child of the `FadeTransition`, return a `Card` containing a `ListTile`. Its `title` will take a `Text` widget containing the

concatenation of `'Pizza'` and the item number for the index that was passed.

8. In the `onTap` parameter of the `ListTile`, call the `removePizza` method, which you will create in the next step (this will show an error, which you can ignore).
9. The full code of the `fadeListTile` method is shown here:

```
Widget fadeListTile(BuildContext context, int index,
Animation<double> animation) {
  return FadeTransition(
      opacity:
Tween(begin: 0.0, end: 1.0).animate(animation),
      child: Card(
        child: ListTile(
          title: Text('Pizza $item'),
          onTap: () {
            int index = _items.indexOf(item);
            if (index != -1) removePizza(index);
          },
        ),
      ),
    );;
}
```

10. Add another method, called `removePizza`, which takes an integer containing the item that will be removed.
11. Inside the function, call the `removeItem` method using the `GlobalKey` that was created at the top of the class. This method takes the index of the item that should be removed, a function that returns the animation (`fadeListTile`), and an optional `duration` of 1 second.
12. At the bottom of the method, also remove the deleted item from the `_items` list.
13. The code for the `removePizza` method is shown here:

```
removePizza(int index) {
  int removedItem = _items[index];
    listKey.currentState!.removeItem(
      index,
      (context, animation) => fadeListTile(context,
removedItem, animation),
```

```
      duration: const Duration(seconds: 1),
    );
    _items.removeAt(index);
}
```

14. Add a new method in the `_AnimatedListScreenState` class called `insertPizza`.
15. Inside the function, call the `insertItem` method using the `GlobalKey` that was created at the top of the class.
16. This method takes an integer with the position of the item that should be added, and an optional duration.
17. At the end of the method, also add the item in the `_itemsList`, incrementing the counter.
18. The code of the `insertPizza` method is shown here:

```
insertPizza() {
  listKey.currentState!.insertItem(
      _items.length,
      duration: const Duration(seconds: 1),
    );
    _items.add(++counter);
}
```

19. In the `build` method of the `_AnimatedListScreenState` class, return a `Scaffold`.
20. In the `body` of the `Scaffold`, add an `AnimatedList`, setting its key with the `GlobalKey` that was defined at the top of the class, an `initialItemCount` that takes the length of the `_items` list, and an `itemBuilder` that returns the `fadeListTile` method, as shown here:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
        appBar: AppBar(
          title: const Text('Animated List'),
        ),
        body: AnimatedList(
          key: listKey,
          initialItemCount: _items.length,
```

```
            itemBuilder:
                (BuildContext context, int index,
Animation<double> animation) {
              return fadeListTile(context, index,
animation);
            },
          ));
}
```

21. Add a `floatingActionButton` to the `Scaffold`. This will show an `add` icon, and when pressed, will call the `insertPizza` method:

```
floatingActionButton: FloatingActionButton(
  child: const Icon(Icons.add),
  onPressed: () {
    insertPizza();
  },
),
```

22. In the `main.dart` file, in the `home` of the `MaterialApp`, import and call the `AnimatedListScreen` widget:

```
home: const AnimatedListScreen(),
```

23. Run the app and try adding items with the floating action button and removing items from the list by tapping on a few items, and notice the animation that shows for each action you perform.

How it works...

The layout for this recipe was very simple, only containing a list view, and a `FloatingActionButton` to add new items in the `AnimatedList`.

Note

An `AnimatedList` is a `ListView` that shows an animation when an item is inserted or removed.

The first step you used in this recipe was setting a `GlobalKey<AnimatedListState>`. This allowed you to store the state of the `AnimatedList` widget:

```
final GlobalKey<AnimatedListState> listKey =
GlobalKey<AnimatedListState>();
```

In the example in this recipe, we used an `AnimatedList`, setting three properties:

- `key`: You need a key whenever you need to access the `AnimatedList` from outside the item itself.
- `initialItemCount`: You use the `initialItemCount` to load the initial items of the list. These won't be animated. The default value is `0`.
- `itemBuilder`: This is a required parameter, necessary to build items in an `AnimatedList`.
- The method inside the `itembuilder` can return any animation widget. In the example in this recipe, we used a `FadeTransition` to show a `Card` with a `ListTile` inside.

The code we used to create the `AnimatedList` and set its properties is shown here:

```
AnimatedList(
key: listKey,
initialItemCount: _items.length,
itemBuilder: (BuildContext context, int index,
Animation<double> animation) {
    return fadeListTile(context, _items[index], animation);
  },
)
```

Specifically, in the `itemBuilder` parameter, we called the `fadeListTile` method:

```
fadeListTile(BuildContext context, int item, Animation
animation) {
```

```
  return FadeTransition(
      opacity: Tween(begin: 0.0, end:
1.0).animate(animation),
      child: Card(
        child: ListTile(
          title: Text('Pizza $item'),
          onTap: () {
            int index = _items.indexOf(item);
            if (index != -1) removePizza(index);
          },
        ),
      ),
    );
```

In this, we used a `FadeTransition` animation, with a `Tween` that starts at 0 and ends at 1. Note that you can use any animation when showing items on an `AnimatedList`.

Note

For a quick recap on how to use the `FadeTransition` animation, have a look at the previous recipe: *Using transition animations*.

In order to add new items to the list, you wrote the `insertPizza` method:

```
insertPizza() {
  listKey.currentState.insertItem(
    _items.length,
    duration: const Duration(seconds: 1),
  );
  _items.add(++counter);
}
```

This calls the `insertItem` method using the `GlobalKey` that was created at the top of the `State` class: the `currentState` property contains the `State` for the widget that currently holds the global key.

Note

You should always make sure that the data that holds the values contained in the `AnimatedList` is also updated when you insert or

remove an item.

In order to remove an item, we used the `removePizza` method:

```
removePizza(int index) {
  int removedItem = _items[index];
    listKey.currentState!.removeItem(
      index,
      (context, animation) => fadeListTile(context,
removedItem, animation),
      duration: const Duration(seconds: 1),
    );
    _items.removeAt(index);
  }
}
```

Now, you might be wondering why we used `animationIndex` instead of the index that was passed to the function: this is because when the `AnimatedList` removes the last item, its index is no longer available. So, in order to avoid an error, we can animate the item before the last one. This works well for a `FadeTransition`.

See also

Another possible option when dealing with animated lists is using the `SliverAnimatedList`: you'll find more information about it at https://api.flutter.dev/flutter/widgets/SliverAnimatedList-class.html.

Implementing swiping with the Dismissible widget

Mobile users expect apps to respond to gestures. In particular, swiping an item left or right in a `ListView` can be used to delete an item from a `List`.

There's a very useful widget in Flutter designed to remove an item in response to a swiping gesture from the user. It's called `Dismissible`. In this recipe, you will see a simple example that shows how to implement a `Dismissible` widget in your apps.

Getting ready

To follow along with this recipe, create a new Flutter project, or use the app created in any of the previous recipes.

How to do it...

You will now create a screen with a `ListView`, and include a `Dismissible` widget in it:

1. Create a new file in the `lib` folder called `dismissible.dart`.
2. At the top of the file, import `material.dart`:

```
import 'package:flutter/material.dart';
```

3. Create a stateful widget, calling it `DismissibleScreen`:

```
class DismissibleScreen extends StatefulWidget {
const DismissibleScreen({super.key});
  @override
  State<DismissibleScreen> createState() =>
_DismissibleScreenState();
}
class _DismissibleScreenState extends
State<DismissibleScreen> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

4. At the top of the `_DismissibleScreenState` class, declare a `List` of `String` values, calling it `sweets`:

```
final List<String> sweets = [
  'Petit Four',
  'Cupcake',
  'Donut',
  'Éclair',
  'Froyo',
  'Gingerbread ',
  'Honeycomb ',
  'Ice Cream Sandwich',
  'Jelly Bean',
  'KitKat'
];
```

5. In the `build` method of the `_DismissibleScreenState` class, return a `Scaffold`, with an `appBar` and a `body` containing a `ListView` with a `builder` constructor:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
        appBar: AppBar(
          title: const Text('Dismissible Example'),
        ),
        body: ListView.builder());
}
```

6. In the `ListView.builder`, set the `itemCount` and `itemBuilder` parameters, as shown here:

```
body: ListView.builder(
            itemCount: sweets.length,
            itemBuilder: (context, index) {
              return Dismissible(
                key: Key(sweets[index]),
                background:
                  const ColoredBox(color: Colors.red),
                child: ListTile(
                  title: Text(sweets[index]),
                ),
                onDismissed: (direction) {
                  sweets.removeAt(index);
                },
              );
            }));
```

7. In the `home` of the `MaterialApp` in the `main.dart` file, import and call `DismissibleScreen`.
8. Run the app, and swipe left or right on any item in the `ListView`: the item should be removed from the screen with animation.

How it works...

With the `Dismissible` widget, the Flutter framework makes the action of deleting an item in a `ListView` with the swiping gesture very easy.

You simply drag a `Dismissible` left or right (these are `DismissDirections`) and the selected item will slide out of view, with a nice-looking animation.

It works like this:

- First, you set the `key` parameter. This allows the framework to uniquely identify the item that has been swiped, and it's a required parameter.
- In the example in this recipe, you created a new `Key` with the name of the sweets at the position that's being created:

```
key: Key(sweets[index]),
```

- Then, you set the `onDismissed` parameter. This is called when you swipe the item.
- In this example, we do not care about the direction of the swipe as we want to delete the item whether the user swipes left or right.
- Inside the function, you just call the `removeAt` method on the `sweets` list to remove the item from the `List`.

Note

The `sweets` list contains the names of the first 10 versions of Android.

I recommend using the `Dismissible` widget whenever you want to remove an item from a `List`.

See also

In some circumstances, it may be useful to perform different actions based on the direction of the user's swipe gesture. For a full list of the available directions, have a look at https://api.flutter.dev/flutter/widgets/DismissDirection.html.

Using the animations Flutter package

The `animations` package, developed by the Flutter team, allows you to design animations based on the Material Design motion specifications. The package contains pre-made animations, which you can customize with your own content: the result is that with very few lines of code, you will be able to add complex and compelling effects to your apps, including the `Container Transform`, which you will implement in this recipe.

Specifically, in this recipe, you will transform a `ListTile` into a full screen, as shown in the following screenshot:

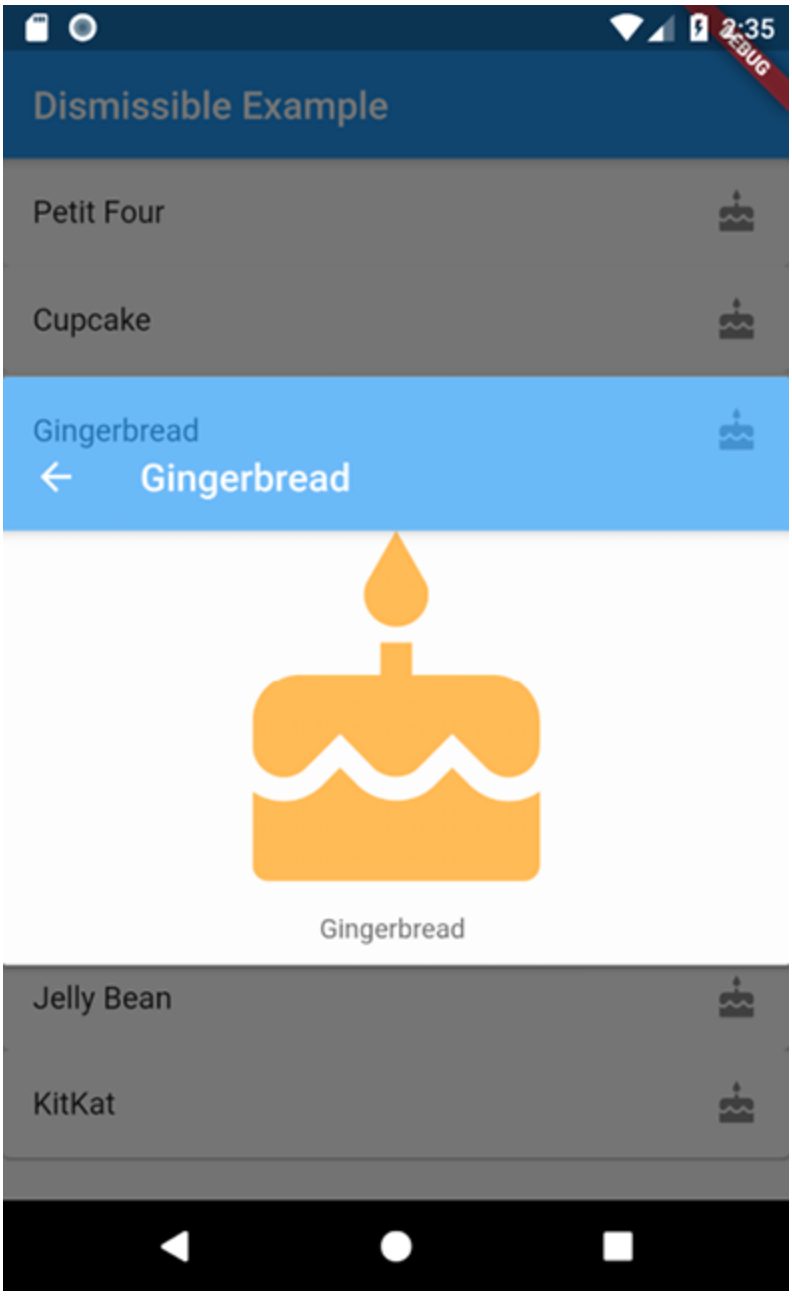


Figure 12.8: An animations package example

Getting ready

To follow along with this recipe, you should have completed the code in the previous recipe: *Implementing swiping with the Dismissible widget.*

How to do it...

In this recipe, you will use the animations **Container Transform**:

1. In your Terminal, add the `animations` dependency to your project by typing the following command:

```
flutter pub add animations
```

2. At the top of the `dismissible.dart` file, import the `animations` package:

```
import 'package:animations/animations.dart';
```

3. In the `dismissible.dart` file, in the `itembuilder` of the `ListView`, wrap the `Dismissible` widget into an `OpenContainer` widget, in its `closedBuilder` parameter, as shown here:

```
return OpenContainer(
  closedBuilder: (context, openContainer) {
    return Dismissible(
      key: Key(sweets[index]),
      child: ListTile(
        title: Text(sweets[index]),
        trailing: Icon(Icons.cake),
        onTap: () {
          openContainer();
        },
      ),
      onDismissed: (direction) {
        sweets.removeAt(index);
      },
    );
},);
```

4. At the top of the `OpenContainer` object, add a `transitionDuration` and a `transitionType`, as shown here:

```
transitionDuration: const Duration(seconds: 3),
transitionType: ContainerTransitionType.fade,
```

5. Still in the `OpenContainer` object, add the `openBuilder` parameter, containing a `Scaffold` with a detailed view of the selected `ListTile`:

```
openBuilder: (context, closeContainer) {
  return Scaffold(
    appBar: AppBar(
      title: Text(sweets[index]),
    ),
    body: Center(
      child: Column(
    children: [
      const SizedBox(
    width: 200,
    height: 200,
    child: Icon(
      Icons.cake,
      size: 200,
      color: Colors.orange,
    ),
      ),
    Text(sweets[index])
],),),);},
```

6. Run the app and observe the transition from the `ListTile` to the `Scaffold` and vice versa.

How it works...

In this recipe, you used a Container Transform transition. Basically, you transformed the `ListTile` in a `ListView` into a fullscreen, made of a `Scaffold`, an `Icon`, and some text.

The transition between the two views (the "containers" of the animation) was completely managed by the `OpenContainer` widget.

There are two important properties that we set to implement this effect: `openBuilder` and `closedBuilder`. Both require a function that builds the user interface for the current view.

In particular, you have implemented the starting view (`closed`), with this code:

```
closedBuilder: (context, openContainer) {
  return Dismissible(
    key: Key(sweets[index]),
    child: ListTile(
      title: Text(sweets[index]),
      trailing: Icon(Icons.cake),
      onTap: () {
    openContainer();
      },
    ),
    onDismissed: (direction) {
      sweets.removeAt(index);
    },
);
```

The function in the `closedBuilder` parameter takes the current `BuildContext` and the method that will be called when the user opens the animation.

In this example, the closed view contains a `Dismissible` widget, whose child is a `listTile` with a `Text` and a trailing icon. When users tap on the `ListTile`, the `openContainer` method is called. You implemented this with the `openBuilder` property of `OpenContainer`:

```
openBuilder: (context, closeContainer) {
  return Scaffold(
    appBar: AppBar(
    title: Text(sweets[index]),
  ),
  body: Center(
    child: Column(
      children: [
        const SizedBox(
          width: 200,
          height: 200,
          child: Icon(
            Icons.cake,
            size: 200,
            color: Colors.orange,
          ),
      ),
    Text(sweets[index])],),
),);},
```

The second view (`open`) of the animation contains a `Scaffold`, with an `appBar` that contains the name of the selected sweet as its title.

In its body, you put a `Column` containing a bigger version of the cake icon and a `Text` with the selected item.

See also

The animation package currently contains four effects, all taken from the Material Motion Pattern specifications: Container Transform, which you have used in this recipe; `Shared Axis`; `Fade`; and `Fade Through`. For a full list of the specifications, go to the package's official documentation at https://pub.dev/packages/animations.

Summary

In this chapter, you began exploring the world of animations in Flutter. You started by using the `AnimatedContainer` widget, allowing you to create smooth transitions between different properties of a container, like its size, color, or position. This has been an introduction to implicit animations, as the `AnimatedContainer` handles the animation process automatically.

You also designed your own custom animations using the `AnimationController`. This gives you greater control over animations, allowing you to create more complex visuals. You also saw how to add multiple animations using the same `AnimationController`.

You learned how to incorporate curves into your animations, which creates more realistic and natural-looking animations. You also saw how to optimize your animations, ensuring they run efficiently without consuming unnecessary resources, which is essential for delivering a high-performance app.

Hero animations create a seamless transition between UI elements when navigating from one screen to another: this can greatly enhance the user experience and make your app stand out.

You used pre-made animation transitions, such as `FadeTransition`, and learned how to leverage the `AnimatedList` widget to create dynamic lists with animated elements, making the process of adding, removing, or updating items more visually appealing.

Finally, you implemented swiping functionality with the `Dismissible` widget and explored the Flutter `animations` package, which offers even more tools and resources for creating great-looking animations.

Using Firebase

Firebase is a set of tools you can leverage to build scalable applications in the cloud. Those tools include databases, file storage, authentication, analytics, notifications, and hosting.

In this chapter, you will start by configuring a Firebase project, and then you will see how to use sign-in, add analytics, synchronize data across multiple devices with Cloud Firestore, send notifications to your users, and store data in the cloud.

As a bonus, a backend created with Firebase can scale over Google server farms, giving it access to virtually unlimited resources and power. There are several advantages to using Firebase as a backend for your Flutter apps: arguably the most important is that Firebase is a **Backend as a Service (BAAS)**, meaning you can easily create backend (or server-side) applications without worrying about the platform and system setup, saving you from writing most of the code that you usually need in order to have a working, real-world app. This is perfect for apps where you can give up some of the implementation details and work mostly on the frontend (the Flutter app itself).

In this chapter, we will cover the following recipes:

- Configuring a Firebase app
- Creating a login screen
- Adding Google Sign-in
- Customizing Sign in
- Integrating Firebase Analytics
- Using Firebase Cloud Firestore
- Sending push notifications with **Firebase Cloud Messaging (FCM)**

- Storing files in the cloud

By the end of this chapter, you will be able to leverage several Firebase services and create full-stack apps without any server-side code.

Configuring a Firebase app

Before using any Firebase service in your app, a configuration process is required. The tasks vary, depending on the system where the app will be installed. In this recipe, you will see the configuration required in Android and iOS. After the configuration, you can add all Firebase services, including data and file storage, sign-in, and notifications.

Getting ready

In this recipe, you will create a new project. A Google account is required to use Firebase, and this is a prerequisite for all recipes in this chapter.

How to do it...

In this recipe, you will see how to configure a Flutter app so that you can use Firebase services. This is a two-part task: first, you will create a new Firebase project, and then you will configure a Flutter app to connect to the Firebase project.

To create a new Firebase project:

1. Open your browser at the Firebase console at https://console.firebase.google.com/.
2. Enter your Google credentials, or create a new account.
3. From the Firebase console, click the **Add Project** link.
4. In the **Create a Project** page, for the project name, type `Flutter CookBook`, as shown in *Figure 13.1*:

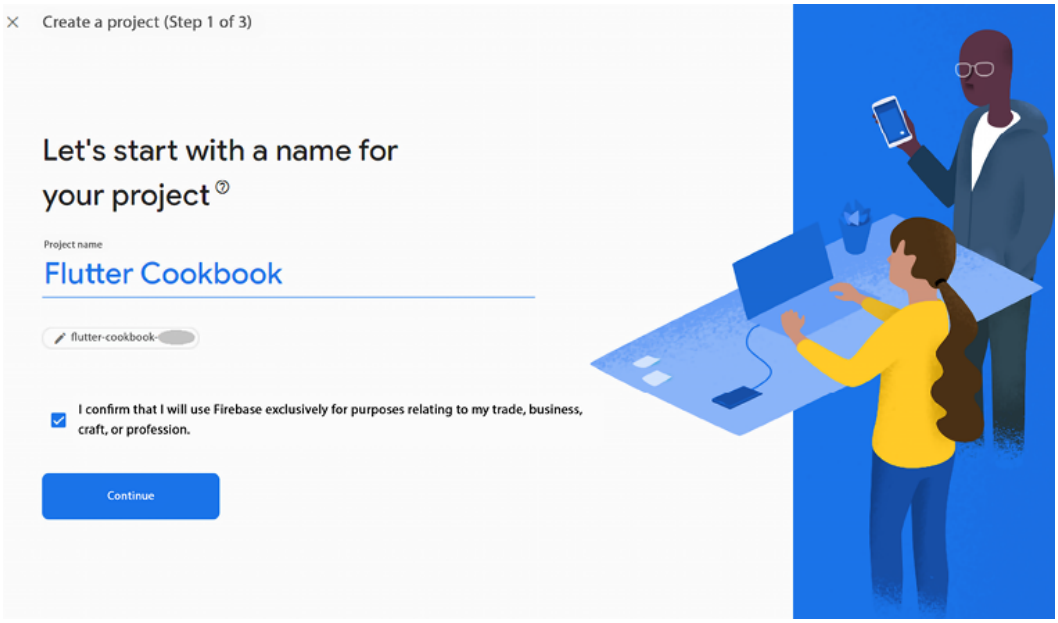


Figure 13.1: Creating a Firebase project

5. Click the **Continue** button.
6. On the **Step 2** page, make sure the **Google Analytics** feature is enabled.
7. On the **Step 3** page, select the analytics location (the closest to your and/or your user's location is recommended) and if necessary accept the required terms.
8. Click the **Create Project** button and wait until the project has been created.
9. Click the **Continue** button. You should land on the Firebase project's overview page, which should look similar to *Figure 13.2*:

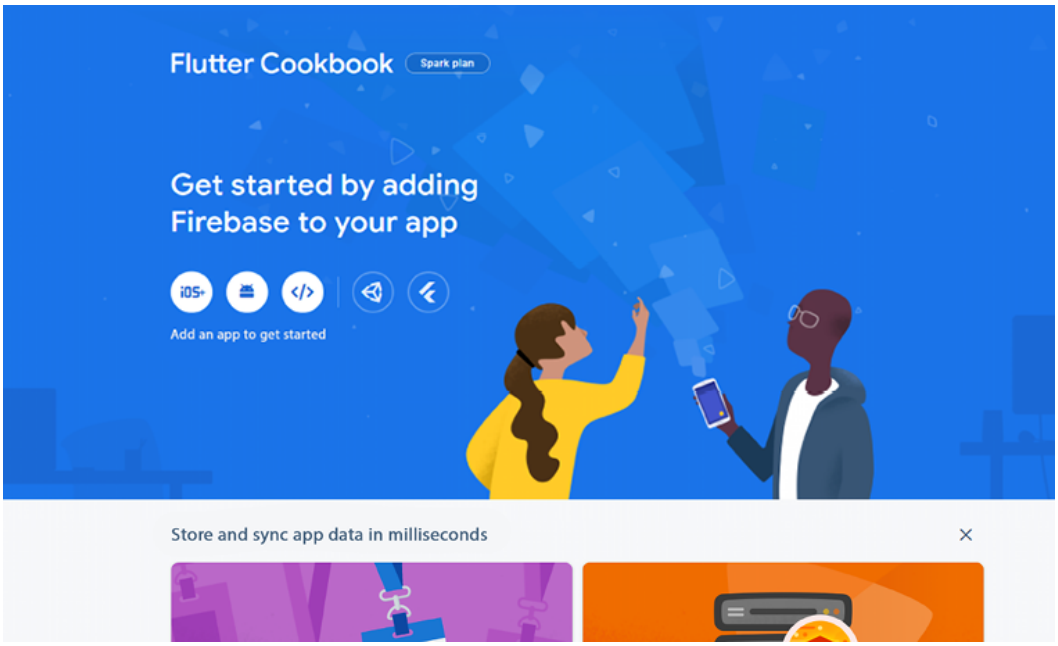


Figure 13.2: The Firebase Project overview page

Your Firebase project is now ready. Now you need to configure a Flutter app so that it can connect to the new project. There is a tool that simplifies the process of adding Firebase to your Flutter application: it's the **FlutterFire command-line interface (CLI)** tool, built on top of the Firebase CLI. In order to configure your app:

1. Create a new Flutter app and call it `firebase_demo`.
2. There are different ways to install the Firebase CLI:
    1. An easy way to install the Firebase CLI is by using **npm** (**Node Package Manager**). If it's not already installed in your system, you can get it by installing `node.js` from https://nodejs.org/. Once npm is available in your system, open a Terminal window and type:

        ```
        npm install -g firebase-tools
        ```

    2. Please note that on a Mac or Linux, this command may need to be run with admin privileges.

    3. What happens is that you leverage npm to install the Firebase CLI globally (that's the `-g` option in the command), so that you can call it from any directory in your Terminal.
    4. Alternatively, you can download and install the Firebase CLI as a standalone program: go to the Firebase CLI reference page at https://firebase.google.com/docs/cli, and in the **Installing** section, choose your developing machine operating system (currently Windows, macOS, or Linux) and follow the steps outlined for it.

3. The next step is installing and activating `flutterfire_cli`. From your Terminal, run the command:

```
dart pub global activate flutterfire_cli
```

You now need to add the path to the `flutterfire_cli` executable to the system path.

On Windows:

1. Open the **Environment variables** window.
2. Click on the **Edit the system environment variables** option.
3. Click on the **Environment Variables** button.
4. In the **System Variables** section, find the variable named `Path` and click on **Edit**.
5. Click on **New** and add the path to the `flutterfire_cli` executable. For example: **C:\Users*your_username*\AppData\Local\Pub\Cache\bin**.
6. Click the OK button to save the new path. You may have to restart your machine.

On macOS:

1. Open your Terminal.

2. Open or create the `~/.bash_profile` file, or `~/.zshrc` if you're using `zsh`, using your favorite text editor.
3. Add the following line to the file, **replacing the path below with the correct path** to the `flutterfire_cli` executable in your system:

```
export PATH="$PATH:/path/to/flutterfire_cli/
```

4. Save the changes and exit your text editor.
5. Close the Terminal and reopen it for the changes to take effect.

This completes the configuration of the `flutterfire_cli` tool in your system.

Adding Firebase dependencies

This final part should be completed in your project's directory. To install the required dependencies for your app, follow the steps below:

1. Open your Terminal, in the project's root directory.
2. To add the `firebase_core` plugin to your app, run the command:

```
flutter pub add firebase_core
```

3. Next, also add `firebase_auth` with the instruction:

```
flutter pub add firebase_auth
```

4. Open the project's `pubspec.yaml` file, and note that the `firebase_core` and `firebase_auth` dependencies have been added to the project.
5. Make sure you are logged in to Firebase by running the command:

```
firebase login
```

6. If you are **not** already logged in to Firebase, a browser window will open, asking for your credentials. Insert them and wait for the success message.

> Note
>
> Sometimes even if the Firebase login command shows that you are already connected, some information about your credentials may be outdated, or expired. In this case, the next step (7) will return an error. When this happens, try to run the `command firebase logout`, and then `firebase login again`.

7. Finally, complete the configuration of your app by running the command:

```
flutterfire configure
```

8. If more than one project is shown in the project's list, choose the `Flutter Cookbook` project using the arrows on your keyboard, and then press *Enter*.
9. Select the platform(s) where you'll be running this app, as shown in *Figure 13.3*, and press *Enter*.

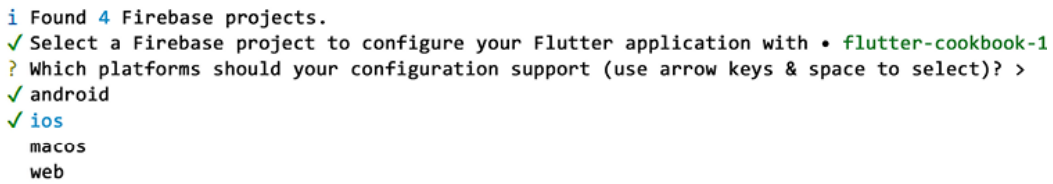


Figure 13.3: Firebase platforms configuration

10. Wait for the success message. You have now configured your app so that you can now use Firebase.

How it works...

Firebase provides several plugins that allow developers to easily integrate Firebase services into Flutter projects.

These are the main steps required to integrate Firebase into a Flutter app:

- **Creating a Firebase project**: This is the top-level entity for Firebase. Each feature you add to your apps belongs to a Firebase project.
- When you create a new project in Firebase, you'll need to choose a **project name**. This is your project's identifier and is on the **Firebase Project Overview** page, which contains the name of the project and the billing plan. It's here that you add all the other features to your app.
- **Adding the dependencies to the `pubspec.yaml` file**: These include `firebase_core`, required for all projects that connect to Firebase, and any other service you require in your app. In this recipe you also added `firebase_auth`, needed for authentication.
- **Running the `flutterfire configure` command**: This automates many of the steps required to integrate Firebase into your Flutter app.
- When you select the target platforms, for each platform the `flutterfire configure` command creates a new app in your Firebase project.
- It also creates a configuration file, called `firebase_options.dart`, in the project's lib directory.

Enabling the analytics feature is a requirement for the Firebase Analytics integration, which will be completed in a later recipe in this chapter: *Integrating Firebase Analytics*.

Note

Firebase is free for apps with small traffic, but as your app grows and requires more power, you may be asked to pay, based on your app's requirements. For more details on the Firebase pricing structure, take a look at the following page: https://firebase.google.com/pricing.

When you include Firebase in your project, you always need the `firebase_core` package. Then, based on the Firebase services you use in your app, you will need specific packages for the features you use. For example, `Cloud Storage` requires the `firebase_storage` package, and the `Firestore` database requires the `cloud_firestore` package.

See also

For a full and updated list of the available Firebase services and packages available for Flutter, take a look at the following page: https://firebase.google.com/docs/flutter/setup#available-plugins.

Creating a login screen

One of the most common features that apps require when connecting to a backend service is an **authentication (login) form**. Firebase makes creating a secure user login form extremely easy, using the `FirebaseAuth` service.

In this recipe, you will create a login form that uses a *username* and a *password* to authenticate the user. But instead of creating the user interface manually, we'll use the `firebase_ui_auth` package to build our login screen.

Getting ready

To follow along in this recipe, you need to complete the previous recipe, *Configuring a Firebase app*.

How to do it...

In this recipe, you will add authentication with a username and password, and you will allow users to sign up and sign in. Perform the following steps:

1. From the Firebase console at https://console.firebase.google.com/, open your project and go to the **Authentication** option inside the

Build section of the project's menu, as shown in *Figure 13.4*. Then, click on the **Get Started** button.

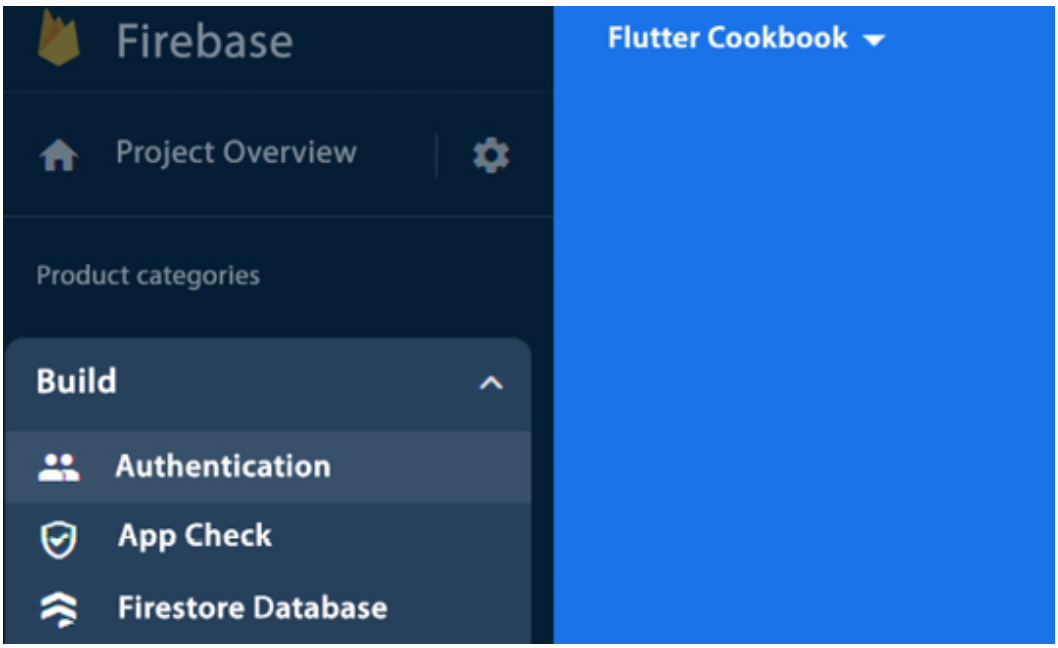


Figure 13.4: The Firebase Authentication menu item

2. Click on the **Sign-in method** tab.
3. Enable the **Email/Password** authentication method, and leave the **Email link** option disabled, as shown in *Figure 13.5*, and click **Save**.

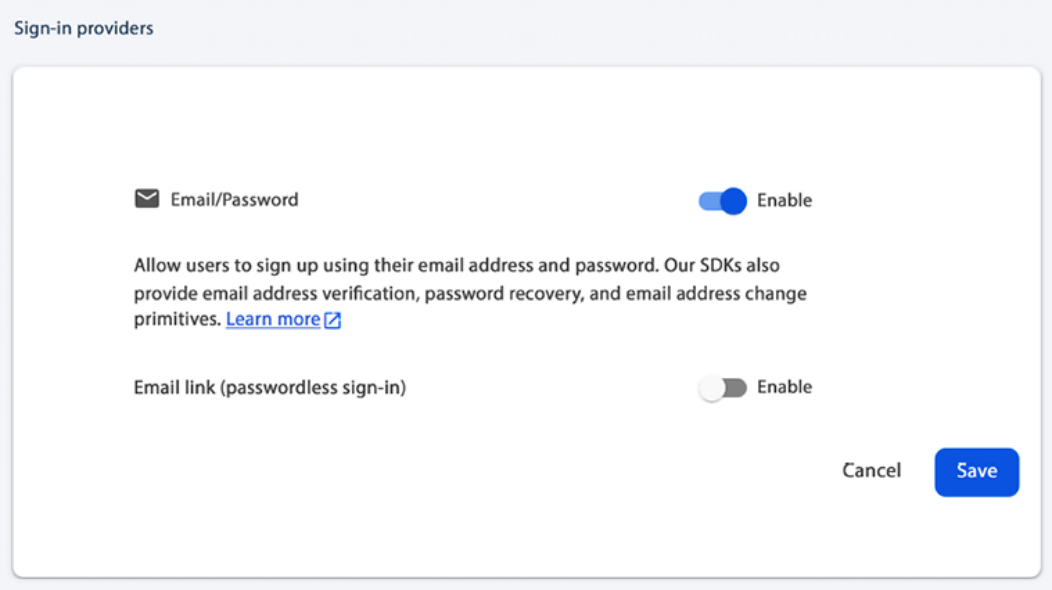


Figure 13.5: The Firebase e-mail sign-in provider configuration page

4. Get back to your Flutter project, and open the `main.dart` file.
5. Delete the `MyHomePage` class and remove the comments from the `MyApp` class.
6. In the home property of the `MaterialApp` in the build method, name a widget `AuthenticationScreen`, which you will create later:

```
home: const AuthenticationScreen(),
```

7. At the top of the `main.dart` file, import the `firebase_core` library and the `firebase_options.dart` file, which we'll use to initialize Firebase:

```
import 'package:firebase_core/firebase_core.dart';
import 'firebase_options.dart';
```

8. Mark the `main()` method as `async`:

```
Future main() async {
  runApp(const MyApp());
}
```

9. In the `main()` method, before calling `runApp()`, call the `WidgetsFlutterBinding.ensureInitialized()` method.
10. Next, await `Firebase.initializeApp`, specifying the options property and setting it to `DefaultFirebaseOptions.currentPlatform`. This will initialize Firebase:

```
Future main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp(
    options: DefaultFirebaseOptions.currentPlatform,
  );
  runApp(const MyApp());
}
```

11. In the project's lib directory, create a directory called `screens`.
12. In the `screens` directory, create a new file called `authentication_screen.dart`.
13. At the top of the new file, import the `material.dart` and `firebase_auth` libraries:

```
import 'package:flutter/material.dart';
import 'package:firebase_auth/firebase_auth.dart';
```

14. Create a stateless widget using the `stless` shortcut and call it `AuthenticationScreen`:

```
class AuthenticationScreen extends StatelessWidget {
  const AuthenticationScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

15. In the build method of the widget, return a `StreamBuilder`, of type `User` (nullable).
16. In the `StreamBuilder`, add a `builder`: this takes a `BuildContext`, and a `snapshot`:

```
return StreamBuilder<User?>(
  stream: FirebaseAuth.instance.authStateChanges(),
  builder: (context, snapshot) {
     return Container();
  });
```

17. Add the `firebase_ui_auth` package in your Terminal:

```
flutter pub add firebase_ui_auth
```

18. In the `main.dart` file, import this new screen and `firebase_ui_auth.dart`:

```
import 'screens/authentication_screen.dart';
import 'package:firebase_ui_auth/firebase_ui_auth.dart';
```

19. In the `main()` method, before calling `runApp`, configure the `FirebaseUiAuth` providers:

```
FirebaseUIAuth.configureProviders([
    EmailAuthProvider(),
 ]);
 runApp(const MyApp());
```

20. Set `AuthenticationScreen` as the home for the `MaterialApp`:

```
home: const AuthenticationScreen(),
```

21. Before moving on, run the app to make sure everything is working.

> **Note**
>
> If you target an Android device, you might get an error message telling you that you need a higher Android SDK version. In case this happens, in the `Android` folder, open the `local.properties` file, and add the instruction:
>
> ```
> flutter.minSdkVersion=21
> ```

Now in the `android/app` folder, open the `build.gradle` file, and in the `minSdkVersion` key, set:

```
localProperties.getProperty('flutter.minSdkVersion').toIntege
r()
```

In the next few steps, you'll leverage FirebaseUI Auth to create the login screen for the app:

1. From a Terminal window, run the following command:

```
flutter pub add firebase_ui_auth
```

2. In `authentication_screen.dart`, import `firebase_ui_auth.dart`:

```
import 'package:firebase_ui_auth/firebase_ui_auth.dart';
```

3. In the builder of the `StreamBuilder`, check whether the snapshot has some data; if the user has not logged in, this will return false. In this case return a `SignInScreen`, which is part of the `firebase_ui_auth` library:

```
if (!snapshot.hasData) {
   return SignInScreen();
}
```

4. Run the app.
5. Note the screen that appears; it should look as shown in *Figure 13.6*:

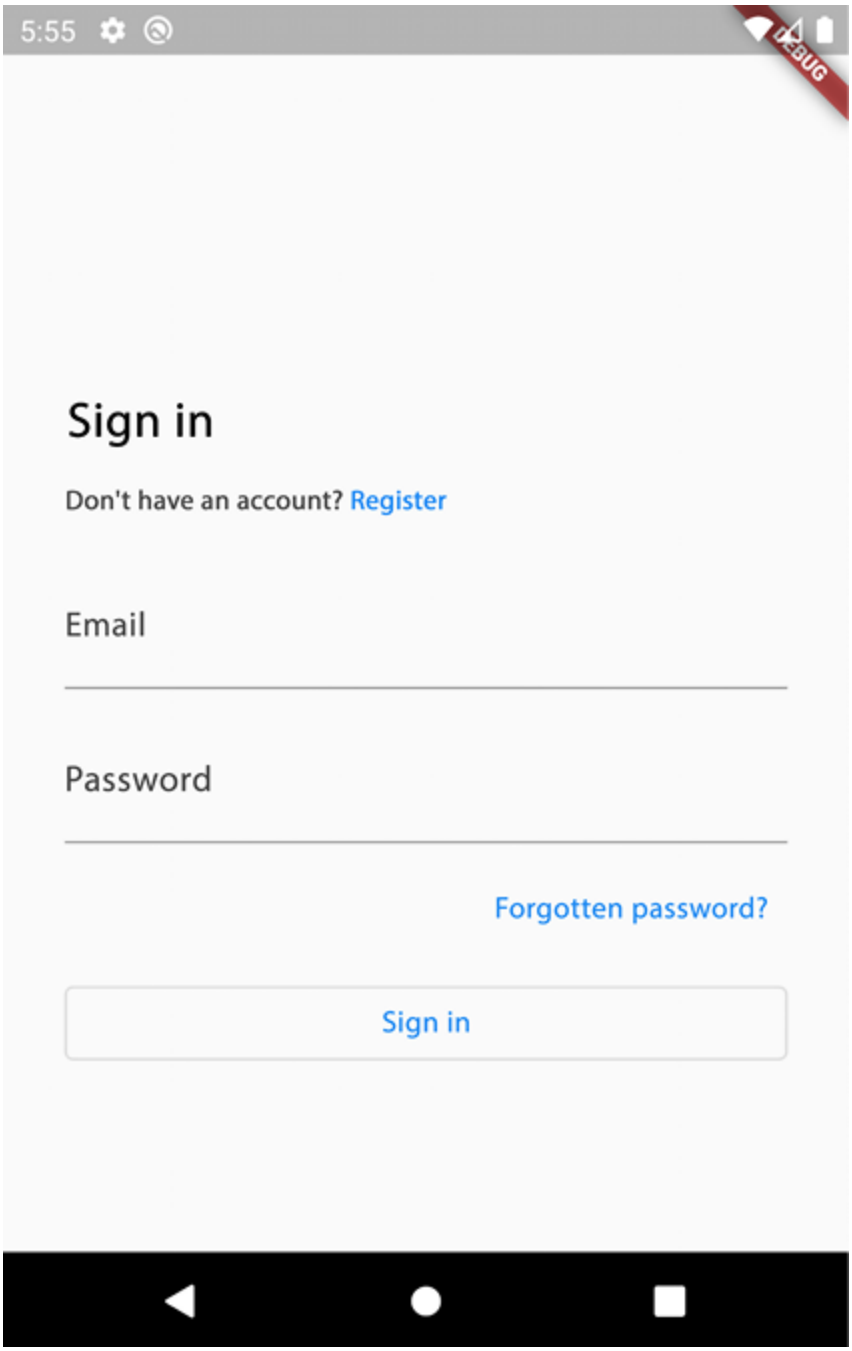


Figure 13.6: The FirebaseUI Auth Sign in screen with Email and Password

6. Create a new user with the **Register** button. At the end of the process you should see a black screen.
7. Stop the app, and run it again. Note that the sign-in screen will not appear, and you will see the black screen again. Stop the app.

8. On your device, uninstall the `firebase_demo` app.

> Note
>
> By default, Firebase Authentication keeps your user logged in after authentication, so in order to delete the authentication information you can uninstall your app, or delete its data.

9. Run the app again.
10. You should now see the sign-in screen. Click on the **Forgot password** button and enter the email you just used in the registration in *step 6* of the current list, then click **Reset password**.
11. In your mailbox, note the reset password message. There is no need to reset your password at this time.
12. Back to the app, sign in with the email and password that you chose. You'll get back to the black screen.
13. Stop and uninstall the app again.

How it works...

I truly hope you like the result you achieved in this recipe. It still astonishes me how easy it is to build fully functional sign-in, sign-up, and forgot password screens, plus all the related backend features with Flutter and Firebase!

In this recipe, you used two of the tools Firebase provides for Flutter: Firebase Authentication and FirebaseUI.

Firebase Authentication includes several ways to provide authentication to your apps. Among them, you find authentication with a username and password, or third-party providers such as Google, Microsoft, and several others. In this recipe, you created a screen that leverages Firebase Authentication with an email and password to provide login, sign-up, and forgotten password features.

In order to leverage Firebase Authentication, you need to enable the method or methods you want to use. You choose the providers you want to enable from the Firebase console; in the **Build** section of the console, you find the **Sign in methods** page.

All authentication methods are disabled by default. So the first step you performed in this recipe was enabling the email/password authentication method. As the name implies, this provider allows you to sign up and log in with an email and a password.

Once the authentication method was enabled, you leveraged `firebase_ui_auth` to create the user interface in `authentication_screen.dart`.

FirebaseUI is a library that contains several pre-built UI components that you can use in your apps to implement Firebase authentication and other Firebase services. In this case you could use a login, registration, and forgotten password screen without writing any code.

Note the instructions:

```
if (!snapshot.hasData) {
   return SignInScreen();
}
```

In this case, the snapshot object contains the current state of the authentication session.

In this code snippet, if `!snapshot.hasData` is true, it means that the user is not authenticated. In this case, the `return SignInScreen()` statement is executed. This displays a FirebaseUI sign-in screen.

By returning `SignInScreen()` when the user is not signed in, you make sure that the user is always redirected to the sign-in screen until they successfully authenticate. Once the user logs in successfully, the app's main content is displayed instead. In this case we just have an empty Container, but we will fix this in the next recipes in this chapter.

As you saw while using the app, when you call `SignInScreen()` you get the registration screen, the login screen, and the forgotten password screen, with the related navigation implementation. All features and links to Firebase are added automatically.

Now, you might wonder where user data is stored in Firebase. To see the user data you have added, go to the Firebase console and then, from the **Build** section, open the **Authentication** link. There you will find all your users, as shown in *Figure 13.7*:

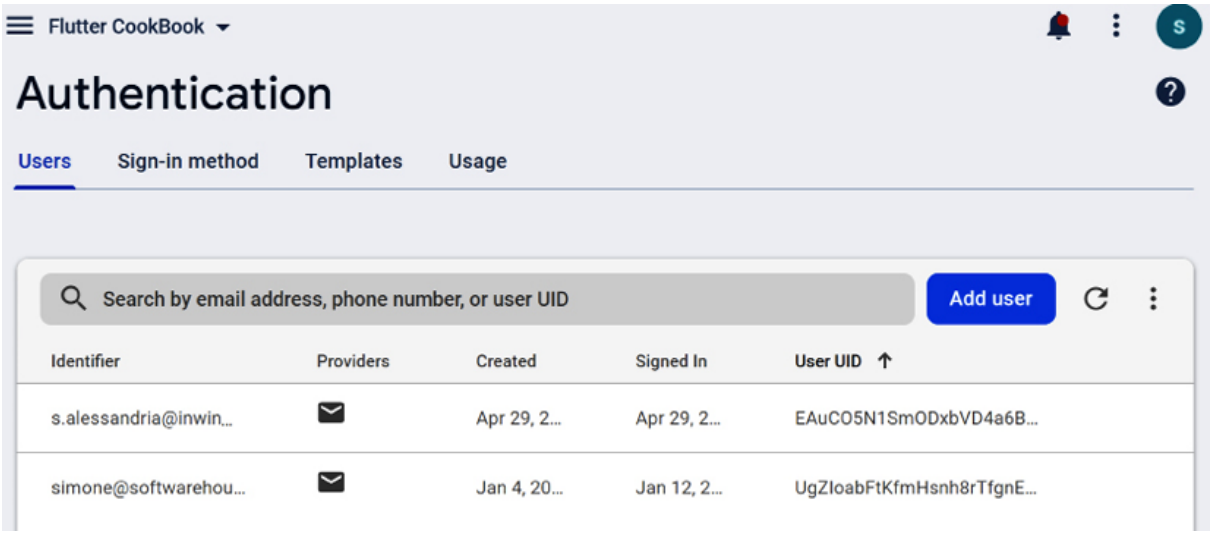


Figure 13.7: Firebase users

See also

For a full description of the features you can leverage in your apps with Firebase authentication, have a look at the official documentation available at https://firebase.google.com/docs/auth?hl=en.

Adding Google Sign-in

While dealing with user authentication with a custom user ID and password is quite common, several users prefer using a single authentication provider to access multiple services. One of the great features you can leverage when

using Firebase authentication is that you can easily include authentication providers, such as Google, Apple, Microsoft, and Facebook.

In this recipe, you will see how to add Google authentication to your app.

Getting ready

To follow along with this recipe, you need to complete the previous two recipes, *Configuring a Firebase app* and *Creating a login screen*.

How to do it...

In this recipe, you will make the Google Sign-in service available to your users.

The first task required to add third-party sign-in features is to enable them in your Firebase project:

1. From the Firebase console, in the **Authentication** page on your project, get to the **Sign In method** page.
2. In the **Sign-in providers** table, click the **Add new provider** button, and note the third-party providers that are available for your projects.
3. Click on **Google**.
4. Enable Google Sign-in and set a name for your project, as shown in *Figure 13.8*:

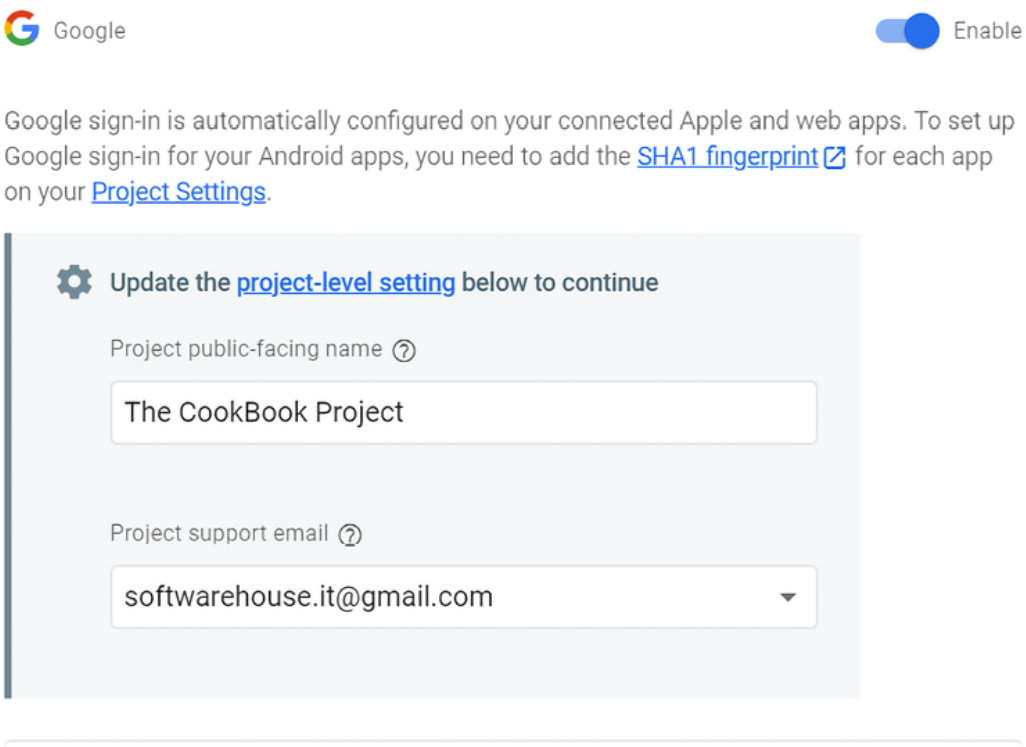


Figure 13.8: Enabling Google Sign-in

5. If using Android, notice that there may be a message, telling you that *you need to add the SHA1 fingerprint for each app on your Project Settings*. Click on the **SHA-1** link (or go to the page https://developers.google.com/android/guides/client-auth?authuser=0). The procedure to obtain an SHA-1 fingerprint varies based on your system. Follow the instructions on the page to obtain an SHA-1 key.
6. Once you have the SHA-1 key, get back to the **Project Overview** page, enter the project settings page, and then in your apps section click the **Add fingerprint** button, as shown in the following screenshot:



Figure 13.9: Certificate fingerprint

7. Insert the SHA1 fingerprint and then click the **Save** button.

> **Note**
>
> You can obtain the SHA1 fingerprint with the `keytool` command. This requires Java and the creation of a keystore. For more information about how to create a keystore, see https://flutter.dev/docs/deployment/android.

8. You need to repeat the process for each app in your project.
9. Back in the Flutter project, add the `google_sign_in` package to your project by typing in your Terminal the command:

```
flutter pub add google_sign_in
```

10. Add the `firebase_ui_oauth_google` package to your project by typing in your Terminal the command:

```
flutter pub add firebase_ui_oauth_google
```

11. Run the `flutterfire configure` command in your Terminal again to update the project's Firebase settings.
12. In `main.dart`, make sure you import `firebase_ui_auth.dart` and `firebase_ui_oauth_google.dart` (you should remove `firebase_auth.dart` as this may generate a conflict for the `EmailAuthProvider`):

```
import 'package:firebase_ui_auth/firebase_ui_auth.dart';
import
'package:firebase_ui_oauth_google/firebase_ui_oauth_goog
le.dart';
```

13. Add `GoogleProvider` to the `configureProviders` method:

```
FirebaseUIAuth.configureProviders([
    EmailAuthProvider(),
    GoogleProvider(clientId: 'Your client ID here'),
]);
```

> **Note**
>
> You can easily retrieve your client ID in the `firebase_options.dart` file: it's in the `appId` key in the `FirebaseOptions` object for your target device.
>
> As an alternative, consider retrieving the `clientId` with:
>
> ```
> clientId: DefaultFirebaseOptions.currentPlatform.appId
> ```

14. Run the app. You should see a screen similar to the screenshot in *Figure 13.10*:

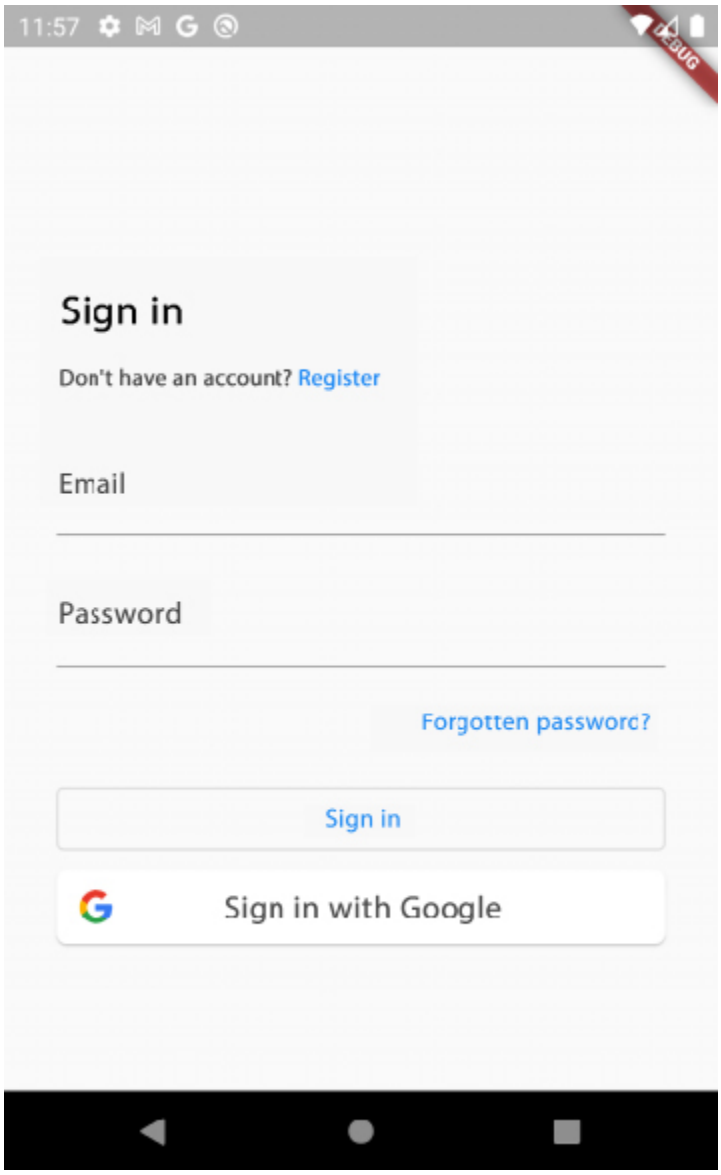


Figure 13.10: The Sign in with Google button added to the screen

15. **Click the Sign in with Google** button. You should be able to use your Google account to log in.
16. Uninstall the app.

How it works...

In this recipe, you performed two tasks: the configuration of the app to enable Google Sign-in, and the actual code to use Google to log in your

users.

The first step involved activating Google Sign-in from the Firebase authentication page. All sign-in methods are disabled by default, so when you want to use a new method, you need to activate it explicitly.

Some services, including Google Sign-in, require the SHA-1 fingerprint of your app's signing certificate. One of the possible ways to get the SHA-1 fingerprint is by using `Keytool`. This is a command-line tool that generates public and private keys and stores them in a Java keystore.

The tool is included in the Java SDK, so you should already have it if you have configured your Android environment or the Java SDK. In order to use it, you need to open your Terminal and reach the `bin` directory of your Java SDK installation (or add it to your environment paths).

This completed the configuration tasks for the **Firebase** project.

In the **Flutter** project, the first step you performed was importing the `google_sign_in`. As the name implies, this package enables Google Sign-in in your app. You also need the `firebase_ui_oauth_google` package. This is used to implement Google Sign-In for FirebaseUI Auth.

Next, you need to add to the `FirebaseUIAuth.configureProviders()` method the `GoogleProvider`. This requires setting a client ID to allow identification of your app. Once the new provider has been added, the **Sign in with Google** button will appear at the bottom of the Sign in screen.

This is all that's required to add Google Sign-in to your apps.

See also

To have a look at the sign-in configuration for other providers, have a look at: https://firebase.google.com/docs/auth/flutter/federated-auth?hl=en.

Customizing Sign in

While having fully functional sign-in features with very little code is a great achievement in itself, in most cases you will need to customize your login and registration user interfaces, for example, by adding a logo or some text.

In this recipe, you will add an image, a title, and a footer to your sign-in screens.

Getting ready

To follow along with this recipe, you need to complete the previous recipes in this chapter.

How to do it...

First you will customize the sign-in by adding a logo and some text:

1. Download a free image for your project (you can use https://bit.ly/logo_ico).
2. At the root of your project, create a new folder called `assets`.
3. Save the image you have downloaded, and rename it `logo.jpg`.
4. In your `pubspec.yaml`, uncomment the assets section, and add the `assets` folder as a source:

```
  assets:
    - assets/
```

5. In the `authenticationscreen.dart` file, in the build method, set the `headerBuilder` of the `SignInScreen` widget:

```
       return SignInScreen(
            headerBuilder: (context, constraints,
shrinkOffset) {
              return Padding(
                padding: const EdgeInsets.all(16),
                child: Image.asset('assets/logo.jpg'),
              );
```

```
            },
        );
```

6. Run the app, and note that the logo image now appears both in the **Sign in** and **Register** screens (*Figure 13.11*).

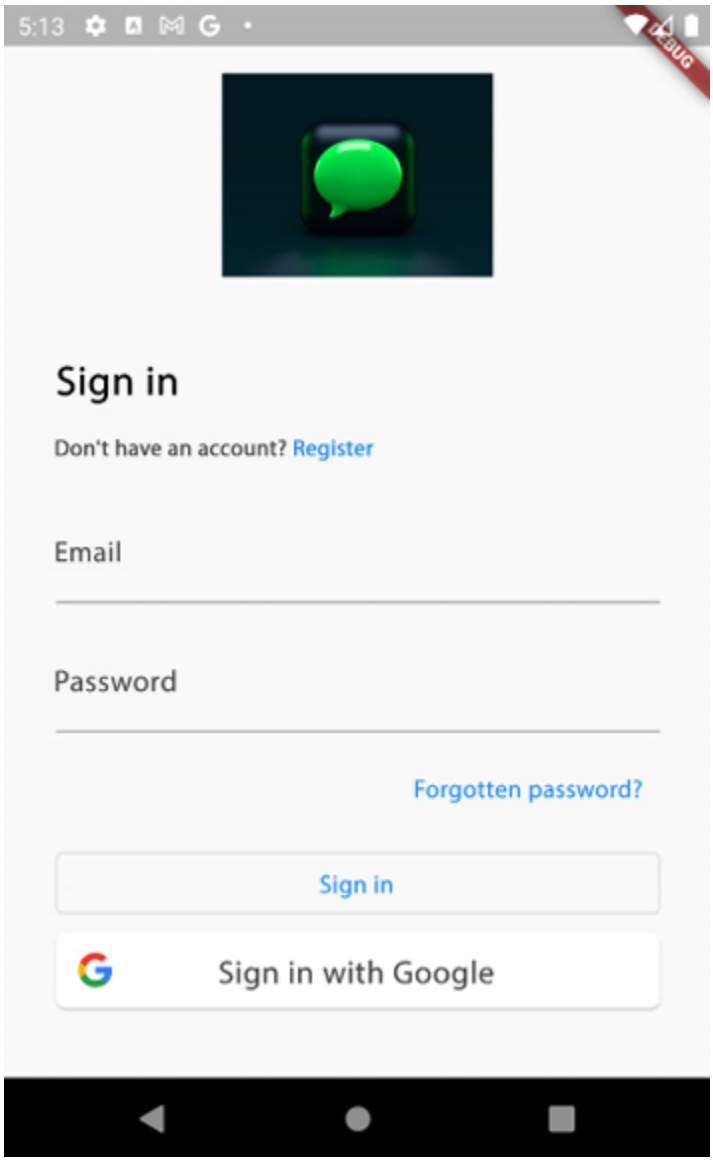


Figure 13.11: Logo added to the Sign in screen

7. Add a `subtitleBuilder` to the `SignInScreen`, showing a subtitle that depends on the current action of the user (sign in or register):

```
subtitleBuilder: (context, action) {
  return Padding(
    padding: const EdgeInsets.only(bottom: 16),
      child: Text(
        action == AuthAction.signIn
           ? 'Welcome to the Cookbook App! Sign in to
continue.'
           : 'Welcome to the Cookbook App! Create an
account' +
               'to continue',
            ),
          );
     },
```

8. Add a `footerBuilder` to the `SignInScreen`, showing the text **Powered by me!**:

```
footerBuilder: (context, _) {
                return const Padding(
                  padding:
EdgeInsets.symmetric(vertical: 32),
                  child: Text(
                    'Powered by me!',
                    style: TextStyle(color:
Colors.grey),
                  ),
                );
              },
```

9. Run the app; you should see your screen with a subtitle and a footer, as shown in *Figure 13.12*:

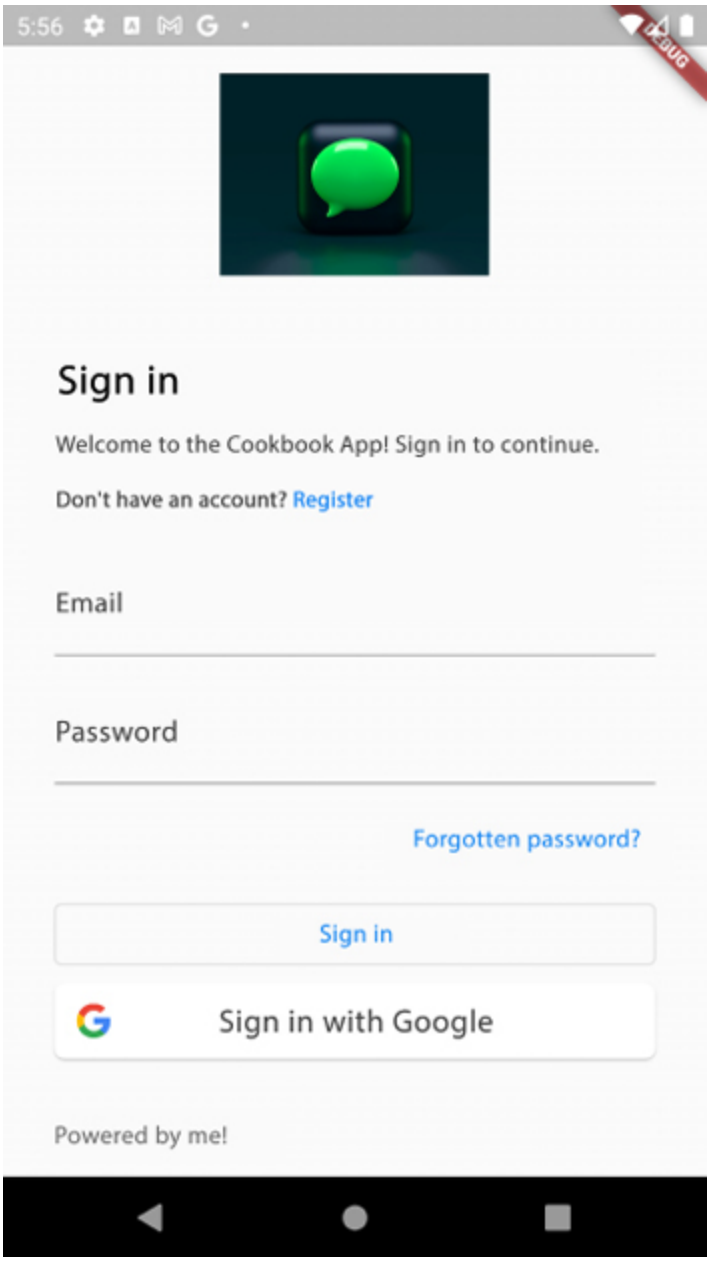


Figure 13.12: Subtitle and footer added to the Sign in screen

How it works...

The `SignInScreen` widget has properties that allow you to customize your screens; in particular, in this recipe, you used three properties: `headerBuilder`, `subtitleBuilder`, and `footerBuilder`.

You used the `headerBuilder` to add an image at the top of the screens. This property is a function that takes in three arguments: the context of the widget, the constraints of the screen, and the `shrinkOffset`. It returns a widget that is displayed at the top of the sign-in screen. In this case, it returns a `Padding` widget with an `Image.asset` widget that displays the image that you put in the assets directory.

In order to add a subtitle to your sign-in, you use the `subtitleBuilder`. This takes a `BuildContext` and an `AuthAction`. An `AuthAction` is an enumerable that contains the authentication actions. In particular, `AuthAction.signIn` tells us that the user is trying to log into the app, and `AuthAction.signUp` tells us that the user is creating a new account. The parameter passed to the function contains the action that's happening on the current screen. In this example you used the action for a ternary operator. When the action is `AuthAction.signIn`, then the text will contain "**Welcome to the Cookbook App! Sign in to continue**", otherwise "**Welcome to the Cookbook App! Create an account to continue**".

You use the `footerBuilder` property of the `SignInScreen` widget to display a footer at the bottom of the screen.

This can contain any additional information or links that you, as a developer, want to display, like copyright information, terms of service, or links to social media accounts. In this case you just added a `Text` with "**Powered by me**".

Integrating Firebase Analytics

One of the most relevant features of Firebase is getting real feedback about how your app is used, and for that, you can use Firebase Analytics.

Firebase Analytics is an incredibly powerful tool, easy to set up and use, and can give you invaluable information about your users and your app. In this recipe, you will set up a custom event and log it into Firebase Analytics.

Getting ready

To follow along with this recipe, you need to complete the previous recipes in this chapter.

How it works...

You will now add `firebase_analytics` to your app and activate a custom event:

1. In a Terminal window, in your project's directory, add the dependency to Firebase Analytics:

```
flutter pub add firebase_analytics
```

2. In the `screens` folder in your project, create a new file, called `happy_screen.dart`.
3. At the top of the new file, import `material.dart` and `firebase_analytics`:

```
import 'package:flutter/material.dart';
import
'package:firebase_analytics/firebase_analytics.dart';
```

4. Under the `import` statements, create a new stateful widget, calling it `HappyScreen`:

```
class HappyScreen extends StatefulWidget {
  const HappyScreen({super.key});
  @override
  State<HappyScreen> createState() =>
_HappyScreenState();
}
class _HappyScreenState extends State<HappyScreen> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

5. In the `build` method of the `_HappyScreenState` class, return a `Scaffold`, with an `appBar` and a `body`. The latter contains a `Center` widget, and its child is `ElevatedButton`. The code is shown here:

```
return Scaffold(
  appBar: AppBar(title: const Text('Happy Happy!'),),
  body: Center(
    child: ElevatedButton(
      child: const Text('I\'m happy!'),
      onPressed: () {},
    ),),
);
```

6. In the `onPressed` parameter of the `ElevatedButton`, add the `FirebaseAnalytics logEvent` method, passing `'Happy'` as the name of the event:

```
onPressed: () {
  FirebaseAnalytics.instance.logEvent(name: 'Happy');
},
```

7. Open the `authentication_screen.dart` file and, at the top of the file, import `happy_screen.dart`:

```
import './happy_screen.dart';
```

8. In the `build()` method, in the `else` statement of the `if` where you check `!snapshot.hasData`, return `HappyScreen`:

```
return const HappyScreen();
```

The next few steps apply to Android devices only. Instructions to run analytics for iOS follow after these Android steps:

1. Open the `android/build.gradle` file, and in the dependencies node, add the current version of `google-services` (you can check the current version at https://developers.google.com/android/guides/google-services-plugin):

```
classpath 'com.google.gms:google-services:4.3.15'
```

2. Open the `android/app/build.gradle` file, and under the other "**apply plugin**" instructions, add the plugin for `google-services`:

```
apply plugin: 'com.google.gms.google-services'
```

3. Open a Terminal window and run the following command:

```
adb shell setprop debug.firebase.analytics.app [your
app name here]
```

> **Note**
>
> The name you should add to the `adb` command is the full name of your app, including the domain name that you set in your Firebase project (in other words, `com.example.yourappname`).

In order to enable debug for iOS devices:

1. Open Xcode, and from there your iOS Runner project.
2. From the menu, choose **Product** > **Scheme** > **Edit Scheme**.
3. In the **Arguments Passed On Launch** section, add `-FIRDebugEnabled` and `-FIRAnalyticsDebugEnabled`.
4. Clean and build your app.

You can now view the messages sent from your app on the Analytics page:

1. Go to the Firebase console, click the **Analytics** menu, and then select **DebugView**. Leave this page open.
2. Run the app, log in with any method, and then tap a few times on the **I'm happy** button.
3. Wait a few seconds, and then go back to the **Firebase Analytics DebugView**. You should see that the event has been logged, as shown in the following screenshot:

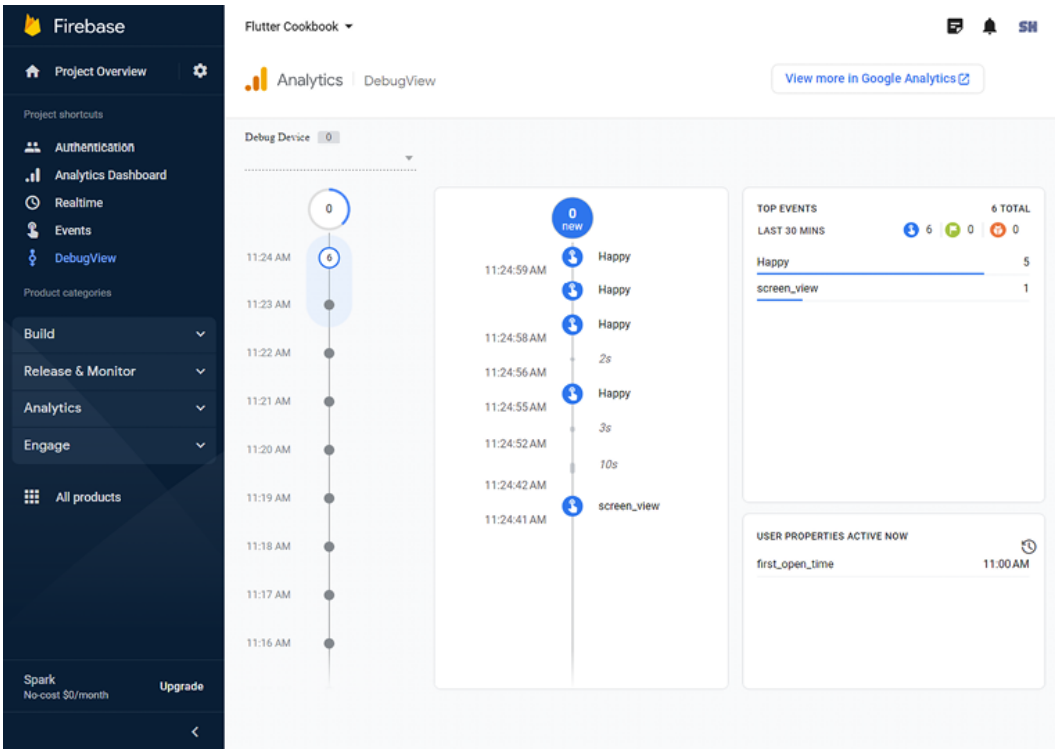


Figure 13.13: Firebase Analytics DebugView

How it works...

Google Analytics for Firebase can give you invaluable information on how your users behave in your app. The great news is that when using Firebase, if you enable Analytics, several statistics are generated automatically for you. These are called **Default Events** and include logging errors, sessions (when your app starts), notifications, and several others.

If you go to the **Analytics** dashboard from the Firebase console, you can see a lot of information about your app and its users, without any action on your part.

> **Note**
>
> You may have to wait up to 24 hours before seeing an event logged in the **Analytics** dashboard. Use **DebugView** to see data in real time.

You can see here a possible outcome of the **Analytics** dashboard in this screenshot:

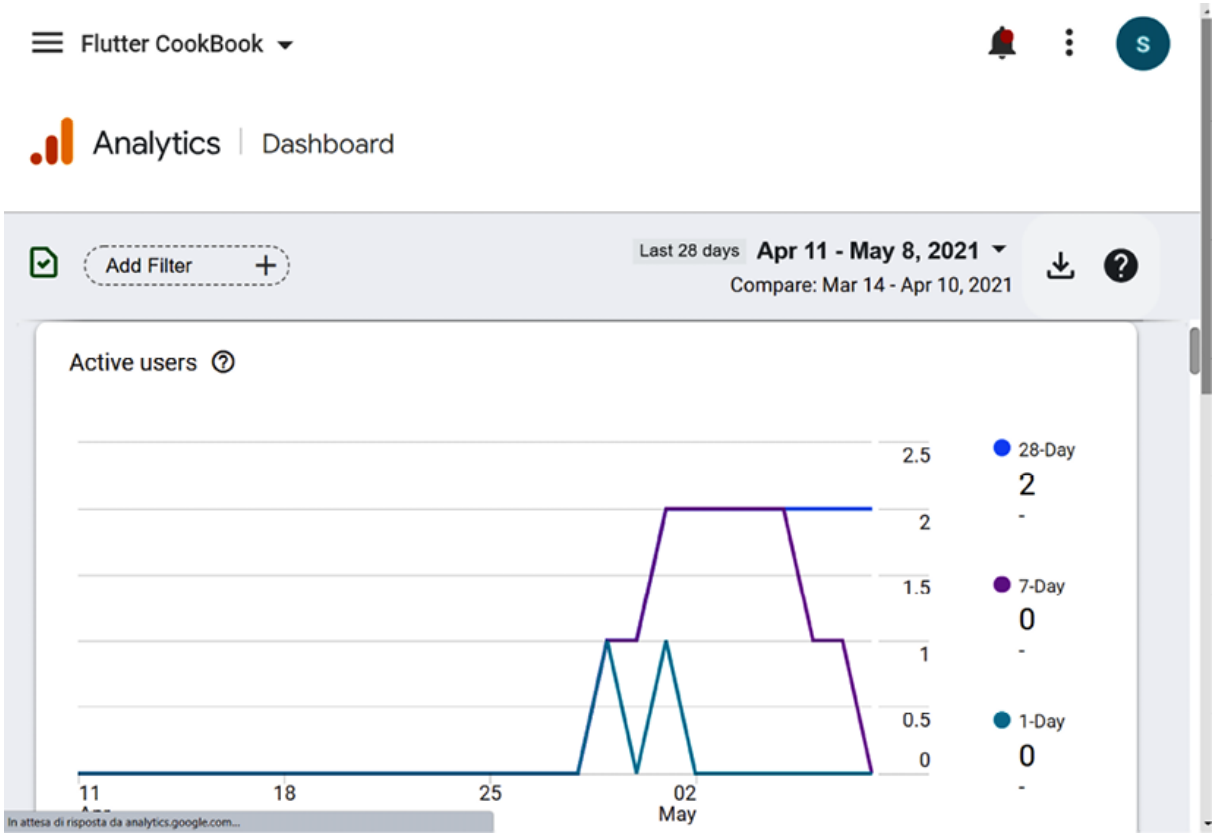


Figure 13.14: Firebase Analytics user statistics

A great feature of Firebase Analytics is that you can capture both pre-built and custom events. In this recipe, we added a custom event called `happy`. This means that you can log virtually anything you consider important within your app.

Using Firebase Analytics is very simple. It requires adding the analytics package to your app, importing it where you want to use it, and then logging the custom events you need. The `logEvent` method adds a new log that will become visible on the analytics page. The command logging the `'Happy'` event in the example of this recipe was:

```
analytics.logEvent(name: 'Happy');
```

When you are developing, it may be important to immediately see what's happening, without waiting the normal 24 hours generally needed before seeing the events in the **Analytics** dashboard. To do that, you enabled **Debug** mode for Android and/or iOS.

By enabling **Debug** mode, and opening the **DebugView** page from the Firebase console, you can immediately see the events that are triggered in your app.

See also

Google Analytics for Firebase can really boost your understanding of how your users behave within your app. For more information about all the statistics and features of the service, see https://firebase.google.com/docs/analytics?hl=en.

Using Firebase Cloud Firestore

When using Firebase, you can choose between two database options: **Cloud Firestore** and **Realtime Database**. Both are solid and efficient NoSQL databases.

Note

In short, you can use both SQL and NoSQL databases to store and manage data.

SQL databases, also called *Relational Databases*, store data in tables and use a **fixed schema**, meaning that columns and data types must be defined before storing any data. You generally prefer SQL databases for data that requires complex queries and tasks that involve joining multiple tables.

NoSQL databases store data in different formats, like key-value pairs, documents, or graphs, and don't require a fixed schema. NoSQL is usually preferred for unstructured or semi-structured data, and simple read/write tasks.

Cloud Firestore is the newer and recommended option in most cases for new projects, so that's the tool you'll be using in this recipe.

Getting ready

To follow along with this recipe, you need to complete the first two recipes in this chapter, *Configuring a Firebase app* and *Creating a login form*.

How to do it...

In this recipe, you will see how to create a Cloud Firestore database and integrate it into your projects. To be more specific, you will ask users whether they prefer ice cream or pizza (pizza is better, by the way), and store the results in your database:

1. From the **Build** menu on the left of the Firebase console, click on the *Firestore Database* menu item. This will bring you to the **Cloud Firestore** page, as shown in *Figure 13.15*:

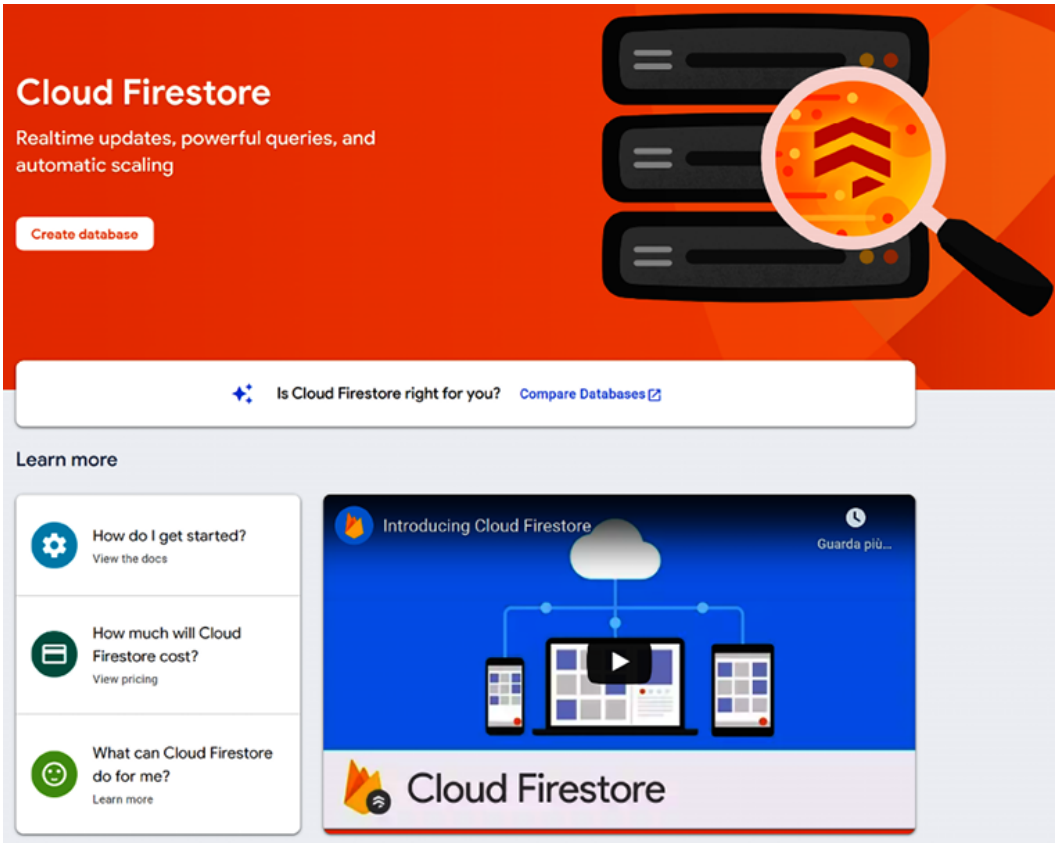


Figure 13.15: The Cloud Firestore page

2. Click on the **Create database** button, choose the **Start in test mode** option, which keeps data open without authorization rules, and then click **Next**.
3. Choose the Cloud Firestore location. Generally, prefer regions that are closer to where you and your users will access data.
4. Click the **Enable** button. After a few seconds, you should see your Cloud Firestore database, as shown in *Figure 13.16*:

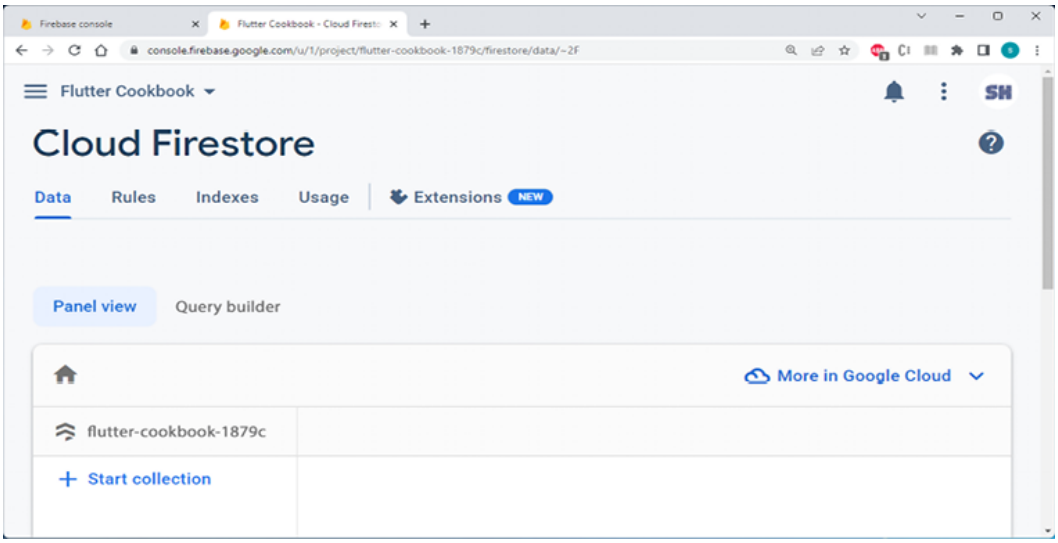


Figure 13.16: The Cloud Firestore Data page

5. Click on the **Start Collection** link.
6. Set the collection ID to `poll` and then click **Next**.
7. For the document ID, click the **Auto-ID** button.
8. Add two fields, one called `icecream`, with a type of `number` and a value of `0`, and another called `pizza`, with a type of `number` and a value of `0`. The result should look like *Figure 13.17*:

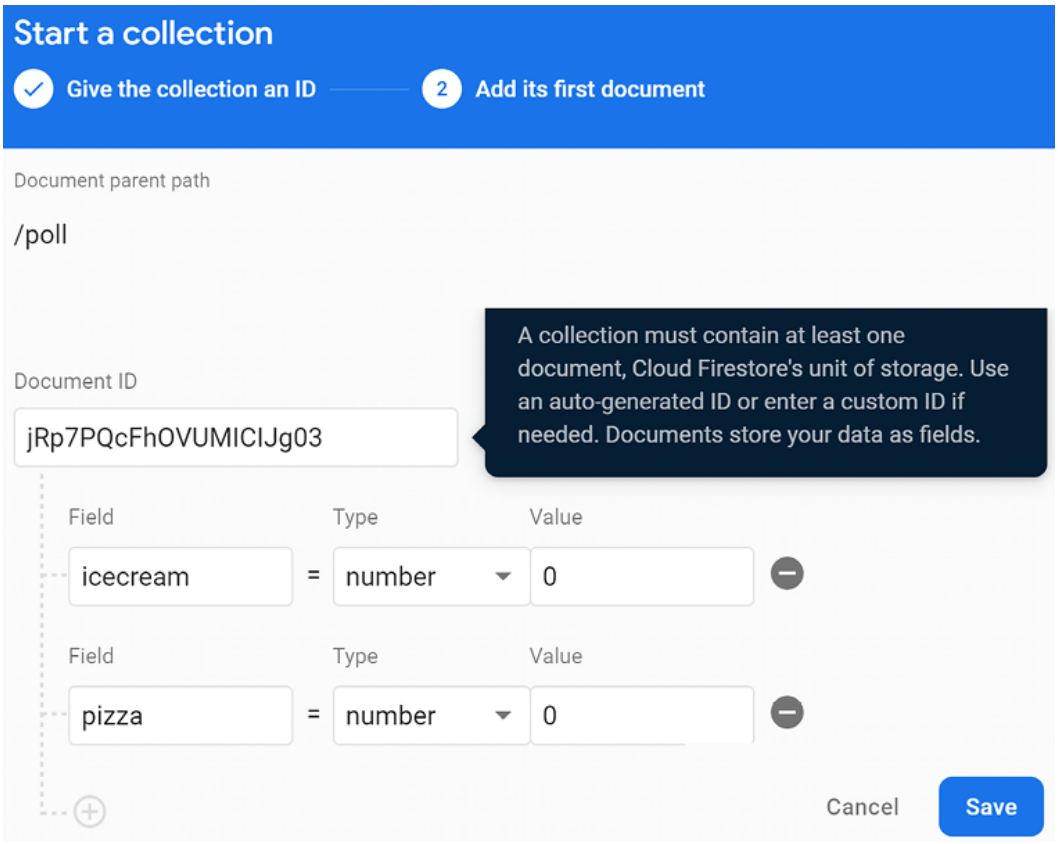


Figure 13.17: Cloud Firestore collection

9. Click the **Save** button.
10. Back in your Flutter project, add the latest version of the `cloud_firestore` package by typing in your Terminal the command:

```
flutter pub add cloud_firestore
```

11. Create a new file in the `screens` folder in your project, called `poll.dart`.
12. At the top of the `poll.dart` file, import the `material.dart` and `cloud_storage.dart` packages:

```
import 'package:flutter/material.dart';
import 'package:cloud_firestore/cloud_firestore.dart';
```

13. Still in the `poll.dart` file, create a new stateful widget, called `PollScreen`:

```
class PollScreen extends StatefulWidget {
    const PollScreen({super.key});
  @override
  State<PollScreen> createState() => _PollScreenState();
}
class _PollScreenState extends State<PollScreen> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

14. In the `build` method of the `_PollScreenState` class, return a `Scaffold`. In the body of the `Scaffold`, place a `Column`, with a `mainAxisAlignement` value of `spaceAround`:

```
return Scaffold(
  appBar: AppBar(
    title: const Text('Poll'),
  ),
  body: Padding(
    padding: const EdgeInsets.all(96.0),
    child: Column(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      children: []
      ),
  ));
```

15. In the `children` property of the `Column` widget, add two `ElevatedButtons`. In each one, place a `Row` as a child, with an `Icon` and `Text`, as shown here:

```
 ElevatedButton(
  child: Row(
    mainAxisAlignment: MainAxisAlignment.spaceAround,
    children: const [Icon(Icons.icecream), Text('Ice-
cream')]),
    onPressed: () {},
  ),
  ElevatedButton(
    child: Row(
    mainAxisAlignment: MainAxisAlignment.spaceAround,
    children: const [Icon(Icons.local_pizza),
Text('Pizza')]),
```

```
    onPressed: () {},
  ),
```

16. At the bottom of the `_PollScreenState` class, add a new asynchronous method, called `vote`, that takes a Boolean parameter called `voteForPizza`:

```
Future<void> vote(bool voteForPizza) async {}
```

17. At the top of the `vote` method, create an instance of `FirebaseFirestore`:

```
FirebaseFirestore db = FirebaseFirestore.instance;
```

18. From the `FirebaseFirestore` instance, retrieve the `'poll'` collection:

```
CollectionReference collection = db.collection('poll');
```

19. Call the asynchronous `get` method on the collection to retrieve a `QuerySnapshot` object:

```
QuerySnapshot snapshot = await collection.get();
```

20. Retrieve a `List` of `QueryDocumentSnapshot` objects, called `list`, by getting the `docs` property of the snapshot:

```
List<QueryDocumentSnapshot> list = snapshot.docs;
```

21. Get the first document on the list and retrieve its `id`:

```
DocumentSnapshot document = list[0];
final id = document.id;
```

22. If the `voteForPizza` parameter is `true`, update the `pizza` value of the document, incrementing it by `1`; otherwise, do the same with the `icecream` value:

```
if (voteForPizza) {
  int pizzaVotes = document.get('pizza');
  collection.doc(id).update({'pizza':++pizzaVotes});
} else {
  int icecreamVotes = document.get('icecream');

collection.doc(id).update({'icecream':++icecreamVotes});
}
```

23. Call the `vote` method, passing `false` from the first `ElevatedButton`, with the text `icecream`:

```
onPressed: () {
  vote(false);
},
```

24. Call the `vote` method, passing `true` from the second `ElevatedButton`, with the text `pizza`:

```
onPressed: () {
  vote(true);
},
```

25. In the `authentication_screen.dart` file, in the `build()` method of the `AuthenticationScreen` class, return `PollScreen` instead of `HappyScreen` (or `Container`) in the `else` block.
26. Run the app, and after logging in, try pressing the two buttons a few times each.
27. Get to the Cloud Firestore page and enter the `poll` collection. You should see the `pizza` and `icecream` values updated with the number of clicks on the two buttons, as shown in *Figure 13.18*:

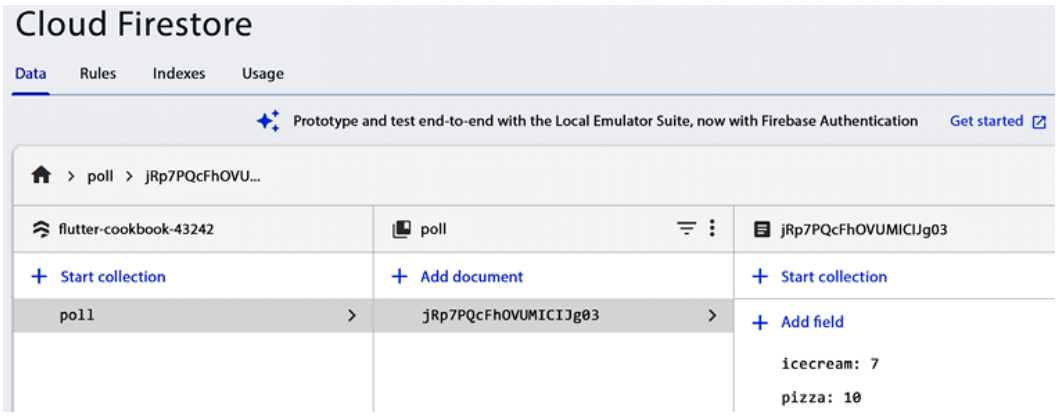


Figure 13.18: Cloud Firestore data

How it works...

In a Firebase project, which is the entry point for any Firebase service that you want to implement into your app, you can create a Firebase Firestore database. The Firestore database is a NoSQL database that stores data in the cloud.

In a Firestore hierarchy, databases contain `Collections`, which, in turn, contain `Documents`. Documents contain key-value pairs. There are a few rules you should be aware of when designing a Firestore database:

- The Firestore root can contain collections but cannot contain documents.
- Collections must contain documents, not other collections.
- Each document can take up to 1 MB.
- Documents cannot contain other documents.
- Documents CAN contain collections.
- Each document has a `documentID`: `documentID` is a unique identifier of each document in a collection.

To sum things up, the hierarchy of Firestore data is as follows:

Database > Collection > Document > key-value

> **Note**
>
> All methods in a Firebase Firestore database are asynchronous.

In the code you have written in this recipe, you retrieved the database instance with the help of the following command:

```
FirebaseFirestore db = FirebaseFirestore.instance;
```

Then, to get to the `'poll'` collection, which is the only collection in the database, you used the `collection` method on the database:

```
CollectionReference collection = db.collection('poll');
```

From `Collection`, you get all the existing documents with the `get` method. This returns a `QuerySnapshot` object, which contains the results of queries performed in a collection. A `QuerySnapshot` object can contain zero or more instances of `DocumentSnapshot` objects. You retrieved the documents in the `poll` collection with the following command:

```
QuerySnapshot snapshot = await collection.get();
```

From the `QuerySnapshot` object, you retrieved a `List` of `QueryDocumentSnapshot` objects by calling the `docs` property of `QuerySnapshot`. A `QueryDocumentSnapshot` object contains the data (key-value pairs) read from a document in a collection. You used the following command to create the `List`:

```
List<QueryDocumentSnapshot> list = snapshot.docs;
```

In this case, we know that `Collection` contains a single document. In order to retrieve it, you used the following command:

```
DocumentSnapshot document = list[0];
```

Once you have the ID of a single document in a collection, you can simply update any key with the `update` method, passing all the keys and values that you want to update. You performed this task with the following command:

```
collection.doc(id).update({'pizza':++pizzaVotes});
```

See also

If you want to learn more about the differences between Cloud Firestore and Realtime Database, have a look at the guide available at the following address: https://firebase.google.com/docs/database/rtdb-vs-firestore.

Sending push notifications with Firebase Cloud Messaging (FCM)

push notifications are an extremely important feature for an app. They are messages sent to your app's users that they can receive even when your app is not open. You can send notifications about anything: offers, updates, discounts, and any other type of message. This may help you keep users' interest in your app. In this recipe, you will see how to leverage Firebase Messaging to send messages to your app's users.

Getting ready

To follow along with this recipe, you need to complete the first two recipes in this chapter, *Configuring a Firebase app* and *Creating a login form*.

How to do it...

In this recipe, you will see how to send notification messages when your app is in the background. Please note that this code will not work on an iOS simulator (although it works perfectly on a physical device):

1. Add the `firebase_messaging` plugin to your app.
2. At the top of the `main.dart` file in your app, import the `firebase_messaging.dart` file:

```
import
'package:firebase_messaging/firebase_messaging.dart';
```

3. In the `main()` method, before calling `runApp`, add the statement below (you will get an error in `_firebaseBackgroundMessageReceived` that we will fix in the next step):

```
  FirebaseMessaging.onBackgroundMessage(
    _firebaseBackgroundMessageReceived);
  runApp(const MyApp());
```

4. Outside of the `MyApp` class, add a method, called `_firebaseBackgroundMessageReceived`, that prints the notification information:

```
Future _firebaseBackgroundMessageReceived(RemoteMessage
message) async {
    print(
        "Notification: ${message.notification?.title} -
${message.notification?.body}");
  }
```

5. Run the app and then close it.
6. Get to the Firebase console and click on the **Engage** section. Then, click on the **Cloud Messaging** link.
7. Click on the **Create your first campaign** button at the top of the page, then choose **Firebase Notification messages** in the following dialog, and click **Create**.
8. In the **Compose Notification** page, insert a title and text, as shown in the following screenshot, and then click the **Next** button:

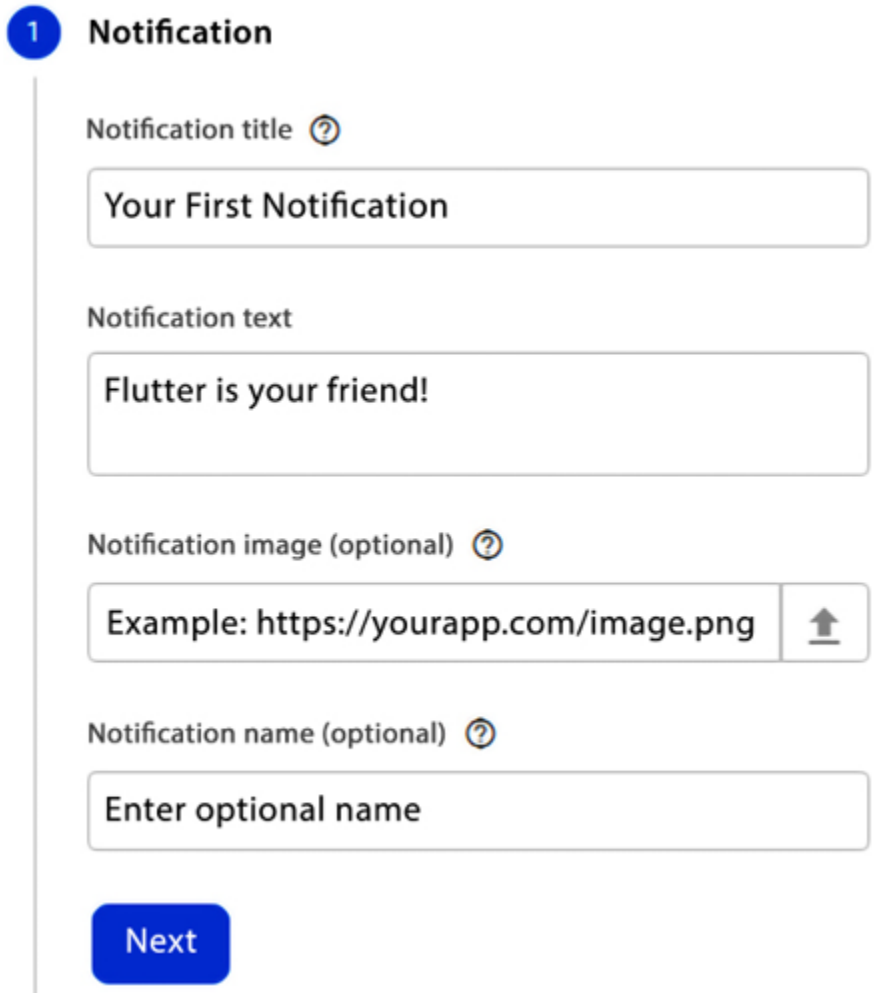


Figure 13.19: Creating a new notification

9. On the target page, select your app and then click **Next**.

> Note
>
> Notifications do not work on iOS simulators: you will need to use a real device and obtain the relevant permissions to see push notifications. See https://firebase.google.com/docs/cloud-messaging/ios/client for more details.

10. On the **Scheduling** page, leave the **Now** choice and then click **Next**.
11. On the **Conversion Events** page, click **Next**.

12. Make sure your app is NOT in the foreground and then, from the **Review** message, click **Review, and then in the dialog, click Publish**.
13. In your device or emulator, you should see a small flutter icon at the top of the screen.
14. Open the notifications tab, and there you should see the message that you just added from the **Firebase Cloud Messaging** page, as shown in *Figure 13.20* (please note that it may take several minutes before you receive a notification):

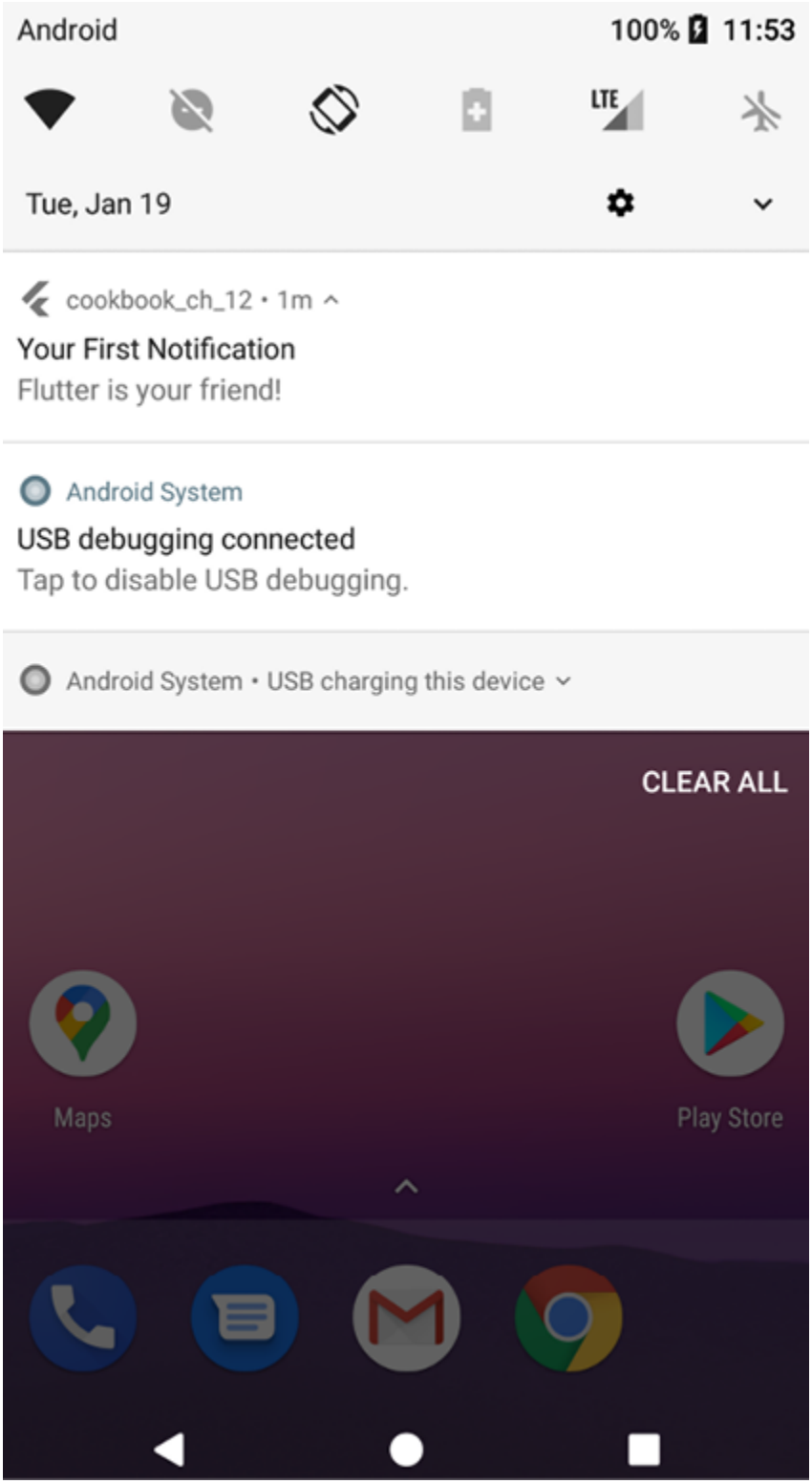


Figure 13.20: A notification on an Android device

How it works...

FCM is a service that allows you to send notifications to your users. In Flutter, this requires enabling the `firebase_messaging` package in your app.

To get an instance of the `FirebaseMessaging` class, you can use the following command:

```
FirebaseMessaging messaging = FirebaseMessaging.instance;
```

Like all services within FlutterFire, messaging services are available only **after** you initialize `FirebaseApp`. This is why you included the `FirebaseMessaging` configuration after initializing the app:

```
await Firebase.initializeApp(
    options: DefaultFirebaseOptions.currentPlatform,
  );
FirebaseUIAuth.configureProviders(…
```

The `onBackgroundMessage` callback is triggered when a new notification message is received and takes the method that deals with the notification:

```
FirebaseMessaging.onBackgroundMessage(_firebaseBackgroundMess
ageReceived);
```

In the function you pass to the `onBackgroundMessage` callback, you could save the message in a local database or `SharedPreferences`, or take any other action over it. In this recipe, you just print the message in the debug console with the following command:

```
Future _firebaseBackgroundMessageReceived(RemoteMessage
message) async {
  print("Notification: ${message.notification.title} -
${message.notification.body}");
```

Now what happens is that when the app is in the background or not running, the notification message will be shown as a system notification, and tapping the notification will launch the app or bring it to the foreground. When the app is in the foreground (active), the notification will not be shown.

See also

In this recipe, you saw how to send notification messages when your app is in the background, but you can also deal with notifications when your app is in use. For more information, have a look at https://pub.dev/packages/firebase_messaging.

Another interesting feature you can add to your app (also with notifications) is dynamic links. These are URLs that allow you to send users to a specific location inside your iOS or Android app. For more information, have a look at https://firebase.google.com/docs/dynamic-links.

Storing files in the cloud

Firebase Cloud Storage is a service that allows the uploading and downloading of files, such as images, audio, video, or any other content. This may add a powerful feature to your apps.

Getting ready

To follow along with this recipe, you need to complete the first two recipes in this chapter, *Configuring a Firebase app* and *Creating a login form*.

Depending on your device, you may need to configure the permissions to access your images library. See the setup procedure at https://pub.dev/packages/image_picker.

How to do it...

In this recipe, you will add a screen that allows images to be uploaded to the cloud:

1. In your Firebase console, click on the **Storage** link in the build section.
2. Click on the **Get started** button.
3. In the prompt window, select **Start** in test mode and click **Next**.
4. On the following page, click **Done**.
5. In your Flutter project, add the latest versions of the `firebase_storage`, `image_picker`, and `path` packages:
6. In the `lib/screens` folder of your project, create a new file, and call it `upload_file.dart`.
7. At the top of the `upload_file_screen.dart` file, add the following import statements:

```
import 'package:flutter/material.dart';
import 'dart:io';
import 'package:path/path.dart';
import 'package:image_picker/image_picker.dart';
import 'package:firebase_storage/firebase_storage.dart';
```

8. Create a new stateful widget, and call it `UploadFileScreen`:

```
class UploadFileScreen extends StatefulWidget {
  const UploadFileScreen({super.key});
  @override
  State<UploadFileScreen> createState() =>
_UploadFileScreenState();
}
class _UploadFileScreenState extends
State<UploadFileScreen> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

9. At the top of the `_UploadFileScreenState` class, declare a file called `_image`, a string called `_message`, and an `ImagePicker` object:

```
File? _image;
String _message = '';
final picker = ImagePicker();
```

10. In the `build` method of the `_UploadFileScreenState` class, return a `Scaffold`, with an `appBar` and a `body`, as shown in the following code block:

```
return Scaffold(
  appBar: AppBar(title: const Text('Upload To
FireStore')),
  body: Padding(
    padding: const EdgeInsets.all(24),
    child: Column(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      crossAxisAlignment: CrossAxisAlignment.stretch,
      children: [],
),),);
```

11. In the `children` parameter of the `Column` widget, add four widgets – `ElevatedButton`, `Image`, another `ElevatedButton`, and a `Text`:

```
children: [
   ElevatedButton(
         child: const Text('Choose Image'),
          onPressed: () {},
      ),
SizedBox(
  height: 200,
  child: _image == null ? null : Image.file(_image!)),
   ElevatedButton(
     child: const Text('Upload Image'),
       onPressed: () {},
      ),
   Text(_message),
]
```

12. At the bottom of the `_UploadFileScreenState` class, create a new `async` method, called `getImage`, that allows users to choose a file from the image gallery:

```
Future getImage() async {
  final pickedFile = await picker.pickImage(
source: ImageSource.gallery);
     if (pickedFile != null) {
         _image = File(pickedFile.path);
          setState(() {});
      } else {
        print('No image selected.');
      } }
```

Note

For iOS, you will need to ask the user's permission before using the ImagePicker and accessing the device gallery. To do that, open the `ios/Runner/Info.plist` file in your project directory, and in the `<dict>` tag, add the following lines:

```
<key>NSPhotoLibraryUsageDescription</key>
<string>Firebase Demo requires access to your photo
library to select images.</string>
```

If you need to access the camera, you also need to ask that specific permission:

```
<key>NSCameraUsageDescription</key>
<string>YOUR_APP_NAME requires access to your camera
to take photos.</string>
```

13. Under the `getImage` method, add another asynchronous method, called `uploadImage`, that leverages the `FirebaseStorage` service to upload the selected image to the cloud, as shown here:

```
Future uploadImage() async {
  if (_image != null) {
      String fileName = basename(_image!.path);
      FirebaseStorage storage =
FirebaseStorage.instance;
      Reference ref = storage.ref(fileName);
      setState(() {
        _message = 'Uploading file. Please wait...';
```

```
      });
      ref.putFile(_image!).then((TaskSnapshot result) {
        if (result.state == TaskState.success) {
          setState(() {
            _message = 'File Uploaded Successfully';
          });
        } else {
          setState(() {
            _message = 'Error Uploading File';
          });
        }
      });
    }
}
```

14. In the `onPressed` parameter of the first `ElevatedButton` (**the one that chooses an image**), add a call to the `getImage` method:

```
getImage();
```

15. In the `onPressed` parameter of the second `ElevatedButton` (the one that **uploads the image**), add a call to the `uploadImage` method:

```
uploadImage();
```

16. At the top of the `authentication_screen.dart` file, import the `upload_file_screen.dart` file:

```
import './upload_file_screen.dart';
```

17. In the `authentication_screen.dart` file, in the `build()` method of the `AuthenticationScreen` class, return `UploadFileScreen` in the `else` block:

```
} else {
   return const UploadFileScreen();
}
```

18. Run the app, and after logging in, pick an image and upload it to Firebase. The end result should look similar to *Figure 13.21*:

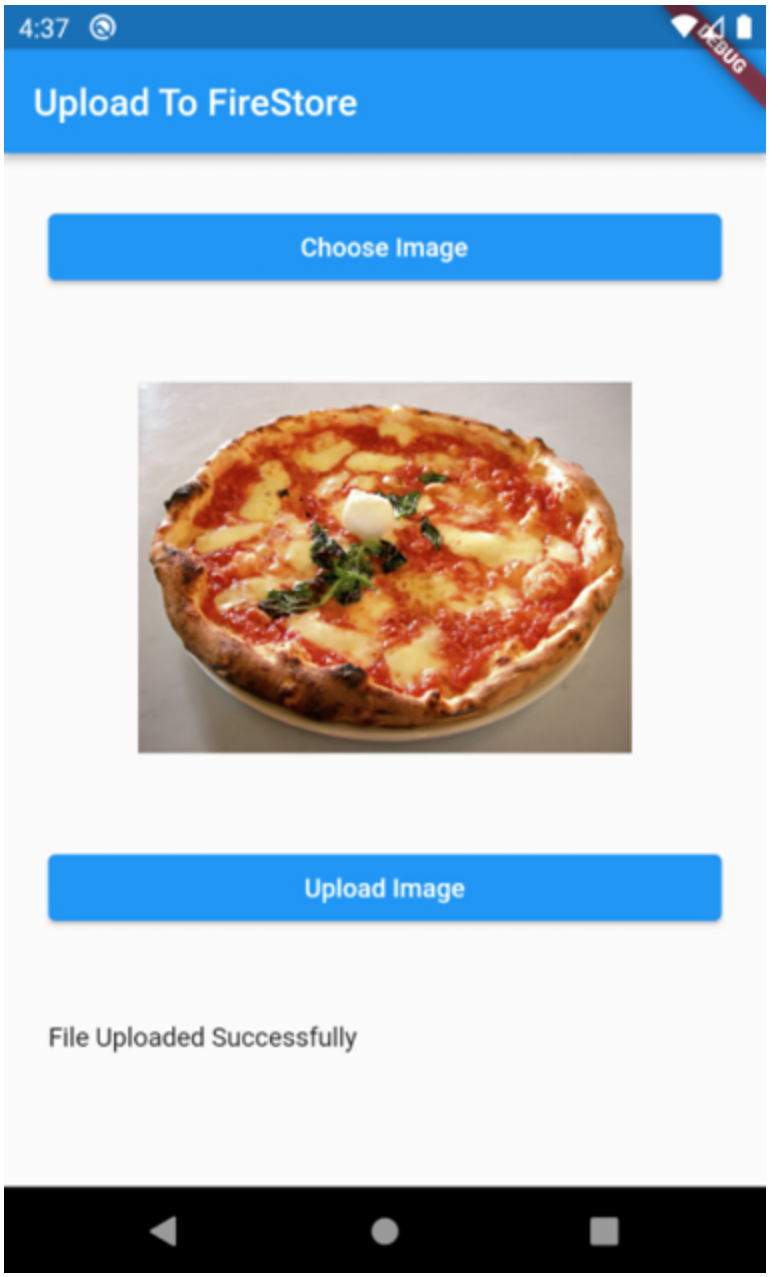


Figure 13.21: Uploading an image to storage

How it works...

The Firebase Storage API lets you upload files to the Google Cloud. These files can then be shared and used based on your app's needs.

The first step to leverage this service in Flutter is adding the `firebase_storage` dependency to your `pubspec.yaml` file and importing it into your files. This way, you get access to the `FirebaseStorage` instance, which is the entry point for any future action on the files. In this recipe, you retrieved the instance with the following command:

```
FirebaseStorage storage = FirebaseStorage.instance;
```

From storage, you can then get a `Reference` object. This is a pointer to a storage object that you can use to upload, download, and delete objects. In this recipe, we created a `Reference` object, passing the name of the file that would later be uploaded, with the following command:

```
Reference ref = storage.ref(fileName);
```

Once you create a `Reference` object, you can then upload a file to that reference with the `putFile` method, passing the file that you want to upload:

```
ref.putFile(_image)
```

Once the `putFile` task is complete, it returns `TaskSnapshot`. This contains a `state` property that indicates whether the task was completed successfully. This allows us to give some feedback to the user with the `_message` state variable:

```
setState(() {
  _message = 'File Uploaded Successfully';
});
```

It's also worth noting that in this recipe, we used `ImagePicker` to select an image from the device gallery. All the magic happens with two lines of code:

```
final picker = ImagePicker();
final pickedFile = await picker.pickImage(source:
ImageSource.gallery);
```

First, you retrieve an instance of `ImagePicker`, and then you call the `pickImage` method, selecting the source where you want to retrieve the images.

Summary

In this chapter, you saw how to integrate Firebase with Flutter. First, you saw to create a new Firebase project and add Firebase to a Flutter app.

Then, you learned how to create an authentication screen leveraging Firebase Authentication and FirebaseUI Auth. You also saw ways to customize the authentication process and how to integrate third-party providers, like Google Sign-in, for a smoother login experience.

Next, you saw how to integrate Firebase Analytics into your apps to gain insights into user behavior and app performance.

To save data in the cloud, we covered the Firebase Cloud Firestore, a document-based database that makes it easy to store and retrieve data in an app. You saw how to use collections and documents.

Push notifications are an important feature for mobile apps, and FCM makes it easy to send push notifications to users. You learned how to set up your app to receive push notifications when your app is in the background.

Finally, you saw how to store files in the cloud with Firebase Storage.

Integrating Firebase and Flutter can help you easily create real-world solutions without creating complex backend projects.

Firebase Machine Learning

Machine Learning (ML) has become a critical topic in software development. In a nutshell, ML means that you import data that "trains" a software application and use this data to refine a model. This trained model can then be used to solve problems that would be virtually impossible with traditional programming. To make this process more manageable, Firebase offers a service called Firebase Machine Learning, an ML kit that we could define as "pre-built ML".

Among other functionalities, it contains text recognition, image labeling, face detection, and barcode scanning. For functionalities that are not already provided by Firebase ML, you can create your own custom model with TensorFlow. Most of the services outlined in this chapter can run both on the cloud and on your device.

In this chapter, you will start by taking a picture with your device; then, you will use Firebase ML to recognize text, barcodes, images, and faces and identify a language. At the end of this chapter, you will also be introduced to TensorFlow Lite, which allows you to build your own ML algorithms.

In particular, we will cover the following topics:

- Using the device's camera
- Recognizing text from an image
- Reading a barcode
- Image labeling
- Building a face detector and detecting facial gestures
- Identifying a language
- Using TensorFlow Lite

By the end of this chapter, you will be able to leverage several Firebase services and take advantage of several ML features.

Using the device's camera

In this recipe, you will use the `Camera` plugin to create a canvas for ML Kit's vision models. The `camera` plugin is not related to ML, but being able to take pictures is one of the prerequisites for ML visual functions that you will use in the following recipes in this chapter. Of course, you can also skip this recipe and use images you download from the web if you prefer.

By the end of this recipe, you will be able to use the device's cameras (front and rear) to take pictures and use them in your apps.

Getting ready

For this recipe, you should create a new project and set up **Firebase** as described in *Chapter 13*, *Using Firebase,* in the *Configuring a Firebase app* recipe.

How to do it...

In this recipe, you will add the camera functionality to your app. Users will be able to take a picture with the front or rear camera of their device. Follow these steps:

1. In the dependencies section of the project's `pubspec.yaml` file, add the `camera` and `path_provider` packages.
2. For Android, change the minimum Android SDK version to 21 or higher in your `android/app/build.gradle` file:

```
minSdkVersion 21
```

3. For iOS, add the following instructions to the `ios/Runner/Info.plist` file:

```
<key>NSCameraUsageDescription</key>
<string>Enable MLApp to access your camera to capture
your photo</string>
```

4. In the `lib` folder of your project, add a new file and call it `camera.dart`.
5. At the top of the `camera.dart` file, import `material.dart`, and the `camera` package:

```
import 'package:flutter/material.dart';
import 'package:camera/camera.dart';
```

6. Create a new stateful widget, calling it `CameraScreen`:

```
class CameraScreen extends StatefulWidget {
  const CameraScreen({super.key});
  @override
  State<CameraScreen> createState() =>
_CameraScreenState();
}
class _CameraScreenState extends State<CameraScreen> {
  @override
  Widget build(BuildContext context) {
    return const Placeholder();
  }
}
```

7. At the top of the `_CameraScreenState` class, declare the following variables:

```
  List<CameraDescription> cameras = [];
  List<Widget> cameraButtons = [];
  CameraController? cameraController;
  CameraDescription? activeCamera;
  CameraPreview? preview;
```

8. At the bottom of the `_CameraScreenState` class, create a new asynchronous method called `listCameras` that returns a list of widgets:

```
Future<List<Widget>> listCameras() async {}
```

9. In the `listCameras` method, call the `availableCameras` method, and based on the result of the call, return the `ElevatedButton` widgets with the name of the camera, as shown:

```
    List<Widget> buttons = [];
    cameras = await availableCameras();
    if (cameras.isEmpty) return [];
    activeCamera ??= cameras.first;
    for (CameraDescription camera in cameras) {
      buttons.add(ElevatedButton(
          onPressed: () {
            setState(() {
              activeCamera = camera;
              setCameraController();
            });
          },
          child: Row(
            children: [const Icon(Icons.camera_alt),
Text(camera.name)],
          )));
    }
    return buttons;    }
```

10. In the `_CameraScreenState` class, create a new asynchronous method called `setCameraController`, which based on the value of `activeCamera` will set the `CameraPreview` preview variable, as shown:

```
  Future setCameraController() async {
    if (activeCamera == null) return;
    if (cameraController != null) {
      await cameraController!.dispose();
    }
    cameraController = CameraController(
      activeCamera!,
      ResolutionPreset.high,
    );
    try {
      await cameraController!.initialize();
    } catch (e) {
      print('Ccamera controller initialize error: $e');
      return;
    }

    setState(() {
```

```
      preview = CameraPreview(
        cameraController!,
      );
    });
  }
```

11. Under the `setCameraController` method, add another asynchronous method and call it `takePicture`. This will return an `XFile`, which is the result of the call to the `takePicture` method of the `cameraController` widget, as shown:

```
Future takePicture() async {
    if (cameraController == null) {
      return;
    }
    if (!cameraController!.value.isInitialized) {
      return;
    }
    if (cameraController!.value.isTakingPicture) {
      return;
    }
    try {
      await
cameraController!.setFlashMode(FlashMode.off);
      XFile picture = await
cameraController!.takePicture();

      return picture;
    } catch (exception) {
      print(exception.toString());
    }  }
```

12. Override the `initState` method. Inside it, set `cameraButtons` and call the `setCameraController` method, as shown here:

```
@override
  void initState() {
    super.initState();
    listCameras().then((result) {
      setState(() {
        cameraButtons = result;
        setCameraController();
      });
    });
```

```
  }
```

13. Override the `dispose` method and inside it, dispose of the `cameraController` widget:

```
 @override
  void dispose() {
    cameraController?.dispose();
    super.dispose();
  }
```

14. In the `build` method, find the current size of the screen and return a `Scaffold` with an `AppBar` whose title is `Camera View` and a `body` with a `Placeholder` widget:

```
final size = MediaQuery.of(context).size;
return Scaffold(
        appBar: AppBar(
          title: const Text('Camera View'),
        ),
        body: const Placeholder());
```

15. Instead of the `Placeholder`, insert a `Padding` with an `EdgeInsets.all` value of `24` and a `child` of `Column`, as shown:

```
Padding(
  padding: const EdgeInsets.all(24),
            child: Column(
                mainAxisAlignment:
MainAxisAlignment.spaceAround,
                children: []))
...
```

16. In the `children` parameter of `Column`, add a row with `cameraButtons`, a `Container` with the camera preview, and another button that takes the picture using the camera:

```
Row(
  mainAxisAlignment: MainAxisAlignment.spaceAround,

  children: cameraButtons.isEmpty
```

```
    ?  [const Text('No cameras available')],
    : cameraButtons,
),
SizedBox(height: size.height / 2 , child: preview),
Row(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: [
    ElevatedButton(
      child: const Text('Take Picture'),
      onPressed: () {
        if (cameraController != null) {
          takePicture().then((dynamic
icture) {
           Navigator.push(
             context,
             MaterialPageRoute(
               builder: (context) =>
                       PictureScreen(picture)));
         });
        }
       },
      )
     ],
  ),],
```

17. In the `lib` folder of your project, create a new file called `picture.dart`.
18. In the `picture.dart` file, import the following packages:

```
import 'package:camera/camera.dart';
import 'package:flutter/material.dart';
import 'dart:io';
```

19. In the `picture.dart` file, create a new stateless widget called `PictureScreen`:

```
class PictureScreen extends StatelessWidget {
const PictureScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return const Placeholder();
  }
}
```

20. At the top of the `PictureScreen` class, add a final `XFile` called `picture`, and in the constructor method, set its value:

```
final XFile picture;
const PictureScreen(this.picture, {super.key});
```

21. In the `build` method of the `_PictureScreenState` class, retrieve the device's height, then return a `Scaffold` containing a `Column` widget that shows the picture that was passed to the screen. Under the picture, also place a button that will later send the file to the relevant ML service, as shown here:

```
final deviceHeight =
MediaQuery.of(context).size.height;
return Scaffold(
      appBar: AppBar(
        title: const Text('Picture'),
      ),
      body: Column(
        mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
        children: [
          Text(widget.picture.path),
          SizedBox(
              height: deviceHeight / 1.5,
              child:
Image.file(File(widget.picture.path))),
          Row(
            children: [
              ElevatedButton(
                child: const Text('Text Recognition'),
                onPressed: () {},
              )
            ],
          )
        ],
      ),
    );
```

22. Back to the `main.dart` file, in the `MyApp` class, call the `CameraScreen` widget and set the title and theme of `MaterialApp` as shown. Also, remove all the code under `MyApp`:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Firebase Machine Learning',
      theme: ThemeData(
        primarySwatch: Colors.deepOrange,
      ),
      home: const CameraScreen(),
); } }
```

23. Run the app. Choose one of the cameras in your device, then press the **Take Picture** button. You should see the picture you have taken, with the path of the file that was saved in your device, as shown in the following screenshot:

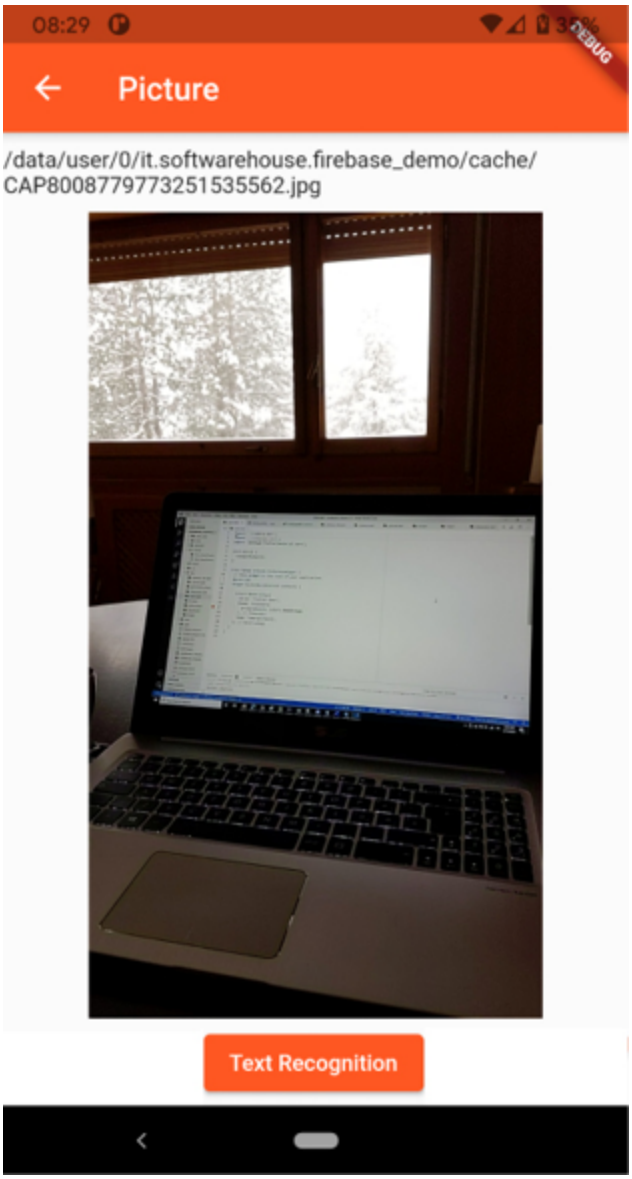


Figure 14.1: A picture taken from a device camera (and snow outside)

How it works...

Being able to use the camera and add pictures to your app is useful not only for ML but also for several other features you might want to add to your app. You can leverage the `camera` plugin to get a list of the available cameras in the device and take photos or videos.

With the `camera` plugin, you get access to two useful objects:

- `CameraController` connects to a device's camera, and you use it to take pictures or videos.
- `CameraDescription` contains the properties of a camera device, including its name and orientation.

Most devices have two cameras, one on the front (for selfies) and the other on the back, but some devices may only have one, and others more than two when they connect an external camera. That's why in our code, we created a dynamic `List` of `CameraDescription` objects to make the user choose the camera they want to use with the following instruction:

```
cameras = await availableCameras();
```

The `availableCameras` method returns all the available cameras for the device in use and returns a `Future` value of `List<CameraDescription>`.

In order to choose the camera, we called the `cameraController` constructor, passing the active camera with the following instruction:

```
cameraController = CameraController(activeCamera,
ResolutionPreset.veryHigh);
```

Note that you can also choose the resolution of the picture with the `ResolutionPreset` enumerator; in this case, `ResolutionPreset.veryHigh` has a good resolution (a good resolution is recommended for use with ML algorithms).

The `CameraController` asynchronous `takePicture` method actually takes a picture; this will save the picture in a default path that we later show on the second screen of the app:

```
XFile picture = await cameraController.takePicture();
```

The `takePicture` method returns an `XFile`, which is a **cross-platform file abstraction**.

Another important aspect to perform is overriding the `dispose` method of the `_CameraScreenState` class, which calls the `dispose` method of `CameraController` when the widget is disposed of.

The second screen that you have built in this recipe is the `PictureScreen` widget. This shows the picture that was taken and shows its path to the user. Please note the following instruction:

```
Image.file(File(widget.picture.path))
```

In order to use the picture in the app, you need to create a `File`, as you cannot use `XFile` directly.

Now your app can take pictures from the camera(s) in your device. This is a prerequisite for several of the remaining recipes of this chapter.

See also

While currently, there is no way to apply real-time filters with the official `camera` plugin (for the issue, see https://github.com/flutter/flutter/issues/49531), there are several workarounds that allow us to obtain the same effects with Flutter. For an example with opacity, for instance, see https://stackoverflow.com/questions/50347942/flutter-camera-overlay.

Recognizing text from an image

We'll start with ML by incorporating ML Kit's text recognizer. You will create a feature where you take a picture, and if there is some recognizable text in it, ML Kit will turn it into one or more strings.

Getting ready

For this recipe, you should have completed the previous one: *Using the device camera*.

How to do it...

In this recipe, after taking a picture, you will add a text recognition feature. Follow these steps:

1. Import the latest version of the `google_mlkit_commons` and `google_mlkit_text_recognition` packages in your `pubspec.yaml` file by typing in your Terminal:

```
flutter pub add google_mlkit_commons
flutter pub add google_mlkit_text_recognition
```

2. Create a new file in the `lib` folder of your project and call it `ml.dart`.
3. Inside the new file, import the `dart:io` and `google_mlkit_text_recognition` packages:

```
import 'dart:io';
import '
package:google_mlkit_text_recognition/google_mlkit_text_
recognition.dart';
```

4. Create a new class, calling it `MLHelper`:

```
class MLHelper {}
```

5. In the `MLHelper` class, create a new async method called `textFromImage` that takes an image file and returns `Future<String>`:

```
Future<String> textFromImage(File image) async { }
```

6. In the `textFromImage` method, process the image with the ML Kit `TextRecognizer` and return the retrieved text, as shown:

```
final textRecognizer = TextRecognizer(script:
TextRecognitionScript.latin);
    final InputImage inputImage =
InputImage.fromFile(image);
    final RecognizedText recognizedText =
```

```
        await textRecognizer.processImage(inputImage);
return recognizedText.text;
```

7. In the `lib` folder of your project, create a new file and call it `result.dart`.
8. At the top of the `result.dart` file, import the `material.dart` package:

```
import 'package:flutter/material.dart';
```

9. Create a new stateful widget and call it `ResultScreen`:

```
class ResultScreen extends StatefulWidget {
  const ResultScreen({super.key});
  @override
  State<ResultScreen> createState() =>
_ResultScreenState();
}
class _ResultScreenState extends State<ResultScreen> {
  @override
  Widget build(BuildContext context) {
    return const Placeholder();
  }
}
```

10. At the top of the `ResultScreen` class, declare a final `String`, called `result`, and set it in the default constructor method:

```
final String result;
const ResultScreen(this.result, {super.key});
```

11. In the `build` method of the `_ResultScreenState` class, return a `Scaffold`, and in its `body`, add `SelectableText`, as shown:

```
return Scaffold(
      appBar: AppBar(
        title: const Text('Result'),
      ),
      body: Padding(
        padding: const EdgeInsets.all(24),
        child: SelectableText(widget.result,
          showCursor: true,
```

```
          cursorColor:
Theme.of(context).colorScheme.secondary,
          cursorWidth: 5,
          scrollPhysics: const ClampingScrollPhysics(),
          onTap: (){},
        )),
    );
```

12. In the `picture.dart` file, in the `onPressed` function, in the **Text Recognition** button, add the following code:

```
onPressed: () {
  final image = File(widget.picture.path);
    MLHelper helper = MLHelper();
    helper.textFromImage(image).then((result) {
      Navigator.push(
        context,
        MaterialPageRoute(
          builder: (context) => ResultScreen(result)));
     }
  );
},
```

13. Stop and run the app, select a camera on your device, and take a picture of some printed text. Then, press the **Text Recognition** button. You should see the text taken from your picture as shown in the following screenshot:

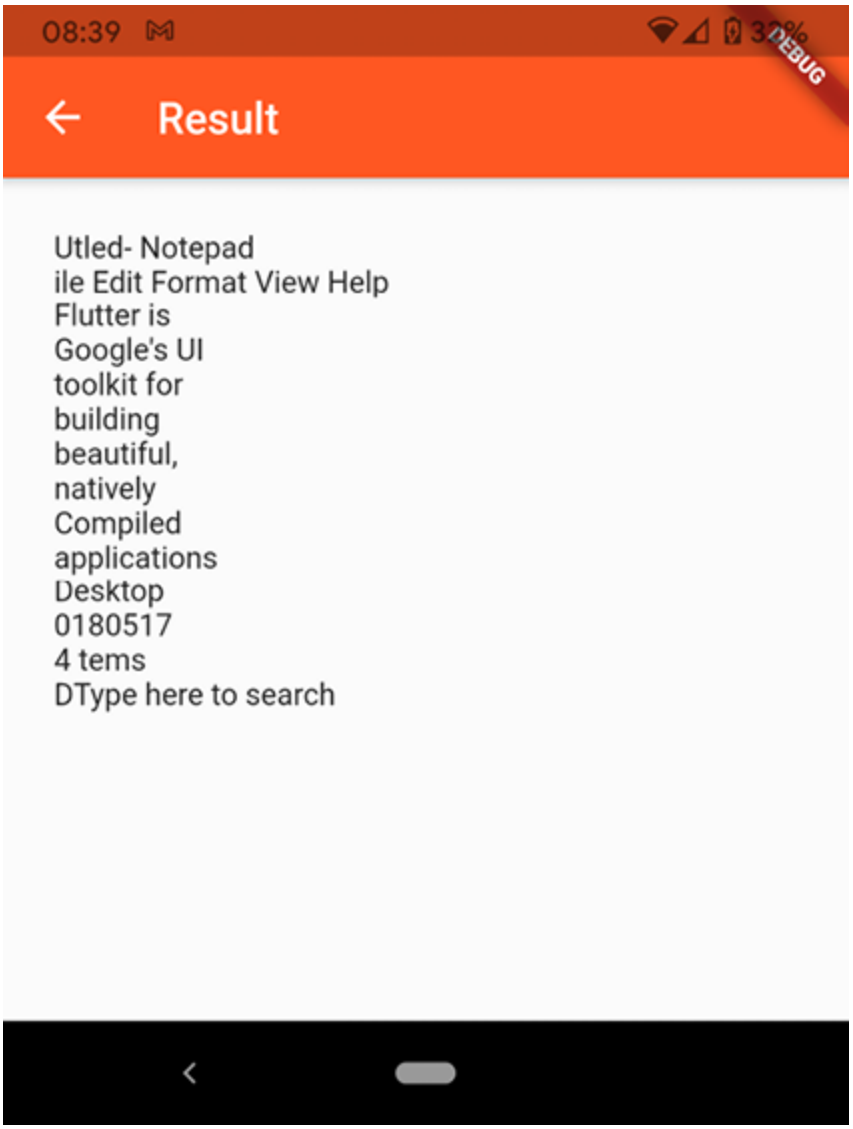


Figure 14.2: ML text recognition

How it works...

When using ML Kit, the process required to get results is usually the following:

1. You get an image.
2. You send it to the API to get some information about the image.

3. The ML Kit API returns data to the app, which can then be used as necessary.

In the example in this recipe, most of the logic is contained in the `textFromImage` method. The first step is to get an instance of the `TextRecognizer` object:

```
final textRecognizer = TextRecognizer(script:
TextRecognitionScript.latin);
```

The next step is creating an `InputImage`, which is the image object used for the API detector. In our example, you created it with the following instruction:

```
final InputImage inputImage = InputImage.fromFile(image);
```

To get the text from the image, you need to call the `processImage` method on `TextRecognizer`. This asynchronous method returns a `RecognizedText` object, which contains several pieces of information, including the `text` property, which contains all the text recognized in the image. You got the text with the following instructions:

```
final RecognizedText recognizedText =
        await textRecognizer.processImage(inputImage);
    return recognizedText.text;
```

You then showed the text on another screen, but instead of just returning a `Text` widget, you used `SelectableText`. This widget allows users to select some text and copy it to other applications.

See also

The text property of a `TextRecognizer` instance returns all the text that was returned by the method, but there are cases where you might want to only return one or more specific texts (see, for instance, plate recognition or invoices). In these cases, you might use the `block`'s property, which contains a list of `TextBlock` instances. `TextBlock` is a single element of text

recognized by the ML Kit. For more information and examples of its use, see https://pub.dev/packages/firebase_ml_vision.

Reading a barcode

Adding barcode reading capabilities to your app can open several scenarios for your projects. It is a fast and convenient way to take some user input and return relevant information.

The ML Kit Barcode Scanner library takes an input image and extracts any barcodes that it finds within the image itself.

ML Kit can read most standard barcode formats, including linear and 2D formats:

- **Linear**: Code 39, Code 93, Code 128, Codabar, EAN-8, EAN-13, ITF, UPC-A, and UPC-E
- **2D**: Aztec, Data Matrix, PDF417, and QR code

In this recipe, you will see an easy way to add this feature to the sample app.

Getting ready

Before following this recipe, you should have completed the previous two: *Using the device camera* and *Recognizing text from an image*.

How to do it...

You will now add the barcode reading feature to the existing app:

1. Add the `google_mlkit_barcode_scanning` package to your project from the Terminal:

```
flutter pub add google_mlkit_barcode_scanning
```

2. At the top of the `ml.dart` file, import `google_mlkit_barcode_scanning.dart`:

```
import
'package:google_mlkit_barcode_scanning/google_mlkit_barc
ode_scanning.dart';
```

3. In the `MLHelper` class in the `ml.dart` file, add a new async method called `readBarCode`, that takes `File` as a parameter and returns a string:

```
Future<String> readBarcode(File image) async {}
```

4. At the top of the `readBarCode` method, declare a `BarcodeScanner`, an `InputImage` that parses the file that was passed as a parameter, and a `String` for the result of the function:

```
final barcodeScanner = BarcodeScanner(formats:
[BarcodeFormat.all]);
final InputImage inputImage =
InputImage.fromFile(image);
String result = '';
```

5. Declare a `List` of `Barcode` objects, called `barcodes`. This takes the result of the `BarcodeScanner.processImage` method:

```
final List<Barcode> barcodes =
       await barcodeScanner.processImage(inputImage);
```

6. For each `Barcode` object, extract the display value of the barcode:

```
 for (Barcode barcode in barcodes) {
      final String displayValue = barcode.displayValue
?? '';
      result += '$displayValue\n';
}
```

7. At the bottom of the `readBarCode` method, return the result string:

```
return result;
```

8. In the `picture.dart` file, in the `Row` widget that contains the **Text Recognition** button, add a second button that calls `readBarCode` and calls `ResultScreen` with the result:

```
ElevatedButton(
  child: const Text('Barcode Reader'),
                onPressed: () {
                  final image =
File(widget.picture.path);
                  MLHelper helper = MLHelper();

helper.readBarcode(image).then((result) {
                    Navigator.push(
                        context,
                        MaterialPageRoute(
                            builder: (context) =>
ResultScreen(result)));
                  });
                },
),
```

9. Run the app and try to scan a barcode under any book (you may also use this book if you have a printed copy). You should see the string value of the barcode you scanned, as shown in *Figure 14.3*:

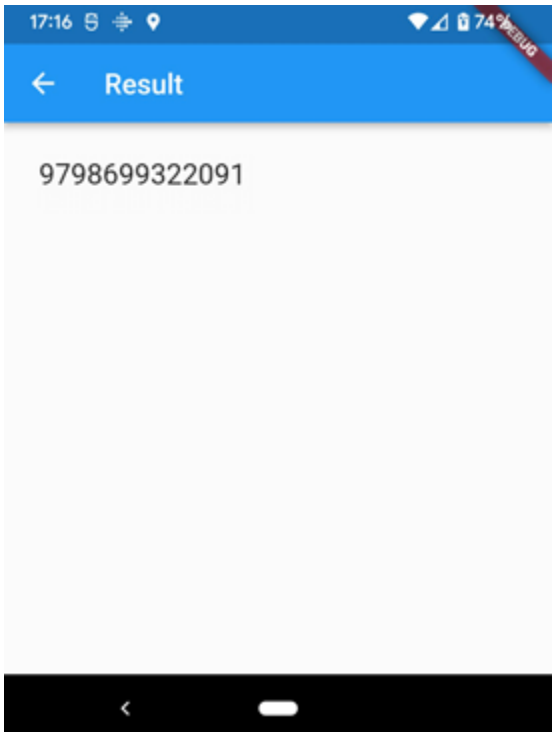


Figure 14.3: Barcode reading result

How it works...

Reading barcodes is a key feature of many business applications. When using ML Kit's barcode scanning API, your app can automatically recognize most of the standard barcode formats, including QR codes, EAN-13, and ISBN codes.

> **Note**
>
> The detection of a barcode happens on the device, so it doesn't require a network connection.

A `Barcode` object contains a `displayValue`, which is a string that contains the value of the barcode as it should be displayed to the user.

In order to scan a barcode, the first step is to create an instance of the `BarcodeScanner` class and pass it an array of `BarcodeFormat` objects that

represent the types of barcodes that the scanner should detect. In our example, the list contains a single item, `BarcodeFormat.all`, which tells the scanner to detect all types of barcodes.

The method to actually detect barcodes is `BarcodeScanner.processImage()`. This takes an `inputImage` object as its argument, then asynchronously processes the image, and finally returns a list of the `Barcode` objects that were detected in the image.

In this recipe, you read the barcodes with the following instruction:

```
List<Barcode> barcodes =
        await barcodeScanner.processImage(inputImage);
```

From the results, you wrote a string containing each `displayValue` of the barcodes that were recognized:

```
for (Barcode barcode in barcodes) {
  final String displayValue = barcode.displayValue ?? '';
  result += '${barcode.displayValue}\n';
}
```

As you can see, adding barcode reading capabilities is extremely easy for Flutter developers.

See also

You can also add barcode scanning capabilities to your app without using ML Kit. One of the most popular packages available in the `pub.dev` repository is `flutter_barcode_scanner`, available at https://pub.dev/packages/flutter_barcode_scanner.

Image labeling

ML Kit contains an image labeling service. With it, you can identify common objects, places, animals, products, and more. Currently, the API

supports over 400 categories, but you can also use a custom TensorFlow Lite model to add more objects. In this recipe, we'll learn how to implement this feature in our sample app.

Getting ready

Before following this recipe, you should have completed the *Using the device camera* and *Recognizing text from an image* recipes in this chapter.

How to do it...

Let's now add an image labeling feature to the existing app:

1. Add the `google_mlkit_image_labeling` package to your project from the Terminal:

```
flutter pub add google_mlkit_image_labeling
```

2. In the `ml.dart` file, import the new package:

```
import
'package:google_mlkit_image_labeling/google_mlkit_image_
labeling.dart';
```

3. Add a new async method and call it `labelImage`. The method takes `File` as a parameter and returns a string:

```
Future<String> labelImage(File image) async {}
```

4. At the top of the `labelImage` method, declare a `String` for the result and an `InputImage` that parses the image passed as a parameter:

```
String result = '';
    final InputImage inputImage =
InputImage.fromFile(image);
```

5. Declare `ImageLabelerOptions` to set `confidenceThreshold` to `0.5`, and an `ImageLabeler` that receives these options:

```
  final ImageLabelerOptions options =
        ImageLabelerOptions(confidenceThreshold: 0.5);
    final imageLabeler = ImageLabeler(options: options);
```

6. Call the `processImage` method to retrieve the list of labels recognized by the image labeler:

```
final List<ImageLabel> labels = await
imageLabeler.processImage(inputImage);
```

7. Loop through the labels, and for each label, add to the results string the index, the labels, and the confidence of the label, then `return result`:

```
for (ImageLabel label in labels) {
      final String text = label.label;
      final int index = label.index;
      final double confidence = label.confidence;
      result += '$index: $text - ${confidence * 100}%
\n';
    }
return result;
```

8. In the `picture.dart` file, in the **Row** widget that contains the **Text Recognition** and **Barcode Reader** buttons, add another `ElevatedButton` that calls the `labelImage` method and calls `ResultScreen` with the result of the labeling:

```
ElevatedButton(
  child: const Text('Image Labeler'),
    onPressed: () {
      final image = File(widget.picture.path);
      MLHelper helper = MLHelper();
      helper.labelImage(image).then((result) {
      Navigator.push(
        context,
        MaterialPageRoute(
          builder: (context) => ResultScreen(result)));
```

```
          });
        },
),
```

9. Run the app and take a picture of your environment. You should see a list of objects, as shown in *Figure 14.4*:



Figure 14.4: ML labelling

How it works...

The image labeling service recognizes different objects in an image. These include people, places, animals, and plants. Use cases for image labeling are

almost limitless; you could use it to automatically categorize your user's pictures or use it for content moderation or more specific tasks.

The pattern is similar to the previous recipes in this chapter: you need to get an image, send it to the API, and retrieve and show the results.

Specific to the code you built in this example, consider this code:

```
final ImageLabelerOptions options =
        ImageLabelerOptions(confidenceThreshold: 0.5);
```

You use `ImageLabelerOptions` to customize the behavior of `ImageLabeler`. In this case, you set the `confidenceThreshold` property to `0.5` (or 50%), which is the minimum level of confidence required for a label to be assigned to an object in the image.

This is the likelihood of the object being that particular label.

For example, if you set `confidenceThreshold` to `0.8`, `ImageLabeler` would only assign labels to objects in the picture where it had a confidence level of at least 80%. Any objects with a confidence level lower than that would not be labeled.

After that, you create a new `ImageLabeler` object and call its `processImage` method, passing the `InputImage` object as an argument. This returns a list of `ImageLabel` objects that contain information about the labels assigned to the image, including its `label`, `index`, and `confidence` properties.

The label is a `String` that contains the label assigned to an object, like *dog* or *guitar*.

The index property contains an integer that identifies the label object: you usually don't need to show it to your users.

The confidence contains a double that represents the confidence level of the label assigned to the object, between 0 (not confident at all) and 1 (virtually certain).

Building a face detector and detecting facial gestures

This recipe will explore ML Kit's face detection. This also includes a model that predicts the probability of a face smiling or having open or closed eyes. This API also includes the identification of key facial features (such as eyes, nose, and mouth) and can get the contours of detected faces.

Getting ready

Before following this recipe, you should have completed the *Using the device camera* and *Recognizing Text from an image* recipes in this chapter.

How to do it...

In this recipe, you will add the face detection feature to the existing project: the model will predict whether the faces in the picture are smiling and have their eyes open. Follow these steps:

1. Add the `google_mlkit_face_detection` package to your project from the Terminal:

```
flutter pub add google_mlkit_face_detection
```

2. In the `ml.dart` file, import the face detection library:

```
import
'package:google_mlkit_face_detection/google_mlkit_face_d
etection.dart';
```

3. Add a new async method called `detectFace()`. The method takes `File` as a parameter and returns a future of type `String`:

```
Future<String> detectFace(File image) async {}
```

4. At the top of the `detectFace()` method, declare a `String` for the result and an `InputImage` that takes the file passed as an argument:

```
    String result = '';
    final InputImage inputImage =
InputImage.fromFile(image);
```

5. Also declare `FaceDetectorOptions`, enabling `Classification`, `Landmark`, and `Tracking`, and setting `FaceDetectorMode` to `accurate`:

```
final options = FaceDetectorOptions(
      enableClassification: true,
      enableTracking: true,
      enableLandmarks: true,
      performanceMode: FaceDetectorMode.accurate,
    );
```

6. Create a `FaceDetector` instance with the options you have set:

```
final faceDetector = FaceDetector(options: options);
```

7. Call the `faceDetector` `processImage` method to retrieve a list of `Face` objects:

```
final List<Face> faces = await
faceDetector.processImage(inputImage);
```

8. Set the result so that it tells users the number of faces in the picture, then for each item in the list, print the probability of the face smiling and having the left and right eyes open:

```
result = 'There are ${faces.length} face(s) in your
picture \n';
for (Face face in faces) {
  result += 'Face #$num: \n';
  result += 'Smiling: ${face.smilingProbability ?? 0 *
100}% \n';
  result +=
          'Left Eye Open:
${(face.leftEyeOpenProbability ?? 0) * 100}% \n';
```

```
  result +=
          'Right Eye Open:
${(face.rightEyeOpenProbability ?? 0) * 100}% \n';

}
```

9. At the bottom of the `detectFace` method, return the `result` string:

```
return result;
```

10. In the `picture.dart` file, add another `Row` widget under the first one in the `build` method. To the `Row` widget, in the `children` parameter, add an `ElevatedButton` that calls the `detectFace()` method and shows the results in the results screen:

```
Row(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: [
    ElevatedButton(
      child: const Text('Face Recognition'),
      onPressed: () {
        final image = File(widget.picture.path);
        MLHelper helper = MLHelper();
        helper.detectFace(image).then((result) {
          Navigator.push(
            context,
            MaterialPageRoute(
              builder: (context) =>
ResultScreen(result)));
            });
      },
    ),
  ],
),
```

11. Run the app, smile, keep your eyes open, and take a selfie. The results should look similar to the following screenshot:

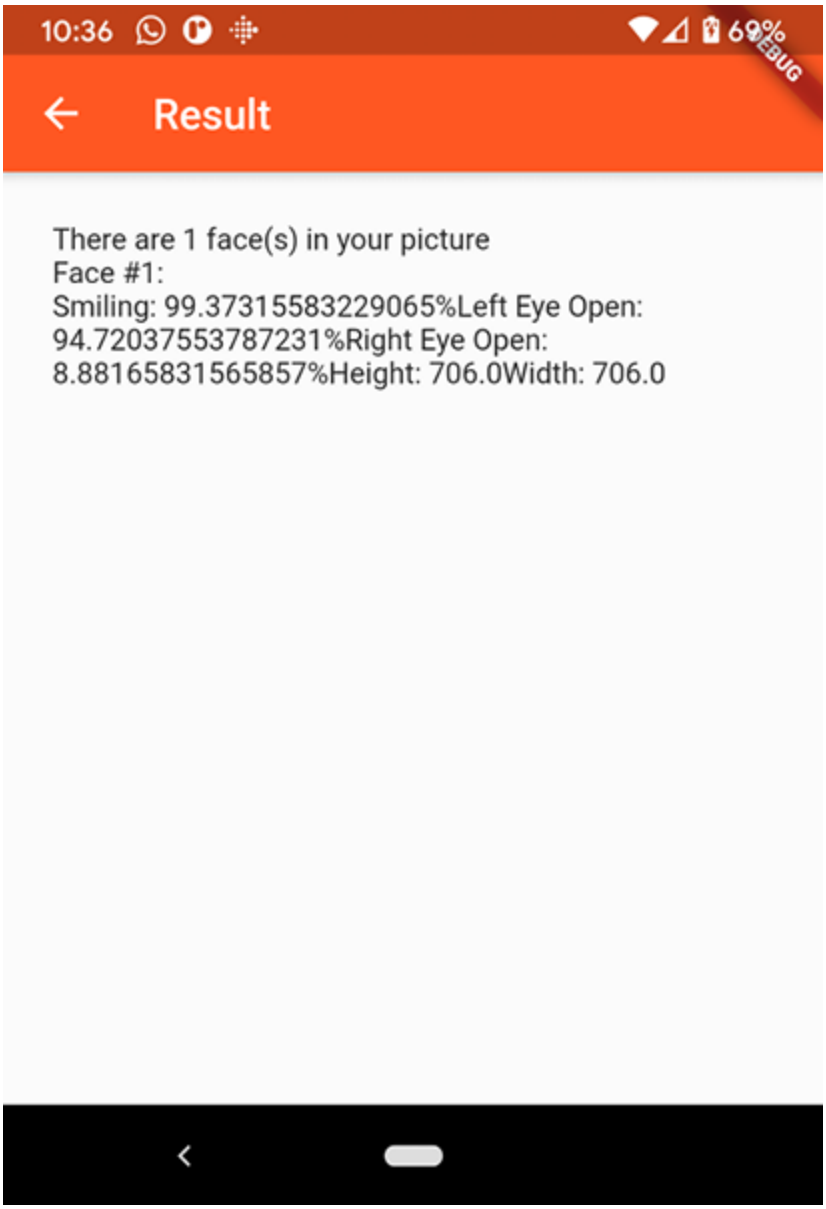


Figure 14.5: ML face detection

How it works...

Among its services, ML Kit provides a **face detection** API: you can use it to detect faces in an image, identify single parts of a face, such as eyes, mouth, and nose, get the contours of detected faces and parts, and identify whether a face is smiling and has the eyes open or closed.

Note

ML Kit provides a face *detection* service, **not** a face *recognition* one. This means that while you can identify faces in a picture, you cannot recognize people.

There are several use cases to implement face detection: among others, you can create avatars, edit pictures, or categorize your images.

In this recipe, after taking a picture, you identified the faces in the image and gave your user some information about them: whether they had their eyes open and were smiling.

Face detection is performed on the device, and no connection is required to use it.

You used a pattern similar to the previous recipes in this chapter: you got an image, sent it to the API, and retrieved and showed the results.

In this case, the object you used was `FaceDetector`, with its `options`:

```
FaceDetectorOptions(
      enableClassification: true,
      enableTracking: true,
      enableLandmarks: true,
      performanceMode: FaceDetectorMode.accurate,
    );
final faceDetector = FaceDetector(options: options);
```

Note that the `faceDetectorOptions` object allows specifying a few options:

- `enableclassification` specifies whether to add attributes such as smiling and eyes open.
- `enableLandmarks` specifies whether to add `FaceLandmarks`, which are points on a face, such as eyes, nose, and mouth.
- `enableTracking` is useful in videos, where more frames are available with the same ID. When enabled, the detector will keep the same ID for each face in the subsequent frames.

- `mode` lets you choose whether to process the image accurately or fast.

The `processImage` method, when called on `FaceDetector`, returns a list of `Face` objects:

```
 List<Face> results = await await
faceDetector.processImage(inputImage);
```

`Face` contains several properties: the ones you used in this recipe are `smilingProbability`, `leftEyeOpenProbability`, and `rightEyeOpenProbability`. They can all contain a value between `0` and `1`, where `1` is the highest probability level and `0` is the lowest.

Note

`Face` also contains the `FaceContour` objects, which contain information about the position (contours) of the face in a picture. You can use it to draw shapes around faces in a picture.

See also

The ML Kit face identification feature can also track faces in video streams. For a full list of the capabilities of this API, see the official guide at https://developers.google.com/ml-kit/vision/face-detection.

Identifying a language

Identifying a language can be useful when you get some text information from your users, and you need to respond in the appropriate language.

This is where the ML Kit language package comes to help. By passing some text to the package, you are able to recognize the language of the text. Once you know the language, you can later translate it into one of the other supported languages.

Getting ready

Before following this recipe, you should have completed the *Using the device camera* and *Recognizing text from an image* recipes in this chapter.

How to do it...

In this recipe, you will build a screen where users can type some text, and you will identify the language of the text. Follow these steps:

1. Add the latest version of the `google_mlkit_language_id` package to your project:

```
flutter pub add google_mlkit_language_id
```

2. In the `ml.dart` file, import the new package:

```
import
'package:google_mlkit_language_id/google_mlkit_language_
id.dart';
```

3. Add a new async method called `identifyLanguage`. The method takes `String` as a parameter and returns a future of type `String`:

```
Future<String> identifyLanguage(String text) async {}
```

4. At the top of the `indentifyLanguage` method, declare four variables: `String`, a `FirebaseLanguage` instance, `LanguageIdentifier`, and a list of `LanguageLabel` objects:

```
String result = '';
final languageIdentifier =
LanguageIdentifier(confidenceThreshold: 0.5);
```

5. Call the `languageIdentifier.identifyPossibleLanguages()` method to retrieve a list of `IdentifiedLanguage` objects:

```
final List<IdentifiedLanguage> languages =
  await
languageIdentifier.identifyPossibleLanguages(text);
```

6. Run a `for..in` loop over the list, and for each `IdentifiedLanguage` that was identified, add to the `result` string the language code and the confidence level, then return the result:

```
 for (IdentifiedLanguage language in languages) {
      result +=
          'Language: ${language.languageTag} -
Confidence: ${language.confidence * 100}%';
    }
    return result;
```

7. In the `lib` folder of your project, create a new file called `language.dart`.
8. At the top of the `language.dart` file, import `material.dart`, `ml.dart`, and `result.dart`:

```
import 'package:flutter/material.dart';
import 'ml.dart';
import 'result.dart';
```

9. Under the `import` statements, declare a stateful widget, and call it `LanguageScreen`:

```
class LanguageScreen extends StatefulWidget {
  const LanguageScreen({super.key});
  @override
  State<LanguageScreen> createState() =>
_LanguageScreenState();
}
class _LanguageScreenState extends
State<LanguageScreen> {
  @override
  Widget build(BuildContext context) {
    return const Placeholder();
  }
}
```

10. At the top of the `_LanguageScreenState` class, declare `TextEditingController` and call it `txtLanguage`:

```
final TextEditingController txtLanguage =
TextEditingController();
```

11. Override the `dispose` method, and dispose of the `txtLanguage` controller:

```
  @override
  void dispose() {
    txtLanguage.dispose();
    super.dispose();
  }
```

12. In the `build` method, return a `Scaffold`, and in its body, add a `TextField` that allows users to insert some text and a button that will call the `identifyLanguage` method and will call the result screen, as shown in the following code snippet:

```
return Scaffold(
      appBar: AppBar(
        title: const Text('Language Detection'),
      ),
      body: Padding(
        padding: const EdgeInsets.all(32),
        child: Column(
          mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
          children: [
            TextField(
              controller: txtLanguage,
              maxLines: 5,
              decoration: const InputDecoration(
                labelText: 'Enter some text in any
language',
              ),
            ),
            Center(
                child: ElevatedButton(
              child: const Text('Detect Language'),
              onPressed: () {
                MLHelper helper = MLHelper();
```

```
helper.identifyLanguage(txtLanguage.text).then((result)
{
                  Navigator.push(
                      context,
                      MaterialPageRoute(
                          builder: (context) =>
ResultScreen(result)));
                });
              },
            ))
          ],
        ),
      ),
    );
```

13. In the `main.dart` file, import `language.dart`:

```
import './language.dart';
```

14. In the `home` parameter of `MaterialApp`, set `LanguageScreen`:

```
home: const LanguageScreen(),
```

15. Run the app and insert some text in any language (if you want to try something in Italian, just type `Mi piace la pizza`), and press the **Detect Language** button. You should see a result similar to the following screenshot:

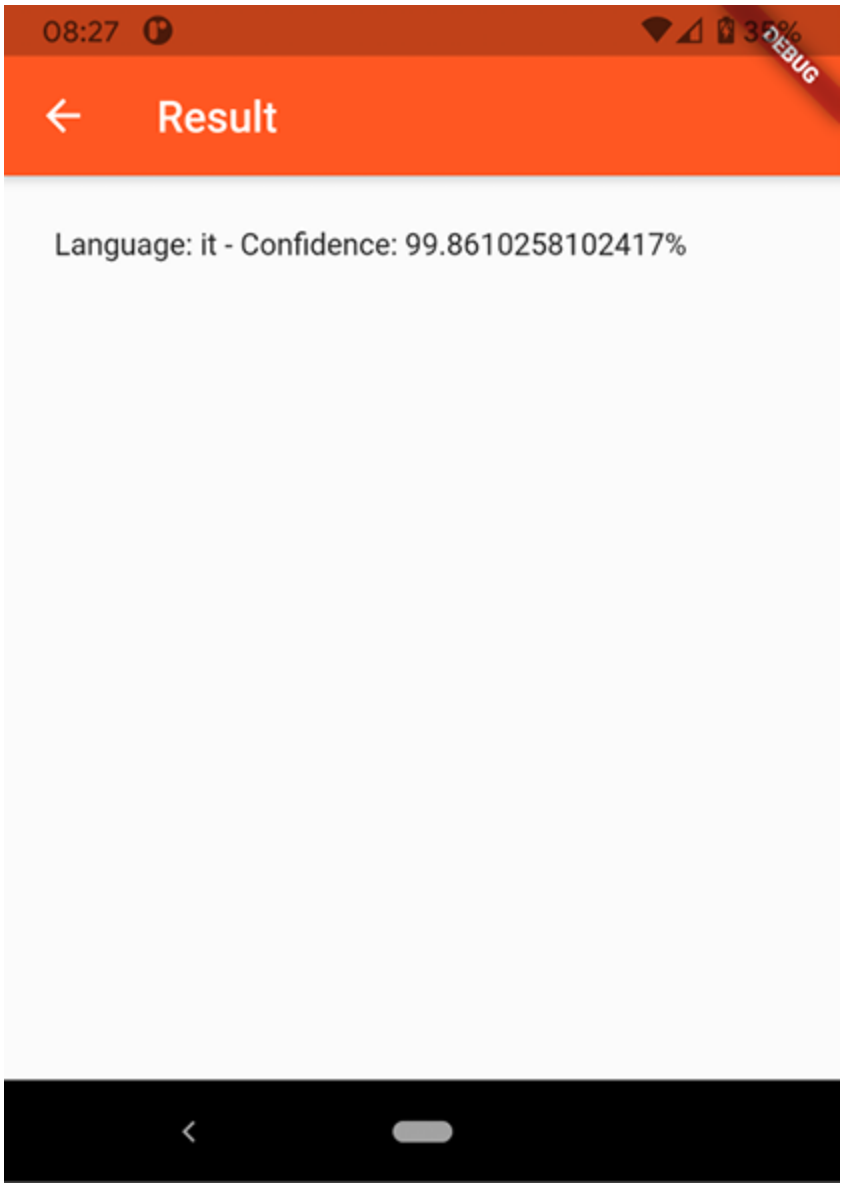


Figure 14.6: ML language identification

How it works...

ML Kit provides several tools that deal with language: **language identification**, text translation, and smart replies, where you can provide relevant responses to user messages, and entity extraction, which understands the content of the text, such as addresses, telephone numbers, or links.

In particular, in this recipe, you have implemented language recognition on a user-provided text.

> **Note**
>
> There are currently over 100 languages available in ML Kit. For a full list of the supported languages, have a look at the official list available at the following address https://developers.google.com/ml-kit/language/identification/langid-support.

This is useful when you want to recognize the language that your users speak and is the starting point of other services, such as translation.

To implement language recognition, you need an instance of the `LanguageIdentifier` class.

In this recipe, you declared it with:

```
final languageIdentifier =
LanguageIdentifier(confidenceThreshold: 0.5);
```

Note that again, you have a `confidenceThreshold`: this is used to filter out languages with confidence levels below a specific level: in this case, `0.5` or 50%.

To identify the language, you call the `LanguageIdentifier identifyPossibleLanguages()` method; this takes a string, which is the text you want to analyze, and returns a list of `IdentifiedLanguage` objects.

This is a list because a text could be valid in more than a single language. A great example of that is the expression "*an amicable coup d'etat*," which can be both English and French:

```
final List<IdentifiedLanguage> languages =
        await
languageIdentifier.identifyPossibleLanguages(text);
```

An `IdentifiedLanguage` object contains two properties that you used in the code in this recipe: `languageTag`, which you use to identify the language, and the confidence level, which is again a value between `0` and `1`, where `1` is the highest confidence level, and `0` is the lowest. In this recipe, you added the `languageTag` and `Confidence` level to the `result` string with the following instruction:

```
result +=    'Language: ${language.languageTag} -
Confidence: ${language.confidence * 100}%';
```

Most texts will only return a single language, but you should be ready to deal with ambiguous results.

See also

Once you recognize a language, you can also use this information to translate it to another; ML Kit also provides translation features. For more information, have a look at https://developers.google.com/ml-kit/language/translation.

Using TensorFlow Lite

While using pre-made models to recognize text, objects, and expressions is a powerful and useful feature to add to your apps, there are cases where you need to create your own models or use third-party models that can add virtually limitless features to your projects.

TensorFlow is an open source platform for creating and using ML models, and **TensorFlow Lite** is a lightweight platform specifically designed to be used on mobile and **Internet of Things (IoT)** devices.

Note

TensorFlow Hub contains hundreds of models already trained and ready to be used in your own apps. See the https://tfhub.dev/ page for more information.

While creating models and TensorFlow itself are beyond the scope of this book, in this recipe, you will use an open source TensorFlow model built by the creators of the `tflite_flutter` package.

Getting ready

Before following this recipe, you should have completed the *Using the device camera*, *Recognizing text from an image*, and *Identifying a language* recipes in this chapter.

How to do it...

In this recipe, you will add the `tflite_flutter` package and use its example model to classify a text and sort whether the sentiment of the text is positive or negative. Follow these steps:

1. Complete the setup procedure necessary to use the `tflite_flutter` package; this depends on your system and is available at the following address: https://pub.dev/packages/tflite_flutter, in the *Initial setup* section.
2. Add the latest version of the `tflite_flutter` package to your project, typing in your Terminal:

```
flutter pub add tflite_flutter
```

3. In the root of the app, create a new folder called `assets`.
4. Download the `text_classification.tflite` model from https://github.com/am15h/tflite_flutter_plugin/tree/master/example/assets to the `assets` folder.
5. Download the `text_classification_vocab.txt` vocabulary file from https://github.com/am15h/tflite_flutter_plugin/tree/master/example/assets to the `assets` folder.

6. Download the `classifier.dart` file from https://github.com/am15h/tflite_flutter_plugin/tree/master/example/lib to the `lib` folder.
7. In your `pubspec.yaml` file, add the `assets` folder to the `assets` node:

```
assets:
    - assets/
```

8. In the `Classifier` class, in the `classifier.dart` file, modify the `classify` method, as shown:

```
Future<int> classify(String rawText) async {
  await _loadModel();
  await _loadDictionary();
  List<List<double>> input = tokenizeInputText(rawText);
  var output = List<double>(2).reshape([1, 2]);
  _interpreter.run(input, output);
  var result = 0;
  if ((output[0][0] as double) > (output[0][1] as
double)) {
    result = 0;
  } else {
    result = 1;
  }
  return result;
}
```

9. Change the return value of the `_loadModel()` and `_loadDictionary()` methods to `Future` instead of `void`:

```
Future _loadModel() async {
…
Future _loadDictionary() async {
```

10. In the `ml.dart` file, add a new method called `classifyText` that takes a string and returns the result of the classification (`Positive sentiment` or `Negative sentiment`), as shown here:

```
Future<String> classifyText(String message) async {
  String result;
    Classifier classifier = Classifier();
    int value = await classifier.classify(message);
```

```
    if (value > 0) {
      result = 'Positive sentiment';
    } else {
      result = 'Negative sentiment';
    }
    return result;
}
```

11. In the `language.dart` file, in the `build` method, add a second `ElevatedButton` with the text `Classify Text` under the first one in the row:

```
ElevatedButton(
  child: const Text('Classify Text'),
    onPressed: () {
      MLHelper helper = MLHelper();

helper.classifyText(txtLanguage.text).then((result) {
          Navigator.push(
            context,
            MaterialPageRoute(
              builder: (context) =>
ResultScreen(result)));
          });
     },
])
```

12. Run the app and write `I like pizza`. This should return a positive sentiment.
13. Then write `Yesterday was a nightmare`. This should return a negative sentiment.

How it works...

You can use the `tflite_flutter` package when you want to integrate a TensorFlow Lite model in your Flutter apps. The advantages of using this package include the following:

- No need to write any platform-specific code.
- It can use any `tflite` model.

- It runs on the device itself (no need to connect to a server).

> **Note**
>
> A TensorFlow Lite model uses the `.tflite` extension. You can also convert existing TensorFlow models into TensorFlow Lite models. The procedure is available at https://www.tensorflow.org/lite/convert.

In the `assets` folder of the app, you placed two files: the `tflite` model and a vocabulary text file. The vocabulary file contains 10,000 words that are used by the model to retrieve the positive and negative sentiments.

> **Note**
>
> You can use this classification model, and all the models available in TensorFlow Hub, in your apps as they are released with an Apache 2.0 license (details here: https://github.com/tensorflow/hub/blob/master/LICENSE).

Before classifying the string, you need to load both the model and the dictionary. In this recipe, this is performed by the following two methods:

```
await _loadModel();
await _loadDictionary();
```

If you have a look inside `loadModel`, you will find a key function of this package:

```
_interpreter = await Interpreter.fromAsset(_modelFile);
```

This creates an `Interpreter` object using the `Interpreter.fromAsset` constructor method. `Interpreter` is an object required to run inference on a model. In other words, `Interpreter` contains the method that runs the model on text, an image, or any other input and returns the relevant identification information.

The next instruction in the `classify` method is as follows:

```
List<List<double>> input = tokenizeInputText(rawText);
```

This takes the string you want to classify and creates a list of single words that, combined, will give the sentiment of the statement.

The instruction that runs the model is as follows:

```
_interpreter.run(input, output);
```

You will notice that running the model is extremely fast, even on mobile devices, which is a feat in itself.

Being able to run any `tflite` model in your apps may add several benefits to your projects. Generating your own tensors might be even better; this requires some Python and ML knowledge, but there are several great resources to get you started if you are new to ML. See the *See also* section that follows for more details.

See also

If you are new to ML, TensorFlow offers great guides and tutorials to get started at the following link: https://www.tensorflow.org/learn

Summary

In this chapter, you explored several powerful features that can be added to your Flutter apps using Google ML Kit.

First, you learned how to use the device camera in your Flutter applications.

Next, you saw how to recognize text from an image using the ML Kit's Text Recognition API.

You saw how to scan barcodes and QR codes from images: this allows building apps for inventory management, ticket scanning, or any other project that involves reading barcodes.

You used ML Kit's Image Labeling API to identify objects and scenes in images. This is useful for building apps that can categorize or tag images without any user interaction.

You learned how to use the Face Detection API to identify faces in images and detect facial expressions.

You've seen how to identify languages in a text using the Language Identification API and developed a feature that could detect the language of a text input and return its language tag and confidence level, which can be useful in apps like translation or content filtering.

Finally, you used TensorFlow Lite and saw how to integrate TensorFlow Lite models into your Flutter applications. This has a wide range of ML capabilities specific to your needs.

Flutter Web and Desktop

Believe it or not, Flutter originally started as a fork of Chrome with a simple question — how fast can the web run if you don't worry about maintaining over 20 years of technical debt? The answer to that question is the blazingly fast mobile framework that we love.

Since Flutter 2.0, developers can create mobile, web, and desktop apps; this means that you can create apps that work on iOS, Android, the web, Windows, macOS, Linux, or embedded devices *with the same code base*.

While you could use exactly the same design and code for all operating systems and devices, there are cases where this might not be the optimal approach: an app screen designed for a smartphone might not be ideal for a large desktop, and not all packages are compatible with all systems. Also, setting up permissions depends on the target destination of your app.

Note

When creating a desktop app, you need to develop *in the same platform as your target app*: you need a Mac to develop for macOS, a Windows PC if you target Windows, and a Linux PC when targeting Linux.

This chapter will focus on how to develop and run your apps on web and desktop devices, and how to create responsive apps based on the screen size where your app is running.

In particular, we will cover the following topics:

- Creating a responsive app leveraging Flutter Web
- Running your app on macOS
- Running your app on Windows

- Deploying a Flutter website
- Responding to mouse events in Flutter Desktop
- Interacting with desktop menus

Note

Building Linux Desktop does not require proprietary software like Visual Studio or Xcode and may depend on the distro you are using.

For more information on building Linux desktop apps, see https://docs.flutter.dev/platform-integration/linux/building.

By the end of this chapter, you will know how to design your apps not only for mobile devices but also for the web and desktop.

Creating a responsive app leveraging Flutter Web

Running a web app with Flutter might be as simple as running the `flutter run -d chrome` command on your Terminal. In fact, most of the apps you create with Flutter may also run on the web without any change, but some plugins might only work for specific platforms, and you may need to adjust the UI for larger screens.

In this recipe, you will see how to make your layout **responsive**, building your app so that it can later be published to any web server. You will also see how to solve a **Cross-Origin Resource Sharing (CORS)** issue when loading images.

You will build an app that retrieves data from the Google Books API and shows text and images. After running it on your mobile emulator or device, you will then make it responsive so that when the screen is large, the books will be shown in two columns instead of one.

Getting ready

There are no specific requirements for this recipe, but in order to debug your Flutter apps for the web, you should have the Chrome browser installed. If you are developing on Windows, Edge will work as well.

How to do it...

In order to create a responsive app that also targets the web, follow these steps:

1. Create a new Flutter project and call it `books_universal`.
2. Add the latest version of the `http` package by typing in your Terminal:

```
flutter pub add http
```

3. In the `lib` directory of your project, create three new directories called `models`, `data`, and `screens`.
4. In the `models` directory, create a new file called `book.dart`.
5. In the `book.dart` file, create a class called `Book`, with the fields specified here:

```
class Book {
  final String id;
  final String title;
  final String authors;
  final String thumbnail;
  final String description;
}
```

6. In the `Book` class, create a constructor that sets all the fields:

```
 const Book(this.id, this.title, this.authors,
this.thumbnail, this.description);
```

7. Create a named `factory` constructor, called `fromJson`, that takes a `Map` and returns a `Book`, as shown in the code sample:

```
 factory Book.fromJson(Map<String, dynamic> parsedJson)
{
   final String id = parsedJson['id'];
   final String title = parsedJson['volumeInfo']
['title'];
   String image = parsedJson['volumeInfo']
['imageLinks'] == null
   ? '' : parsedJson['volumeInfo']['imageLinks']
['thumbnail'];
   image.replaceAll('http://', 'https://');
   final String authors = (parsedJson['volumeInfo']
['authors'] ==
   null) ? '' :   parsedJson['volumeInfo']
['authors'].toString();
   final String description =
   (parsedJson['volumeInfo']['description'] == null)
   ? ''
   : parsedJson['volumeInfo']['description'];
   return Book(id, title, authors, image, description);
 }
```

8. In the `data` directory, create a new file and call it `http_helper.dart`.
9. At the top of the `http_helper` file, add the required `import` statements, as shown here:

```
import 'package:http/http.dart' as http;
import 'dart:convert';
import 'dart:async';
import 'package:http/http.dart';
import '../models/book.dart';
```

10. Under the `import` statements, add a class and call it `HttpHelper`:

```
 class HttpHelper {}
```

11. In the `HttpHelper` class, add the strings and `Map` required to build the `Uri` object that will connect to the Google Books API:

```
static const String authority = 'www.googleapis.com';
static const path = '/books/v1/volumes';
```

12. Still in the `HttpHelper` class, create a new asynchronous method, called `getBooks`, that takes a `String` for the query and returns a `Future` of a `List` of `Book` objects, as shown here:

```
Future<List<Book>> getBooks(String query) async {
  Map<String, dynamic> params = {
     'q': query,
     'maxResults': '40',
   };
  Uri uri = Uri.https(authority, path, params);
  Response result = await http.get(uri);
  if (result.statusCode == 200) {
    final jsonResponse = json.decode(result.body);
    final booksMap = jsonResponse['items'];
    List<Book> books = booksMap.map<Book>((i) =>
     Book.fromJson(i)).toList();
    return books;
  } else {
    return [];
  }
}
```

13. In the `screens` directory, create a new file called `book_list_screen.dart`.

14. At the top of the `book_list_screen.dart` file, add the required imports:

```
import 'package:flutter/material.dart';
import '../models/book.dart';
import '../data/http_helper.dart';
```

15. Under the `import` statements, create a new stateful widget and call it `BookListScreen`:

```
class BookListScreen extends StatefulWidget {
  const BookListScreen({super.key});
 @override
 State<BookListScreen> createState() =>
_BookListScreenState();
```

```
}
class _BookListScreenState extends
State<BookListScreen> {
  @override
  Widget build(BuildContext context) {
    return const Placeholder();
  }
 }
```

16. At the top of the `_BookListScreenState` class, create two variables: a `List` of books called `books`, and a Boolean called `isLargeScreen`:

```
List<Book> books = [];
```

17. In the `_BookListScreenState` class, override the `initState` method, and then call the `getBooks` method to set the value of the `books` variable, as shown here:

```
@override
void initState() {
  super.initState();
  final HttpHelper helper = HttpHelper();
  helper.getBooks('flutter').then((List<Book> value) {
    setState(() {
      books = value;
    });
  });

}
```

18. At the top of the `build` method, use the `MediaQuery` class to read the width of the current device, and based on its value, set the `isLargeScreen` Boolean to `true` when the number of device-independent pixels is higher than `600`:

```
bool isLargeScreen;
if (MediaQuery.of(context).size.width > 600) {
   isLargeScreen = true;
 } else {
   isLargeScreen = false;
 }
```

19. Still in the `build` method, return a `Scaffold` that, in its `body`, contains a responsive `GridView`. Based on the value of the `isLargeScreen` variable, set the number of columns to `2` or `1`, and `childAspectRatio` to `8` or `5`, as shown in the following code sample:

```
 return Scaffold(
   appBar: AppBar(title: const Text('Books')),
   body: GridView.count(
     childAspectRatio: isLargeScreen ? 8 : 5,
     crossAxisCount: isLargeScreen ? 2 : 1,
     children: List.generate(books.length, (index) {
       return ListTile(
         title: Text(books[index].title),
         subtitle: Text(books[index].authors),
         leading: CircleAvatar(
           backgroundImage: (books[index].thumbnail) ==
'' ? null :
            NetworkImage(books[index].thumbnail),
         ),
       );
   }))
);
```

20. Edit the `main.dart` file so that it calls the `BookListScreen` widget:

```
import 'package:flutter/material.dart';
import 'screens/book_list_screen.dart';
void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
      primarySwatch: Colors.blue,
    ),
    home: const BookListScreen(),
  );
}}
```

21. Run the app on your mobile device. You should see an app screen similar to the screenshot:

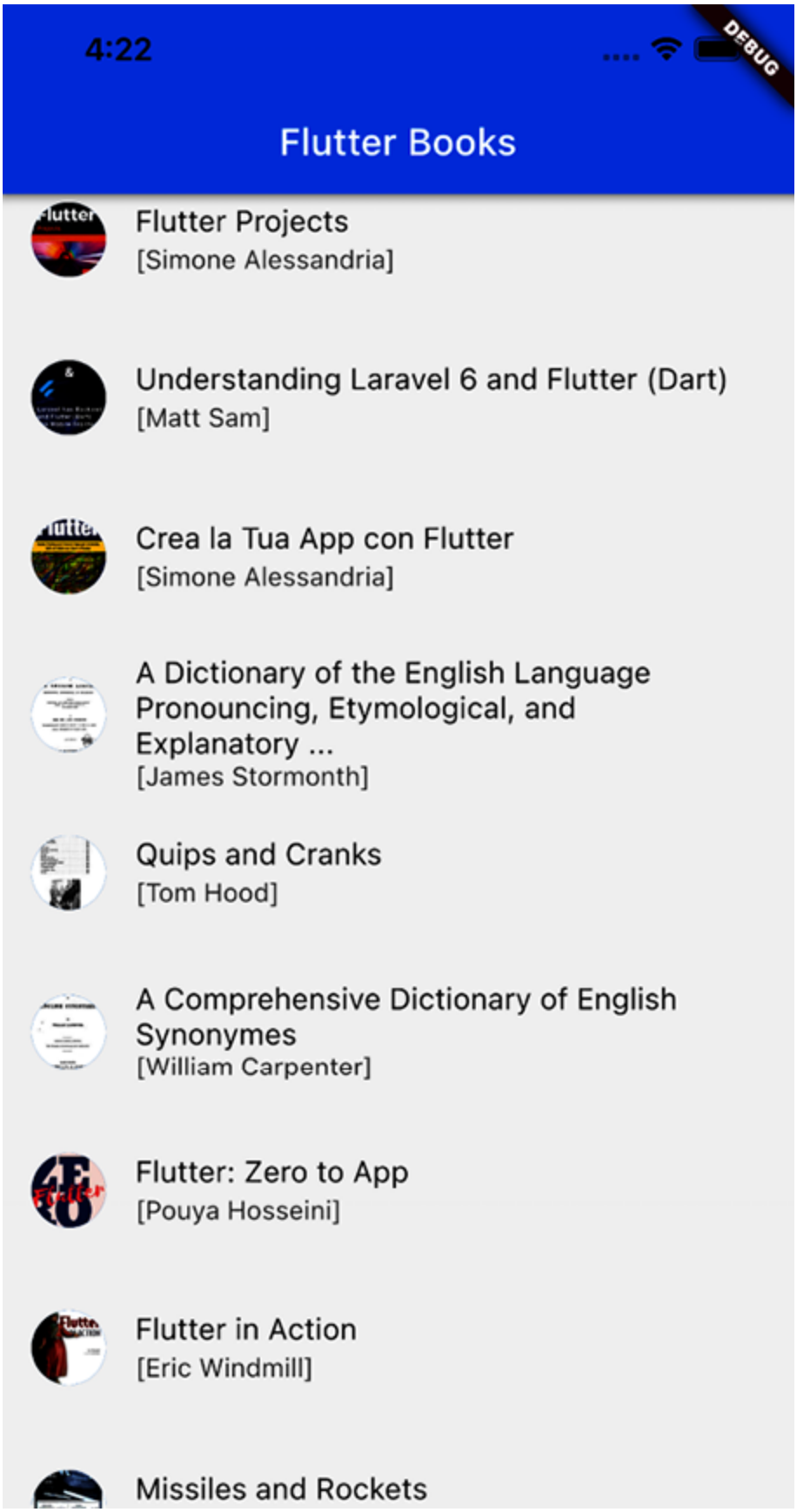

4:22
DEBUG
Flutter Books
Flutter Projects
[Simone Alessandria]
Understanding Laravel 6 and Flutter (Dart)
[Matt Sam]
Crea la Tua App con Flutter
[Simone Alessandria]
A Dictionary of the English Language Pronouncing, Etymological, and Explanatory ...
[James Stormonth]
Quips and Cranks
[Tom Hood]
A Comprehensive Dictionary of English Synonymes
[William Carpenter]
Flutter: Zero to App
[Pouya Hosseini]
Flutter in Action
[Eric Windmill]
Missiles and Rockets



Figure 15.1: App on a mobile device

22. Stop the app, and run it on Chrome; you can select the device in your editor, or in the Terminal window in your project's directory, run the command:

```
flutter run -d chrome
```

23. Note that the app is running on your web browser and showing two columns, but the images are not showing:

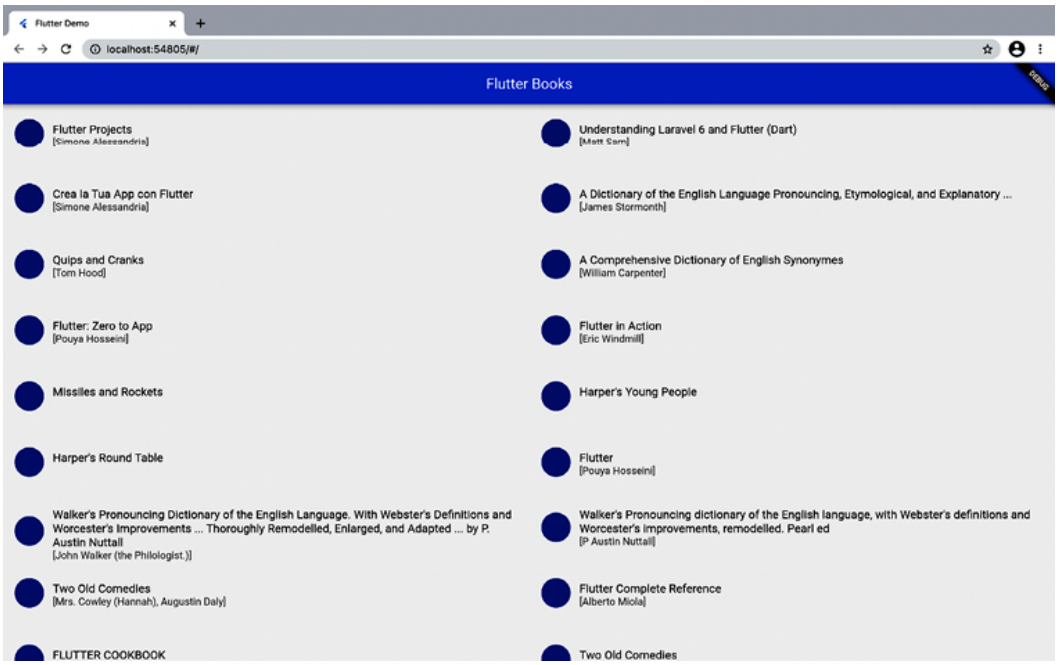


Figure 15.2: App in a browser — images are not showing

24. Stop the app.
25. In the Terminal, run the following command:

```
flutter run -d chrome --web-renderer html
```

26. This time, note that the images show correctly:

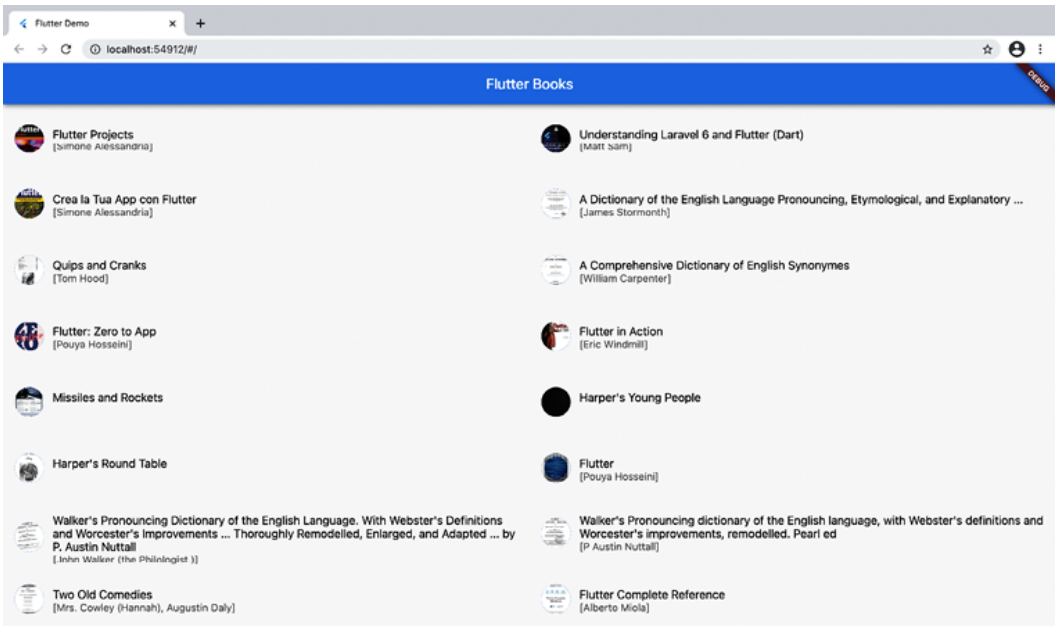


Figure 15.3: App in a browser, with images showing

How it works...

When you retrieve data from a web API, the first step is usually to define which fields you are interested in. In this case, we only want to show a selection of the available data that Google Books publishes: in particular, the ID, title, authors, thumbnail, and a description of each book.

When a web service returns data in JSON format, in Dart and Flutter, you can treat it as a `Map`. The name of the fields are strings, and the values can be any data type; that's why we created a constructor that takes a `Map<String, dynamic>` and returns a `Book`.

In the `Book.fromJson` constructor, based on the JSON format returned by the service, you read the relevant data. Note the instructions:

```
 String image = parsedJson['volumeInfo']['imageLinks'] ==
null
   ? ''
   : parsedJson['volumeInfo']['imageLinks']['thumbnail'];
 image.replaceAll('http://', 'https://');
```

When you read data returned by a web service, it's always recommended to check whether the data you expect is available or `null`.

In the case of the `imagelinks` field, Google Books returns an HTTP address; as this is not enabled by default on several platforms, including iOS and Android, it may be a good idea to change the address from `http://` to `https://`. This is the purpose of the `replaceAll` method called on the image string.

Note

You can also enable retrieving data through an `http` connection; the procedure depends on your target system. See https://docs.flutter.dev/release/breaking-changes/network-policy-ios-android for more information.

The `http.get` method requires a `Uri` object. You build a `Uri`, passing the `authority` (domain name) and the `path`. If you want to add one or more parameters, you add them with a `Map`:

```
 final String authority = 'www.googleapis.com';
 final String path = '/books/v1/volumes';
 Map<String, dynamic> params = {'q':'flutter dart',
'maxResults': '40', };
```

When building a `Uri`, `authority` and `path` are required; the `parameters` are optional:

```
 Uri uri = Uri.https(authority, path, params);
```

There are several strategies to make your app responsive: one of them is using the `MediaQuery` class. In particular, `MediaQuery.of(context).size`

allows you to retrieve the device screen size in pixels. These are actually logical (or device-independent) pixels. Most devices use a `pixelRatio` to make images and graphics uniform across devices.

Note

If you want to know the number of physical pixels in your device, you can multiply the size and the `pixelRatio`, like this:

```
MediaQuery.of(context).size.width *
MediaQuery.of(context).devicePixelRatio.
```

The `MediaQuery.of(context).size` property returns a `Size` object; this contains a width and a height. In our example, we only need to get the device width. We consider a screen "large" when it has a width higher than 600 logical pixels. In this case, we set the `isLargeScreen` Boolean value to `true`. This is an arbitrary measure and may depend on the objects you want to show in your designs, or the orientation of the device (portrait or landscape).

Based on the value of the `isLargeScreen` Boolean value, we leverage the `GridView` `childAspectRatio` and `crossAxisCount`:

```
childAspectRatio: isLargeScreen ? 8 : 5,
crossAxisCount: isLargeScreen ? 2 : 1,
```

By default, each box in a `GridView` is a square with the same height and width. By changing the `childAspectRatio`, you can specify a different aspect ratio. For example, the width of the child will be 8 when the screen is large, and 5 when it's not. The `crossAxisCount` value in this example sets the number of columns in the `GridView`.

Note this instruction:

```
flutter run -d chrome
```

The `-d` option allows you to specify which device you want to run your app on — in this case, the web browser. But if you run your app with this option, the images in the `GridView` don't show. The solution is running the app with this command:

```
flutter run -d chrome --web-renderer html
```

Before it can be displayed on a browser, your app must be transformed (or rendered) in a language compatible with your browser. Flutter uses two different web-renderers: `HTML` and `CanvasKit`.

The default renderer on desktop and web is `CanvasKit`, which uses WebGL to display the UI. This requires access to the pixels of the images you want to show, while the `HTML` renderer uses the `<img>` HTML element to show images.

When you try to show the images in these recipes with the default web-renderer, you encounter a `CORS` error.

Note

CORS, as mentioned earlier, stands for **Cross-Origin Resource Sharing**. For security reasons, browsers block JavaScript requests that originate from a different destination. For example, if you want to load a resource from `www.thirdpartydomain.com`, and the origin of the request is `www.mydomain.com`, the request will be automatically blocked.

When you use an `<IMG>` element, cross-origin requests are allowed.

See also...

For more information about web-renderers, and specifically the issue with images, have a look at https://docs.flutter.dev/development/platform-integration/web/web-images#flutter-renderers-on-the-web.

While, for simple results, parsing JSON manually is an option, when your classes become more complex, some automation will greatly help you. The `json_serializable` package is available at https://pub.dev/packages/json_serializable.

For a thorough overview of the differences between the two web-renderers and when to use them, see https://flutter.dev/docs/development/tools/web-renderers.

If you are interested in how logical pixels work, and how they are different from physical pixels, see https://material.io/design/layout/pixel-density.html.

Running your app on macOS

You can compile Flutter projects to *native* macOS, Windows, and Linux apps.

In this recipe, you will see how to run your app on a Mac, and solve a specific permission issue that prevents your code from retrieving data from the web.

Getting ready

Before following this recipe, you should have completed the app in the previous recipe, *Creating a responsive app leveraging Flutter Web*.

In order to develop for macOS with Flutter, you should also have Xcode and CocoaPods installed on your Mac, as described in *Chapter 1*, *Getting Started with Flutter*.

How to do it...

In order to run the app you have built on your Mac, implement the following steps:

1. In a Terminal window, run this command:

```
flutter devices
```

2. In the list of devices returned by the command, make sure that **macOS (desktop)** now shows among the available devices.
3. From your editor, choose macOS as your target and run the app, or in the Terminal window, run the command:

```
flutter run -d macos
```

> **Note**
>
> Note that the app does not show any book, as shown in *Figure 15.4*, and you get a `SocketException` error (*Figure 15.5*):

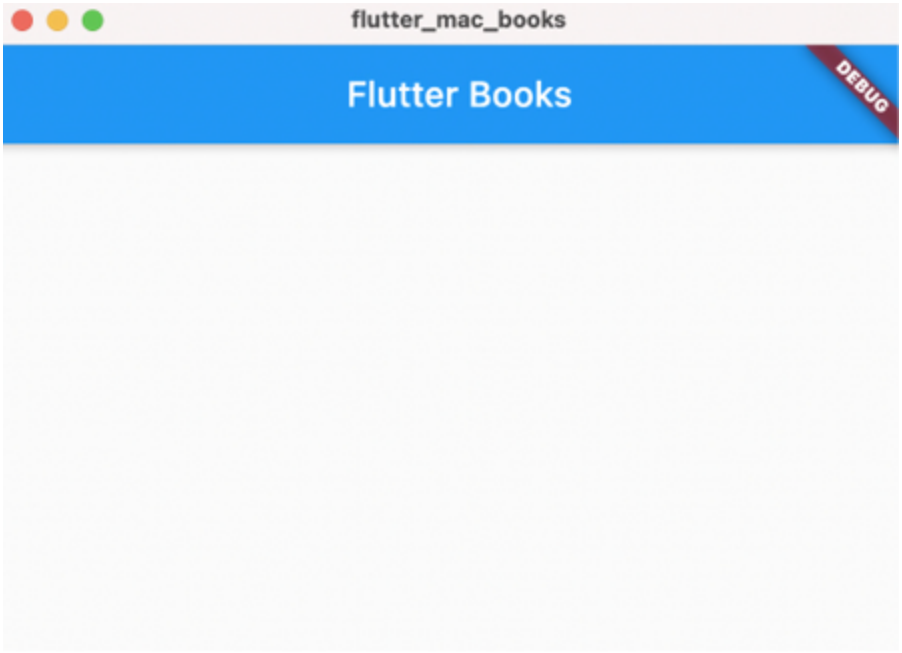


Figure 15.4: No books showing on the macOS app

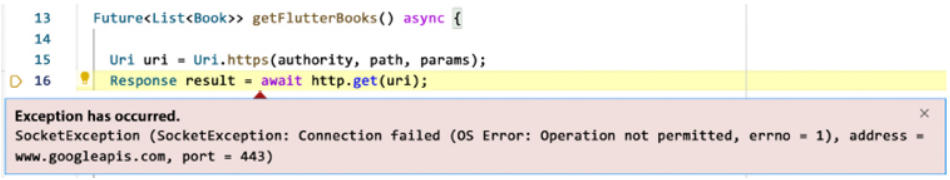


Figure 15.5: SocketException error on macOS

4. Open the `macos/Runner/DebugProfile.entitlements` file in your project and add this key:

```
<key>com.apple.security.network.client</key>
<true/>
```

5. Open the `macos/Runner/Release.entitlements` file and add the same key you added in the `DebugProfile.entitlements` file.
6. Run the app again, and note that the books' data and images are showing correctly.
7. In your Terminal, run the command:

```
flutter build macos
```

How it works...

You have two options in order to run your app on a specific device: one is using the Flutter **command-line interface** (**CLI**) and specifying the device where you want to run your app, as with this instruction:

```
flutter run -d macos
```

The other way is choosing the device from your editor. With VS Code, you will find your devices in the bottom-right corner of the screen. In Android Studio (and IntelliJ IDEA), you will find it in the top-right corner of the screen.

As with Android and iOS, an app build for macOS requires specific permissions that must be declared before running the app; these permissions are called `entitlements` in macOS. In a Flutter project, you will find two files for the entitlements: one for development and debugging, called `DebugProfile.entitlements`, and another for release, called `Release.entitlements`. In these two files, you should add the entitlements that are required for your app. In the example in this recipe, a client connection is required, so you just need to add the network client entitlement with the key:

```
<key>com.apple.security.network.client</key>
<true/>
```

Note

Even if you can debug your app by just adding your entitlements in the `DebugProfile.entitlements` file, I recommend you always add them in the `Release.entitlements` file as well so that when you actually publish your app, you do not need to copy the entitlements there.

See also

For a full and updated list of the desktop support capabilities of Flutter, see https://flutter.dev/desktop. If you later want to publish your app to the Mac App Store, see https://docs.flutter.dev/deployment/macos.

Running your app on Windows

Most desktop computers run on Windows, and that's a fact. Being able to deploy an app built with Flutter to the most widespread desktop operating system is a huge add-on to the Flutter framework.

In this recipe, you will see how to run your apps on Windows.

Getting ready

Before following this recipe, you should have completed the app in the first recipe in this chapter, *Creating a responsive app leveraging Flutter Web*.

In order to develop for Windows with Flutter, you should also have the full version of Visual Studio (this is *not* Visual Studio *Code*) with the Desktop Development with C++ workload installed on your Windows PC.

You can download Visual Studio for free at https://visualstudio.microsoft.com.

Note

Visual Studio is a full-featured **integrated development environment (IDE)**, *mainly* designed for development using the .NET framework and other languages like C++, Python, and more.

Visual Studio Code is a lightweight editor, *mainly* designed for frontend applications (including Dart and Flutter, of course!).

How to do it...

In order to run the app that you built in the previous recipe on a Windows desktop, take the following steps:

1. In a Command Prompt window, get to the project you completed in the first recipe in this chapter, and run the command:

```
flutter devices
```

2. In the list of devices returned by the command, note that **Windows (desktop)** shows as a device.
3. Run the `flutter doctor` command, and note that `Visual Studio – develop for Windows` is showing, as shown in the screenshot:

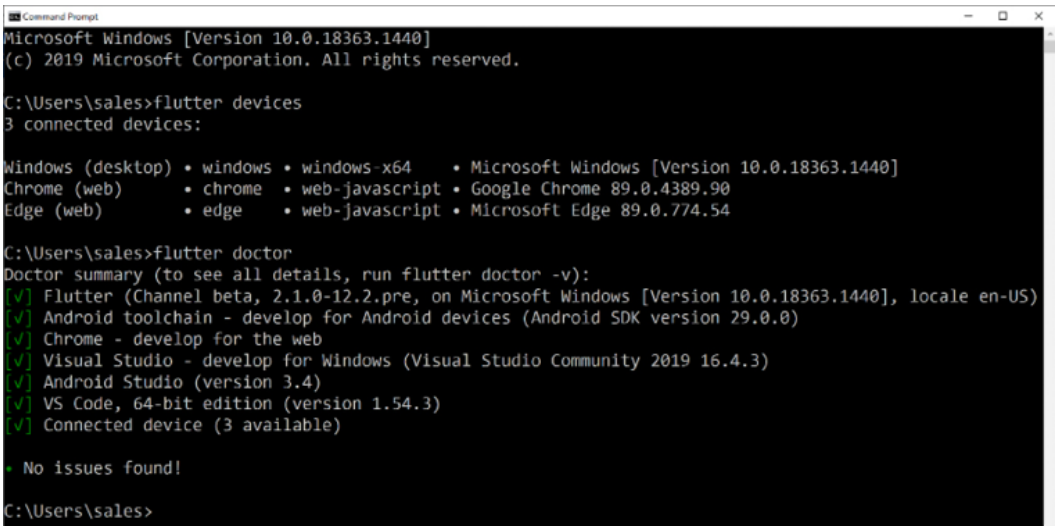


Figure 15.6: Flutter doctor results

4. Run the app on Windows with your editor, or by calling the command:

```
flutter run -d Windows
```

5. Note that the app runs correctly, and the books and images are shown on the screen.
6. In your Terminal, run the command:

```
flutter build Windows
```

How it works...

You have two options in order to run your app on a specific device: one is using the Flutter CLI and specifying the device where you want to run your app, as with this instruction:

```
flutter run -d windows
```

The other way is choosing the device from your editor. With VS Code, you find your devices in the bottom-right corner of the screen. In Android

Studio (and IntelliJ IDEA), you find it in the top-right corner of the screen.

When you build a Flutter app for Windows, a native executable is created. The `build\windows\runner\Release` folder in your Flutter project contains the executable of your app, for example, in this recipe, the file `books_universal.exe`.

If you want to distribute your app to other Windows desktops, different from your development machine, you'll need to copy the entire `Release` folder. This contains your executable and all other necessary files:

- The `data` folder: This contains assets, fonts, and other resources that you use in your app
- The `flutter` folder: This contains the Flutter engine and other required libraries (only when required)
- `flutter_windows.dll`: A C++ runtime library necessary to run the app

Different from what happens on a Mac, in the Windows client, HTTP connections are currently enabled by default.

See also...

Visual Studio is a full IDE available for Windows and macOS; it allows you to develop frontend and backend applications and supports most modern languages. For more information, have a look at https://visualstudio.microsoft.com.

Deploying a Flutter website

Once you build your Flutter app for the web, you can deploy it to any web server. One of your options is using Firebase as your hosting platform.

In this recipe, you will deploy your project as a website to Firebase.

Getting ready

Before following this recipe, you should have completed the first recipe in this chapter, *Creating a responsive app leveraging Flutter Web*.

How to do it...

In order to deploy your web app to the Firebase hosting platform, please take the following steps:

1. Open a Terminal window and move it to your project folder.
2. Type the following command:

```
flutter build web --web-renderer html
```

3. Open your browser and go to the Firebase console at this address: https://console.firebase.google.com/u/0/.
4. Create a new Firebase project (you can also use an existing one).

> **Pro Tip**
>
> For a refresher on Firebase projects, see the *Configuring a Firebase app* recipe in *Chapter 13*, *Using Firebase*.

5. In your Terminal, type the following command:

```
firebase login
```

6. If you are not already logged in, from the browser window that asks for your login details, confirm your credentials and allow the required permissions. At the end of the process, note the success message in your browser:



Figure 15.7: Successful Firebase message

7. In your Terminal, note the success message: **Success! Logged in as [yourusername]**.
8. Type this command:

```
firebase init
```

9. When prompted, choose the **Hosting: Configure files for Firebase Hosting and (optionally) set up GitHub Action deploys** option.
10. When prompted, choose the **Use an existing project** option and the project you created above.
11. At the public directory prompt, type `build/web` to choose the files that will be deployed.
12. When prompted, confirm that you want to configure a single-page application.
13. When prompted, choose whether you want to automatically deploy your changes to GitHub (select **no** for this example).
14. When prompted, answer **no** when asked whether you want to overwrite the `index.html` file. A screenshot of the full configuration process is shown below:



Figure 15.8: Firebase hosting setup

15. In your Terminal, type the following:

```
firebase deploy
```

16. At the end of the process, copy the URL of your project and paste it into a web browser (usually `https://[yourappname].web.app`). Your app has been published!

How it works...

When using Flutter, the web is just another target for your apps. When you use the following command:

```
flutter build web
```

what happens is that the Dart code gets compiled to JavaScript and then minified for use in production. After that, your source *can be deployed to any web server*, including Firebase. As you saw in the first recipe in this chapter, you can also choose a web-renderer to suit your needs.

I would recommend using Flutter for the web when you want to build progressive web applications, single-page applications, or when you need to port mobile apps to the web. It is not recommended for static text-based HTML content, even though Flutter fully supports this scenario as well.

The reason is that Flutter's main strength is in building interactive apps with a rich user experience. You can probably build static websites more efficiently using web development tools like HTML, CSS, and JavaScript, as they offer smaller bundle sizes and easier **search engine optimization (SEO)**.

The Firebase **CLI** makes publishing a web app with Flutter extremely easy. Here are the steps that you should take, and which you followed in this recipe:

1. You create a Firebase project.
2. You install the Firebase CLI (only once for each developing machine).
3. You run `firebase login` to log in to your Firebase account.
4. You run `firebase init`. For this step, you need to provide Firebase with some information, including your target project and the position of the files that will be published.

Pro Tip

Make sure you answer **no** when asked whether you want to overwrite the `index.html` file; otherwise, you'll have to build your web app again.

Once you have initialized your Firebase project, publishing it just requires typing the following:

```
firebase deploy
```

Unless you set up a full domain, your web address will be a third-level domain, like `yourapp.web.app`.

Publishing your site is just as easy: you only need to copy the files in the `build/web` folder of your project to your web server. As the Dart code is compiled to JavaScript, no further setup is needed on your target server.

See also...

Another common way to publish your Flutter web apps is by leveraging GitHub Pages. For a detailed guide of the steps required to deploy your Flutter web app to GitHub Pages, see https://pahlevikun.medium.com/compiling-and-deploying-your-flutter-web-app-to-github-pages-be4aeb16542f.

If you want to learn more about GitHub Pages themselves, see https://pages.github.com/.

Responding to mouse events in Flutter Desktop

While, with mobile devices, users generally interact with your apps through *touch* gestures, with bigger devices, they may use a *mouse*, and from a developer perspective, mouse input is different from touch.

In this recipe, you will learn how to respond to common mouse gestures such as click and right-click. In particular, you will make the user select and deselect items in a `GridView`, and change the color of the background to give some feedback to your users.

Getting ready

Before following this recipe, you should have completed the first recipe in this chapter, *Creating a responsive app leveraging Flutter Web*.

How to do it…

In order to respond to mouse gestures within your app, please follow these steps:

1. Open the `book_list_screen.dart` file in your project.
2. At the top of the `_BookListScreenState` class, add a new `Color list`, called `bgColors`, and set it to an empty `List`:

```
List<Color> bgColors = [];
```

3. In the `initState` method, in the `then` callback of the `getBooks` method, add a `for` cycle that adds a new color (`white`) for each item in the `List` of books that was retrieved by the method:

```
helper.getBooks('flutter').then((List<Book> value) {
  int i;
  for (i = 0; i < value.length; i++) {
    bgColors.add(Colors.white);
}
…
```

4. At the bottom of the `_BookListScreenState` class, add a new method, called `setColor`, that takes a `Color` and an `integer`, called `index`, and sets the value of the `bgColors` list at the `index` position to the color that was passed:

```
void setColor(Color color, int index) {
  setState(() {
    bgColors[index] = color;
  });
}
```

5. In the `build` method of the `_BookListScreenState` class, in the `Gridview` widget, wrap the `ListTile` in a `ColoredBox` widget, and the `ColoredBox` itself in a `GestureDetector` widget, as shown here:

```
body: GridView.count(
  childAspectRatio: isLargeScreen ? 8 : 5,
  crossAxisCount: isLargeScreen ? 2 : 1,
  children: List.generate(books.length, (index) {
    return GestureDetector(
```

```
        child: ColoredBox  (
          child: ListTile(
[...]
```

6. In the `ColoredBox`, set the `color` property based on the value of the `bgColors` list at the `index` position:

```
 color: bgColors.isNotEmpty ? bgColors[index] :
Colors.white,
```

7. In the `GestureDetector` widget, add the callbacks for the `onTap`, `onLongPress`, and `onSecondaryTap` events, as shown here:

```
onTap: () => setColor(Colors.lightBlue, index),
onSecondaryTap: () => setColor(Colors.white, index),
onLongPress: () => setColor(Colors.white, index),
```

8. Run the app on your browser or desktop; left-click with your mouse on some of the items in the grid, and note that the background color of the item becomes blue, as shown in the following screenshot:

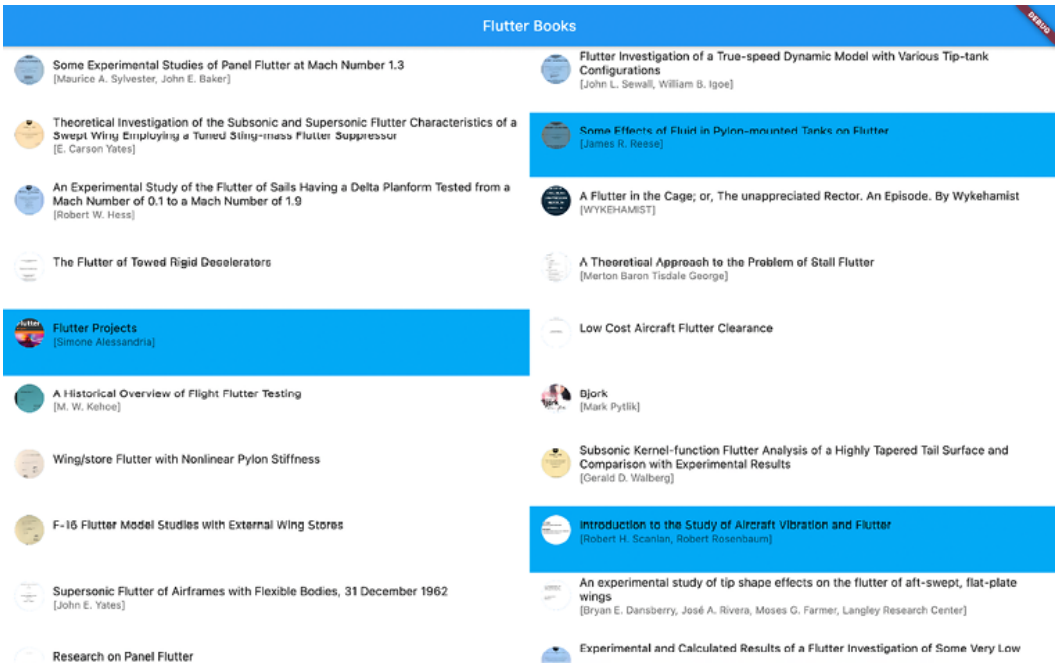


Figure 15.9: Responding to mouse events

9. Right-click on one or more of the items that you selected previously. The background color will turn back to white.
10. Long-press with the left button of the mouse on one of the items that you selected previously. The background color will turn back to white.

How it works...

When you design an app that works both on mobile and desktop, you should take into account the fact that some gestures are platform-specific. For instance, you cannot right-click with a mouse on a mobile device, but you can long-press.

In this recipe, you used the `GestureDetector` widget to change the color of items in a `GridView`. You can use a `GestureDetector` both for touchscreen gestures, such as swipes and long-presses, and for mouse gestures, such as right-clicks and scrolling.

To select items and add a light-blue background color, you used an `onTap` event:

```
onTap: () => setColor(Colors.lightBlue, index),
```

`onTap` gets called both when the user taps with a finger on a touchscreen and when they click on the main button of a mouse, stylus, or any other pointing device. This is an example of a callback that works both on mobile and desktop.

To deselect an item, you used both `onSecondaryTap` and `onLongPress`:

```
onSecondaryTap: () => setColor(Colors.white, index),
onLongPress: () => setColor(Colors.white, index),
```

In this case, you dealt differently with desktop and mobile: the “secondary tap” is mainly available for web and desktop apps that run on a device with

an external pointing tool, such as a mouse. This will typically be a mouse right-click, or a secondary button press on a stylus or graphic tablet.

`onLongPress` is mainly targeted at mobile devices or touchscreens that don't have a secondary button; it is triggered when the user keeps pressing an item for a "long" period of time (the time depends on the device, but it's usually 1 second or longer).

In order to keep track of the selected items, we used a `List` called `bgColors`, which, for each item in the list, contains its corresponding color (light blue or white). As soon as the screen loads, the `List` gets filled with the white color for all the items, as nothing is selected at the beginning:

```
helper.getBooks('Flutter').then((List<Book> value) {
  int i;
  for (i = 0; i < value.length; i++) {
    bgColors.add(Colors.white);
} …
```

When the user taps/clicks on an item, its color changes to blue, with the `setColor` method, which takes the new color and the position of the item that should change its color:

```
void setColor(Color color, int index) {
  setState(() {
    gbColors[index] = color;
  });
}
```

In this way, you leveraged a `GestureDetector` widget to respond to both touch and mouse events.

See also

The `GestureDetector` widget has several properties that help you respond to any kind of event. For a full guide on `GestureDetector`, have a look at https://api.flutter.dev/flutter/widgets/GestureDetector-class.html.

Interacting with desktop menus

Desktop apps have menus that do not exist on mobile form factors. Typically, you expect to see the important actions of a desktop app in its top menu.

In this recipe, you will learn how to add a menu that works on Windows and macOS.

Getting ready

Before following this recipe, you should have completed the first recipe in this chapter, *Creating a responsive app leveraging Flutter Web*.

Depending on where you want to run your app, you should also have completed one of the previous recipes in this chapter:

- For Windows: *Running your app on Windows*
- For macOS: *Running your app on macOS*

How to do it...

In order to add a menu to your desktop app, follow the next steps:

1. In your project, add the dependency to the `menu_bar` package by typing in your Terminal:

```
flutter pub add menu_bar
```

2. At the top of the `book_list_screen.dart` file, add the `menubar` import, and hide `MenuBar` from the `material.dart` import:

```
import 'package:flutter/material.dart' hide MenuBar;
import 'package:menu_bar/menu_bar.dart';
```

> **Note**
>
> At the time of writing, this menu implementation is still subject to change. Please see https://github.com/flutter/flutter/issues/23600 for the latest changes.

3. Move the `HttpHelper helper` declaration to the top of the `_BookListScreenState` class:

```
 class _BookListScreenState extends
State<BookListScreen> {
   final helper = HttpHelper();
[...]
```

4. At the bottom of the `_BookListScreenState` class, add a new method, called `updateBooks`, that takes a `String` as a parameter, calls the `getBooks` method, and updates the `books` list:

```
 void updateBooks(String key) {
   helper.getBooks(key).then((List<Book> value) {
     List<Color> newBgColors =
List.generate(value.length, (index) => Colors.white);
     setState(() {
       books = value;
       bgColors = newBgColors;
     });
   });
 }
```

5. At the bottom of the `_BookListScreenState` class, add another new method, called `buildBarButton`, that adds three search terms as menu items in the menu bar, as shown here:

```
List<BarButton> buildBarButton() {
   return [
     BarButton(
       text: const Text(
         'Search',
         style: TextStyle(color: Colors.white),
       ),
       submenu: SubMenu(menuItems: [
```

```
          MenuButton(
            onTap: () {
              updateBooks('Flutter');
            },
            text: const Text('Flutter'),
          ),
          MenuButton(
            onTap: () {
              updateBooks('C#');
            },
            text: const Text('C#'),
          ),
          MenuButton(
            onTap: () {
              updateBooks('JavaScript');
            },
            text: const Text('JavaScript'),
          ),
        ]),
      )
    ];
  }
```

6. In the `build` method, call the `buildBarButton()` method and include the Scaffold widget in `MenuBar`, as shown here:

```
@override
Widget build(BuildContext context) {
    if (MediaQuery.of(context).size.width > 600) {
      isLargeScreen = true;
    } else {
      isLargeScreen = false;
    }
    List<BarButton> myMenu = buildBarButton();
    return MenuBarWidget(
      barButtons: myMenu,
      child: Scaffold(
[…]
```

7. Run the app. In the menu bar, you should see a new menu, called **Search**. If you click on it, you should see the three search keys you set previously. The screenshot shows the menus as they appear on a Mac:

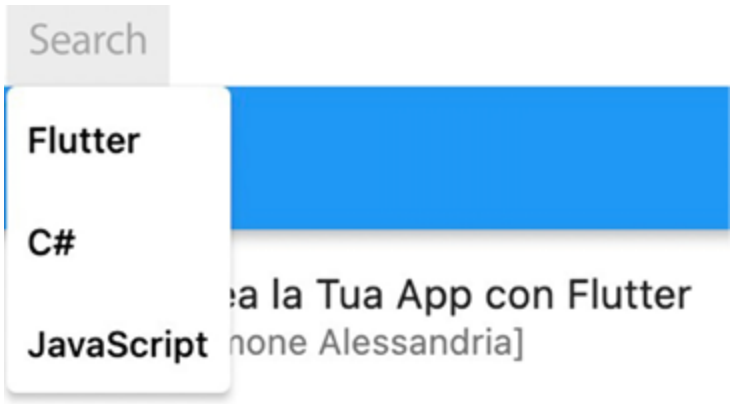


Figure 15.10: Menu on a Mac

How it works...

For this recipe, you used the `menu_bar` plugin to deal with menus for desktop apps. When using this package, the hierarchy of classes is:

MenuBar > **BarButton** > **SubMenu** > **MenuButton**

In our example, we included the `Scaffold` in a `MenuBarWidget` (the element at the top of the hierarchy:

```
return MenuBarWidget(
      barButtons: myMenu,
      child: Scaffold(
```

The `buildBarButton()` function returns a list with an instance of the `BarButton` class, the top-level menu item (and the second item in the hierarchy).

The `submenu` property of the `BarButton` contains an instance of the `SubMenu` class, which in turn contains a list of three `MenuButton` instances:

```
List<BarButton> buildBarButton() {
    return [
      BarButton(
        text: const Text(
          'Search',
```

```
          style: TextStyle(color: Colors.white),
        ),
        submenu: SubMenu(menuItems: [
          MenuButton(
            onTap: () {
              updateBooks('Flutter');
            },
            text: const Text('Flutter'),
          ),
          MenuButton(
            onTap: () {
              updateBooks('C#');
            },
            text: const Text('C#'),
          ),
          MenuButton(
            onTap: () {
              updateBooks('JavaScript');
            },
            text: const Text('JavaScript'),
          ),
        ]),
      )
    ];
  }
```

Each `MenuButton` has its own text and an `onTap` callback function that calls the `updateBooks()` function, with a string argument that corresponds to the text of the button.

The great thing is that, depending on the system you use, the menus will change accordingly. Here, you can see an example of the menu as it appears on a Windows desktop:

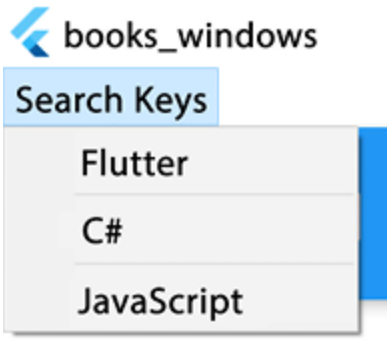


Figure 15.11: Menu on a Windows PC

See also...

While using the `menu_bar` package is probably one of the most straightforward ways to add menus to your desktop apps, you can also choose to use the official `PlatformMenuBar` class. This only works on Macs at the time of writing, but this may change in the near future; see https://api.flutter.dev/flutter/widgets/PlatformMenuBar-class.html for details.

Summary

The focus of this chapter has been on developing and running apps on web and desktop devices, and creating responsive apps based on screen size.

First, you explored how to create responsive apps using Flutter Web. You also saw how to deploy a Flutter website leveraging Firebase hosting.

You saw how to run apps on macOS and Windows devices, and learned how to respond to mouse events in Flutter Desktop, such as right-clicks. Lastly, you saw how to create and interact with desktop menus.

Now you should know how to develop and run your apps on web and desktop devices, and create responsive apps based on the screen size where the app is running. Flutter capabilities go beyond mobile devices, and understanding how to target web and desktop devices may become a great asset in your toolbox.

Join us on Discord

Read this book alongside other app developers.

Ask questions, participate in challenges, provide solutions to other readers, network and much more.

Scan the QR code or visit the link to join the community

https://packt.link/fluttercookbook



Distributing Your Mobile App

In this chapter, you will explore how to publish your app in the main mobile stores: Google Play and the Apple App Store. There are many small tasks that app developers need to perform when distributing their apps. Thankfully, some of these tasks can be automated. This includes code signing, writing metadata, incrementing build numbers, and creating icons.

We will be using the platform portals and `fastlane`, a tool currently owned by Google. At its core, `fastlane` is a set of scripts that automates the deployment of iOS and Android apps. Some scripts are only available for iOS, but it's still a great tool that can save you a lot of time.

The main `fastlane` tools you can leverage for your app's deployment include the following:

- `cert` to create and maintain signing certificates
- `sigh` to manage provisioning profiles
- `gym` to build and sign your apps
- `deliver` to upload your apps, screenshots, and metadata to the stores
- `pilot` to upload your projects to TestFlight and manage them
- `scan` to run automated tests

In this chapter, we will cover the following recipes:

- Registering your iOS app on App Store Connect
- Registering your Android app on Google Play
- Installing and configuring `fastlane`
- Generating iOS code signing certificates and provisioning profiles

- Generating Android release certificates
- Configuring your app metadata
- Adding icons to your app
- Publishing a beta version of your app in the Google Play Store
- Using TestFlight to publish a beta version of your iOS app
- Publishing your app to the stores

By the end of this chapter, you will understand the tasks required in order to publish an app in the main stores and automate some of the required tasks.

Technical requirements

All the recipes in this chapter assume you have purchased a developer account from Apple and Google. You can also decide to use only one of the two stores. In this case, just follow the instructions for your platform: Android for the Play Store and iOS for the App Store. Also, note that you need a Mac with Xcode to publish to the App Store.

Registering your iOS app on App Store Connect

App Store Connect is a set of tools that you can use to manage the apps you want to publish into the Apple App Store. These include all apps made for mobile devices, such as iPhone, iPad, and Apple Watch, and larger devices such as Mac and Apple TV.

Note

Before using the App Store Connect service, you need to enroll in the *Apple Developer Program*. See https://developer.apple.com/programs/ for more information.

In this recipe, you will complete the first steps required to publish your app to the App Store.

Getting ready

Before publishing an app to the App Store, you should have an Apple Developer Program subscription. This is a paid subscription and a prerequisite for this recipe and all the other recipes in this chapter that target iOS.

In order to follow the App Store Connect publication steps, you should have a Mac with Xcode installed.

You should also have an app that's ready to be published, at least in beta.

How to do it...

You will now register your app into the Apple App Store and obtain a bundle ID. Follow these steps:

1. Go to the **App Store Connect** page at https://appstoreconnect.apple.com/ and log in with your username and password. After a successful login, you should see a page similar to the following screenshot:

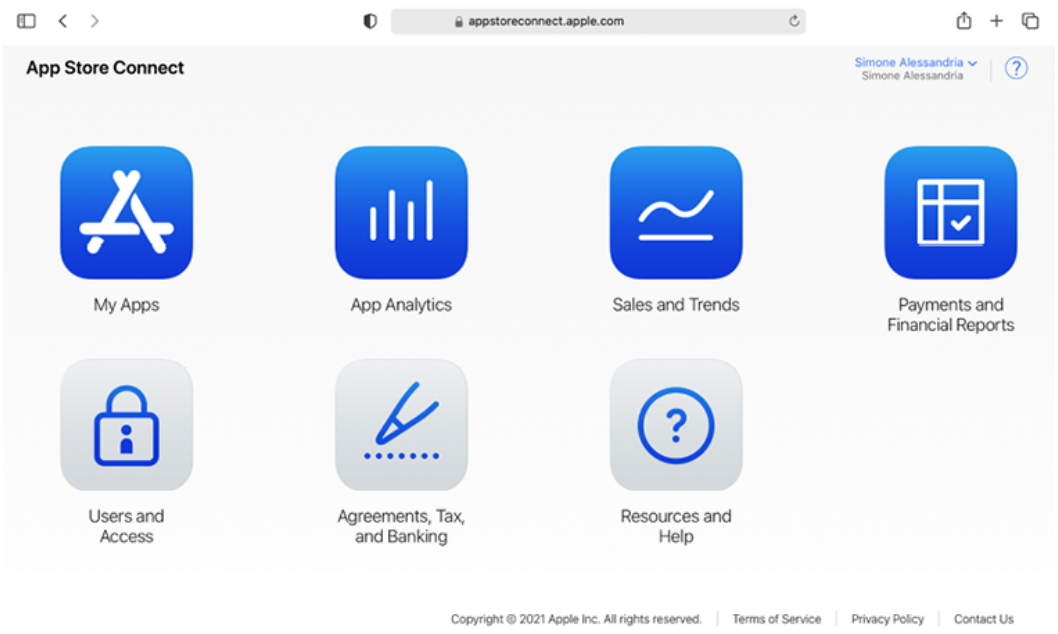


Figure 16.1: App Store Connect

2. Go to the **Identifiers** section of App Store Connect at https://developer.apple.com/account/resources/identifiers/list/bundleId and click on the + button to create a new bundle ID.
3. Select the **App IDs** option, as shown in the following screenshot, then click the **Continue** button at the top right of the page:

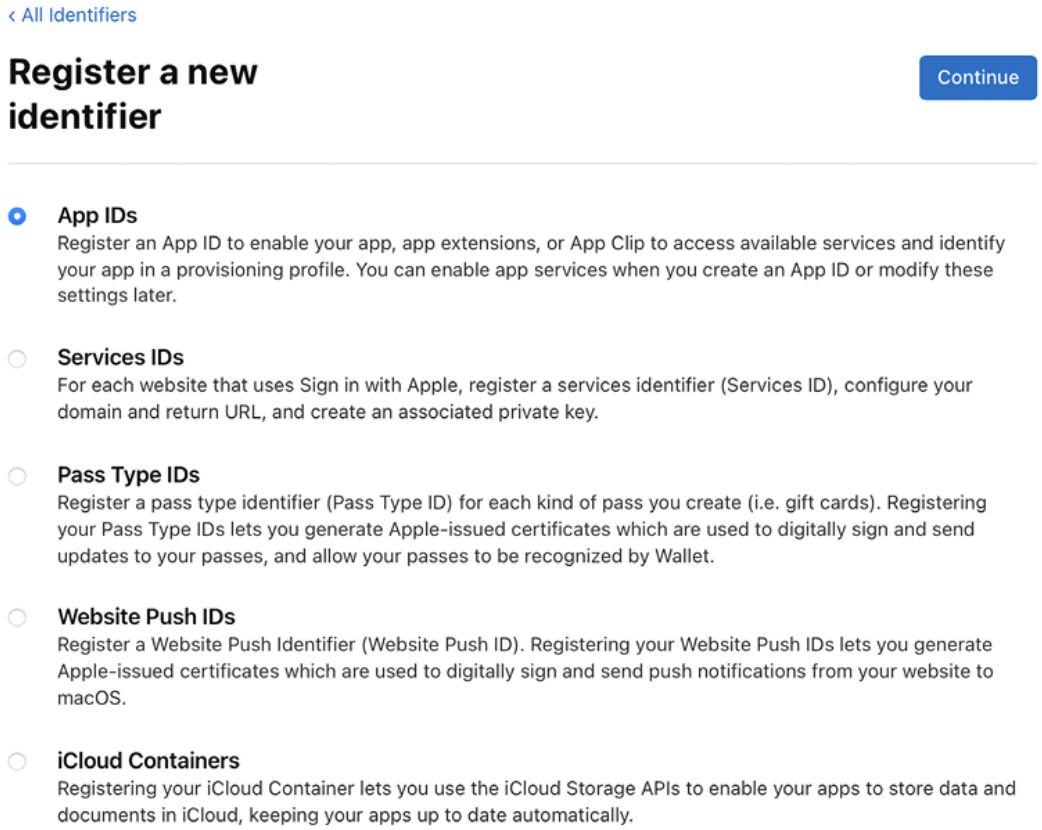


Figure 16.2: Register a new app identifier

4. On the screen after clicking **Continue**, make sure the **App** option is selected and click **Continue** again.
5. On the **Register an App ID** page, fill in the details for the **Description** and **Bundle ID** fields. For the bundle ID, a reverse domain name is recommended (such as `com.yourdomainname.yourappname`).
6. Choose the capabilities or permissions (if any) for your app, then click **Continue**:

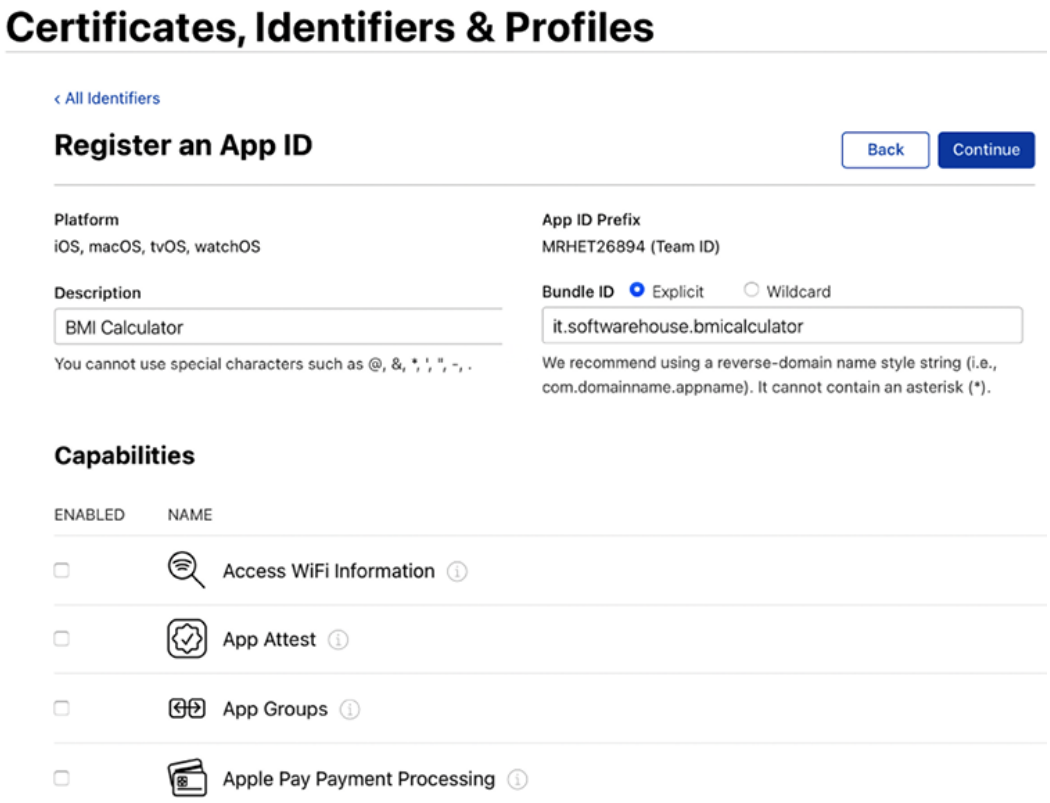


Figure 16.3: App ID configuration

7. On the confirmation screen, click the **Register** button.
8. At the end of the process, note that an app ID has been created.
9. Go back to the **Apps** page at https://appstoreconnect.apple.com/apps.
10. On the **My Apps** page, click on the + icon at the top of the page and select the **New App** link. Note that this action will create a new app *for the store*, which is a prerequisite to publishing your *existing* app there.

> **Note**
>
> When you click on the + icon to add a new app, you have two choices: **New App** and **New App Bundle**. With **App Bundle**, you can create offers in the store that bundle up to 10 apps, so that your customers can get 2 or more of your apps with a single purchase in the store.

11. On the **New App** page, complete the required fields. Choose the available platforms for your new app, its name, and its primary language, select the bundle ID you created in the previous steps, and then select an SKU (a unique ID for your app, which will not be visible on the App Store). An example of the data you can insert is shown in *Figure 16.4*:

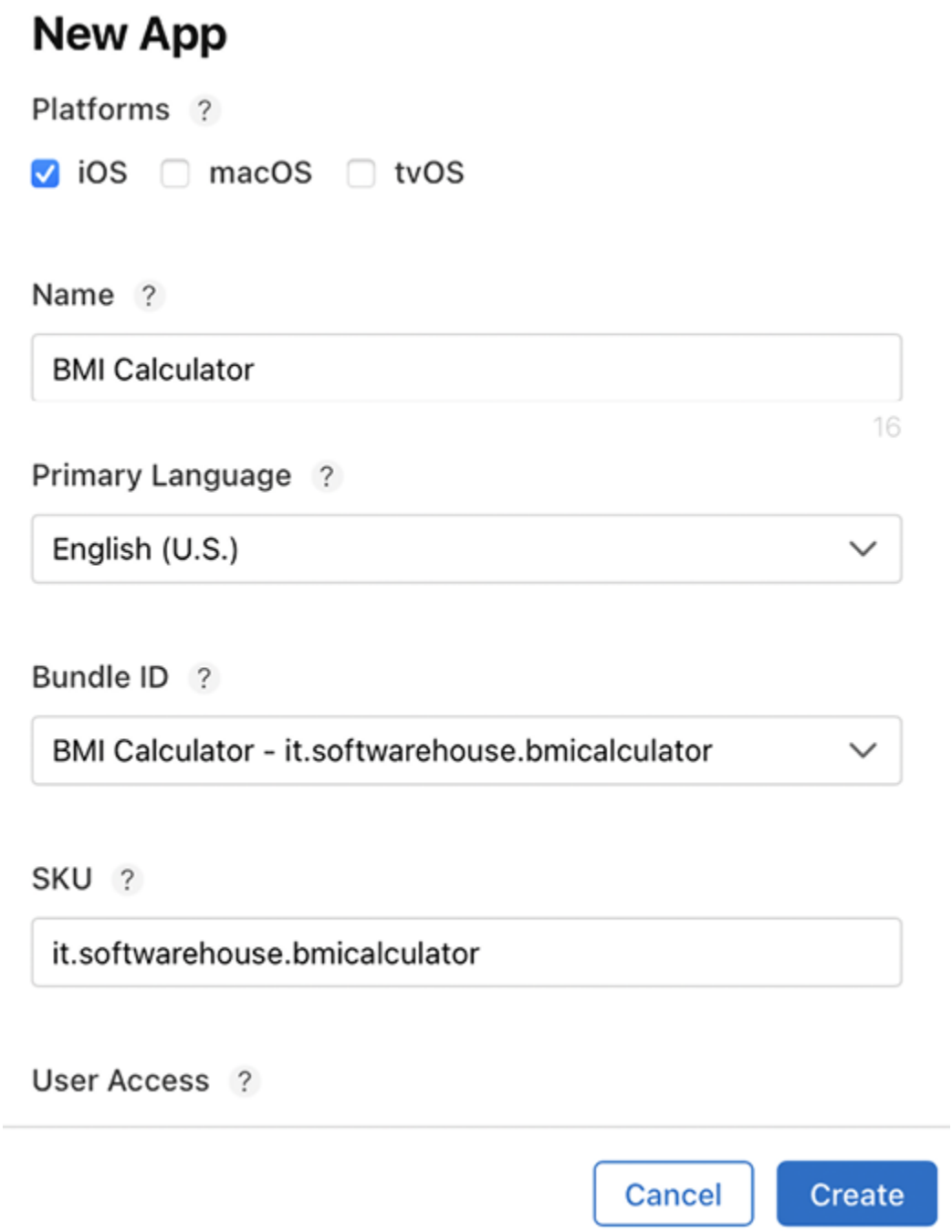


Figure 16.4: App Store new app configuration

12. Click the **Create** button. Your App Store Connect registration is now ready.

How it works...

The first step when registering an app in the App Store is retrieving a **bundle ID**. This is a unique identifier. This identifier can be used for an app or other objects, including websites that use sign-in with Apple, push IDs, or iCloud containers.

The configuration of a bundle ID requires choosing a description, the bundle ID itself, and the capabilities of your app.

> Note
>
> The configuration process gives you the option to choose an *explicit* or *wildcard* bundle ID. You must choose an **explicit bundle ID** if you want to enable push notifications or in-app purchases. If you choose a **wildcard bundle ID**, just leave an asterisk in the text field: this will allow you to not specify your bundle ID.

As the bundle ID must be universally unique (there cannot be another app with the same bundle ID in all the Apple ecosystems), Apple suggests using the **reverse domain name notation**: `ext.yourdomain.yourappname`. You can choose either an existing domain if you have one, your name and surname, or any other name that might be unique to your products.

Choosing the capabilities is also an important part of the creation of a bundle ID. This can enable services provided by Apple such as the Apple Push Notification service, CloudKit, Game Center, and in-app purchases. Capabilities can also be changed later.

Once you get a bundle ID, you can register an app. This requires choosing a name, a primary language that you can select from the supported languages,

the bundle ID that you created previously, and an **SKU**. The SKU is another identifier.

> Note
>
> **SKU** stands for **stock-keeping unit**. It's a unique tracking number for your application. For more information on the use of SKUs, see https://en.wikipedia.org/wiki/Stock_keeping_unit.

The SKU can be the same as your bundle ID, or any other unique identifier. Its purpose is to track your app for accounting purposes.

Registering your app is the first step required for the publication of your app.

See also

There are several small tasks that need to be performed, contracts that must be accepted, and descriptions and images that you should add in order to publish your app to the App Store. For a high-level overview of the workflow, see the official Apple guide at https://help.apple.com/app-store-connect/#/dev300c2c5bf.

For this chapter, we are using `fastlane` for the publishing process. There are also other alternatives, including Codemagic at https://flutterci.com/ and App Center at https://appcenter.ms/.

Registering your Android app on Google Play

The hub where you publish and manage apps in the Google Play Store is the **Google Play Console**, which you can access at https://play.google.com/console.

In this recipe, you will create an entry for your app in the Google Play Console, and thus understand the first step to publish your apps into the main Android store.

This is actually a very quick process.

Getting ready

Before publishing an app to the Google Play Store, you should have a Google Developer account; you can get one for a one-time fee, and it is a prerequisite for this recipe and all the other recipes in this chapter that target Android.

You should also have an app ready to be published, at least in beta.

How to do it...

You will now create a new entry for your app in the Google Play Store. Follow these steps:

1. Go to the Google Play Console at https://play.google.com/console.
2. Click on the **Create app** button at the top right of the page, and you will get to the page shown in *Figure 16.5*:

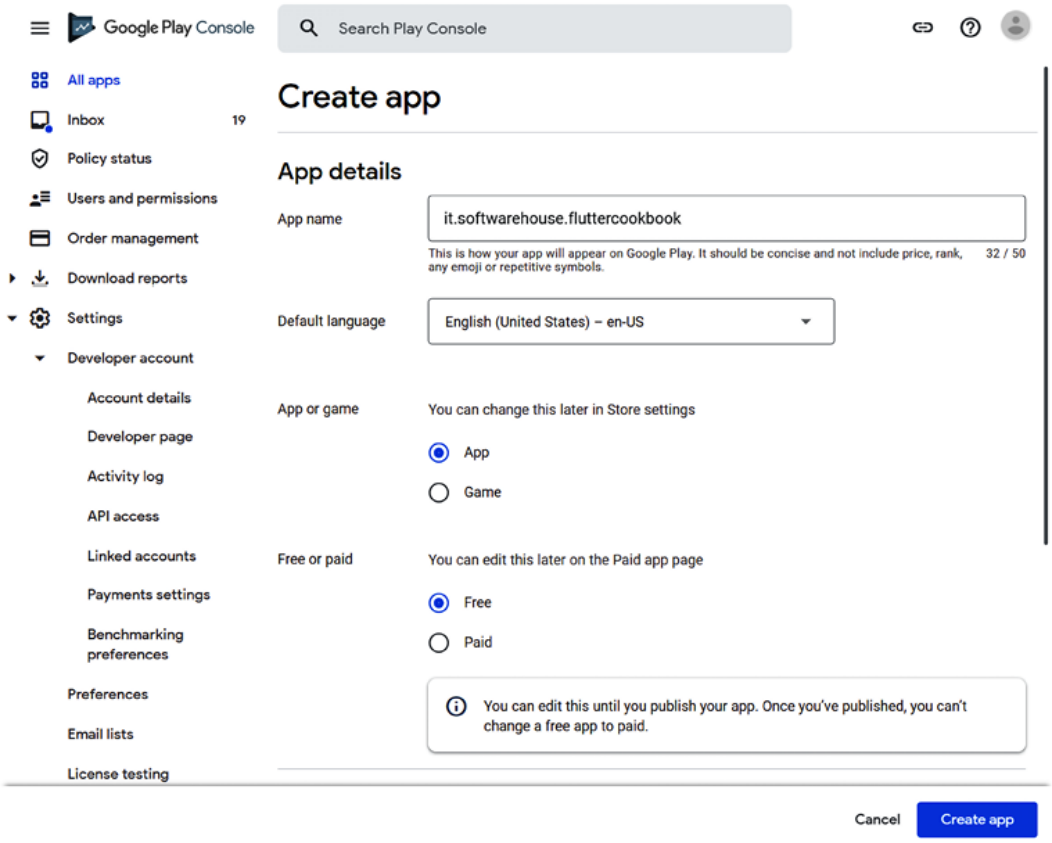


Figure 16.5: Play Store Create app page

3. On the **Create app** page, choose an app name and default language and select whether this is a game or an app and whether it's free or paid. Then accept the required policies and click the **Create app** button.

> Note
>
> **Create app** creates a new entry for your app in the Play Store. This does not create a new Android app, which you create with Flutter or other frameworks.

4. You will be brought to the app dashboard. Note that your app has been successfully created.

How it works...

As you can see, the process to register a new Android app is extremely easy.

> **Note**
>
> Currently, in order to develop for Google Play, you only need to pay a one-time registration fee, instead of the annual subscription needed to develop for iOS.

You should pay attention to a few details though:

- The app name can contain a maximum of 50 characters, and the name you choose will appear to your users in the Play Store, so it should be clear, concise, and appealing.
- Almost everything can be managed and changed later.

See also

There are several small tasks that need to be performed, contracts that must be accepted, and descriptions and images that you should add in order to publish your app to the Play Store. For a high-level overview of the workflow, see the official Android guide at https://developer.android.com/studio/publish.

Installing and configuring fastlane

Publishing an app to these stores is a long and cumbersome process; it may well take a full day of work for the initial publishing, and then hours for each update you make to the app. That's why automation tools such as `fastlane` can boost your productivity by automating several tasks required to deal with the publishing process and keep you focused on designing and developing your apps.

Note

For more information about `fastlane`, have a look at https://fastlane.tools/.

You will now see the setup and configuration process for `fastlane`.

Getting ready

Before following along with this recipe, you should have completed the previous recipes in this chapter: *Registering your iOS app on App Store Connect* for iOS and *Registering your Android app on Google Play* for Android.

How to do it...

In this recipe, you will set up `fastlane` for Windows, macOS, Android, and iOS. Follow the steps in the following sections.

Installing fastlane on Windows

`fastlane` depends on Ruby, so you need to install it on Windows, and then, with it, install `fastlane`. Follow these steps:

1. Go to https://rubyinstaller.org/ and download the latest supported version of Ruby for your system (*with* Devkit). Then, execute the file and follow the installation process, leaving the default options. You may have to restart your system.
2. Open a PowerShell window (the **Command Prompt** should *not* be used for this task), and type:

```
gem install fastlane
```

Note

If you work in a team or use a **continuous integration** (**CI**) system, you may need to lock specific versions of `fastlane` for your project. In this case, you may use `Bundler`, a popular `Ruby` package manager; have a look at this link for the installation instructions: https://docs.fastlane.tools/getting-started/android/setup/.

Installing fastlane on a Mac

If you use macOS, you'll already have Ruby installed on your system, but using the system version of Ruby is not recommended. On macOS, you can use Homebrew, which will automatically install the recommended Ruby version for `fastlane`:

1. If `Homebrew` is not installed on your system yet, install it by typing the following:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/
install.sh)"
```

2. To install `fastlane` on a Mac, run the following command:

```
brew install fastlane
```

From now on, the tasks depend on whether you are building an app for Android or iOS.

Configuring fastlane for Android

In order to configure `fastlane` on Android, follow these steps:

1. Go to the Google Play Console at https://play.google.com/console.
2. Click on the **Setup** menu, then **API access**.

3. In the **Service accounts** section, click the **Learn how to create service accounts** button. This will bring a pop-up menu, with the link to create a service account in Google Cloud Platform. Click on the link.
4. On the **Service account** page, click on the **Create Service Account** button at the top of the page.
5. Fill in the required data fields including, a name for the service account and a description, as shown in the following screenshot. Then, click the **CREATE AND CONTINUE** button:

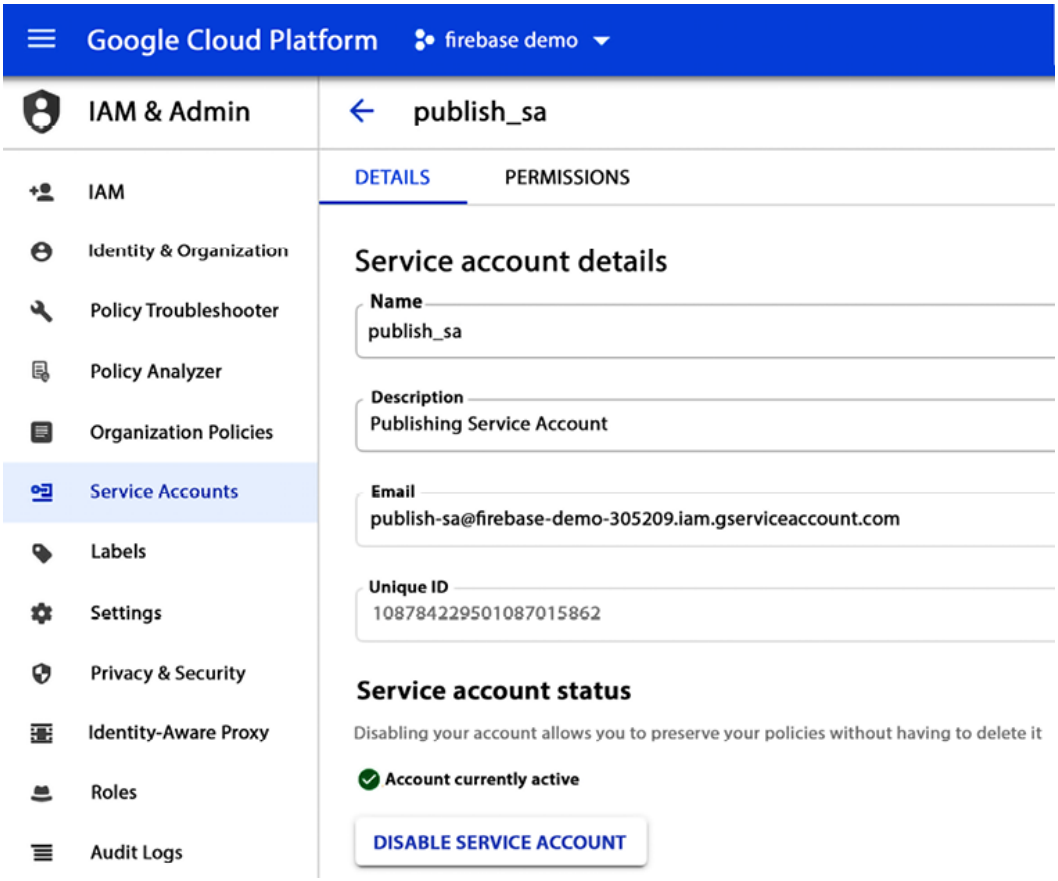


Figure 16.6: Google Cloud Platform Service account details page

6. In the **Grant this service account access to project** section, choose **Service Account User** and click **Continue**.
7. In the **Grant users access to this service account** section, leave the fields blank and click **Done**.

8. On the service account page, click on the **Actions** button of the service account, and select the **Manage Keys** menu option.
9. On the **Keys** page, click the **ADD KEY** button, then **Create new key**.
10. Leave **JSON** as the key format and click **Create**. This will automatically download a JSON file on your PC or Mac. *You will need this file later*.
11. Go back to the **Play Store Console**, to the **API Access** page. You should see the new service account available in the **Service Accounts** section.
12. Click on the **Manage Play Console Permissions** link near your service account.
13. On the **Account Permissions** page, enable all the permissions in the **Releases** section and disable all the remaining permissions. Then, click the **Save Changes** button. Your service account is now ready.
14. Back to the app, make sure you choose a valid `applicationId` identifier in the app's **Gradle** file, which you can find in the `android/app/build.gradle` folder, as in the following example (instead of `it.softwarehouse`, please choose your own reverse domain):

```
applicationId "it.softwarehouse.sh_bmi_calculator"
```

15. In your Terminal, run the command:

```
flutter build appbundle
```

16. Note the success message, as shown in the following screenshot:

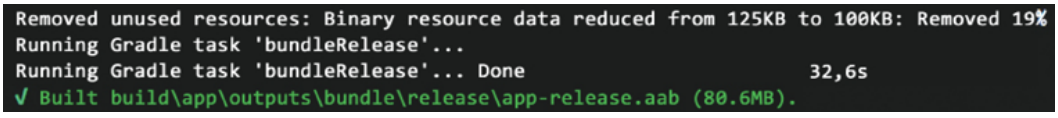


Figure 16.7: Flutter build appbundle success message

17. Go to the `android` directory in your project, then run the following command:

    ```
    fastlane init
    ```

18. Follow the setup instructions. When prompted, add the location of the JSON file you downloaded previously. The Android setup is now complete.

Installing fastlane for iOS

In order to configure `fastlane` on iOS, follow these steps:

1. Open a Terminal window and install the Xcode command-line tools by typing the following command:

    ```
    xcode-select --install
    ```

2. In your Terminal, get to the `ios` directory in your project.
3. Initialize `fastlane` with the following command:

    ```
    fastlane init
    ```

4. When prompted, choose the **Automate app store distribution** option.
5. When prompted, insert your developer Apple ID username and password.
6. If appropriate, answer yes (**y**) at the prompt to create the App Store Connect app.
7. At the end of the process, `fastlane` should have been correctly configured for iOS distribution.

See also

There are some tasks that can go wrong when you set up `fastlane` for the first time. For an overview of the installation process and some troubleshooting, refer to the following links:

- For Android, see https://docs.fastlane.tools/getting-started/android/setup/.
- For iOS, see https://docs.fastlane.tools/getting-started/ios/setup/.

Generating iOS code signing certificates and provisioning profiles

`fastlane` contains a set of command-line tools. One of them is `fastlane match`, which allows sharing a code signing identity with your development team.

In this recipe, you will learn how to leverage `fastlane match` to create signing certificates for your apps and store them in a Git repository.

Getting ready

Before following along with this recipe, you should have completed the previous recipes in this chapter: *Registering your iOS app on App Store Connect* and *Installing and configuring fastlane*.

How to do it...

You will now create a new Git repo and use `fastlane match` to retrieve and store your certificates in your repository. Follow these steps:

1. Create a *private* Git repository at github.com. You can find the details of the procedure at https://docs.github.com/en/github/getting-started-

with-github/create-a-repo.

2. In the `ios` directory of your project, from the Terminal, type the following:

```
fastlane match init
```

3. When prompted, choose the Git storage mode, then insert your private Git repository address.
4. In your Terminal window, type the following:

```
fastlane match development
```

5. When prompted, enter the passphrase that will be used to encrypt and decrypt your certificates.

> **Note**
>
> Make sure to remember your password, as it's needed when running `match` on another machine.

6. When prompted, enter the password for your keychain, which will be stored in the `fastlane_keychain_login` file and used in future runs of `match`.
7. At the end of the process, you should see the following success messages on your prompt:

```
Successfully installed certificate [YOUR CERTIFICATE ID]
Installed Provisioning Profile
```

8. In the Terminal, type the following:

```
fastlane match appstore
```

9. At the end of the process, you should see a success message:

```
All required keys, certificates and provisioning
profiles are installed
```

How it works...

While on an Android device, you can install an unsigned app, this is not possible on iOS devices. *Apps must be signed first*.

This is why you need a **provisioning profile**. It is a link between the device and the developer account and contains the information that identifies developers and devices. It is downloaded from your developer account and included in the app bundle, which is then code-signed.

Once you have a provisioning profile, you should keep it in a safe and easy-to-retrieve space; this is why the first step in this recipe was creating a private Git repository.

In this recipe, you created two provisioning profiles:

- A **development provisioning profile**. You created it with the `fastlane match development` command. This must be installed on each device on which you wish to run your application code; otherwise, the app will not start.
- A **distribution profile** with the `fastlane match appstore` command. This allows distributing an app on any device, without the need to specify any device ID.

A provisioning profile requires a **certificate**, which is a public/private key pair that identifies who developed the app.

`fastlane match`, which you used in this recipe, is a huge time-saver as it automatically performs several tasks:

- It creates your code signing identity and provisioning profiles.
- It stores the identity and profiles in a repository (on GitHub or another platform).

- It installs the certificates from the repository on one or more machines.

When you run `fastlane match` for the first time, for each environment, it will create the provisioning profiles and certificates for you. From then on, it will import the existing profiles.

See also

`fastlane match` can simplify your signing workflow, especially if you want to share your profile with a team of developers. For a full guide on the capabilities and features of this tool, see https://docs.fastlane.tools/actions/match/.

Generating Android release certificates

`keytool` is a Java utility that manages keys and certificates. The `keytool` utility stores the keys and certificates in a keystore.

In this recipe, you will understand how to use the `keytool` utility to generate Android certificates. This step is required to sign and publish your app in the Google Play Store.

Getting ready

Before following along with this recipe, you should have completed the previous recipes in this chapter: *Registering your Android app on Google Play* and *Installing and configuring fastlane*.

How to do it...

You will now create an Android keystore. Follow these steps:

1. On Mac/Linux, use the following command:

```
keytool -genkey -v -keystore ~/key.jks -keyalg RSA -
keysize 2048 -validity 10000 -alias key
```

2. On Windows, use the following command:

```
keytool -genkey -v -keystore c:\Users\USER_NAME\key.jks
-storetype JKS -keyalg RSA -keysize 2048 -validity
10000 -alias androidkey
```

> **Note**
>
> The `keytool` command stores the `key.jks` file in your home directory. If you want to store it elsewhere, change the argument you pass to the `-keystore` parameter. However, keep the keystore file private; don't check it into public source control!

3. Create a file named `<app dir>/android/key.properties` that contains a reference to your keystore:

```
storePassword=your chosen password
keyPassword= your chosen password
keyAlias=androidkey
storeFile=[YOUR_PATH]/key.jks
```

4. In the app `build.gradle` file (`android/app/build.gradle`), add the following instructions *before* the `android{}` block:

```
def keystoreProperties = new Properties()
def keystorePropertiesFile =
rootProject.file('key.properties')
if (keystorePropertiesFile.exists()) {
  keystoreProperties.load(new
     FileInputStream(keystorePropertiesFile))
}
```

5. Add the following code before the `buildTypes` block:

```
signingConfigs {
  release {
    keyAlias keystoreProperties['keyAlias']
    keyPassword keystoreProperties['keyPassword']
    storeFile keystoreProperties['storeFile'] ?
     file(keystoreProperties['storeFile']) : null
    storePassword keystoreProperties['storePassword']
  }
}
```

6. In the `buildTypes` block, edit `signingConfig` to accept the release signing certificate:

```
buildTypes {
  release {
    signingConfig signingConfigs.release
  }
}
```

7. Add the `key.properties` file to the project's `.gitgnore` file.

How it works...

The first step you performed in this recipe was generating a private key and a keystore with the `keytool` command-line utility.

For instance, say you have the following instruction:

```
keytool -genkey -v -keystore ~/key.jks -keyalg RSA -keysize
2048 -validity 10000 -alias key
```

This does the following:

- Generates a new private key with the `-genkey` option
- Creates an alias for this key named `key` with the `-alias` option
- Stores the private key in a file called `key.jks` with the `-keystore` option

- Specifies the size (in bytes) and validity (in days) with the `-keysize` and `-validity` options

A `build.gradle` file is an Android build configuration file. In Flutter, you generally need to interact with two `build.gradle` files: one is at the project level, in the `android` directory, and one is at the app level, in the `android/app` folder. In order to add the signing configuration, you need the app-level `build.gradle` file.

The app `build.gradle` file contains the keystore properties and loads the signing configuration in the `signingConfigs {}` object.

It's also important to make sure that the signing configuration is set to **`release`** mode and not **`debug`**; otherwise, the Play Store will not accept your app. You can perform this task with the following instruction:

```
signingConfig signingConfigs.release
```

Once you add the correct configuration to your app's `build.gradle` file, every time you compile your app in **`release`** mode, the app will automatically be signed.

See also

`keytool` supports several commands that allow you to work with certificates, keys, and keystores. For a full tutorial on this executable, see http://tutorials.jenkov.com/java-cryptography/keytool.html.

Configuring your app metadata

Before uploading your app to the stores, you need to complete some settings specific to the platform (iOS or Android) you are targeting.

In this recipe, you will learn how to set the `AndroidManifest.xml` file for Android to add the required metadata and permissions, and the

`runner.xcworkspace` file to edit your app name and bundle ID for iOS.

Getting ready

Before following along with this recipe, you should have completed all the previous recipes in this chapter, or at least those that target your specific platform (iOS or Android).

How to do it...

You will now set some required metadata in the `AndroidManifest.xml` and `runner.xcworkspace` files.

Adding Android metadata

By following these steps, you will edit the `AndroidManifest.xml` file and add the required configuration to publish your app:

1. Open the `AndroidManifest.xml` file, in the `android/app/src/main` directory.
2. Set the `android:label` value in the application node to the public name of your app, for example, `BMI Calculator`.
3. If your app needs an HTTP connection, for example, to read data from a web service, make sure you add the internet permission at the top of the manifest node with the instruction that follows:

```
<uses-permission
android:name="android.permission.INTERNET"/>
```

4. In the `package` property of your manifest node, set the unique identifier you chose in the app setup at the beginning of this chapter, such as the following:

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/androi
```

```
d" package="com.yourdomain.bmi_calculator">
```

5. Copy the package value you chose, then open the `AndroidManifest.xml` file in the `android/app/src/debug` directory (this is another `AndroidManifest.xml` file, not the same as before) and paste the value into the package name of this file as well.
6. Open the `MainActivity.kt` file in the `android/app/src/main/kotlin/[your project name in reverse domain]` directory.
7. At the top of the file, change the package name to the value you copied:

```
package com.yourdomain.bmi_calculator
```

Adding metadata for iOS

Using the following steps, you will edit the `runner.xcworkspace` file and update the app display name and bundle ID:

1. On your Mac, open Xcode and open the `runner.xcworkspace` file in the `ios` directory in your project.
2. Select **Runner** in the Project Navigator and set the **Display Name** property to the public name of your app, for example, `BMI Calculator`.
3. Make sure you set the bundle ID correctly, with a reverse domain and the name of your app.

How it works...

When you create a new app with Flutter, the default package name is `com.example.your_project_name`. This must be changed before you upload your app to the stores. Instead of `com.example`, you should use your own domain, if you have one, or some other unique identifier. This package

name must then be set in the `AndroidManifest.xml` file in the manifest node, and in `runner.xcworkspace` in the `Bundle Identifier` setting.

> **Note**
>
> The iOS bundle identifier and the Android package name do not need to be the same; just make sure they are both unique in the respective stores.

When you change the package name in the `AndroidManifest.xml` file, you also have to update the `MainActivity.kt` file to avoid compilation errors.

The `android:label` property contains the name of the app that your users will see on their screen, so it's particularly important that you choose a good name for your app here. On iOS, you perform the same with the **Display Name** property.

Another setting you should pay attention to in Android is the internet permission, which you set with the following node:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

This is something you should always add when you use an HTTP connection; when you develop and run the app in **`debug`** mode, this is automatically added to use Flutter features such as hot reload, but if you try to run your app in **`release`** mode and forget to add this permission, your app will crash. On iOS, this is not required.

See also

For a full list of both the compulsory and the optional metadata you can add to your apps to list them in the stores, refer to the following links:

- If you want to localize your app's name, you could use the **`strings.xml`** for Android, or **`InfoPlist.strings`** for iOS: see this article for further details: https://medium.com/@ykaito21/flutter-from-zero-to-one-how-to-localize-app-display-name-c4deb5aa4c04

- For the App Store, see https://developer.apple.com/documentation/appstoreconnectapi/app_metadata.
- For the Play Store, see https://support.google.com/googleplay/android-developer/answer/9859454?hl=en.

Adding icons to your app

Icons help users identify your app on their devices, and they are also a requirement to publish your app into the stores.

In this recipe, you will see how to automatically add icons to your app in all the required formats by using a Flutter package called `flutter_launcher_icons`.

Getting ready

Before following along with this recipe, you should have completed all the previous recipes in this chapter, or at least those that target your specific platform (iOS or Android).

How to do it...

You will now add the `flutter_launcher_icons` package to your app and use it to add icons to your iOS and Android apps. Follow these steps:

1. Create a new directory in the root of your project, called `icons`.
2. Create or retrieve an icon for your app if you haven't already added one, and place it in the `icons` directory.

> **Note**
>
> You can get an icon for free at several websites, including https://remixicon.com/. If you use this specific service, make sure to download your icon at the maximum available resolution.

3. Add the `flutter_launcher_icons` package to the `dev_dependencies` node in your `pubspec.yaml` file.
4. At the same level as `dependencies` and `dev_dependencies`, add the `flutter_launcher_icons` node, as shown:

```
Flutter_launcher_icons:
  android: true
  ios: true
  image_path: "icons/scale.png"
  adaptive_icon_background: "#DDDDDD"
  adaptive_icon_foreground: "icons/scale.png"
```

5. From a Terminal window in your project folder, run the following command:

```
flutter pub run flutter_launcher_icons:main
```

6. Make sure the icons for your app have been generated.
7. For Android, check that the icons are available in the `android/main/res` folder, under the "drawable" directories.
8. For iOS, check that the icons are available in the `ios/Runner/Assets.wcassets/AppIcon.appiconset` directory, in several different formats.

How it works...

`flutter_launcher_icons` is a command-line tool that allows you to create the launcher icon for your app and is compatible with iOS, Android, and even web and desktop apps.

In order to use `flutter_launcher_icons`, you need to add its dependency in the `dev_dependencies` node in your `pubspec.yaml` file. This is where you put the dependencies that will not be exported to the release app.

The configuration of this package happens in the `pubspec.yaml` file with the following instructions:

```
flutter_icons:
  android: true
  ios: true
  image_path: "icons/scale.png"
  adaptive_icon_background: "#DDDDDD"
  adaptive_icon_foreground: "icons/scale.png"
```

Here you specify the following:

- The target platforms for the icons (in this case, `android` and `ios`)
- The path of the source image for the icon with the `image_path` property
- The color used to fill the background of the adaptive icon (Android only) with the `adaptive_icon_background` property
- The image used for the icon foreground of the adaptive icon (Android only) with the `adaptive_icon_foreground` property

Adaptive icons are used when you generate Android launcher icons.

Icons, together with the screenshots, are a requirement for publishing your app to the stores.

Note

`fastlane snapshot` automatically takes the screenshots you need for your app. Unfortunately, this is not currently available for Flutter. There is a Flutter package called `screenshots` that solves this issue, but it received its last update in 2019 and, at the time of writing, is not currently working.

See also

At a future time, and possibly when you read this book, you will probably be able to leverage the `fastlane snapshot` feature in Flutter. For more information about this tool, see the official documentation at https://docs.fastlane.tools/actions/snapshot/.

Publishing a beta version of your app in the Google Play Store

Before making your app available in **`release`** mode, it's considered a best practice to have a smaller group of people test your app. It can be a small group of specific people, such as your team or your customers, or a larger group of people. In any case, publishing a beta version can give you invaluable feedback on your app. In this recipe, you will see how to publish your app in a beta track in the Google Play Store.

Getting ready

Before following along with this recipe, you should have completed all the previous recipes in this chapter, or at least those that target Android.

How to do it...

You will compile your app in **`release`** mode and upload it to the Google Play Store in beta. To do so, follow these steps:

1. Open the `fastfile` file into your project's `android/fastlane` directory.
2. If, in the file, there is already a `beta` lane, comment it out.
3. At the bottom of the `platform: android` lane, add the following instructions:

```
lane :beta do
  gradle( task: 'assemble', build_type: 'Release' )
end
```

4. In a Terminal window, from the `android` folder of your project, run the following command:

```
fastlane beta
```

> **Note**
>
> If you are using Windows, you might need to change the character table for the Terminal. One way to do this is by typing the `command: chcp 1252` before running `fastlane beta`.

5. Make sure you find the **`release`** version of your app file in the `build\app\outputs\apk\release` directory of your project.
6. Go to the Google Play Console at https://play.google.com/console.
7. Click on the **Open Testing** link in the **Release | Testing** section.
8. Click on the **Create New Release** button.
9. On the **App signing preferences** screen, choose **Let Google manage and protect your app signing key**, as shown in the following screenshot, then click the **Update** button:

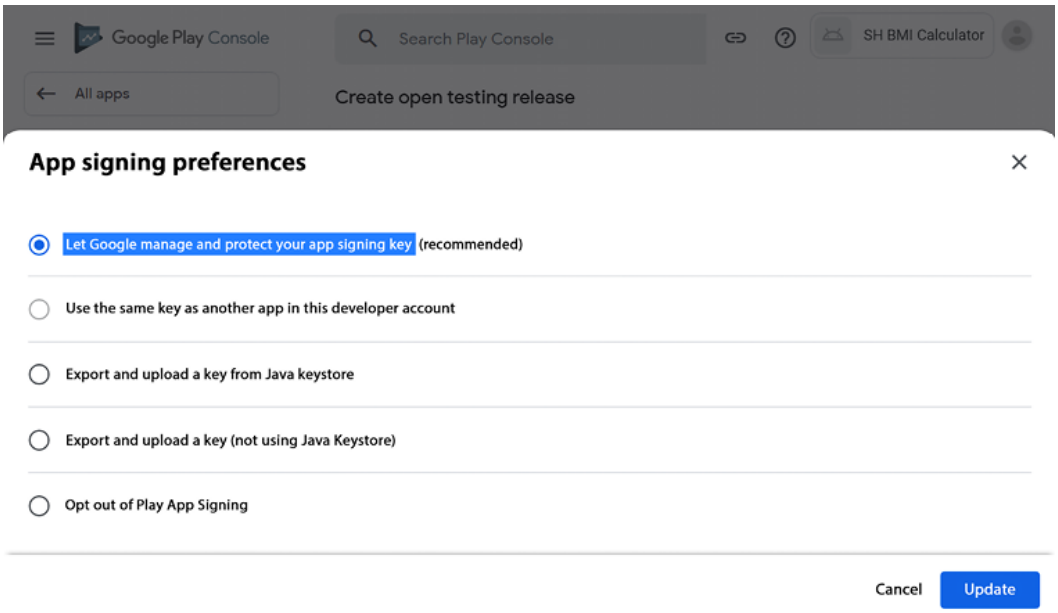


Figure 16.8: Signing preferences screen

10. In the **Create open testing release** section of the Google Play Console, upload the APK file that you have built, which you will find in the `build\app\outputs\apk\release` directory.
11. After uploading your file, you might see some errors and warnings, which will guide you in completing the beta publishing of your app, as shown in the following screenshot:

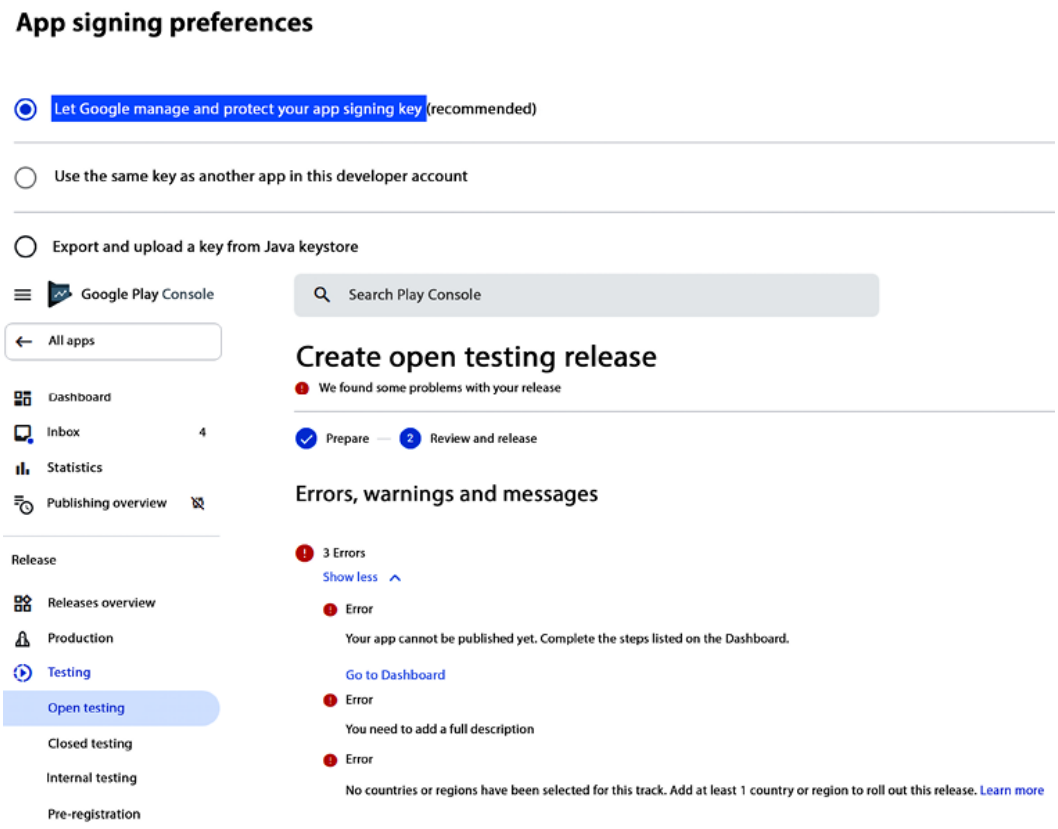


Figure 16.9: Google Play open testing page

12. Click on the **Go to dashboard** link and click on the **setup your store listing** link.
13. Insert a short description (80 characters or less) and a long description (4,000 characters or less) for your app.
14. Insert the required graphics and save.
15. Back in the **Open Testing** track, select the country (or countries) where your beta testing should be available.
16. Click on the **Review and Rollout release** link, and then click on the **Start rollout to open testing** button and confirm the dialog:

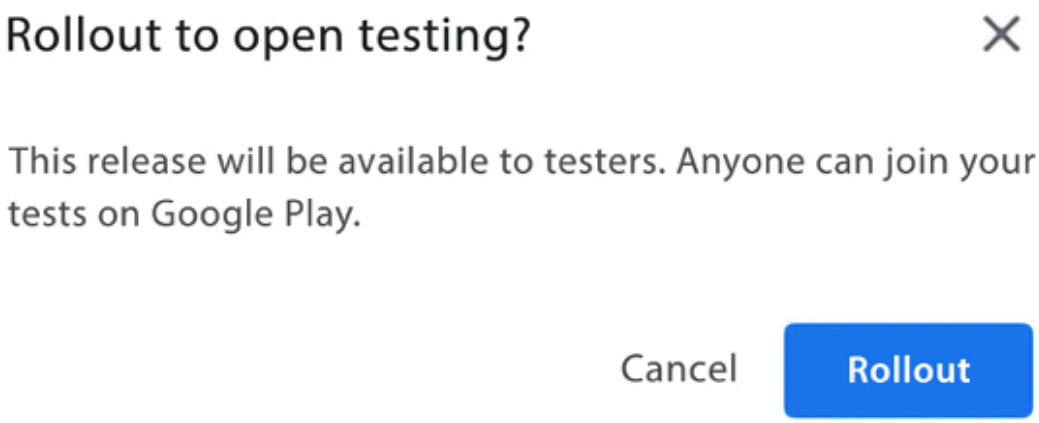


Figure 16.10: Google Play Rollout to open testing page

Your app has now been uploaded to the beta release channel in the Play Store.

How it works...

It's generally a very good idea to create a testing track for your app before it gets officially published. In the Google Play Store, you can choose between three tracks before release:

- **Internal testing**: You can use internal testing to distribute your app to up to 100 testers. This is ideal when you want to show your app to a selected group of testers at the early stages of your development.
- **Closed testing**: This is to reach a wider audience. You can invite testers by email, and this currently supports up to 200 lists of emails with 2,000 people each.
- **Open testing**: This is the track we used in this recipe. Open testing means that anyone can download your app after joining your testing program.

Edit the `beta` lane in the file with the following instructions:

```
lane :beta do
  gradle( task: 'assemble', build_type: 'Release' )
```

```
end
```

This allows you to build an APK file in **`release`** mode with the `fastlane` **`beta`** command from the Terminal. Once the file has been generated, you can upload it to the Google Play Store.

> **Note**
>
> Currently, the first time you publish an Android release to the Google Play Store, you must do it manually. The second time, you can leverage the `fastlane supply` command to upload your builds.
>
> As an alternative, you could use Codemagic (https://codemagic.io/start/) or GitHub Actions (https://github.com/features/actions) to fully automate the process.

The first time you publish an app, you need to provide the required data in the store; this is a rather time-consuming activity, but it's only required once.

See also

In order to have a complete view of the beta testing options in the Google Play Console, have a look at https://support.google.com/googleplay/android-developer/answer/9845334?hl=en.

Using TestFlight to publish a beta version of your iOS app

TestFlight is a tool developed by Apple that allows you to share your app with other users by inviting them with an email or by creating a link. In this recipe, you will see how to create, sign, and publish an iOS app leveraging `fastlane` and TestFlight.

Getting ready

Before following along with this recipe, you should have completed all the previous recipes in this chapter that target iOS.

How to do it...

Using the following set of steps, you will build and sign your iOS app and publish it to TestFlight:

1. From Xcode, open the `Runner.xcodeproj` file.
2. Click on the **Signing** tab and select or add your development team (this is the team associated with your Apple developer account).
3. Sign in to your Apple ID account page at https://appleid.apple.com/account/manage.
4. In the **Security** section, click the **Generate Password** link under the **APP-SPECIFIC PASSWORDS** label.
5. Choose a name for your new password and copy it somewhere secure; you will need it later.
6. From Visual Studio Code, open `Info.plist` in the `ios` directory and add the following key before the end of the `</dict>` node:

```
<key>ITSAppUsesNonExemptEncryption</key>
 <false/>
```

7. In the `ios/fastlane` directory, open the `fastfile` file and add a new `beta` lane, as shown:

```
lane :beta do
  get_certificates
  get_provisioning_profile
  increment_build_number()
  build_app(workspace: "Runner.xcworkspace", scheme:
"Runner")
  upload_to_testflight
end
```

8. From a Terminal window, run the following command:

```
bundle exec fastlane beta
```

9. When prompted, insert the app-specific password you generated previously.

> **Note**
>
> Uploading your app to TestFlight might take a long time, depending on your machine and the speed of your network. Do not interrupt this task until it's completed.

10. After a few minutes, make sure you see a success message. This means your app has been published to TestFlight.

How it works...

With a script that takes just a few lines, you can literally save yourself and your team hours of work.

The first step in this recipe was adding the development team to your Xcode project. This is needed because signing the app requires a development team.

Then, you generated an app-specific password. This is a password that you can use with your Apple ID and allows you to sign in to your account from any app. So, instead of using your main password, you can give access to other apps (in this case, `fastlane`).

In this way, when a security breach happens, you only need to remove the password that was compromised and everything else will keep working securely.

Each iOS app contains an `Info.plist` file, where you keep configuration data. There you put the following node:

```
<key>ITSAppUsesNonExemptEncryption</key>
 <false/>
```

The purpose of this node is to tell TestFlight that you are not using any special encryption tool within the app; some encryption tools cannot be exported in all countries, which is why adding this key is recommended for most apps.

> **Note**
>
> For more information about encryption and App Store publishing, have a look at https://www.cocoanetics.com/2017/02/itunes-connect-encryption-info/.

The first instruction in the `beta` lane that you have created, `get_certificates`, checks whether there are signing certificates already installed on your machine, and if necessary, it will create, download, and import the certificate.

> **Note**
>
> You can also call `get_certificates` with the `fastlane cert` Terminal tool, which is an alias for the same task.

`get_provisioning_profile` creates a provisioning profile and saves it in the current folder. You can achieve the same result through the `fastlane sign` Terminal instruction.

`increment_build_number()` automatically increments by 1 the current build number for your app.

You use the `build_app()` instruction to automatically build and sign your app, and `upload_to_testflight` uploads your generated file to the App Store Connect TestFlight section, which you can use to invite users for beta testing.

To install your app and provide feedback, testers will use the TestFlight app for iPhone, iPad, iPod Touch, Apple Watch, and Apple TV.

Now every time you want to publish a new beta to TestFlight, you can just run the `bundle exec fastlane beta` command in your Terminal window, and `fastlane` will do everything else. This can save you hours of work every time you need to update an app for beta testing.

See also

TestFlight contains several features that you can leverage to make the most of your testing process. For more information about TestFlight, see https://developer.apple.com/testflight/.

Publishing your app to the stores

Once you have released your app as a beta version in the Google Play Console and in TestFlight, publishing it to production is very easy. In this recipe, you will see the steps required to finally see your app in the stores.

Getting ready

Before following along with this recipe, you should have completed all the previous recipes in this chapter, or at least those that target your specific platform (iOS or Android).

How to do it...

You will now see how to move your beta app to production, both in the Google Play Store and in the Apple App Store.

Moving your app to production in the Play Store

Using the following steps, you will move your Android app from beta to production:

1. Go to the Google Play Console at https://play.google.com/apps/publish and select your app.
2. On your app dashboard, click on the **Releases overview** link on the left.
3. In the **Latest releases** section, click on your testing release.
4. Click on the **Promote release** link, then select **Production**, as shown in the following screenshot:

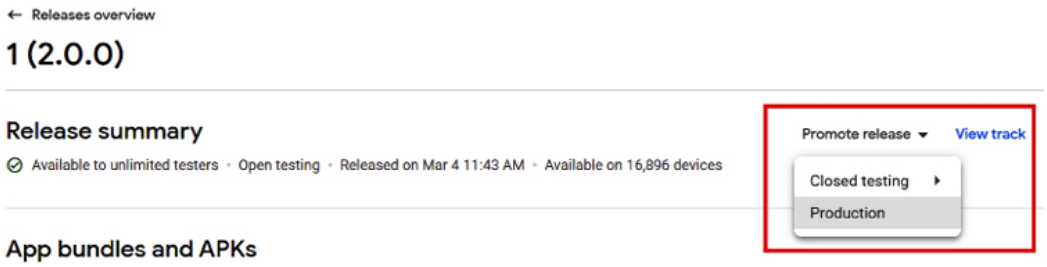


Figure 16.11: Play Store Release summary

5. Add or edit the release notes in the text field, then click the **Review Release** button.
6. Check the release summary and then click the **Start rollout to production** button.

Moving your app to production in the App Store

In the next steps, you will move your iOS app from beta to production:

1. Get to the App Store Connect page at appstoreconnect.apple.com, then click on the **My Apps** button and select your app.
2. On the **Prepare for Submission** page, make sure all the required previews, screenshots, and texts are complete; otherwise, add the

missing data.

3. In the **Build** section, click on **Select a build before you submit your app** button.
4. Select the build you have uploaded through `fastlane`, then click on the **Done** button.
5. Click on the **Submit for Review** button at the top of the page and confirm your choice.

How it works...

After clicking the **Start rollout to production** button, you can usually expect to see your app in the Play Store in about a week. For iOS, the times are currently similar.

See also

Now your app is finally published to the store(s)! But you are not done yet; you need to measure the performance and errors and monitor the usage statistic. Fortunately, the stores can give you some basic information that can help you monitor your app. For more information, have a look at https://support.google.com/googleplay/android-developer/answer/139628?co=GENIE.Platform%3DAndroid&hl=en for the Google Play Store and https://developer.apple.com/app-store-connect/analytics/ for the Apple App Store.

Firebase Analytics (https://firebase.google.com/docs/analytics) also provides very detailed information and statistics on your users' interactions with your app.

Summary

In this chapter, you have seen how to publish your app in the main mobile stores, Google Play and the Apple App Store, and the many small tasks that app developers need to perform when distributing their apps. For some of

those, you have leveraged **`fastlane`**, a powerful tool that offers a set of scripts to automate the deployment of iOS and Android apps.

You have learned how to register your iOS app on App Store Connect and your Android app on Google Play.

You have learned how to install and configure `fastlane`, generate iOS code signing certificates and provisioning profiles, and generate Android release certificates.

You have configured your app metadata, providing essential information about your app to users and the stores, and seen how to add icons to your app.

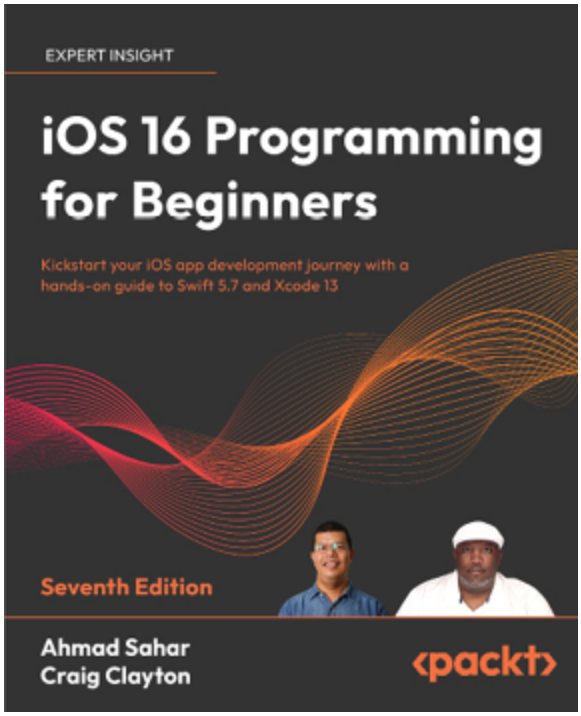
You have looked into the process of publishing beta versions of your app using the Google Play Store for Android and TestFlight for iOS: this enables you to receive feedback from beta testers before releasing the final version of your app to the public.

Finally, you learned how to publish your app to the Google Play Store and Apple App Store, completing the app distribution process.

As you continue to create and distribute your mobile apps, this knowledge will hopefully serve as a foundation to ensure an efficient distribution process.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



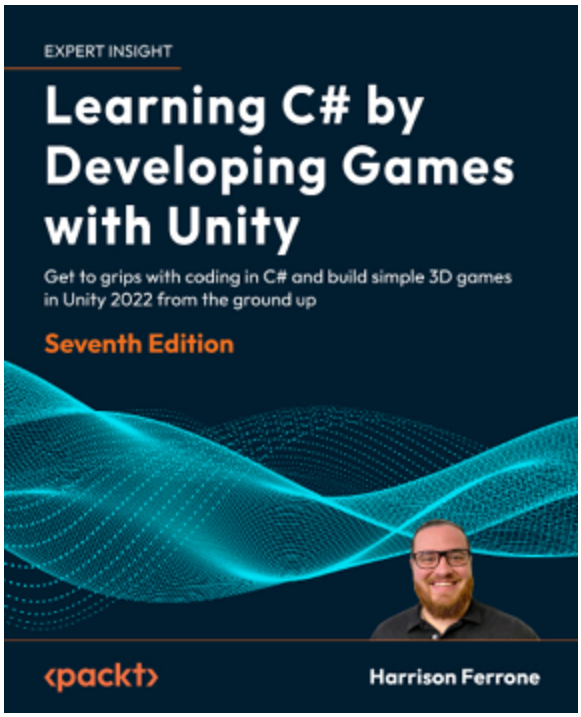
iOS 16 Programming for Beginners, Seventh Edition

Ahmad Sahar

ISBN: 9781803237046

- Get to grips with the fundamentals of Xcode 14 and Swift 5.7, the building blocks of iOS development
- Understand how to prototype an app using storyboards
- Discover the Model-View-Controller design pattern and how to implement the desired functionality within an app
- Implement the latest iOS 16 features such as SwiftUI, Lock screen widgets, and WeatherKit

- Convert an existing iPad app into a Mac app with Mac Catalyst
- Design, deploy, and test your iOS applications with design patterns and best practices



Learning C# by Developing Games with Unity 2022, Seventh Edition

Harrison Ferrone

ISBN: 9781837636877

- Understanding programming fundamentals by breaking them down into their basic parts
- Comprehensive explanations with sample codes of object-oriented programming and how it applies to C#
- Follow simple steps and examples to create and implement C# scripts in Unity
- Divide your code into pluggable building blocks using interfaces, abstract classes, and class extensions

- Grasp the basics of a game design document and then move on to blocking out your level geometry, adding lighting and a simple object animation
- Create basic game mechanics such as player controllers and shooting projectiles using C#
- Become familiar with stacks, queues, exceptions, error handling, and other core C# concepts
- Learn how to handle text, XML, and JSON data to save and load your game data



101 UX Principles, Second Edition

Will Grant

ISBN: 9781803234885

- Work with user expectations, not against them

- Make interactive elements obvious and discoverable
- Optimize your interface for mobile
- Streamline creating and entering passwords
- Use animation with care in user interfaces
- How to handle destructive user actions

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Flutter Cookbook, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

2D barcode format 565

A

Ahead-Of-Time (AOT) 51, 178

compilation 36

AlertDialog

results, obtaining from 314-317

Android

fastlane, configuring 632, 633

Google Maps, adding on 441

Android app

registering, on Google Play 628-630

Android emulator

creating 20-23

Android metadata

adding 640

Android release certificates

generating 637-639

Android SDK

setup, configuring 17

Android Studio 24-26

installing 18-20

Android Virtual Device Manager (AVD Manager) 21

AnimatedContainer 460

reference link 461

animated widgets 487

reference link 487

using 488-494

Animation API 455

AnimationController

animations, designing 461-467

animations 455

optimizing 473-475

animations Container Transform

using 499-502

animations package 498

reference link 502

using 498-502

animations tutorial

reference link 467

app

beta version, publishing in Google Play Store of 644-648

Google Maps, adding to 439-441

icons, adding to 642-644

moving, to production in App Store 653

moving, to production in Google Play Store 652, 653

publishing, to stores 652

running, on macOS 601-603

running, on Windows 604-606

state, applying to 202-207

AppBar 128

application programming interface (API) 176

app metadata

adding, for iOS 641

configuring 639

app state

making, visible across multiple screens 267-274

App Store

app, moving to production in 653

App Store Connect

iOS app, registering on 622-627

AspectRatio widget 130

async/await pattern

reference link 291

using, to avoid callbacks 286-290

asynchronous code

errors, resolving in 297

asynchronous functions

navigation routes, turning into 308-313

asynchronous operations 278

asynchronous pattern, in Flutter

reference link 285

authentication (login) form 510

await for method 388

await keyword 290

B

background decorations 134

bang operator 109

barcode

reading 565-568

basic Container animations

creating 456-460

beta channel 28

beta version

publishing, of app in Google Play Store 644-649

publishing, with TestFlight of iOS app 649-651

Big-O notation

reference link 86, 88

bottom sheets

presenting 241-246

brightness property 198

broadcast stream 402

BSD license

reference link 436

builder pattern

reference link 101

build method 207

Business LOgic Component (BLoC) pattern

using 410-415

buttons

interacting with 208-215

C

callbacks

avoiding, with async/await pattern 286-290

Canvas class

reference link 178

cascade operator 259

advantage 98-101

Center widget 130

class constructor

using 78-81

classes

creating 77-81

closures 70

functions as variables, using 70-72

CocoaPods 13, 14

collection library

reference link 88

command-line interface (CLI) 10, 609

tool 506

complex widget trees

nesting 178-184

constrained spacing 169

Container widget

using 130-136

continuous integration (CI) 631

Create, Read, Update, and Delete (CRUD) 88

Cross-Origin Resource Sharing (CORS) 590, 600

cross-platform file abstraction 560

crypto

reference link 356

Cupertino 124

curves

reference link 473

used, for designing animations 470-473

CustomPaint

used, for drawing shapes 171-177

CustomPaint class

reference link 178

D

Dart class

transforming, into JSON string 330, 331

Dart models

converting, into JSON 320-328

Dart Null Safety 104

example 104-106

working 107-109

DartPad 3

reference link 52

Dart streams

using 382-388

data

obtaining 356-363

saving, with SharedPreferences 339-344

storing, with secure storage 353-356

data layer

managing, with InheritedWidget and InheritedNotifier 260-264

data transforms

injecting, into streams 394-396

data, with collections

grouping 82-87

manipulating 82-87

subscript syntax 86

decoding 321

Default Events 530

deferred rendering 224

DELETE action

performing, on web service 376-378

dependencies

importing 419-423

Dependency injection 363

de-serialization 321

desktop menus

interacting with 614-619

dev channel 28

dev dependencies

using 424, 425

development provisioning profile 636

device-independent pixels (DIPs) 169

device's camera

using 552-561

DevTools 120

reference link 120

dialogs, in Flutter

reference link 317

didChangeDependencies 207

directories

working with 349-353

DismissDirection enum

reference link 497

Dismissible widget

swiping, implementing with 494-497

dispose method 208

distribution profile 636

Document Object Model (DOM) 149

dynamic type 70

E

Emacs 28

encoding 321

encrypt

reference link 356

error handling

enabling 393, 394

error handling, in Dart

reference link 300

errors

resolving, in asynchronous code 297

errors, with async/await

dealing with 299

errors, with then() callback

dealing with 297-299

escape character 63

Expanded widget

proportional spacing with 161-170

explicit bundle ID 627

Extensible Application Markup Language (XAML) 149

Extensible Markup Language (XML) 149

extensions

using 102, 103

working 103

extract method 186

F

face detection

reference link 577

face detector

building 572-576

facial gestures

detecting 572-576

fastlane

configuring 630, 631

configuring, for Android 632, 633

installing 630, 631

installing, for iOS 634

installing, on Mac 631

installing, on Windows 631

files

storing, in Firebase Cloud Storage 544-549

filesystem

accessing 344-348

filtering 94

Firebase Analytics

integrating 526-531

Firebase app

configuring 504-507

dependencies, adding 508-510

Firebase Cloud Firestore

using 531-539

Firebase Cloud Messaging (FCM)

used, for sending Push Notifications 539-544

Firebase Cloud Storage

files, storing 544-549

Firebase Machine Learning 551

FirebaseUI 517

first-class functions 96

versus higher-order functions 97

flattening 95

Flexible widget

proportional spacing with 161-170

Floating Action Button 34

Flutter

channels, selecting 28, 29

command line, setting up 6

framework 2

installing 3, 4

Linux command-line setup 11

macOS command-line setup 6, 7

need for 2

path variables, saving 6

Windows command-line setup 7-10

Flutter app

code, placing considerations 35-37

creating 31-34

fonts, importing into 143-148

images, importing into 143-148

platform language, selecting 34

stateful hot reload 37-42

flutter_barcode_scanner

reference link 569

flutter_bloc package

reference link 415

Flutter Camera Overlay

reference link 561

Flutter Desktop

mouse events, responding in 610-614

Flutter Doctor

used, for confirming environment 11

Flutter SDK

clone and configure, steps 6

Git, installing 5

installing 4, 5

Flutter Version Management (FVM) 5

reference link 5

Flutter Web

leveraging, by creating responsive app 590-600

Flutter website

deploying 606-609

Flutter Widgets 101 246

fonts

importing, into Flutter app 143-148

foreground decorations 134

fuchsia 241

functions

writing 66-70

functions, as variables

using, with closures 70-72

FutureBuilder

reference link 308

using, to manage Future with Flutter 305-308

Future class

states 285

using 278-285

FutureGroup 294

Future objects

completing 291-294

Futures, with StatefulWidgets

using 300-304

G

generator function 387

Git

installation link 5

installing 5

used, for managing Flutter SDK 5

global themes

applying 192-198

GNU project 28

Google Maps

adding, on Android 441

adding, on iOS 441-445

adding, to app 439-441

markers, adding to 449-453

Google Play

Android app, registering on 628-630

Google Play Console

reference link 628

Google Play Store

app, moving to production in 652, 653

app, publishing 652

beta version, publishing of app 644-649

tracks, selecting 648

Google Sign-in

adding 518-522

graphics processing unit (GPU) 177

H

handleData 397

handleDone 397

handleError 397

Hardware Accelerated Execution Manager (HAXM) 20

Hero animations 476

using 477-481

heroes

reference link 482

higher-order functions 88

chaining 96

filtering 94

flattening 95

iterating 96

less code, writing 88-92

mapping 92, 93

reducing 94

reference link 97

sorting 93, 94

versus first-class functions 97

Homebrew 16

hot reload 256

hot restart 256

HTTP client

designing 356-363

HttpHelper class

usage 363, 364

HyperText Markup Language (HTML) 149

I

icons

adding, to app 642-644

images

importing, into Flutter app 143-148

labeling 569-572

text, recognizing from 561-565

immutable variables 53

immutable widgets

creating 116-123

implicit animations 456

inheritance 80

InheritedNotifier

data layer, managing with 260-264

working 264-266

Inherited Widgets 363

data layer, managing with 260-263

reference link 266

working 264-266

initState 207

integrated development environment (IDE) 4, 604

selecting 24

IntelliJ IDEA 27

internet of things (IoT) 583

interpolation 466

iOS

fastlane, installing 634

Google Maps, adding on 441-445

metadata, adding 641

registering, on App Store Connect 622-627

TestFlight, used for publishing beta version of 649-651

iOS code signing certificates

generating 634-636

iOS SDK

checking, in with Doctor 16, 17

CocoaPods 13, 14

configuring 12

Homebrew 16

Xcode command-line tools 14-16

Xcode, downloading 12, 13

itemExtent 225

J

JavaScript Object Notation (JSON)

Dart models, converting into 320-328

errors, catching 337-339

file, reading 328

JSON schemas

handling, that are incompatible with models 331-337

JSON string

Dart class, transforming into 330, 331

transforming, into list of Map objects 328

Just-In-Time (JIT) compilation 36, 51

L

language

identifying 577-582

Language Server Protocol mode (LSP mode) 28

large datasets

handling, with list builders 222-225

linear barcode format 565

Linux

command-line setup 11

list builders

large datasets, handling with 222-225

listen method 388

ListView widgets 487

location services

using 446-448

login screen

creating 510-518

M

Mac

fastlane, installing on 631

Machine Learning (ML) 551

Machine Learning Vision, for Firebase

reference link 565

macOS

app, running on 601-603

command-line setup 6, 7

Map objects

JSON string, transforming into list of 328

transforming, into Pizza objects 329

mapping 92, 93

Markdown format 438

markers

adding, to Google Map 449-453

master channel 28

Material Design 124

ML Kit language identification

reference link 582

model conversion overview

reference link 586

models 249

mouse events

responding, in Flutter Desktop 610-614

multiple animations

adding 468-470

multiple Futures

firing, at same time 294-296

multiple stream subscriptions

allowing 402-405

N

named optional parameters 68

named parameters 69

navigation routes

turning, into asynchronous functions 308-313

next screen

navigating to 232-235

NoSQL databases 531

npm (Node Package Manager) 506

null-aware operators

reference link 113

null-coalescing operator 109

null safety

reference link 109

using, in classes 109-111

working 111-113

O

object-oriented programming (OOP) 77

building blocks 81

of-contex pattern 142

P

package (Part 1)

creating 426-433

package (Part 2)

creating 433-436

package (Part 3)

creating 436-439

packages

importing 419-423

Padding widget 115

path_provider 344-348

documents directory 348

temporary directory 348

payload 371

permission_handler

reference link 448

pixel densities 169

pixels per inch (PPI) 169

Pizza objects

Map objects, transforming into 329

platform-aware widgets 130

plugin 419, 426

POST action

performing, on web service 364-372

Postman 371

predicate 94

premade animation transitions

using 482-487

Provider 363

provisioning profiles

provisioning 634-636

Push Notifications

sending, with Firebase Cloud Messaging (FCM) 539-544

PUT action

performing, on web service 372-376

R

reactive user interfaces

creating, with StreamBuilder 406-410

Realtime Database 531

reducing 94

refactoring 185

relative import 119

responsive app

creating, for leveraging Flutter Web 590-600

RESTful web services 371

Riverpod

reference link 275

S

Scaffold widget 115

using 124-130

Scalable Vector Graphics (SVG) 177

screen

dialogs, showing on 235-240

ScrollController 225

scrolling widgets 221, 222

search engine optimization (SEO) 609

secure storage

using, to store data 353-356

separation of concerns

implementing, for UI and models 250-260

serialization 321

Service Locators 363

shapes

drawing, with CustomPaint 171-177

SharedPreferences

used, for saving data 339-344

sign-in

customizing 523-526

singleton pattern 363

sinks

using 389-393

Skia 177

SliverAnimatedList

reference link 494

software development kit (SDK) 3

sorting 93, 94

sound null safety 104

Sourcetree

URL 5

stable channel 28

state

applying, to app 202-207

State class 207

stateful hot reload 37

StatefulWidgets 202

reference link 208

stateless widget 116

stock-keeping unit (SKU) 627

stopwatch app

laps, displaying in scrollable list 215-221

StreamBuilder

using, to create reactive user interfaces 406-410

stream controllers

using 389-393

stream events

subscribing 397-402

streams

data transforms, injecting into 394-396

string 60-66

string interpolation 60-66

stylish text

printing, on screen 136-142

super.key 123

swiping

implementing, with Dismissible widget 494-497

Switch Expressions 73-77

T

TensorFlow 583

reference link 587

TensorFlow Hub

reference link 583

TensorFlow Lite 583

using 583-587

TestFlight

used, for publish beta version of iOS app 649-651

text

recognizing, from image 561-565

TextFields

working with 226-232

Text widget 115

tflite_flutter

reference link 583

Ticker class 467

to-do app

building 250

translation features, ML Kit

reference link 583

tweens 468

U

Uniform Resource Identifier (URI) 284

unit test

creating 42-49

V

variables 53

declaring 53-59

views 249

VS Code 26, 27

W

web API 356

web-renderers

reference link 601

web service 356

widgets 34

placing, one after another 150-160

widget trees

refactoring, to improve legibility 185-192

wildcard bundle ID 627

Windows

app, running on 604-606

command-line setup 7-10

fastlane, installing on 631

Wire Mock Cloud 356

Wiremock service

reference link 364

X

Xcode

downloading 12, 13

Xcode command-line tools 14-16

Y

YAML Ain't Markup Language (YAML) 37

Z

Z shell (zsh)

setting up 7

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?Is your eBook purchase not compatible with the device of your

choice?

Don’t worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don’t stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



https://packt.link/free-ebook/9781803245430

2. Submit your proof of purchase
3. That’s it! We’ll send your free PDF and other benefits to your email directly