

Flutter Cookbook

Over 100 proven techniques and solutions for app development with Flutter 2.2 and Dart

Simone Alessandria | Brian Kayfitz



Flutter Cookbook

Over 100 proven techniques and solutions for app development with Flutter 2.2 and Dart

Simone Alessandria
Brian Kayfitz

Packt

BIRMINGHAM - MUMBAI

Flutter Cookbook

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Associate Group Product Manager: Pavan Ramchandani

Publishing Product Manager: Ashitosh Gupta

Senior Editor: Hayden Edwards

Content Development Editor: Aamir Ahmed

Technical Editor: Shubham Sharma

Copy Editor: Safis Editing

Project Coordinator: Manthan Patel

Proofreader: Safis Editing

Indexer: Tejal Daruwale Soni

Production Designer: Jyoti Chauhan

First published: June 2021

Production reference: 1170621

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-83882-338-2

www.packt.com

To Renata and Giusy, the two wonderful ladies that are my past, present, and future.

– Simone Alessandria

Contributors

About the authors

Simone Alessandria wrote his first program when he was 12. It was a text-based fantasy game for the Commodore 64. Now, he is a trainer (MCT), author, speaker, passionate software architect, and always a proud coder. He is the founder and owner of Softwarehouse.it. His mission is to help developers achieve more through training and mentoring. He has authored several books on Flutter, including *Flutter Projects*, published by Packt, and web courses on Pluralsight and Udemy.

Brian's career had him working on video games, eCommerce, productivity, finance, and travel apps. After spending years working as an iOS and Android developer, he sees Flutter as the next big pillar for mobile.

About the reviewer

Luke Greenwood is a freelance software developer with over 5 years of experience in the industry, a master's degree in computer science, and a passion for Flutter. Having started his mobile development career with native Android, he soon discovered the potential of cross-platform development, first through Xamarin Forms and then later with Flutter. He has since been working exclusively with Flutter and has produced large-scale production apps across a variety of domains. He has also published articles and hosted webinars on how to make the most of the SDK.

Table of Contents

Preface	1
Chapter 1: Getting Started with Flutter	7
Technical requirements	8
How to use Git to manage the Flutter SDK	9
Installing Git	9
How to do it...	9
See also	10
Setting up the command line and saving path variables	10
macOS command-line setup	10
Windows command-line setup	11
Confirming your environment is correct with Flutter Doctor	14
Configuring the iOS SDK	15
Downloading Xcode	15
CocoaPods	16
Xcode command-line tools	17
Homebrew	18
Checking in with the Doctor	18
Configuring the Android SDK setup	20
Installing Android Studio	20
Creating an Android emulator	22
Which IDE/editor should you choose?	26
Android Studio	26
VS Code	28
IntelliJ IDEA	29
Picking the right channel	30
How to create a Flutter app	31
How to do it...	31
How to choose a platform language for your app	33
Where do you place your code?	34
Hot reload – refresh your app without recompiling	36
Summary	40
Chapter 2: Dart: A Language You Already Know	41
Technical requirements	42
Declaring variables – var versus final versus const	42
Getting ready	43
How to do it...	44
How it works...	46
There's more...	48

See also	48
Strings and string interpolation	48
Getting ready	49
How to do it...	49
How it works...	51
There's more...	52
See also	53
How to write functions	54
Getting ready	54
How to do it...	54
How it works...	57
How to use functions as variables with closures	58
Getting ready	58
How to do it...	58
How it works...	59
Creating classes and using the class constructor shorthand	60
Getting ready	61
How to do it...	61
How it works...	62
The building blocks of OOP	63
See also	64
How to group and manipulate data with collections	64
Getting ready	65
How to do it...	65
How it works...	67
Subscript syntax	68
There's more...	69
See also	69
Writing less code with higher-order functions	70
Getting ready	70
How to do it...	70
How it works...	73
Mapping	74
Sorting	75
Filtering	75
Reducing	76
Flattening	76
There's more...	77
First-class functions	77
Iterables and chaining higher-order functions	78
See also	78
How to take advantage of the cascade operator	79
Getting ready	79
How to do it...	80
How it works...	81
See also	82

Understanding Dart Null Safety	82
Getting ready	83
How to do it...	83
How it works...	86
See also	88
Chapter 3: Introduction to Widgets	89
Technical requirements	89
Creating immutable widgets	90
How to do it...	90
How it works...	93
Using a Scaffold	96
Getting ready	96
How to do it...	97
How it works...	102
Using the Container widget	102
Getting ready	103
How to do it...	103
How it works...	107
Printing stylish text on the screen	108
Getting ready	109
How to do it...	109
How it works...	113
There's more...	114
See also	115
Importing fonts and images into your app	115
Getting ready	115
How to do it...	116
How it works...	119
See also	120
Chapter 4: Mastering Layout and Taming the Widget Tree	121
Placing widgets one after another	122
Getting ready	122
How to do it...	123
How it works...	128
Proportional spacing with the Flexible and Expanded widgets	129
Getting ready	130
How to do it...	130
How it works...	135
See also	137
Drawing shapes with CustomPaint	137
Getting ready	138
How to do it...	138
How it works...	141
There's more...	143

See also	144
Nesting complex widget trees	144
Getting ready	145
How to do it...	145
How it works...	150
See also	151
Refactoring widget trees to improve legibility	151
Getting ready	151
How to do it...	152
How it works...	156
See also	158
Applying global themes	158
Getting ready	158
How to do it...	159
How it works...	163
There's more...	165
See also	165
Chapter 5: Adding Interactivity and Navigation to Your App	166
 Adding state to your app	167
Getting ready	167
How to do it...	167
How it works...	170
There's more...	172
See also	173
 Interacting with buttons	173
Getting ready	173
How to do it...	174
How it works...	177
 Making it scroll	178
Getting ready	179
How to do it...	179
How it works...	182
There's more...	184
 Handling large datasets with list builders	185
How to do it...	186
How it works...	187
There's more...	188
 Working with TextFields	189
Getting ready	189
How to do it...	189
How it works...	193
See also	195
 Navigating to the next screen	195
How to do it...	195

How it works...	197
Invoking navigation routes by name	198
How to do it...	198
How it works...	200
Showing dialogs on the screen	201
How to do it...	202
How it works...	205
There's more...	205
Presenting bottom sheets	206
How to do it...	207
How it works...	209
See also	210
Chapter 6: Basic State Management	212
Technical requirements	212
Model-view separation	213
Getting ready	213
How to do it...	214
How it works...	219
See also	220
Managing the data layer with InheritedWidget	220
Getting ready	221
How to do it...	221
How it works...	223
See also	224
Making the app state visible across multiple screens	225
Getting ready	225
How to do it...	225
How it works...	229
Designing an n-tier architecture, part 1 – controllers	230
Getting ready	231
How to do it...	231
How it works...	235
See also	236
Designing an n-tier architecture, part 2 – repositories	237
Getting ready	237
How to do it...	237
How it works...	239
Designing an n-tier architecture, part 3 – services	241
How to do it...	241
How it works...	245
There's more...	247
See also	247
Chapter 7: The Future is Now: Introduction to Asynchronous Programming	248

Technical requirements	249
Using a Future	249
Getting ready	250
How to do it...	251
How it works...	254
See also	255
Using async/await to remove callbacks	256
Getting ready	257
How to do it...	258
How it works...	259
See also	260
Completing Futures	260
Getting ready	261
How to do it...	261
How it works...	262
There's more...	262
See also	263
Firing multiple Futures at the same time	263
Getting ready	264
How to do it...	264
How it works...	265
See also	266
Resolving errors in asynchronous code	266
Getting ready	266
How to do it...	266
Dealing with errors using the then() callback:	267
Dealing with errors using async/await	268
How it works...	269
See also	269
Using Futures with StatefulWidget	269
Getting ready	270
How to do it...	270
How it works...	271
There's more...	272
See also	273
Using the FutureBuilder to let Flutter manage your Futures	273
Getting ready	273
How to do it...	274
How it works...	275
There's more...	275
See also	275
Turning navigation routes into asynchronous functions	276
Getting ready	277
How to do it...	278
How it works...	280

Getting the results from a dialog	280
Getting ready	281
How to do it...	282
How it works...	283
See also	284
Chapter 8: Data Persistence and Communicating with the Internet	285
Technical requirements	286
Converting Dart models into JSON	286
Getting ready	287
How to do it...	288
How it works...	293
Reading the JSON file	294
Transforming the JSON string into a list of Map objects	294
Transforming the Map objects into Pizza objects	295
There's more...	296
See also	297
Handling JSON schemas that are incompatible with your models	298
Getting ready	298
How to do it...	298
How it works...	301
There's more...	303
See also	305
Catching common JSON errors	305
Getting ready	305
How to do it...	306
How it works...	307
See also	307
Saving data simply with SharedPreferences	308
Getting ready	308
How to do it...	308
How it works...	311
See also	312
Accessing the filesystem, part 1 – path_provider	313
Getting ready	313
How to do it...	313
How it works...	315
See also	316
Accessing the filesystem, part 2 – working with directories	317
Getting ready	317
How to do it...	317
How it works...	319
See also	320
Using secure storage to store data	321
Getting ready	321
How to do it...	321

How it works...	323
See also	324
Designing an HTTP client and getting data	324
Getting ready	325
How to do it...	325
How it works...	329
There's more...	330
See also	331
POST-ing data	331
Getting ready	331
How to do it...	331
How it works...	337
PUT-ing data	338
Getting ready	338
How to do it...	338
How it works...	341
DELETE-ing data	342
Getting ready	342
How to do it...	342
How it works...	344
Chapter 9: Advanced State Management with Streams	345
Technical requirements	346
How to use Dart streams	346
Getting ready	347
How to do it...	347
How it works...	350
There's more...	352
See also	352
Using stream controllers and sinks	352
Getting ready	353
How to do it...	353
How it works...	356
There's more...	357
See also	358
Injecting data transform into streams	358
Getting ready	359
How to do it...	359
How it works...	360
See also	361
Subscribing to stream events	361
Getting ready	362
How to do it...	362
How it works...	365
See also	366

Allowing multiple stream subscriptions	367
Getting ready	367
How to do it...	367
How it works...	370
See also	370
Using StreamBuilder to create reactive user interfaces	370
Getting ready	370
How to do it...	371
How it works...	374
See also	375
Using the BLoC pattern	375
Getting ready	376
How to do it...	376
How it works...	379
See also	380
Chapter 10: Using Flutter Packages	381
 Technical requirements	382
 Importing packages and dependencies	382
Getting ready	383
How to do it...	383
How it works...	384
See also	386
 Creating your own package (part 1)	386
Getting ready	386
How to do it...	387
How it works...	390
See also	392
 Creating your own package (part 2)	392
Getting ready	392
How to do it...	392
How it works...	394
See also	395
 Creating your own package (part 3)	395
Getting ready	395
How to do it...	395
How it works...	397
See also	398
Adding Google Maps to your app	398
Getting ready	398
How to do it...	398
How it works...	403
See also	404
Using location services	404
Getting ready	404

How to do it...	404
How it works...	406
See also	407
Adding markers to a map	407
Getting ready	407
How to do it...	407
How it works...	410
There's more...	411
Chapter 11: Adding Animations to Your App	412
 Creating basic container animations	413
Getting ready	413
How to do it...	413
How it works...	416
See also	417
 Designing animations part 1 – using the AnimationController	418
Getting ready	418
How to do it...	419
How it works...	422
See also	424
 Designing animations part 2 – adding multiple animations	424
Getting ready	424
How to do it...	425
How it works...	426
 Designing animations part 3 – using curves	426
Getting ready	426
How to do it...	427
How it works...	428
See also	429
Optimizing animations	430
Getting ready	430
How to do it...	430
How it works...	431
See also	432
Using Hero animations	432
Getting ready	433
How to do it...	433
How it works...	436
See also	437
Using premade animation transitions	438
Getting ready	439
How to do it...	439
How it works...	441
See also	443
Using the AnimatedList widget	443

Getting ready	444
How to do it...	444
How it works...	447
See also	449
Implementing swiping with the Dismissible widget	450
Getting ready	450
How to do it...	450
How it works...	452
See also	453
Using the animations Flutter package	453
Getting ready	454
How to do it...	454
How it works...	455
See also	457
Chapter 12: Using Firebase	458
Configuring a Firebase app	459
Getting ready	459
How to do it...	459
Android configuration	460
iOS configuration	462
Adding Firebase dependencies	463
How it works...	463
See also	465
Creating a login form	465
Getting ready	465
How to do it...	465
How it works...	472
See also	474
Adding Google Sign-in	474
Getting ready	475
How to do it...	475
How it works...	480
See also	482
Integrating Firebase Analytics	482
Getting ready	482
How it works...	482
How it works...	485
See also	487
Using Firebase Cloud Firestore	487
Getting ready	487
How to do it...	488
How it works...	493
See also	495
Sending Push Notifications with Firebase Cloud Messaging (FCM)	495
Getting ready	495

How to do it...	495
How it works...	499
See also	499
Storing files in the cloud	500
Getting ready	500
How to do it...	500
How it works...	504
Chapter 13: Machine Learning with Firebase ML Kit	505
Using the device camera	506
Getting ready	506
How to do it...	506
How it works...	513
See also	515
Recognizing text from an image	515
Getting ready	515
How to do it...	515
How it works...	518
See also	519
Reading a barcode	520
Getting ready	520
How to do it...	520
How it works...	522
See also	523
Image labeling	523
Getting ready	523
How to do it...	523
How it works...	525
See also	526
Building a face detector and detecting facial gestures	526
Getting ready	526
How to do it...	527
How it works...	529
See also	531
Identifying a language	531
Getting ready	531
How to do it...	531
How it works...	535
See also	536
Using TensorFlow Lite	536
Getting ready	537
How to do it...	537
How it works...	539
See also	540
Chapter 14: Distributing Your Mobile App	541

Technical requirements	542
Registering your iOS app on App Store Connect	542
Getting ready	542
How to do it...	543
How it works...	546
See also	548
Registering your Android app on Google Play	548
Getting ready	548
How to do it...	548
How it works...	549
See also	550
Installing and configuring fastlane	550
Getting ready	551
How to do it...	551
Installing fastlane on Windows	551
Installing fastlane on a Mac	551
Configuring fastlane for Android	551
Installing fastlane for iOS	554
See also	554
Generating iOS code signing certificates and provisioning profiles	555
Getting ready	555
How to do it...	555
How it works...	556
See also	557
Generating Android release certificates	557
Getting ready	557
How to do it...	558
How it works...	559
See also	560
Auto-incrementing your Android build number	560
Getting ready	560
How to do it...	561
How it works...	561
See also	562
Configuring your app metadata	563
Getting ready	563
How to do it...	563
Adding Android metadata	563
Adding metadata for iOS	564
How it works...	564
See also	565
Adding icons to your app	565
Getting ready	566
How to do it...	566
How it works...	567

See also	568
Publishing a beta version of your app in the Google Play Store	568
Getting ready	568
How to do it...	569
How it works...	572
See also	573
Using TestFlight to publish a beta version of your iOS app	573
Getting ready	573
How to do it...	574
How it works...	575
See also...	576
Publishing your app to the stores	576
Getting ready	576
How to do it...	577
Moving your app to production in the Play Store	577
Moving your app to production in the App Store	577
How it works...	578
See also...	578
Chapter 15: Flutter Web and Desktop	579
Creating a responsive app leveraging Flutter Web	580
Getting ready	580
How to do it...	580
How it works...	587
See also...	590
Running your app on macOS	590
Getting ready	590
How to do it...	591
How it works...	593
See also	594
Running your app on Windows	594
Getting ready...	594
How to do it...	594
How it works...	596
See also...	596
Deploying a Flutter website	596
Getting ready	596
How to do it...	597
How it works...	599
See also...	600
Responding to mouse events in Flutter Desktop	600
Getting ready	600
How to do it	600
How it works...	602
See also	604

Table of Contents

Interacting with desktop menus	604
Getting ready...	604
How to do it...	605
How it works...	607
See also...	608
About Packt	609
Index	610

Preface

This book contains over 100 short recipes that will help you learn Flutter by example. These recipes cover the most important Flutter features that will allow you to develop real-world apps. In every recipe, you will learn 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, and now that **Flutter 2.2** has been released, you can also use the same code base for the web and desktop.

There are 15 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 introduced in earlier chapters.

Flutter uses Dart as a programming language. *Chapter 2, 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 2, 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*, will help you set up your development environment.

Chapter 2, *Dart: A Language You Already Know*, introduces Dart, its syntax, and its patterns.

Chapter 3, *Introduction to Widgets*, shows how to build simple user interfaces with Flutter.

Chapter 4, *Mastering Layout and Taming the Widget Tree*, shows how to build more complex screens made of several widgets.

Chapter 5, *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 6, *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 7, *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 8, *Data Persistence and Communicating with the Internet*, gives you the tools to connect to web services and persist data into your device.

Chapter 9, *Advanced State Management with Streams*, shows how to deal with Streams, which are arguably the best tool to create reactive apps.

Chapter 10, *Using Flutter Packages*, teaches you how to choose, use, build, and publish Flutter packages.

Chapter 11, *Adding Animations to Your App*, gives you the tools you need to build engaging animations in your apps.

Chapter 12, *Using Firebase*, shows how to leverage a powerful backend without any code!

Chapter 13, *Machine Learning with Firebase MLKit*, shows how to add machine learning features to your apps by using Firebase.

Chapter 14, *Distributing Your Mobile App*, outlines the steps required to publish an app into the main stores: the Google Play Store and the Apple App Store.

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

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 covered in the book	OS requirements
Visual Studio Code, Android Studio, or IntelliJ Idea	Windows, macOS, or Linux
Flutter SDK	Windows, macOS, or Linux
An emulator/simulator or an iOS or Android device	Windows, macOS, or Linux (macOS is needed only for iOS)

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 (link 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 authors' and reviewer's everlasting gratitude.

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Flutter-Cookbook>. In case there's an update to the code, it will be updated on the existing GitHub repository.

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://static.packt-cdn.com/downloads/9781838823382_ColorImages.pdf.

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. Here is an example: "Edit the `increaseValue` method again. This time, use the null-check operator."

A block of code is set as follows:

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

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

```
final int numValue = 42; // this is ok  
// NOT OK: const int or var int.
```

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

```
$ mkdir css  
$ cd css
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "In DartPad, make sure **Null Safety** is **disabled**"

Warnings or important notes appear like this.





Tips and tricks appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@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 would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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@packt.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 authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Getting Started with Flutter

Creating your development environment is the first task that every developer has to go through when starting with a new platform. In some ways, the ease in which you can go from nothing to building software can be seen as a litmus test for how your experience with the platform is going to be. If the environment is difficult and painful to set up, then it might be very likely that it will be difficult and painful to work with.

The Flutter engineers must have taken this to heart because getting started with Flutter is easier than with other frameworks. You can divide the process into three distinct sections. First, you have to install the Flutter **software development kit (SDK)**. Then, you have to install at least one platform SDK—iOS or Android, or both if you are working on a Mac. Since Flutter 2.0, you can also install a desktop SDK to develop apps for Windows, macOS, or Linux. 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 that will scan your environment and offer you step-by-step guides for what you need to do to successfully complete your environment setup. This means that the Flutter team has made every effort to help you successfully install and use Flutter to develop your projects.

By the end of this chapter, you will have Flutter fully installed and will have learned how to create an app and run code on a virtual device.

In this chapter, we'll be covering the following recipes:

- 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
- How to create a Flutter app
- How Flutter projects are structured
- How to run a Flutter app
- How to use Hot reload to refresh your app without recompiling



While Flutter is compatible with Windows, macOS, and Linux, if you are interested in building applications for Apple's platforms (iOS and macOS), you will need a Mac to build your app.

Technical requirements

Building mobile applications can be a taxing task for your computer.

Your computer should have the following:

- 8 GB of **random-access memory (RAM)** (16 **gigabytes (GB)** preferred)
- 50 GB of available hard drive space
- A **solid-state drive (SSD)** hard drive is recommended
- At least a 2 **gigahertz (GHz)** + processor

If you want to build for iOS, you will also need a Mac instead of a PC.

These are not strict system requirements, but anything less than this may lead to you spending more time waiting rather than working.

How to use Git to manage the Flutter SDK

Before you can build anything, you need to download the Flutter SDK. If you go to the main Flutter website at <https://flutter.dev>, they currently recommend that you download one of their prebuilt packages for macOS, Windows, or Linux. This is certainly OK, and if you feel comfortable with this approach, you can certainly follow it. However, we can do better. Since Flutter is completely open source and hosted on GitHub, if you just clone the main Flutter repository, you'll already have everything you'll need, and you can easily change to different versions of the Flutter SDK if 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, so why not go straight to the source?

Installing Git

First, you need to make sure you have Git installed on your computer. If you are developing on macOS, you can skip this step.

For Windows, you can download and install Git here: <https://git-scm.com/download/win>.

You might also want to get 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 greatly 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, if you 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.

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 location does not explicitly matter, but it will be simpler to install the SDK closer to the root of your hard drive.
2. On macOS, type in the following command:

```
cd $HOME
```

This ensures that the terminal is pointing to your home directory. It might be redundant since most terminal windows automatically open to the home directory when they are opened.

3. We can now install Flutter with this command:

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

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

See also

If you do not feel comfortable using Git, you can certainly install your Flutter SDK by following the instructions at <https://flutter.dev/docs/get-started/install>.

Setting up the command line and saving path variables

Now that you have cloned the Flutter repository, there are 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, Linux, and Windows in the following sections.

macOS command-line setup

To actually use Flutter, we need to save the location of the Flutter executable to your system's environment variables.



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:

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

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.

2. Type the following command at the bottom of the file:

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

3. If you chose to install Flutter at a different location, then replace `$HOME` with the appropriate directory.
4. Exit `nano` by typing `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.

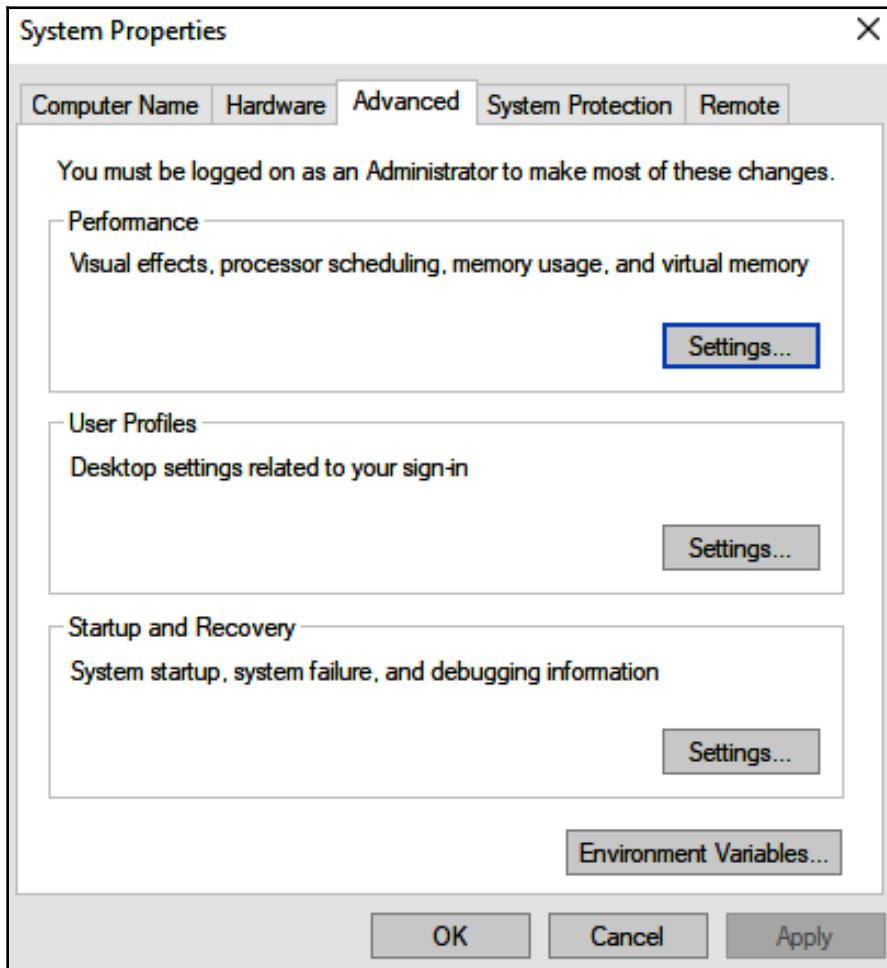
Windows command-line setup



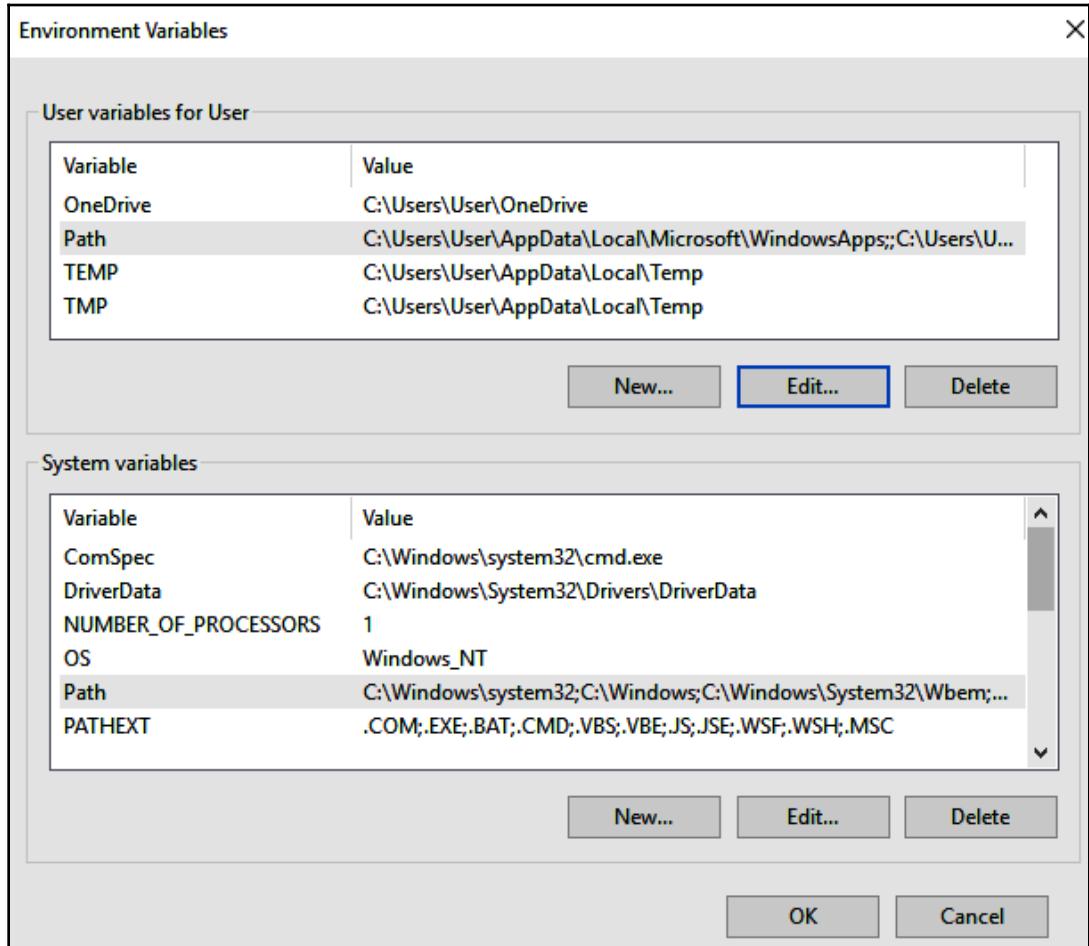
These instructions assume you are using Windows 10.

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

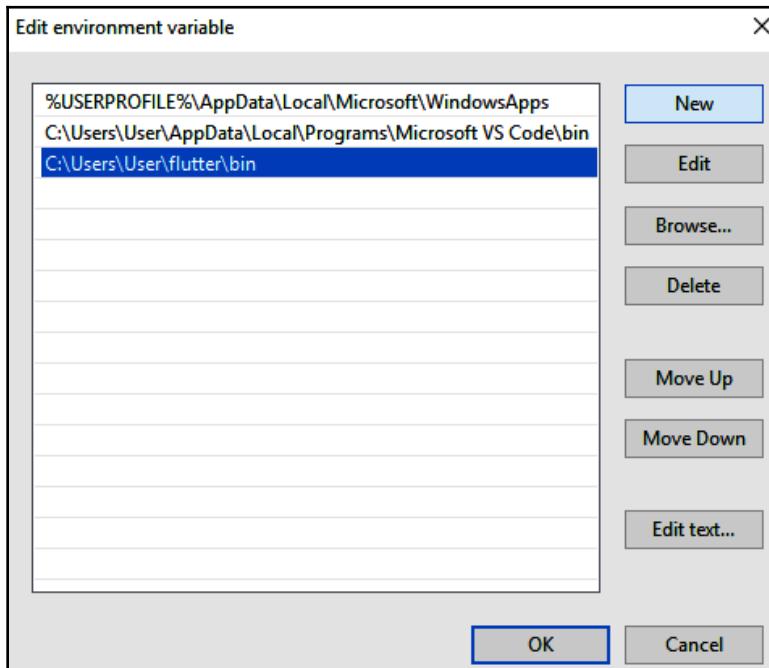
1. In the search bar at the bottom of the desktop, type env. You should see an **Edit the system environment variables** option appear. Select the icon to open the **System Properties** window, and at the bottom of the screen click the **Environment Variables...** button:



2. In the next dialog, select the **Path** variable in the **User variables for User** section and click the **Edit...** button:



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



- Type `C:\Users\{YOUR_USER_NAME}\flutter\bin`, then select **OK**. Flutter should now be added to your path.
- Restart your system.
- 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.

Confirming your environment is correct with Flutter Doctor

Flutter comes with a tool called **Flutter Doctor** that will be your new best friend when setting up the SDK. Flutter Doctor will give you 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 to check whether your system is up to date.

In your terminal window, type the following command:

```
flutter doctor
```

Flutter Doctor will tell you if your platform SDKs are configured properly and whether Flutter can see any devices, including the web browser.

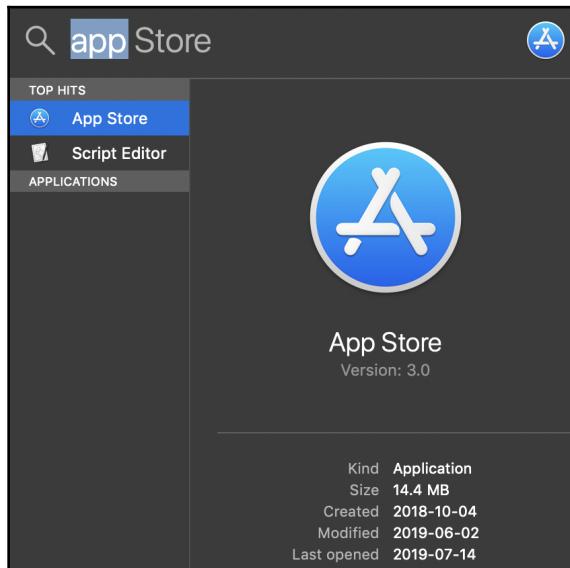
Configuring the iOS SDK

The iOS SDK is mostly provided in a single application, Xcode. Xcode is one behemoth of an application and it controls all the official ways in which you will interact with Apple's platforms. 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. Flutter uses both of these tools in its build system.

Downloading Xcode

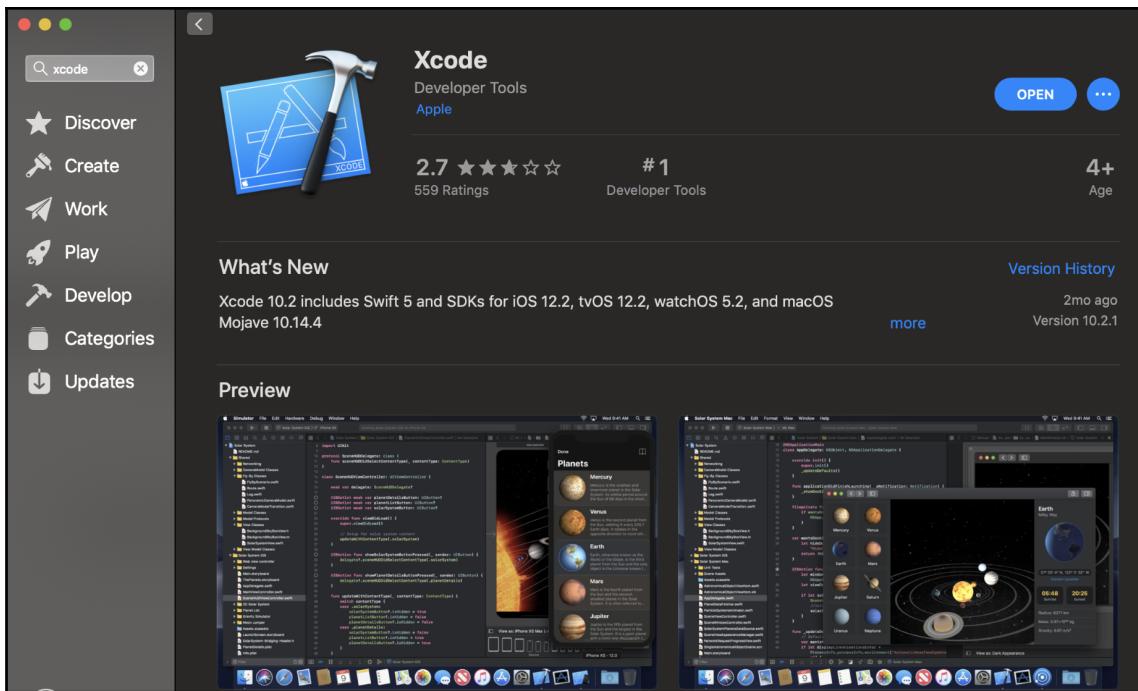
The iOS SDK comes bundled with Apple's IDE, Xcode. The best place to get Xcode is through the Apple App Store:

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



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**:



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 lead dependency manager for iOS development. It is essentially the equivalent of npm for the web community. Flutter requires CocoaPods in its build process to link any libraries you have added to your project:

1. To install cocoapods, type this command:

```
sudo gem install cocoapods
```

2. 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.
3. After cocoapods has finished installing, type this command:

```
pod setup
```

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

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. After it is done, open the application to allow Xcode to fully configure itself. Once you see the **Welcome to Xcode** screen appear, you can close the application:



2. Type this command in the Terminal window to install the command-line tools:

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

3. 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
```

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/install)"
```

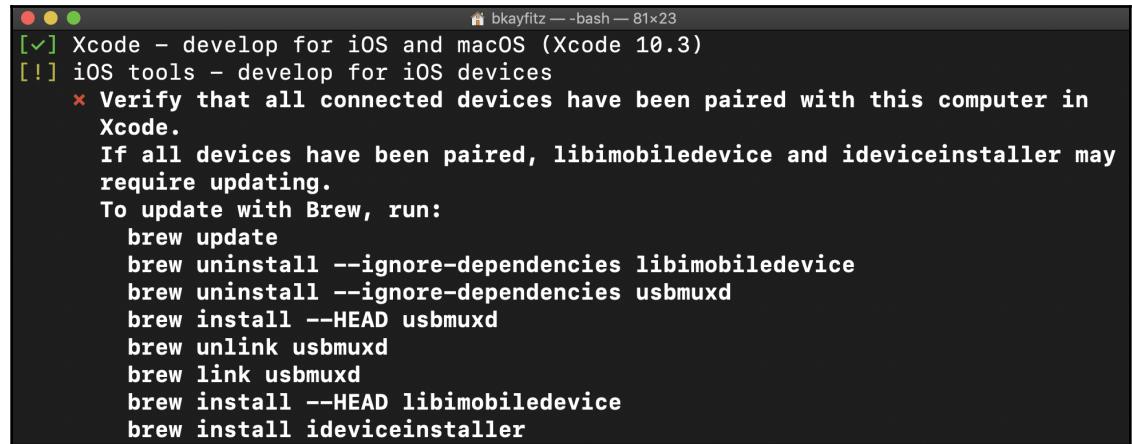
We will be using Homebrew primarily as a mechanism to get other, smaller tools.

You can also get more information about Homebrew from 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:



```
[v] Xcode - develop for iOS and macOS (Xcode 10.3)
[!] iOS tools - develop for iOS devices
✖ Verify that all connected devices have been paired with this computer in
  Xcode.
  If all devices have been paired, libimobiledevice and ideviceinstaller may
  require updating.
To update with Brew, run:
  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
```

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.

Hopefully, you prefer option 2.

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

```
nano update_ios_toolchain.sh
```

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

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

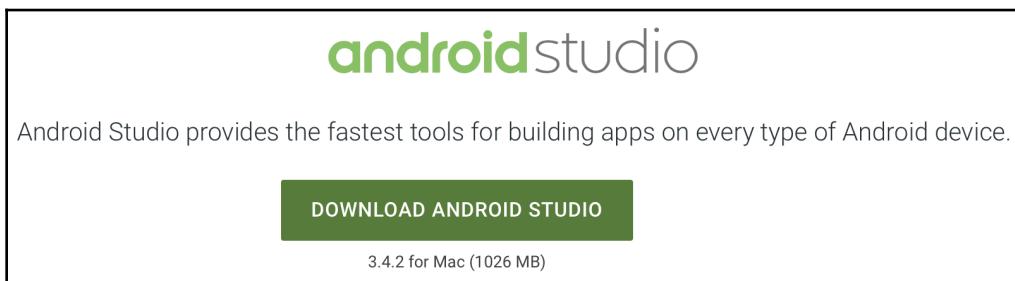
Configuring the Android SDK setup

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

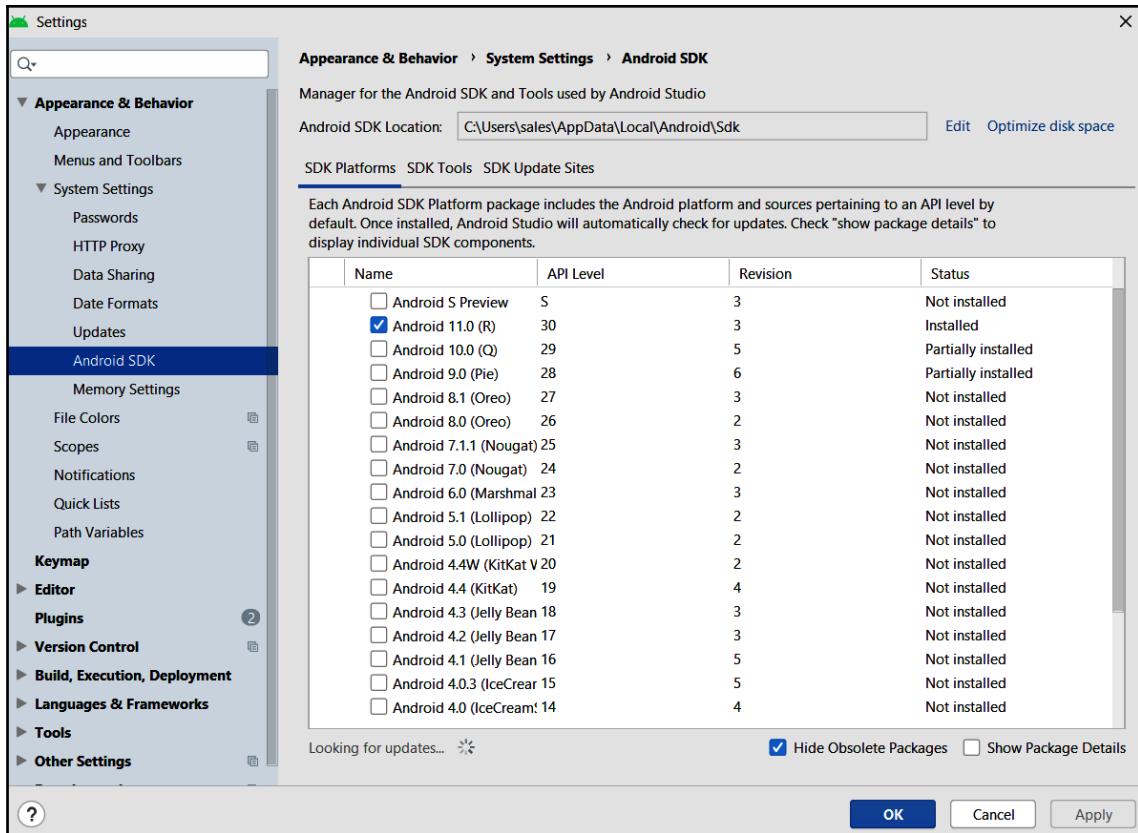
Installing Android Studio

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:



2. After Android Studio is installed, you'll need to download at least one Android SDK. From the **Android Studio** menu, select **Preferences** and then type `android` into the search field:

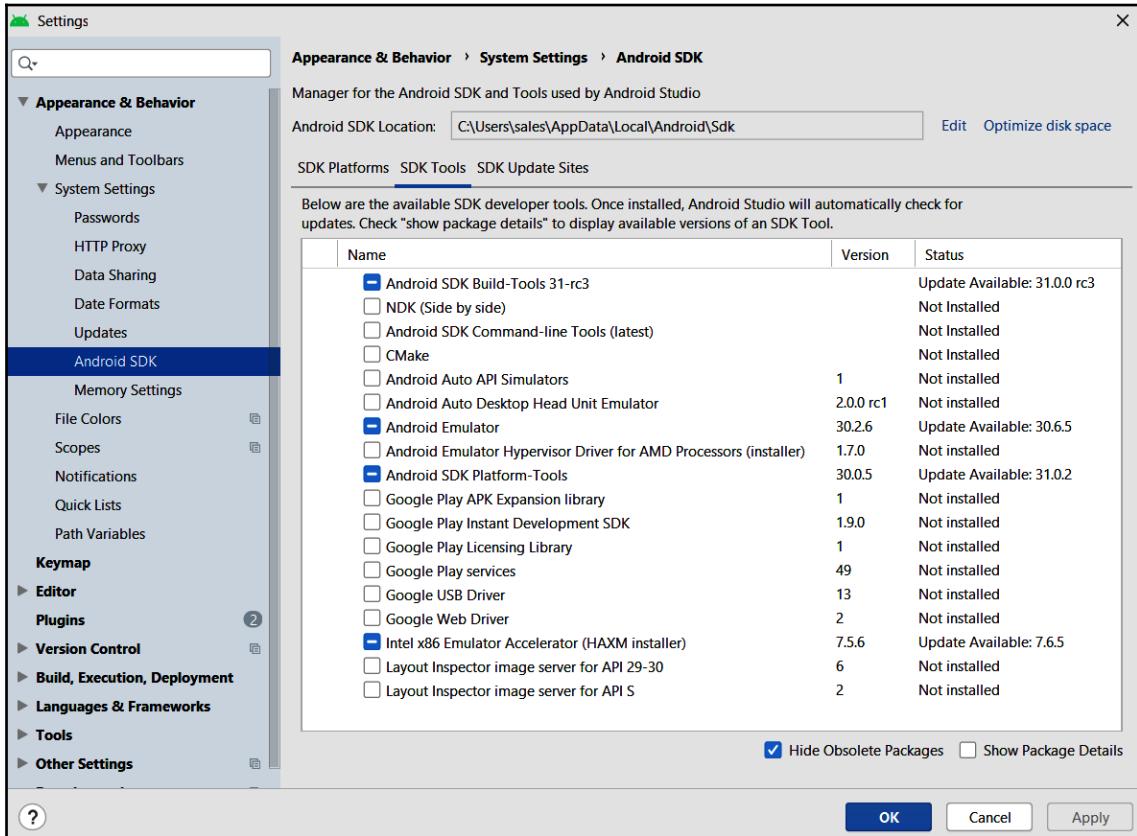


While it may be tempting to grab the most recent version of the Android SDK, you might want to choose the second most recent version, because the Flutter SDK is sometimes a bit behind Android. In most cases, it shouldn't matter, but Android is notorious for breaking compatibility, so be aware of this.

If you ever need to change your version of the Android SDK, you can always uninstall and reinstall it from this screen.

3. You will also need to download the latest build tools, emulator, SDK platform tools, SDK tools, the **Hardware Accelerated Execution Manager (HAXM)** installer, and the support library.

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



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, it is recommended that you try to use that device for development as much as possible.

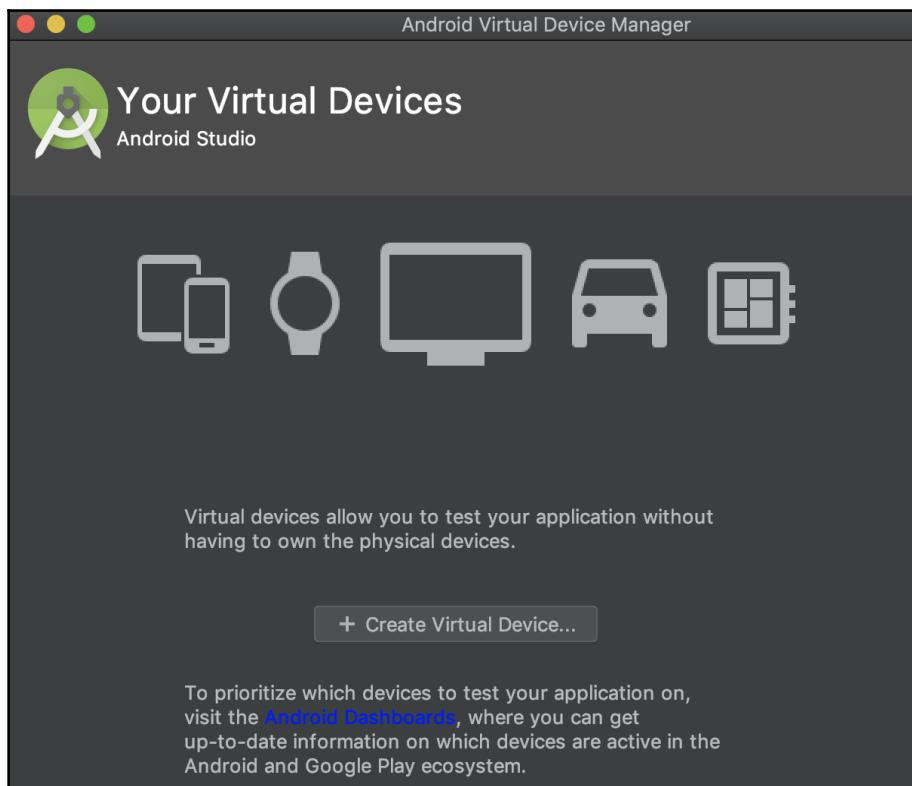
However, there are advantages to using an Android emulator (and an iOS simulator). It is often simpler to have a virtual device next to your code rather than having to carry around real devices with the required cables.

Follow these steps to set up your first emulator:

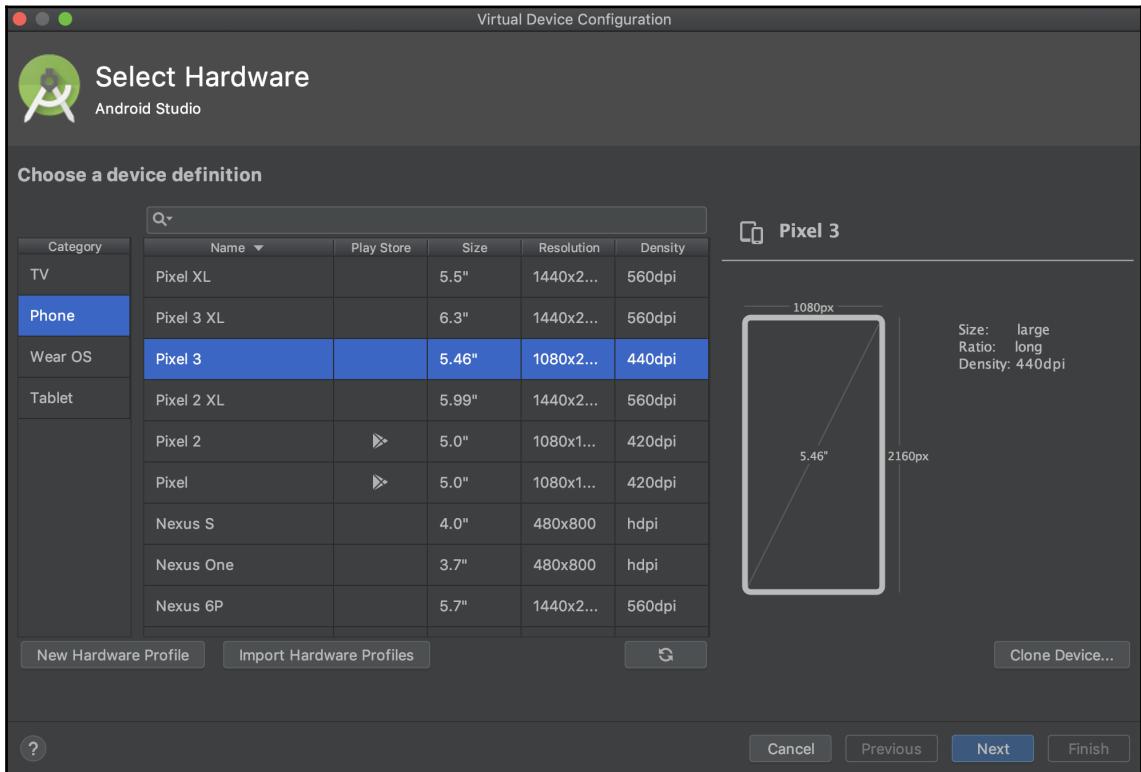
1. Select the **Android Virtual Device Manager (AVD Manager)** from the toolbar in Android Studio:



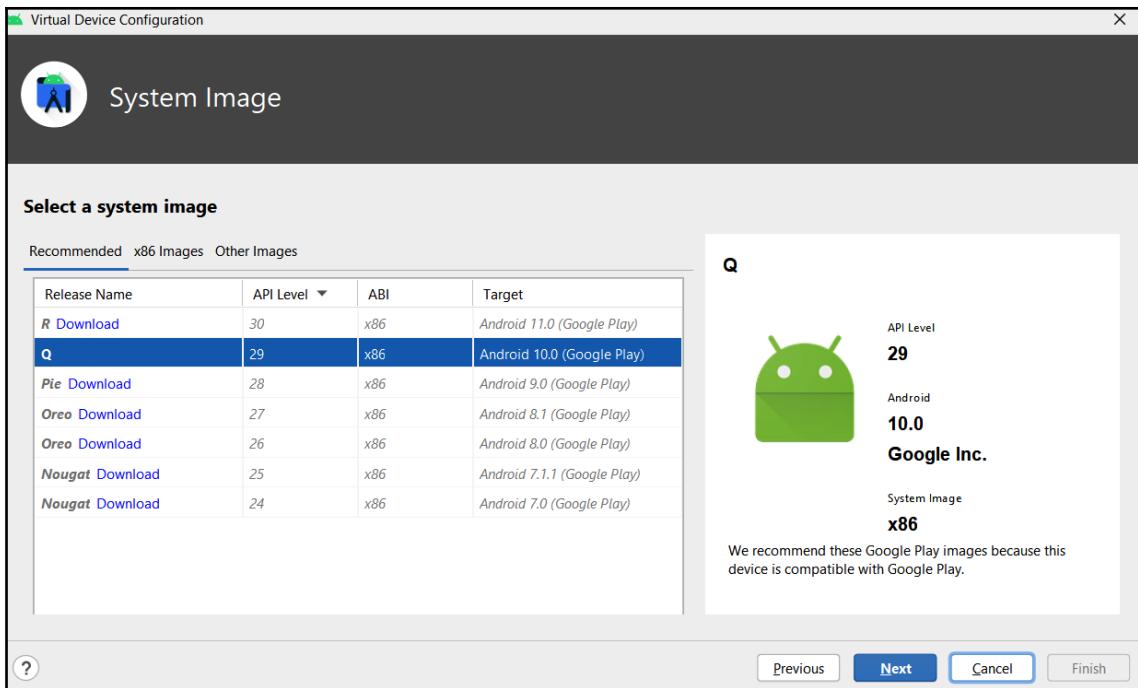
2. The first time you open the AVD Manager, you'll get a splash screen. Select the **Create Virtual Device...** button in the middle to start building your virtual device:



3. The next text screen allows you to configure which Android hardware you want to emulate. I recommend using a **Pixel** device:



4. In 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 **gigabytes (GB)** in size, so only download what you need:



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
```

Keep typing `y` when prompted to accept all the licenses (nobody really reads them, right?). 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. In the olden days, developers would wage proverbial wars over the choice between Emacs or Vim. Today, we are apparently more cool-headed (at least some of us). Ultimately, the choice is dependent on which tool you are most productive 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.

Flutter provides official plugins for three popular IDEs:

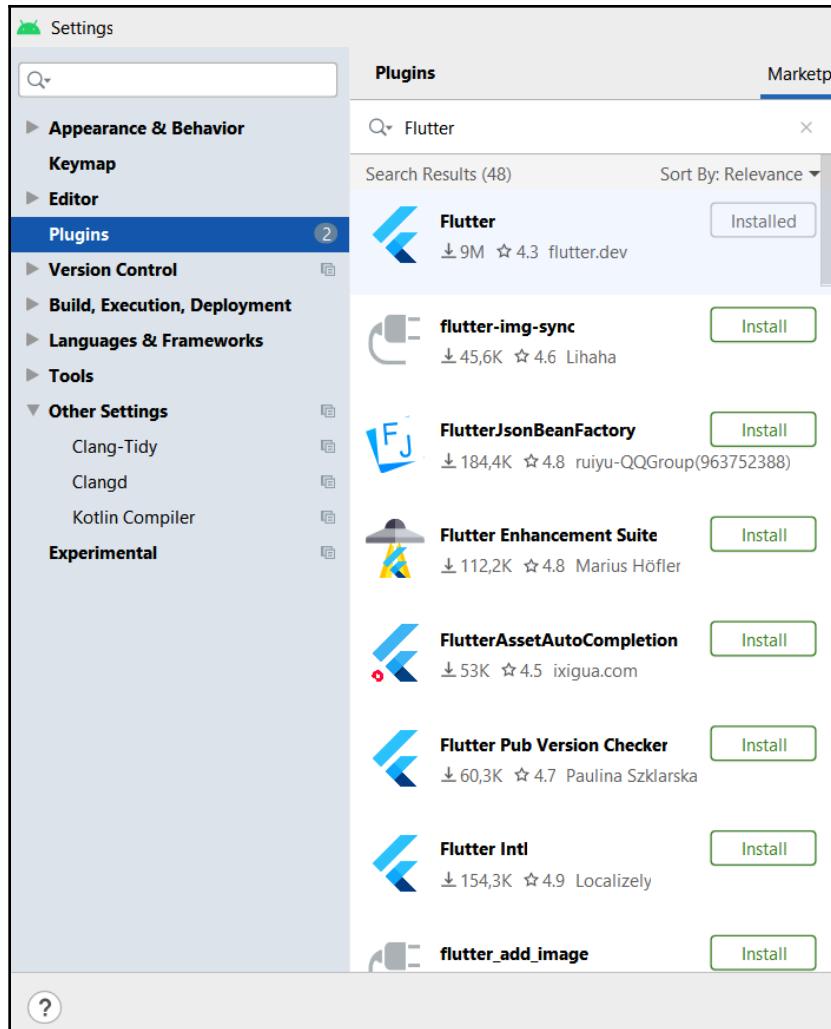
- Android Studio
- **Visual Studio Code (VS Code)**
- IntelliJ IDEA

Let's compare and configure all three 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. The biggest argument in favor of using Android Studio is that you already have it installed. In order to get the Android SDK, you have to download Android Studio.

To add the Flutter plugin, select the **Android Studio** menu, then select **Preferences**. Click on the **Plugins** tab to open the plugins marketplace. Search for Flutter and install the plugin. You will then be prompted to restart Android Studio:



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.

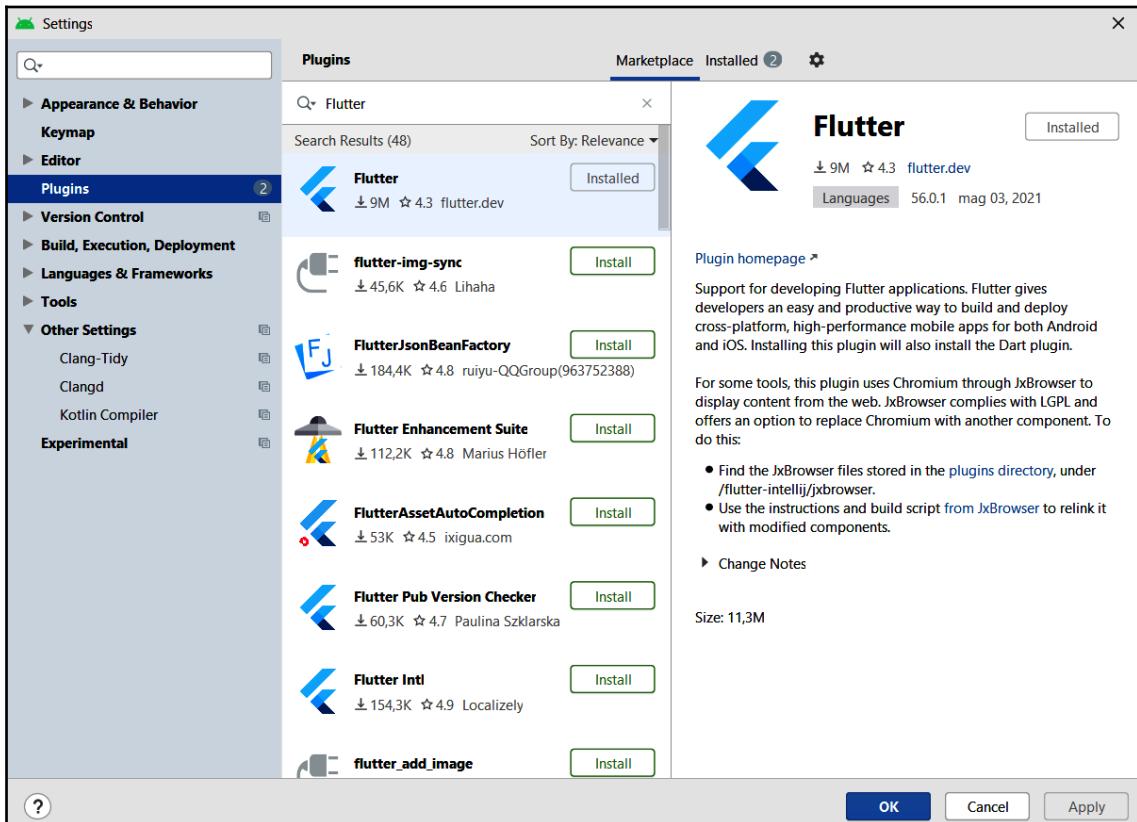
With all this power comes consequence: Android Studio is a very demanding application. On a laptop, the IDE can drain your battery very quickly, so be prepared to keep your power cable nearby. You should also make sure you have a relatively powerful computer; otherwise, you might spend more time waiting than writing code.

VS Code

VS Code is a 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>.

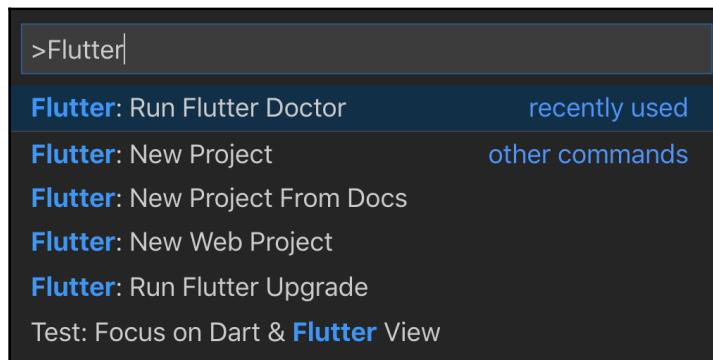
After you've installed the application, click on the fifth button in the left sidebar to open the **Extensions Marketplace**. Search for **flutter** and then install the extension:



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**.

Type *Ctrl + Shift + P* on Windows or *Shift + Command + 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:



If you want a lightweight but complete environment that you can customize to your needs, then VS Code is the right tool for you.

IntelliJ IDEA

IntelliJ IDEA is another extremely powerful and flexible IDE. You can download the tool from this website: <https://www.jetbrains.com/idea/>.

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 are going to have to use IntelliJ IDEA to get the same experience. With Flutter now supporting the web as one of its targets, this might just be enough of a reason to reach for IntelliJ IDEA over Android Studio.

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 first get promoted to the **dev** channel, then to **beta**, and finally to the **stable** channel.

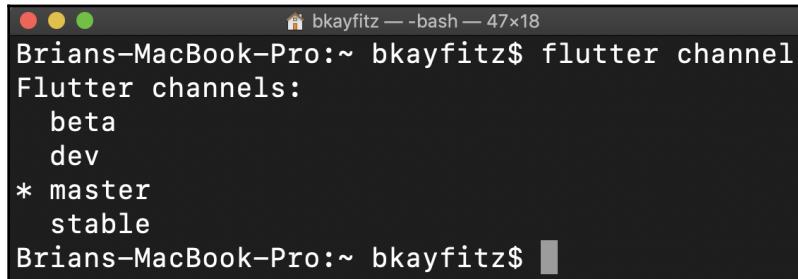
When learning Flutter, you will probably want to stick to the stable channel. This will make sure that your code should mostly run without any issues.

If you were interested in cutting-edge features that may not be completely finished, you'd probably be more interested in the master, dev, or beta channels.

In your terminal window, type in the following command:

```
flutter channel
```

You'll probably see output that looks like this:

A screenshot of a macOS terminal window titled "bkayfitz — bash — 47x18". The window shows the command "flutter channel" being run, followed by the output "Flutter channels:" and a list of four channels: "beta", "dev", "* master", and "stable". The asterisk (*) is positioned next to "master", indicating it is the current active channel.

```
Brians-MacBook-Pro:~ bkayfitz$ flutter channel
Flutter channels:
  beta
  dev
* master
  stable
Brians-MacBook-Pro:~ bkayfitz$
```

When you clone the Flutter repository, it defaults to the master channel, which is normally fine, but for training purposes, let's stick to something more reliable.

Type in these commands:

```
flutter channel stable
flutter upgrade
```

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



You may have noticed references to Git when switching channels. This is because, under the hood, Flutter channels are just fancy names for Git branches. 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` after switching channels/branches.

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 subsequent 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.

So, 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:



If you are curious about what's available for any Flutter command-line tool, 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 on how to use them.

1. Let's type this command to generate our first project:

```
flutter create hello_flutter
```

This command assumes you have an internet connection since it will automatically reach out to the public website to download the project's dependencies.

If you don't currently have an internet connection, type the following instead:

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



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.

2. Now that a project has been created, let's run it and take a look. You'll need to either connect a device to your computer or spin up an emulator. Type this command to see the emulators currently available on your computer:

```
flutter emulators
```

3. You should see a list of available emulators. You will find at least one emulator if you followed the instructions in the previous recipe. Now, type the commands outlined next to run your app.

On Windows/Linux:

```
flutter emulators --launch [your device name, like:  
Nexus_5X_API_28]  
cd hello_flutter  
flutter run
```

On a Mac:

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

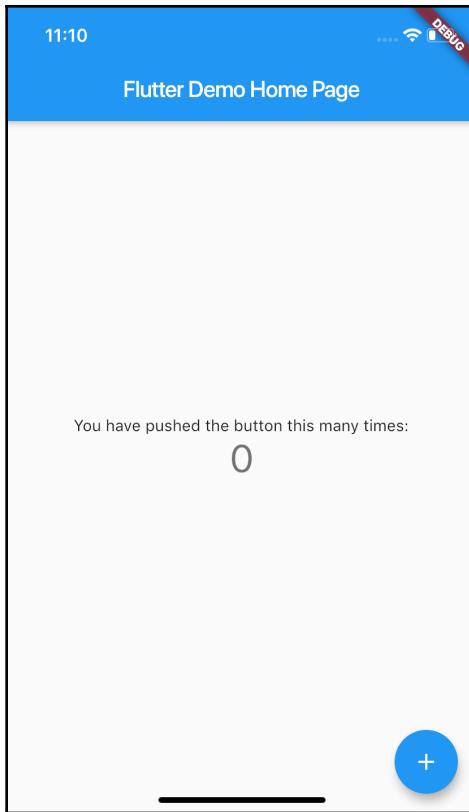
4. For physical devices, in order to see all the connected devices, run the following command:

```
flutter devices
```

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

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

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



7. Go ahead and play around with it! 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. To select these newer languages when creating an app, enter this command into your terminal:

```
flutter create \
--ios-language swift \
```

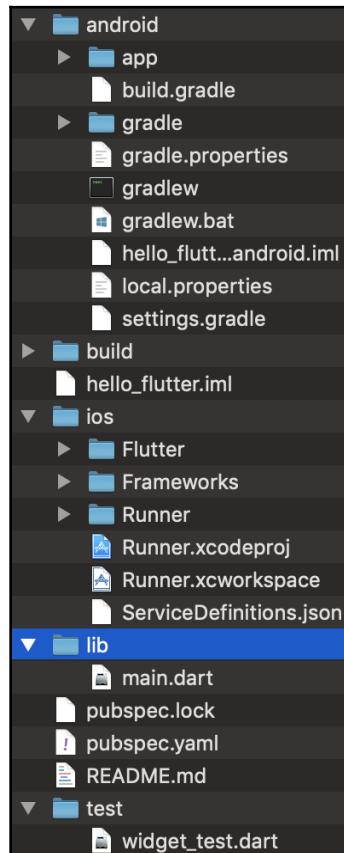
```
--android-language kotlin \
hello_modern_languages
```

Flutter will now create platform shells that use Swift and Kotlin. If you don't specify anything, 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:



The main folders in your projects are listed here:

- android
- build
- ios
- lib
- test

The `android` and `ios` 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 should be placed in these folders.

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

The `lib` folder is the heart and soul of your Flutter app. This is where you will 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 results of your `pubspec.yaml` file. It can be added to your Git repository, but it shouldn't be edited.

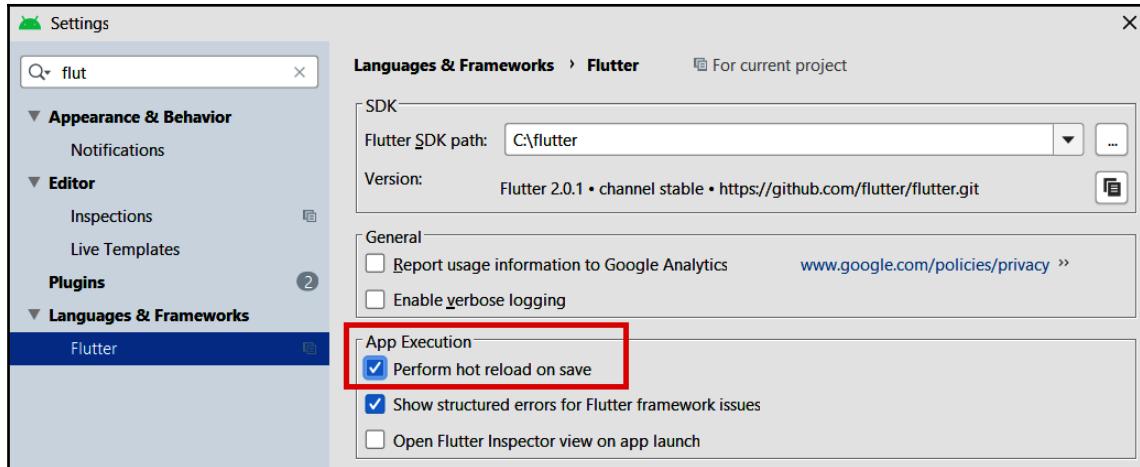
Finally, the last folder is `test`. Here, you can put your unit and widget tests, which are also just Dart code. As your app expands, automated testing will become an increasingly important technique to ensure the stability of your project. Unit testing is an advanced topic and outside the scope of this book, but you can find more information on testing in Flutter here: <https://flutter.dev/docs/testing>.

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 in the app. The time it takes to update code and see the results in an app programmed in a platform language could take several minutes. In Flutter, this edit/update cycle is down to seconds. This feature alone gives Flutter developers a competitive edge.

The best way to use Hot Reload and its cousin, Hot Restart, is through your IDE. You can configure your Flutter plugin to execute 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:



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

In **VS Code**, this setting is enabled by default. If this setting ever disappears, you can check it by opening up VS Code's **Command Palette** with *Shift + Command + P* and then typing *>Open Keyboard Shortcuts*. You can filter to Flutter-specific shortcuts by typing `flutter` in the search field:

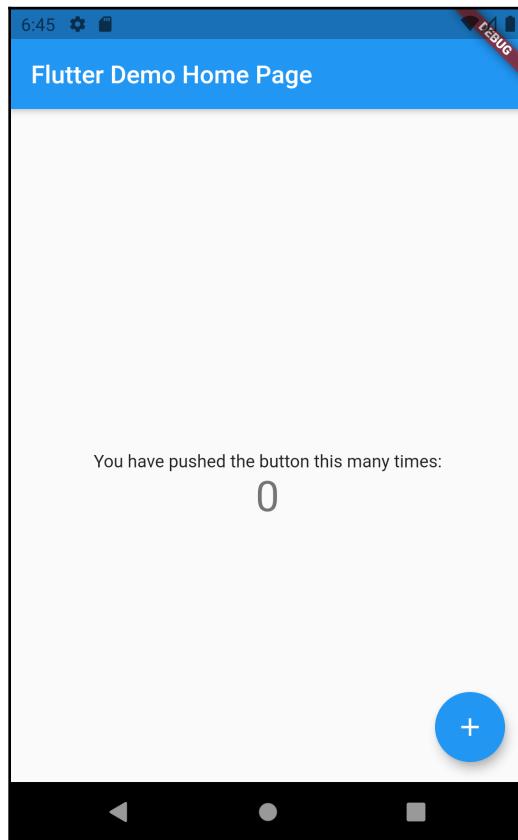
Command	Keybinding	When	Source
Flutter: Hot Reload flutter.hotReload	⌃ F5	dart-code:anyFlutterProjectLoaded && dart-code:service.reloadSources && inDebugMode	Default
Flutter: Hot Restart flutter.hotRestart		—	Default

Let's see this in action:

1. In Android Studio, 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:



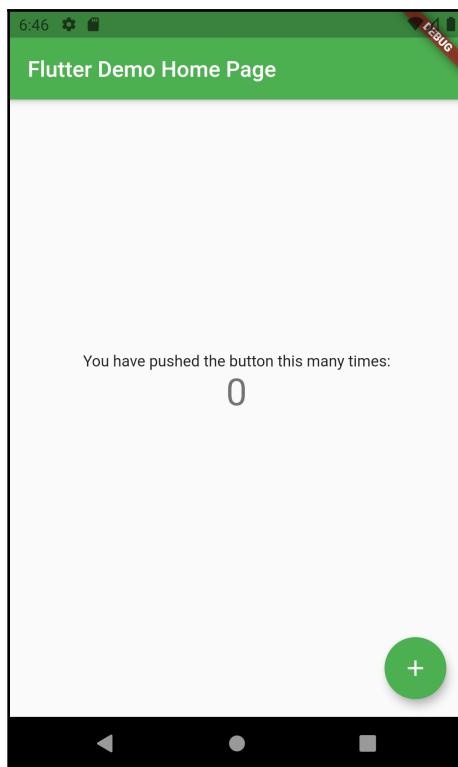
3. When the build finishes, you should see the app running in the emulator/simulator. For the best effect, adjust the windows on your computer so you can see both, side by side:



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

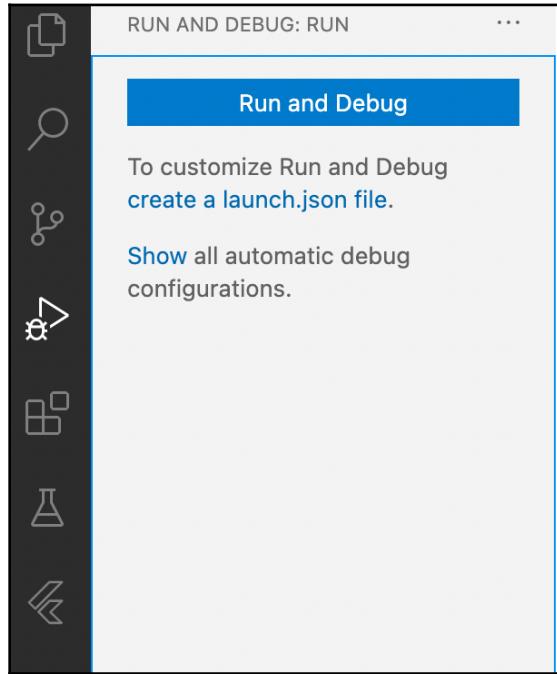
```
class MyApp extends StatelessWidget {  
  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'Flutter Demo',  
      theme: ThemeData(  
        primarySwatch: Colors.green,  
      ),  
      home: MyHomePage(title: 'Flutter Demo Home Page'),  
    );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

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



In VS Code, the pattern is very similar:

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



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

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

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



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

Summary

By now, you should have a working Flutter environment set up. It is recommended that from time to time you rerun Flutter Doctor 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 young framework that is growing fast, so you should always keep your environment up to date.

The same can be said for your iOS and Android SDKs. Mobile development is in a constant state of growth and change. Things sometimes break when they change. With the techniques we covered in this chapter and Flutter Doctor at your side, there should be no challenge you cannot overcome.

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 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>.

In the next chapter, we will go through recipes to understand the Dart programming language.

2

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 and stable programming environment. The language was created at Google in 2011, with the goal of unseating JavaScript as the language of the web.

JavaScript is a very flexible language, but its lack of a type system and misleadingly simple grammar can make projects very 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 also assumes that Dart is not your first programming language. Consequently, we will be skipping the 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.

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
- Defining abstract classes
- Implementing generics

- How to group and manipulate data with collections
- Writing less code with higher-order functions
- Using the cascade operator to implement the builder pattern
- Understanding Dart Null Safety



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.dartlang.org>): DartPad is a simple web app where you can execute Dart code. It's a great playground for trying out new ideas and sharing code.
- **IDEs**: If you wish to try out these samples locally with complete code support, then you can use either Visual Studio Code or IntelliJ.

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.

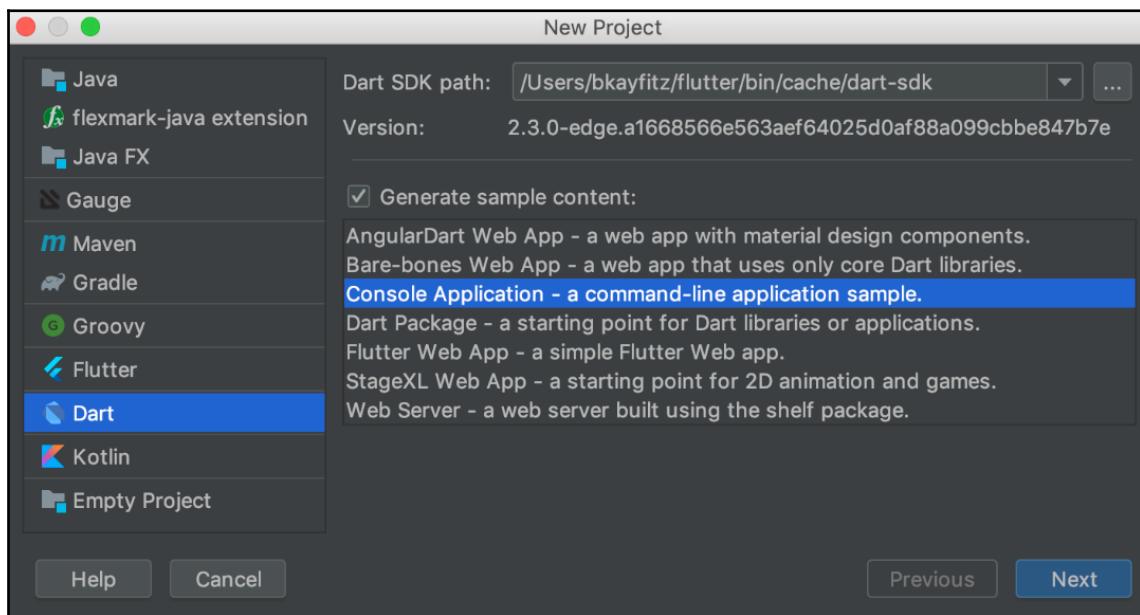
Recently, a new trend has appeared in programming that emphasizes immutability. This means that once the values are stored in a variable, that's it – they cannot change. Immutable variables are safer, produce no side effects, and lead to fewer bugs as a consequence.

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

Getting ready

Install the following before you get started with this recipe:

- DartPad:
 1. In your browser, navigate to <https://dartpad.dartlang.org>.
- Visual Studio Code:
 1. Double-check that the DartCode plugin has been installed. If you followed the steps in the previous chapter, you should be good to go.
 2. Press *Command + N* to create a new file and save it as `main.dart`.
- IntelliJ:
 1. Double-check that you have the Dart plugin installed.
 2. Select **Create new project**. The following dialog will appear, asking what language and configuration you want to use:



3. Pick Dart as your language and then select **Console Application**. This effectively runs the same commands as the command-line instructions but wraps everything in a nice GUI.



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 and DartFmt make it easier for you to type code. If you copy and paste, you'll get a working program, but you will also learn nothing.

How to do it...

Let's get started with our first Dart project. We will start from a blank canvas:

1. Open `main.dart` and delete everything. At this point, the file should be completely empty. Now, let's add the `main` function, which is the entry point for every Dart program:

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

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

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

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);
}
```

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 can optionally supply a value for the variable after the assignment operator. If you don't supply a value, that variable will be set to `null`.

4. 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:

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

5. 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, Dart will remember the type and it cannot be changed later. The values, however, are still mutable:

```
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);
}
```

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



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.

7. Add the following function to the `main.dart` file:

```
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 the variable is mutable. `var` is completely 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 its fields. `const` variables are compile-time constants and are fully immutable; nothing about these variables can be changed once they've been assigned.

Please note that you can only specify a type when you're using the `final` modifier, as follows:

```
final int numValue = 42; // this is ok
// NOT OK: const int or var int.
```

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. 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 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, 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 must also be completely sealed. Typically, 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 be faster 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:

- It's faster. When you declare a `const` value, the compiler 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.
- Immutable data does not have side effects. One of the most common sources of bugs in programming is where 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 resources:

- 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>.
- The *Dart Essentials* book from Packt covers more aspects of the Dart language and how it can be used to write web and server applications: <https://www.packtpub.com/web-development/dart-essentials>.

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 *stringed* together to create text.

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

Getting ready

To follow along with this recipe, you should write the code in DartPad or add the code to the existing project you created in the previous recipe, both in a new file or in the `main.dart` file.

How to do it...

Just like in the previous project, you are going to create a playground function where every sub-function will demonstrate a different aspect of the 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. The first section demonstrates the ways in which you can 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);  
  
    // 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 for cases where you have a text block that you want to print to the screen. The following example gets a little Shakespearean:

```
void multiLineStrings() {  
    final withEscaping = 'One Fish\nTwo Fish\nRed Fish\nBlue Fish';  
    print(withEscaping);  
  
    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 both the traditional method of concatenation, as well as a more modern method called **string interpolation**. 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 = 35;  
    final howOld = 'I am $age ${age == 1 ? 'year' : 'years'} old.';  
    print(howOld);  
}
```

5. Now, all we have to do to run this code is update `main.dart` so that it points this file to a new file. Replace the top of `main.dart` with the following code:

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

How it works...

Just like JavaScript, 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 aMoreMildOpinion = "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 actually 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.

One other interesting feature of strings in Dart is multi-line strings.

If you 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 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.

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 fully supports this method of constructing strings, the language also supports 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 = 35;  
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 35. 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`. When you run this code, you'll see the following output:

I am 35 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's guide entry to 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 functioning 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, or add the code to the existing project you created in the previous recipe, either in a new file or in the `main.dart` file.

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)}');
```

3. One of the new features that Dart has added is *optional parameters*. If you wrap your function's parameter list in square brackets, then those parameters can be omitted without the compiler throwing errors.



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

4. 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.');
```

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



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!');`.

5. Add this function right after the unnamed optional function:

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

6. 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}) {  
    String merged = '';  
    for (int i = 0; i < times; i++) {  
        merged += name;  
        if (i != times - 1) {  
            merged += ' ';  
        }  
    }  
  
    return merged;  
}
```

7. 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);  
}
```

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

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

How it works...

With Dart, you can write functions with *unnamed* (the old way), *named*, and *unnamed optional* parameters. In Flutter, *unnamed optional* parameters are the most common style you will be using, especially with widgets (more on this in the following chapters).

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: 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.

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 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. They are even written inline when consuming a function that expects a closure as a property.

Getting ready

To follow along with this recipe, you can write the code in DartPad, or add the code to the existing project you created in the previous recipe, both in a new file or in the `main.dart` file.

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

2. 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. Let's create a `typedef` called `NumberGetter`, which will be a function that returns an integer:

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

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

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

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

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

5. Finally, add an invocation to `consumeClosure` at the top of the playground method or in your main method:

```
consumeClosure();
```

How it works...

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

We showed two ways to define closures in this recipe:

- Function prototypes
- `typedefs`

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:

```
final 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:

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

Arrow functions are very helpful for removing unneeded syntax, but they should only be used for simple statements. For 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.

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. The main differences have more to do with what is missing rather than what has been added. Dart can fully support most OOP paradigms, but it can also do so without a large number of keywords. Here are a few examples of some common keywords that are generally associated with OOP that are not available in Dart:

- `private`
- `protected`
- `public`
- `struct`

- interface
- protocol

It may take a while to let go of using these, especially for longtime adherents of OOP, but you don't need any of these keywords and you can still write type-safe encapsulated, object-oriented code.

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, create a new file in your existing project or add your code in DartPad.

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', 'Francois', 'Rabelais');
    final message = name.toString();
    print(message);
}
```

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

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

How it works...

Just like functions, Dart implements the expected behavior for classical object-oriented programming.

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

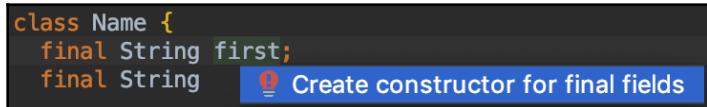
```
class OfficialName extends Name {
    ...
}
```

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);
```

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. Try deleting the constructors from the `Name` class. You should see red underlines underneath the first and last properties. Move your cursor to one of those properties (it doesn't matter which one) and press *Option + Enter*:



You should see a popup appear that generates constructions for final fields. If you hit *Enter*, your constructor will appear without you having to type anything. It's convenient.

The building blocks of OOP

Where Dart does deviate from other OOP languages, such as Java, C#, Kotlin, and Swift, is its lack of explicit keywords for interfaces and abstract classes. In Dart, objects are more defined by how they are used rather than how they are defined.

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 <code>implements</code> when you want to create your own implementation of another class, as all classes are implicit interfaces. When the <code>FullName</code> class implements the <code>Name</code> class, all the functions that were defined in the <code>Name</code> class must be implemented. This means that when you implement a class, you do not inherit any code , just the type. Classes can implement any number of interfaces, but be reasonable and don't make 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.



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 also

This recipe touched on a bunch of topics that warrant more detail:

- *Design Patterns: Elements of Reusable Object-Oriented Software*: Affectionately referred to as the "Gang of Four." This is the quintessential book about design patterns: <https://www.pearson.com/us/higher-education/program/Gamma-Design-Patterns-Elements-of-Reusable-Object-Oriented-Software/PGM14333.html>.
- *Mastering Dart*: https://subscription.packtpub.com/book/web_development/9781783989560.

How to group and manipulate data with collections

All programming languages possess some mechanism to organize data. We've already covered the most common way – objects. These class-based structures allow you, the programmer, 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`.

- Lists are linear collections where the order of the elements is maintained.
- Maps are a non-linear collection of values that can be accessed by a unique key.
- 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

Getting ready

Create a new file in your project or type this code in Dartpad.

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(10);  
    numbers.addAll([4, 1, 35]);  
  
    // Assigning via subscript  
    numbers[1] = 15;  
  
    print('The second number is ${numbers[1]}');  
  
    // enumerating a list  
    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 list. Add this function to see Map in action:

```
void mapPlayground() {
    // Map Literal syntax
    final MapString, int ages = {
        'Mike': 18,
        'Peter': 35,
        'Jennifer': 26,
    };

    // Subscript syntax uses the key type.
    // A String in this case
    ages['Tom'] = 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 = false;
    final randomNumbers = [
        34,
        232,
        54,
        32,
        if (addMore) ...[
            534343,
            4423,
            3432432,
        ],
    ];
}

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

print(duplicated);
}
```

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:

- Does the order matter? **Choose a List.**
- Should all the elements be unique? **Choose a Set.**
- Do you need to access elements from a dataset quickly? **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 speed of a computer algorithm.

Subscript syntax

One thing these collections have in common is subscript syntax. Subscripts are a way to quickly access elements in a collection, and they tend to work identically from language to language:

```
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 with strings instead of integers:

```
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 gracefully fail and 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 duplicated = [
    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) ...[
    534343,
    4423,
    3432432,
  ],
];
```

By putting the three dots before the sublist, Dart will *unbox* the second list and flatten all these numbers into a single 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:

- Collection library: <https://api.dartlang.org/stable/2.4.0/dart-collection/dart-collection-library.html>

- Big-O notation: https://en.wikipedia.org/wiki/Big_O_notation
- Article on Dart 2.3: <https://medium.com/dartlang/announcing-dart-2-3-optimized-for-building-user-interfaces-e84919ca1dff>

Writing less code with higher-order functions

If there was a different name we could give programmers, it would be Data Massager. Essentially, that is all we do. 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 most of your time writing CRUD code. It doesn't matter if you are working with 3D graphics or training machine learning models – CRUD will consume the majority of your life.

Being able to manipulate mass quantities of data quickly, your standard control flows, along with your repertoire of `do`, `while`, and `for` loops isn't going to cut it. Instead, we should use higher-order functions, one of the primary aspects of functional programming, to help us get to the fun stuff faster.

Getting ready

Create a new file in your project or type this code in Dartpad.

How to do it...

Higher-order functions can be divided into categories. This recipe will explore all their different facets. 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:

```
import 'package:introduction_to_dart/04-classes.dart';

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.



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.

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},
];
```

3. 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';
    }
}
```

4. The first higher-order function is `map`. Its purpose is taking data in one format and quickly outputting it in 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 names:

```
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();

  return names;
}
```

5. 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();

  // Alphabetize the list by last name
  names.sort((a, b) => a.last.compareTo(b.last));

  print('');
  print('Alphabetical List of Names');
  names.forEach(print);
}
```

6. 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);
}
```

7. 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');
}
```

8. 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 exact same result with `for` loops, but you will 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 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.

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**. Pure functions are supposed to simply return **new** data, so this one is an exception.

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

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 succinctly 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.

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 higher-order functions are mostly abstractions 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 is helpful for non-numeric types such as strings or if you do not want your code to start reducing from 0:

```
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 destructure 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!

See also

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

- Iterable documentation: <https://api.dart.dev/stable/2.4.0/dart-core/Iterable-class.html>
- *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, we've covered how Dart follows many of the same patterns of other modern languages. Dart, in some ways, is the culmination of the best ideas from multiple languages – you have the expressiveness of JavaScript and the type safety of Java. Dart can be interpreted as JIT, but it can also be compiled.

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**. Builders are a type of class whose only purpose is to build other classes. They are often used for 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.

Create a new file in your project or 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() {
        assert(scheme != null);
        assert(host != null);
        final paths = [host, if (routes != null) ...routes];
        final path = paths.join('/');

        return '$scheme://$path';
    }
}
```

2. Next, you are going to 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 = [342, 23423, 53, 232, 534]  
    ..insert(0, 10)  
    ..sort((a, b) => a.compareTo(b));  
print('The largest number in the list is ${numbers.last}');
```

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 pick apart 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 resources will help you to understand cascades and the builder pattern:

- **Builder pattern:** https://en.wikipedia.org/wiki/Builder_pattern
- **Method cascades in Dart:** <https://news.dartlang.org/2012/02/method-cascades-in-dart-posted-by-gilad.html>

Understanding Dart Null Safety

When Dart 2.12 was shipped in Flutter 2 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 the **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.

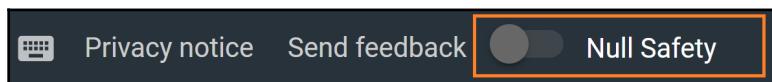
Getting ready

Create a new pad in Dartpad. This is required to easily turn null safety on and off.

How to do it...

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

1. In DartPad, make sure **Null Safety** is **disabled**. You can toggle **Null Safety** with the control at the bottom of the screen:



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

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

3. 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);  
}
```

4. Run your code. You should see a null error in the console, as shown in the following screenshot:

The screenshot shows the Dart Editor interface. On the left, there is a code editor with the following Dart code:

```
1 void main() {  
2   int someNumber;  
3   increaseValue(someNumber);  
4 }  
5  
6  
7 void increaseValue(int value) {  
8   value++;  
9   print (value);  
10 }
```

Next to the code editor is a blue "RUN" button. To the right of the code editor is a "Console" window displaying the output of the run:

```
Console  
Uncaught TypeError: Cannot read property '$add' of nullError: TypeError: Cannot read property '$add' of null
```

Below the code editor, there is a "Documentation" section for the variable `someNumber`:

`int someNumber`
local variable

At the bottom of the screen, there are several buttons and status indicators:

- A keyboard icon followed by "Privacy notice".
- A "Send feedback" link.
- A toggle switch labeled "Null Safety" which is currently off (gray).
- Status indicators: "no issues" and "Based on Flutter 2.2.1 Dart SDK 2.13.1".

5. Enable **Null Safety** with the switch at the bottom of the screen, and note that `someNumber` at line 3 raises a compile error before execution on `someNumber`: "**The non-nullable local variable 'someNumber' must be assigned before it can be used.**"
6. Add a question mark after the two `int` declarations:

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

7. Note that the error has changed to: "**The method '+' can't be unconditionally invoked because the receiver can be 'null'.**"
8. 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);  
}
```

```
    } else {
        value = 1;
    }
    print (value);
}
```

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

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

11. Run the app, and note that in the console you still find the value 1.
12. 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);
}
```

13. Run the app, and note that you get an execution null exception.
14. 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);
}
```

15. 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.



At the time of writing, not all parts of the Flutter SDK are null safe yet. Some packages are also null safe.

You can still implement null safety in your apps while using *null unsafe* packages.

In the first code snippet, which you run without null safety, the code raised a runtime error at the following instruction:

```
value++
```

This is because you cannot increment a `null` value.

Simply put, when you enable 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; //this is ok
int someOtherNumber = null; //compile error
```

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 of the 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 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 (`int value`). The exclamation mark (`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. The main issue with this new (at the time of writing) feature is that not all libraries have been migrated to null safety yet, so you might still get unexpected null values in your code when you use them. Once this transition period is over though, we might expect Flutter apps to be more solid and secure.

See also

A thorough and complete resource to understanding null safety in Dart and Flutter is available at <https://flutter.dev/docs/null-safety>.

3

Introduction to Widgets

It's time to finally get our hands dirty with Flutter! By now, you should have your environment set up and have a pretty good handle on Dart, so there shouldn't be any surprises.

In this chapter, we will be building the layout for static elements in Flutter while showing how to build *widget trees*. Everything in Flutter should be thought of as living in a tree structure.

Every widget in Flutter is 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 chain them together to create expressive interfaces.

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_03.

Creating immutable widgets

A stateless widget is the primary building block for creating user interfaces. This widget is simple, lightweight, and performant. Flutter can render hundreds of stateless widgets without breaking sweat.

Stateless widgets are immutable. Once they are created and drawn, 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:

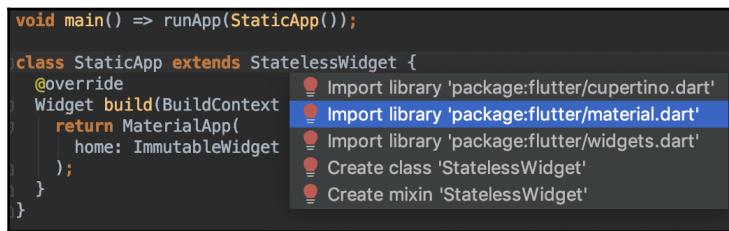
1. Open `main.dart` and delete everything! Then, type the following code into the editor:

```
void main() => runApp(StaticApp());  
  
class StaticApp extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: ImmutableWidget(),  
        );  
    }  
}
```

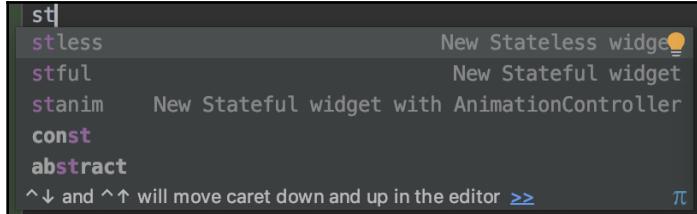
2. Notice that you have a bunch of red underlines. We need to fix this by importing the `material.dart` library. This can be done manually, but it's more fun to let your IDE do that job. Move your cursor over the word `StatelessWidget`.
3. In VS Code, press `Ctrl + .`, or `Command + .` on a Mac. In Android 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.

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.

4. Select `material.dart` and most of the errors will go away:



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:



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 {
  @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: 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';
```

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:



Congratulations! You have now created your first 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(StaticApp());
```

This line initializes the Flutter framework and places a `StaticApp`, which is just another stateless widget, at the root of the tree.

Our root class, `StaticApp`, **is just 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` and the custom `ImmutableWidget`.

One phrase that you frequently see in the official Flutter documentation is, "*It's all widgets, all the way down.*" This phrase implies two things:

- Every 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 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*.



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>.

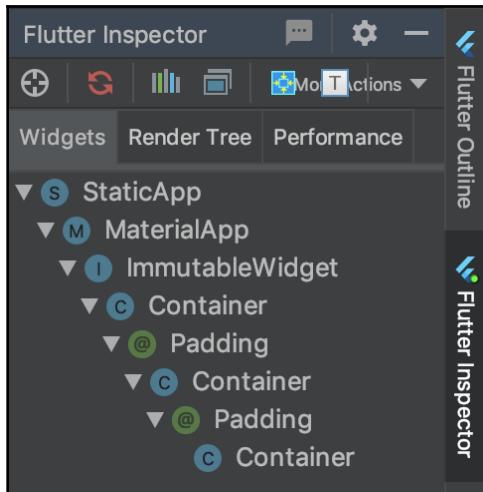
You can see and explore the widget tree using one of the most useful features of the Flutter DevTools from your IDE. To open the inspector while your app is running, in VS Code, perform the following steps:

- Open the command palette (*Ctrl + Shift + P*, or *Cmd + Shift + P* on a Mac).
- Select the **Flutter: Open DevTools Widget Inspector Page** command.

In Android Studio/IntelliJ, perform the following steps:

- Click on the **Flutter Inspector** tab on the right of your screen.

Here you can see an image of the Flutter widget inspector:



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: ImmutableWidget(),
    );
}
```

This first build method returns a `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 creates several helper properties, such as navigation, theming, and localization. You can also use a `CupertinoApp` if you want to follow Apple's design language, or a `WidgetsApp` if you want to create your own. We will typically use material apps at the root of the Flutter tree in this book.



There are some widgets that use different property names to describe their child. `MaterialApp` uses `home`, and `Scaffold` uses `body`. Even though these are named differently, they are still 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: 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, and which contains the last `Container` widget (blue) of this tree.

A container is similar to a `div` in HTML. It's rendered as a box that has many styling options. The three `Container` widgets are separated by two paddings of slightly different sizes.

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.

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 and Cupertino, provide a set of widgets that mirror the user experience of their respective platforms.

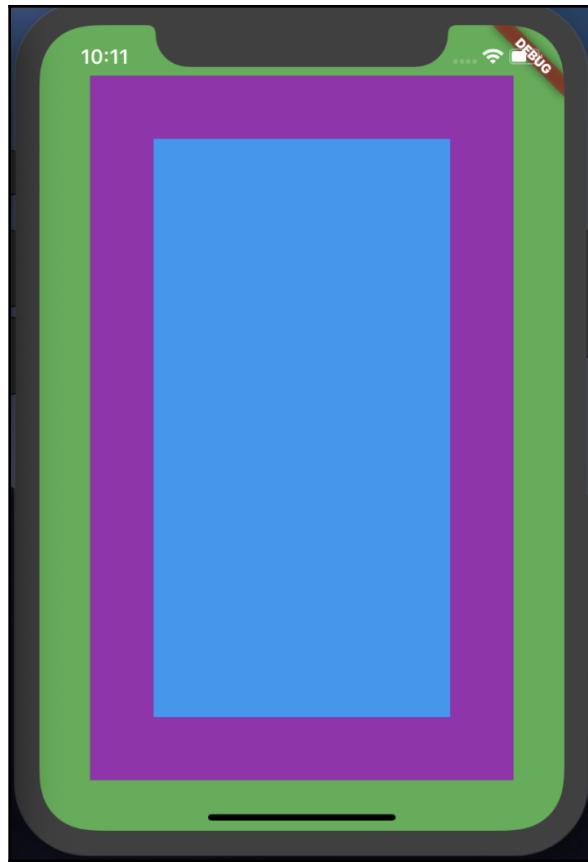
These frameworks use a widget called `Scaffold` (`CupertinoScaffold` in the `Cupertino` framework) that provides a basic structure of a screen.

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`. Make sure that the app is running while you are making these code changes. You could also adjust the size of your IDE so that the iOS simulator or Android emulator can fit beside it:



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 library as well:

```
import 'package:flutter/material.dart';

class BasicScreen extends StatelessWidget {
  @override
```

```
Widget build(BuildContext context) {  
    return Container();  
}  
}
```

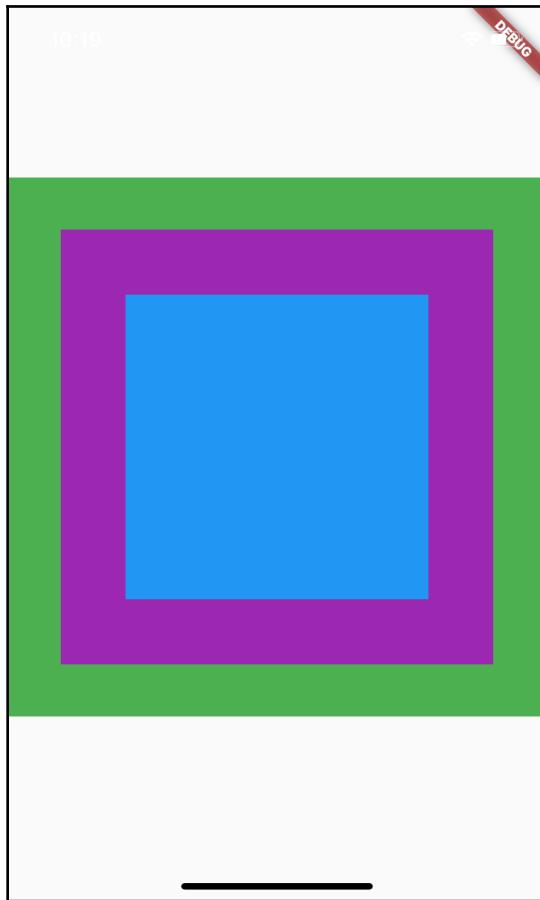
2. Now, in `main.dart`, replace `ImmutableWidget` with `BasicScreen`. Hit the save button to hot reload and your simulator screen should be completely white:

```
import 'package:flutter/material.dart';  
import './basic_screen.dart';  
  
void main() => runApp(StaticApp());  
  
class StaticApp extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: BasicScreen(),  
        );  
    }  
}
```

3. Now it's time to bring in the scaffold. In `basic_screen.dart`, we're going to add the widget that was created in the previous recipe, but bring it under control somewhat with the `AspectRatio` and `Center` widgets:

```
import 'package:flutter/material.dart';  
import './immutable_widget.dart';  
  
class BasicScreen extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return Scaffold(  
            body: Center(  
                child: AspectRatio(  
                    aspectRatio: 1.0,  
                    child: ImmutableWidget(),  
                ),  
            ),  
        );  
    }  
}
```

The screen should now look like this:

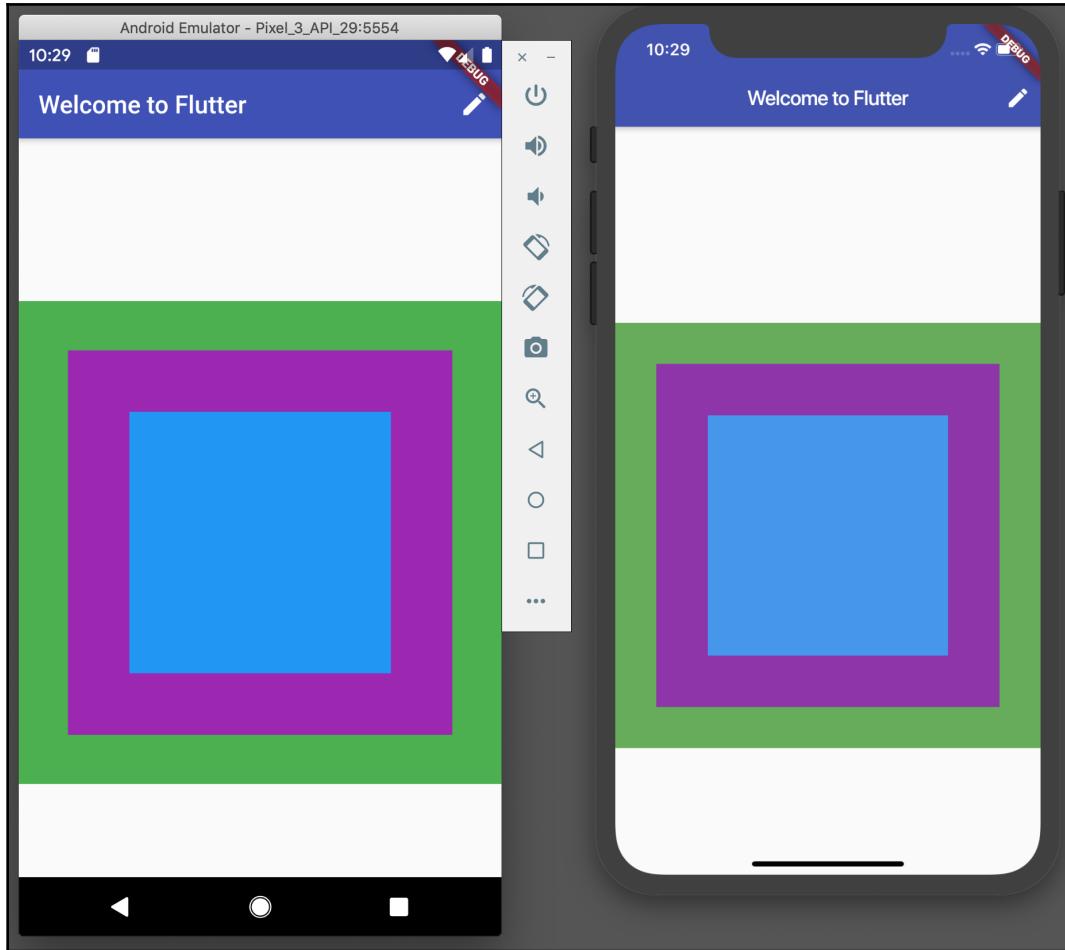


4. Probably one of the most popular widgets in an app is `AppBar`. This is a persistent header that lives at the top of the screen and helps you navigate from one screen to the next. Add the following code to the scaffold:

```
return Scaffold(  
    appBar: AppBar(  
        backgroundColor: Colors.indigo,  
        title: Text('Welcome to Flutter'),  
        actions: <Widget>[  
            Padding(  
                padding: const EdgeInsets.all(10.0),  
                child: Icon(Icons.edit),  
            ),
```

```
    1,  
    ),  
    body: Center(  
    ...  
)
```

5. Hot reload the app and you will now see an app bar at the top of the screen:

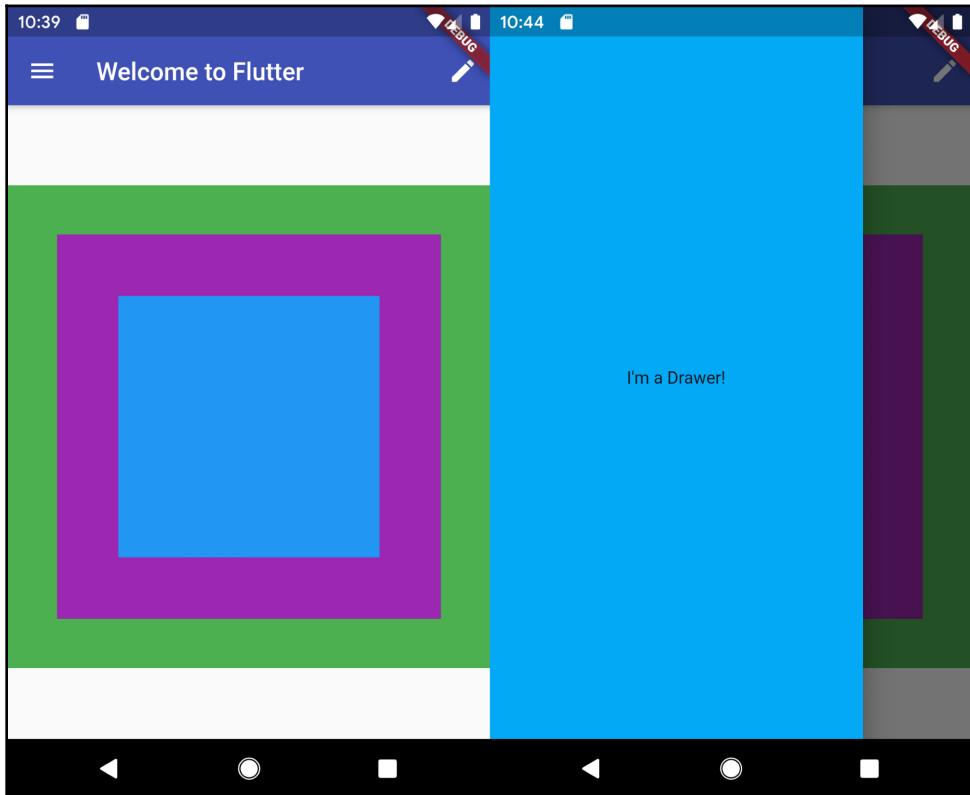


6. Finally, let's add a drawer to the app. Add this code to `Scaffold`, just after `body`:

```
body: Center(  
    child: AspectRatio(  
        aspectRatio: 1.0,  
        child: ImmutableView(),
```

```
) ,  
) ,  
drawer: Drawer(  
    child: Container(  
        color: Colors.lightBlue,  
        child: Center(  
            child: Text("I'm a Drawer!"),  
        ),  
    ),  
) ,  
) ,
```

The final app should now have a hamburger icon in AppBar. If you press it, the drawer will be shown:



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, but it is not 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 metrics. `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 other tools in a scaffold that we will cover in the next chapters.



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 aspect ratio 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, but 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 shiny ball:

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 method for the shiny circle. You will be adding `BoxDecoration` to a `Container`, which can include gradients, shapes, shadows, borders, and even images.



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.

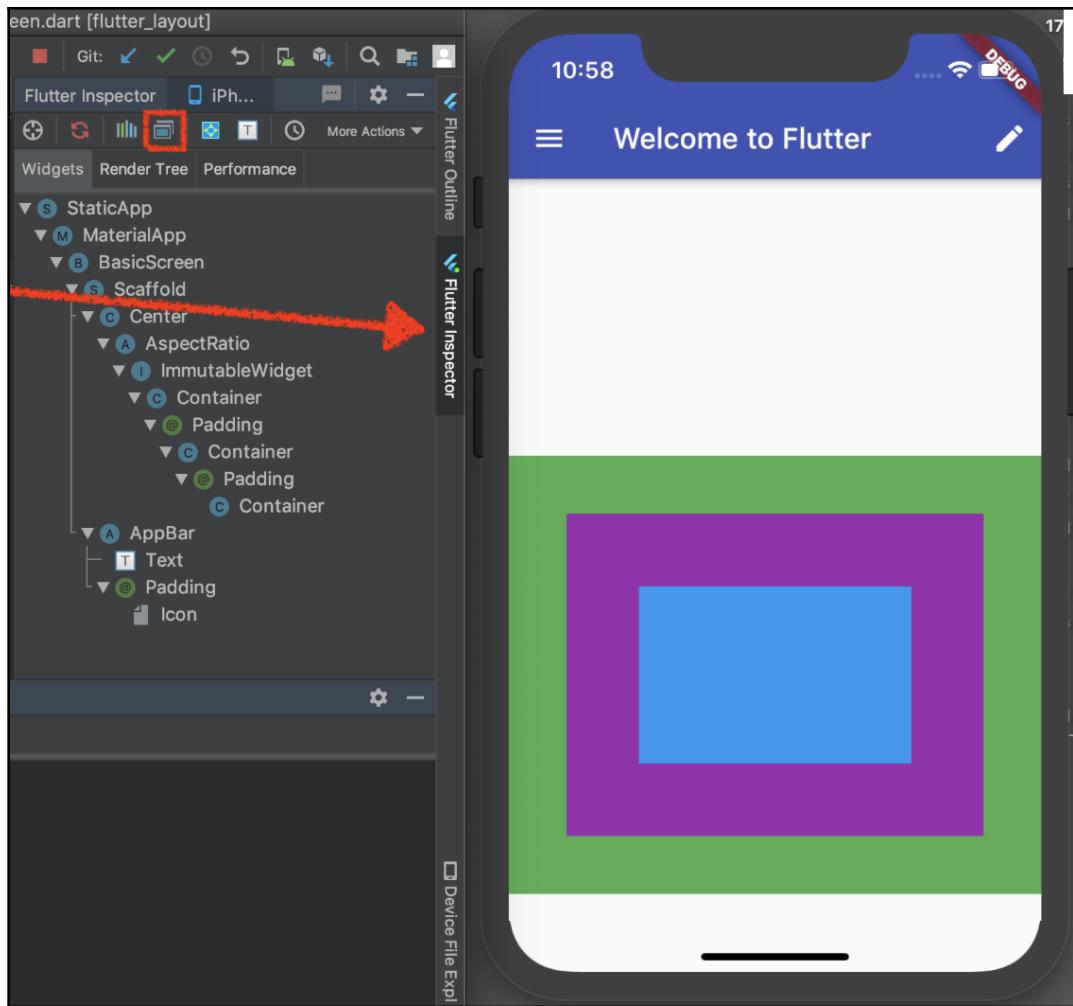
3. Add the following code at the end of the `ImmutableWidget` class:

```
Widget _buildShinyCircle() {
    return Container(
        decoration: BoxDecoration(
            shape: BoxShape.circle,
            gradient: RadialGradient(
                colors: [
                    Colors.lightBlueAccent,
                    Colors.blueAccent,
                ],
                center: Alignment(-0.3, -0.5),
            ),
            boxShadow: [
                BoxShadow(blurRadius: 20),
            ],
        ),
    );
}
```

4. 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.
5. Before we begin, we're going to require access to some math functions.
6. 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;
```

7. Now, update the second container with this decoration 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:



8. 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:

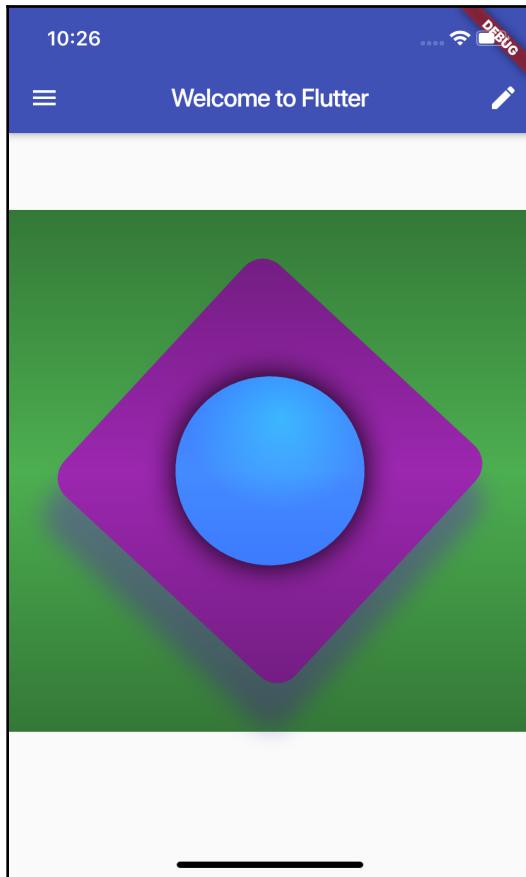
```
return Container(
  color: Colors.green,
  child: Center(
    child: Transform.rotate(
      angle: 180 / Math.pi, // Rotations are supplied in radians
      child: Container(
        width: 250,
        height: 250,
```

```
        decoration: BoxDecoration(
            color: Colors.purple,
            boxShadow: [
                BoxShadow(
                    color: Colors.deepPurple.withOpacity(120),
                    spreadRadius: 4,
                    blurRadius: 15,
                    offset: Offset.fromDirection(1.0, 30),
                ),
                ],
            borderRadius: BorderRadius.all(Radius.circular(20)),
        ),
        child: Padding(
            padding: const EdgeInsets.all(50.0),
            child: _buildShinyCircle(),
        ),
    ),
),
),
),
),
),
);
;
```

9. You will now add some style to the top widget. Containers can actually support two decorations: foreground and background decoration. The two decorations can even be blended together to create interesting effects. Add this code to the root container in `ImmutableWidget`:

```
return Container(
    decoration: BoxDecoration(color: Colors.green),
    foregroundDecoration: 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 like the following:



How it works...

Container widgets can add all manner of effects to their child. Like scaffolds, they enable several customizations that can be explored and experimented with.

The primary property you will be designing with is `BoxDecoration`, which can draw the following:

- 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 to add 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 exactly the same result, so it's up to you really.

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.



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 instructions. 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 and you will rarely have to edit them or even acknowledge their existence.

Printing stylish text on the screen

Almost every app has to display text at some point. Even when we were experimenting with the Dart language in the previous chapter, all those recipes did was display 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 {
  @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 ellipted:

```
import 'package:flutter/material.dart';
import 'package:flutter_layout/imutable_widget.dart';
import 'package:flutter_layout/text_layout.dart';

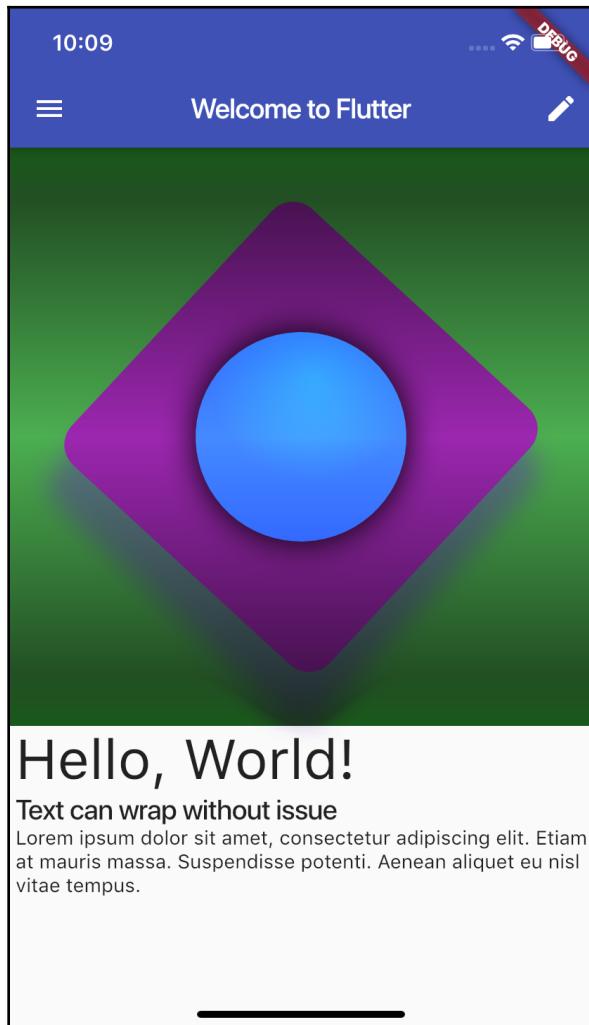
class BasicScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(...),
      body: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          AspectRatio(
```

```
        aspectRatio: 1.0,
        child: ImmutableWidget(),
    ),
    TextLayout()
],
),
drawer: Drawer(...),
);
}
}
```

3. Now, moving back to the `text_layout.dart` file, let's add a column containing three `Text` widgets:

```
class TextLayout extends StatelessWidget {
@override
Widget build(BuildContext context) {
return Column(
crossAxisAlignment: CrossAxisAlignment.start,
children: <Widget>[
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:
Text(
'Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Etiam at mauris massa. Suspendisse potenti.
Aenean aliquet eu nisl vitae tempus.'),
],
);
}
}
```

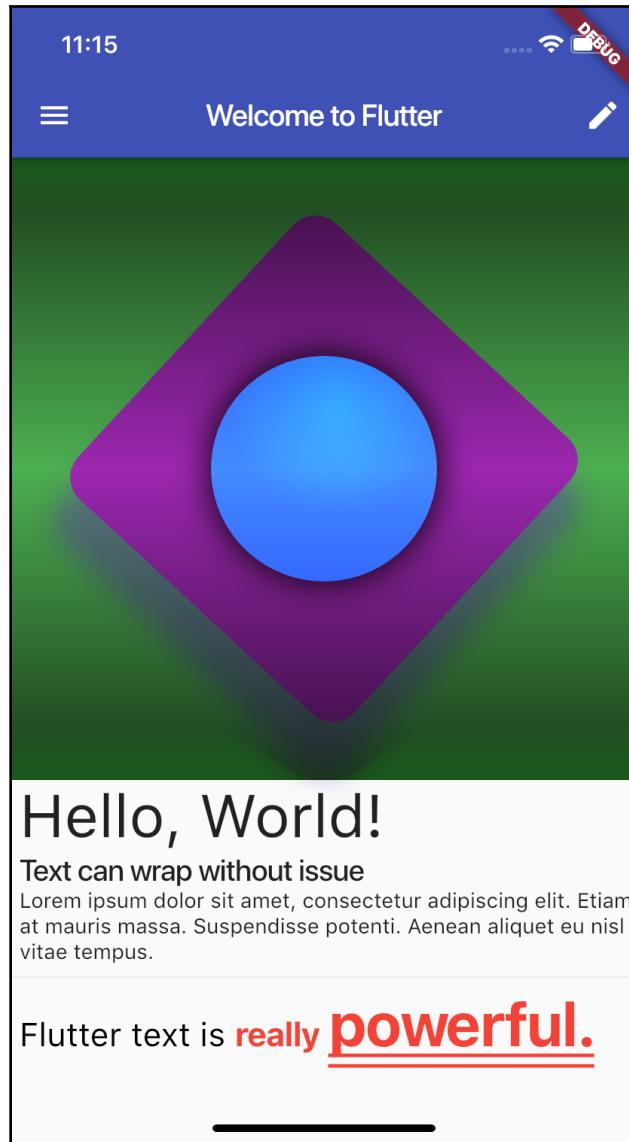
4. When you run the app, it should look like this:



5. 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:

```
Divider(),
RichText(
    text: 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:



How it works...

Hopefully, most of the code for the `Text` widget should be self-evident. It's just a matter of creating hundreds and 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 known as "*of-context*," which 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://flutter.dev/docs/cookbook/design/themes>.

Importing fonts and images into your app

Text and colors are nice, but pictures are worth a thousand words. The process of adding custom images and fonts to your app is a little more complex than you might be expecting. 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 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 to the screen*, before following along with this one.

You will add an image to the app. You can get some gorgeous 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 under `flutter`, as shown here:

```
dependencies:  
  flutter:  
    sdk: flutter  
  google_fonts: ^2.0.0
```

3. After this is done, run `flutter packages get` to rebuild your asset bundle.
4. 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';
```

5. Update the first `Text` widget to reference the `leckerliOne` font:

```
Text(  
  'Hello, World!',  
  style: GoogleFonts.leckerliOne(fontSize: 40),  
) ,
```

The `leckerliOne` font will now be rendered on the screen:



6. Now you will add a picture to the screen. At the root of your project, create a new folder, called `assets`.
7. 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.
8. 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/
```

9. In `basic_screen.dart`, replace `ImmutableWidget` with this code:

```
body: Column(  
    mainAxisAlignment: MainAxisAlignment.start,  
    children: <Widget>[  
        Image.asset('assets/beach.jpg'),  
        TextLayout(),  
    ],  
) ,
```

The final layout should show the image at the top of 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.

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, as you did with the following command:

```
google_fonts: ^2.0.0
```



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: 16),
```

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.

See also

Flutter Assets Guide: <https://flutter.dev/docs/development/ui/assets-and-images>.

4

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)** layout is a tree. 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 pruned.

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 simple. As Flutter is a relatively 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 simple to use.

In this recipe, we're going to create a prototype for a profile screen for dogs.

Getting ready

Create a new Flutter project with your favorite editor or **integrated development environment (IDE)**.

You're going to need a picture for the profile image. You can choose any image you want, but you can also download this image of a dog from *Unsplash*: <https://unsplash.com/photos/k1LNp6dnyAE>.

Rename the file as `dog.jpg` and add it to the `assets` folder (create `assets` in the root folder of your project).

We will also reuse the beach picture that we used in the previous chapter: you can download this at <https://unsplash.com/photos/nXOB-wh4Oyc>. Rename it as `beach.jpg`.

Don't forget to add the `assets` folder to your `pubspec.yaml` file:

```
assets:  
  - assets/
```

You are also going to need a new file for this page. Create a `profile_screen.dart` file in the `lib` folder.

Don't forget to have the app running while inputting this 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 {
    @override
    Widget build(BuildContext context) {
        return Container();
    }
}
```

3. In the `main.dart` file, remove the `MyHomePage` class, and use the new `ProfileScreen` class as the home of `MyApp`:

```
class MyApp extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            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 {
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            body: Column(
                children: <Widget>[
                    _buildProfileImage(context),
                    _buildProfileDetails(context),
                    _buildActions(context),
                ],
            ),
        );
    }
}
```

```
Widget _buildProfileImage(BuildContext context) {  
    return Container();  
}  
Widget _buildProfileDetails(BuildContext context) {  
    return Container();  
}  
Widget _buildActions(BuildContext context) {  
    return Container();  
}  
}  
}
```

5. Now, update the `_buildProfileImage` method to actually show the image of the dog:

```
Widget _buildProfileImage(BuildContext context) {  
    return Container(  
        width: 200,  
        height: 200,  
        child: ClipOval(  
            child: Image.asset(  
                'assets/dog.jpg',  
                fit: BoxFit.fitWidth,  
            ),  
        ),  
    );  
}
```

6. The next section will add a `Column` widget to describe some of the dog's best features. Replace the `_buildProfileDetails()` method with this code. This code also includes the `Column` widget's horizontal sibling, `Row`:

```
Widget _buildProfileDetails(BuildContext context) {  
    return Padding(  
        padding: const EdgeInsets.all(20.0),  
        child: Column(  
            crossAxisAlignment: CrossAxisAlignment.start,  
            children: <Widget>[  
                Text(  
                    'Wolfram Barkovich',  
                    style: TextStyle(fontSize: 35, fontWeight:  
                        FontWeight.w600),  
                ),  
                _buildDetailsRow('Age', '4'),  
                _buildDetailsRow('Status', 'Good Boy'),  
            ],  
        ),  
    );  
}
```

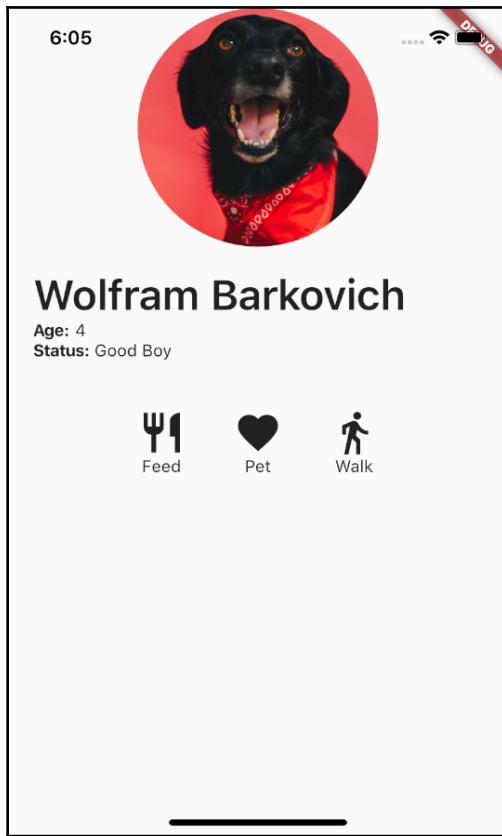
```
Widget _buildDetailsRow(String heading, String value) {
    return Row(
        children: <Widget>[
            Text(
                '$heading: ',
                style: TextStyle(fontWeight: FontWeight.bold),
            ),
            Text(value),
        ],
    );
}
```

7. Let's add some fake controls that simulate interactions with our pet. Replace the `_buildActions()` method with this code block:

```
Widget _buildActions(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)
            ],
        ),
    );
}
```

- Run the app—your device screen should now look similar to this:

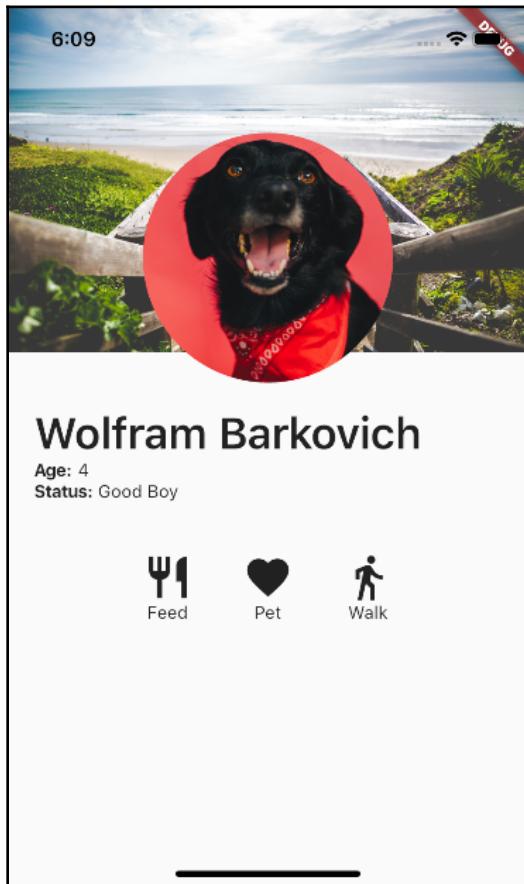


- In order to place widgets on top of each other, you can use a `Stack` widget.
Replace the code in the `build` method to add a billboard behind the dog photo:

```
@override  
Widget build(BuildContext context) {  
    return Scaffold(  
        body: Stack(  
            children: <Widget>[  
                Image.asset('assets/beach.jpg'),  
                Transform.translate(  
                    offset: Offset(0, 100),  
                    child: Column(  
                        children: <Widget>[  
                            _buildProfileImage(context),  
                            _buildProfileDetails(context),  
                        ]  
                    )  
                )  
            ]  
    );  
}
```

```
        _buildActions(context),  
    ],  
    ),  
    ),  
],  
),  
);  
}
```

The final screen will look like this:



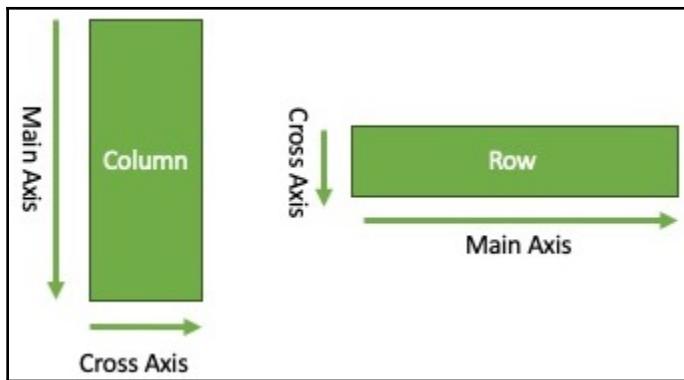
How it works...

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`

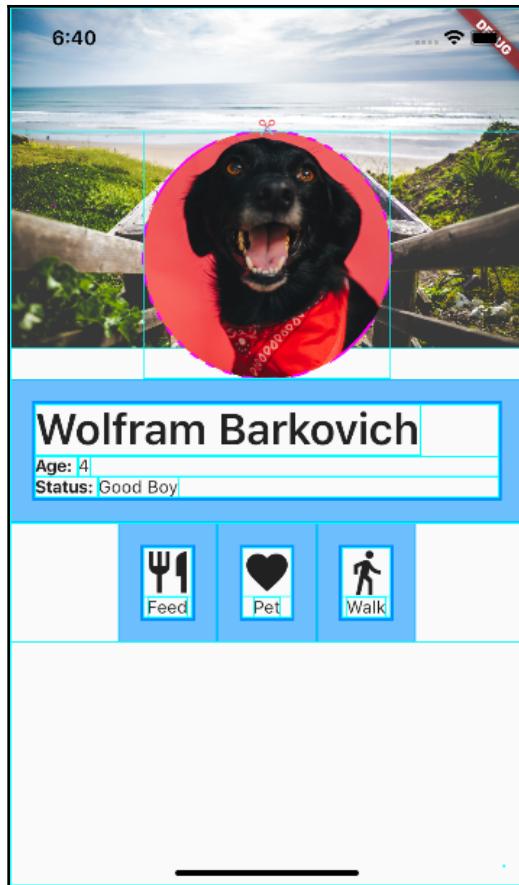
These are abstractions on the *x* and *y* axis. They are also referring to different axes depending on whether you are using a `Row` widget or a `Column` widget, as shown in the following diagram:



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.

The Flutter tools also have a setting called **Debug Paint**, which you can activate from the Flutter tools in your editor or from the command line:



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.

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—you have to be flexible.

`Column` and `Row` widgets do not just position widgets one after another—they also implement a variable on the `FlexBox` algorithm, allowing you to write UIs that should always look correct, regardless of the screen size.

In this recipe, we will demonstrate two ways to develop proportional widgets: `Flexible` and `Expanded` widgets.

Getting ready

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 double width;
    final double height;
    final Color color;
    final String text;
    final Color textColor;

    const LabeledContainer({
        Key key,
        this.width,
        this.height = double.infinity,
        this.color,
        @required this.text,
        this.textColor,
    }) : super(key: key);

    @override
```

```
Widget build(BuildContext context) {
    return Container(
        width: width,
        height: height,
        color: color,
        child: 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 'labeled_container.dart';

class FlexScreen extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                title: Text('Flexible and Expanded'),
            ),
            body: Column(
                mainAxisAlignment: MainAxisAlignment.start,
                children: <Widget>[
                    ..._header(context, 'Expanded'),
                    _buildExpanded(context),
                    ..._header(context, 'Flexible'),
                    _buildFlexible(context),
                ],
            ),
        );
    }

    Iterable<Widget> _header(BuildContext context, String text) {
        return [
            SizedBox(height: 20),
            Text(
                text,
                style: Theme.of(context).textTheme.headline,
```

```
        ),
    ];
}

Widget _buildExpanded(BuildContext context) {
    return Container();
}

Widget _buildFlexible(BuildContext context) {
    return Container();
}

Widget _buildFooter(BuildContext context) {
    return Container();
}
}
```

4. Now, fill in the `_buildExpanded` method:

```
Widget _buildExpanded(BuildContext context) {
    return SizedBox(
        height: 100,
        child: Row(
            children: <Widget>[
                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 `Flexible` section. Don't forget to hot reload while writing this code:

```
Widget _buildFlexible(BuildContext context) {
  return SizedBox(
    height: 100,
    child: Row(
      children: <Widget>[
        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 `buildFooter` method to create a rounded banner:

```
Widget _buildFooter(BuildContext context) {
  return Center(
    child: Container(
      decoration: BoxDecoration(
        color: Colors.yellow,
        borderRadius: BorderRadius.circular(40),
      ),
      child: Padding(
        padding: const EdgeInsets.symmetric(
          vertical: 15.0,
          horizontal: 30,
        ),
        child: Text(

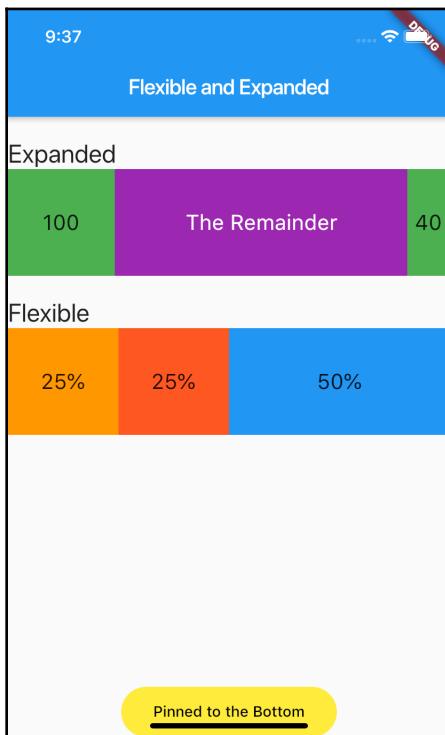
```

```
        'Pinned to the Bottom',
        style: Theme.of(context).textTheme.subtitle,
    ),
),
),
);
}
```

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 the `_buildFlexible` method:

```
_buildFlexible(context),
Expanded(
    child: Container(),
),
_buildFooter(context)
```

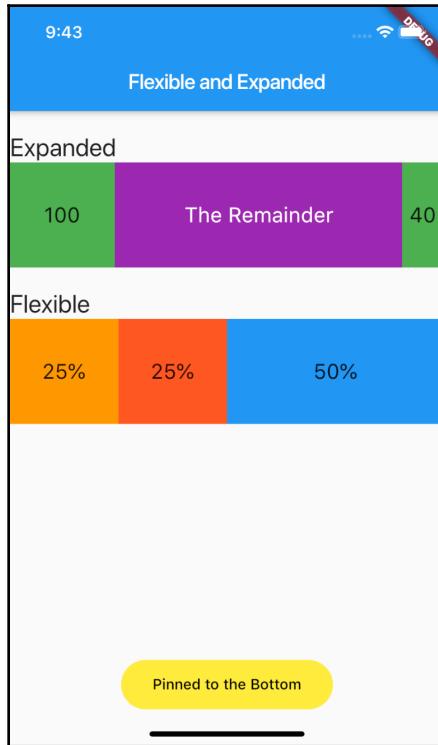
8. When running the app, you should see a screen similar to this:



9. On some devices, the header or footer covers up some software or hardware features (such as the notch or the **Home** button). To fix this, just wrap the **Scaffold** in a **SafeArea** widget:

```
return SafeArea(  
    child: Scaffold(  
        ...  
    )
```

10. Perform a hot reload and everything should now render correctly, as shown here:



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 in the first row. The container was given a width of 100 units. The last container was given a width of 40 units. 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:

$$\text{Width} = \text{ParentWidth} - 100 - 40$$

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:

```
Expanded(  
    child: Container(),  
) ,
```

It is very common to create an `Expanded` widget with an empty container that will simply consume the remaining space in a `Row` or `Column`.



Another widget you can use as a space-filler is the `Spacer` widget. Have a look at <https://api.flutter.dev/flutter/widgets/Spacer-class.html> for more information on this.

The `Flexible` widget behaves similarly to the `Expanded` widget, but it also has the ability to set a `flex` value, which can be used to help calculate how much space it should use.

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.



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({  
    Key key,
```

```
int flex = 1,  
    @required Widget child,  
  }) : super(key: key, flex: flex, fit: FlexFit.tight, child: child);  
}
```

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` to `Flexible` and the app would be unchanged.



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 6, 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://flutter.dev/docs/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, and even 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 fully 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 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({
        Key key,
        this.color,
        this.size,
    }) : super(key: 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();
  path.moveTo(size.width * 0.5, 0);
  path.lineTo(size.width * 0.618, size.height * 0.382);
  path.lineTo(size.width, size.height * 0.382);
  path.lineTo(size.width * 0.691, size.height * 0.618);
  path.lineTo(size.width * 0.809, size.height);
  path.lineTo(size.width * 0.5, size.height * 0.7639);
  path.lineTo(size.width * 0.191, size.height);
  path.lineTo(size.width * 0.309, size.height * 0.618);
  path.lineTo(size.width * 0.309, size.height * 0.618);
  path.lineTo(0, size.height * 0.382);
  path.lineTo(size.width * 0.382, size.height * 0.382);

  path.close();

  canvas.drawPath(path, paint);
}
```

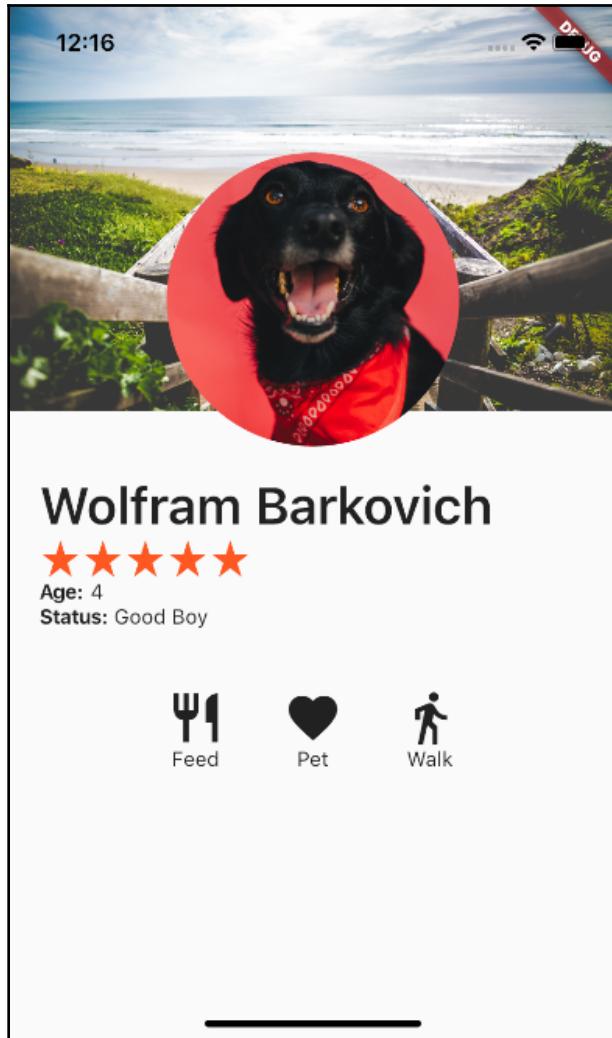
4. Create another class that uses these stars to create a rating. For the sake of brevity, we can include this new widget in the same file, as shown here:

```
class StarRating extends StatelessWidget {  
    final Color color;  
    final int value;  
    final double starSize;  
  
    const StarRating({  
        Key key,  
        @required this.value,  
        this.color = Colors.deepOrange,  
        this.starSize = 25,  
    }) : super(key: key);  
  
    @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 `_buildProfileDetails` method to add the `StarRating` widget:

```
Text(  
    'Wolfram Barkovich',  
    style: TextStyle(fontSize: 35, fontWeight: FontWeight.w600),  
)  
,  
StarRating(  
    value: 5,  
),  
_buildDetailsRow('Age', '4'),
```

The final app, as shown, should now have five stars under the dog's name:



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({  
  Key key,
```

```
    this.color,  
    this.size,  
) : super(key: key);
```

This custom constructor is asking for a color and a size, which will then be passed down to the painter. This 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. You can also pass the size directly into the constructor of `CustomPaint` if you want, but this method has been found to be more reliable.

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 when the widget is redrawn and provide information about the old custom painter that was used to draw the image. In most cases, you can just return `false` unless you need to change the drawing, to allow Flutter to cache your image.

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:

```
path.lineTo(size.width * 0.309, size.height * 0.618);
```

We wrote the preceding code instead of the following:

```
path.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 look in 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.



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.

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 a `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 tell Dart that we are ignoring this value, while still complying with the required API.

We could explicitly reference the index value with this code:

```
(index) => Padding(...)
```

If we did that, however, the compiler might give a warning that the index value is unused. It's usually considered a faux pas to declare variables and not use them. By replacing the `index` symbol with `_` 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

Your effectiveness with any given platform is measured by how fast you can make changes. 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's 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:

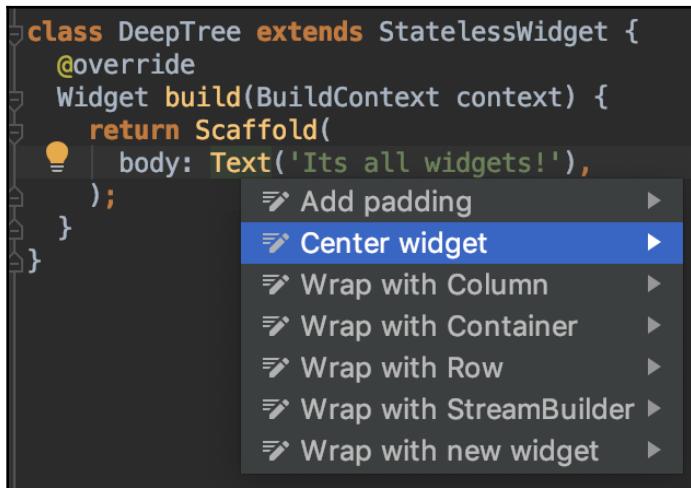
```
class DeepTree extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return Scaffold(  
            body: Text('Its all widgets!'),  
        );  
    }  
}
```

2. Compile and run the code. You'll get the following result:



3. 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?
4. 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`.

5. Perform a hot reload, and the text will move to the center of the screen:



6. 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.



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**.

7. You can now remove the Center widget.
8. Put your cursor on the Center widget and call the intentions methods, then select **Remove this widget**.
9. Add two more Text widgets, as shown here:

```
class DeepTree extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return Scaffold(  
            body: Center(  
                child: Column(  
                    children: <Widget>[  
                        Text('Its all widgets!'),  
                        Text('Flutter is amazing.'),  
                        Text('Let\'s find out how deep the rabbit hole goes.'),  
                    ],  
                ),  
            ),  
        );  
    }  
}
```

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

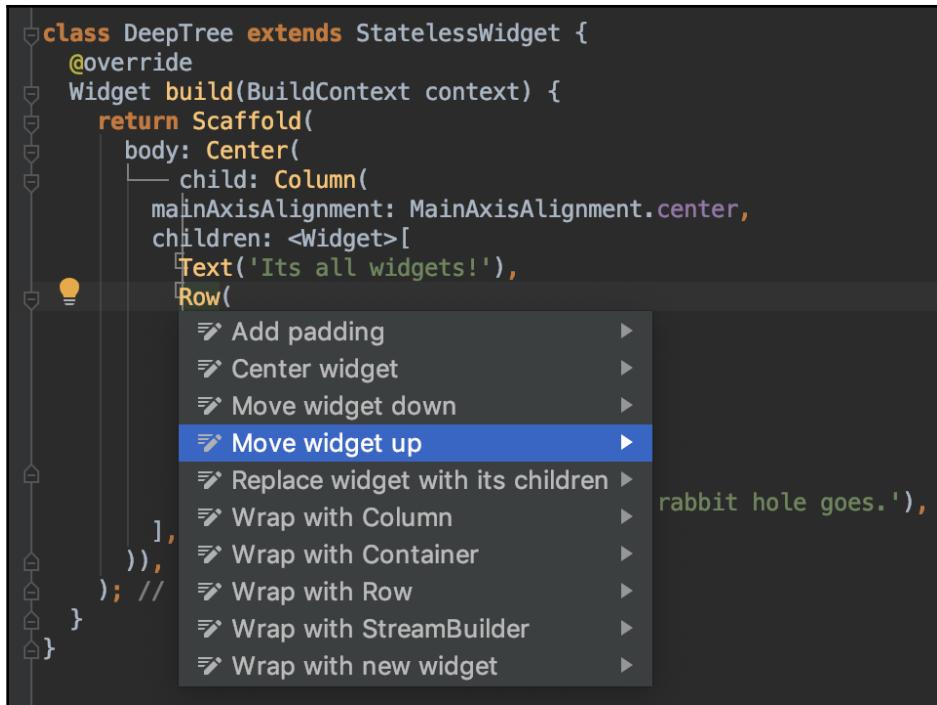
- Uh oh! The text has moved to the top again. We can fix that by setting the main axis on the column back to the center:

```
Column(  
  mainAxisSize: MainAxisSize.center,  
  children: <Widget>[  
    Text('Its all widgets!'),
```

- 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: <Widget>[  
    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 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. Bring up the intentions dialog and select **Move widget up**:

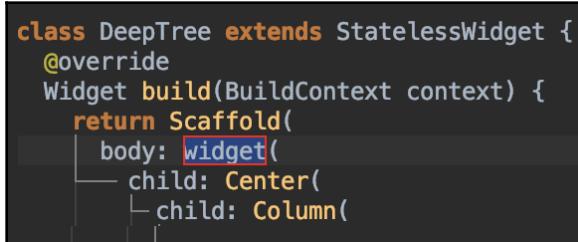


13. Add a purple container between the row and the 'It's all widgets!' Text widget:

```
Row(
  mainAxisSize: MainAxisSize.center,
  children: <Widget>[
    FlutterLogo(),
    Text('Flutter is amazing.'),
  ],
),
Expanded(
  child: Container(
    color: Colors.purple,
  ),
),
Text('It's all widgets!'),
```

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.

14. Bring up the intentions methods and select **Wrap with (a new) widget**. A placeholder will appear.
15. Replace the word `widget` with the `SafeArea` widget:



A screenshot of a Dart code editor showing a code snippet for a `DeepTree` class. The `build` method is defined, returning a `Scaffold` widget. Inside the `Scaffold`, there is a `body` child, which is currently highlighted with a red box. Below it is a `child` child, which contains a `Center` widget, and that contains a `child` child, which is a `Column` widget. This visual cue helps the developer quickly identify where the placeholder needs to be inserted.

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 intentions dialog 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 intentions 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. Both Android Studio and VS Code talk to the same server. When you open the intentions dialog, your IDE sends the token you are currently highlighting to the server, which analyzes your code and 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 this chapter, we've tried to keep our widget trees fairly shallow, but none of the examples so far is 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.

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 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:

1. The first thing we're going to do with this file is divide it up using a refactoring tool called **extract method**.

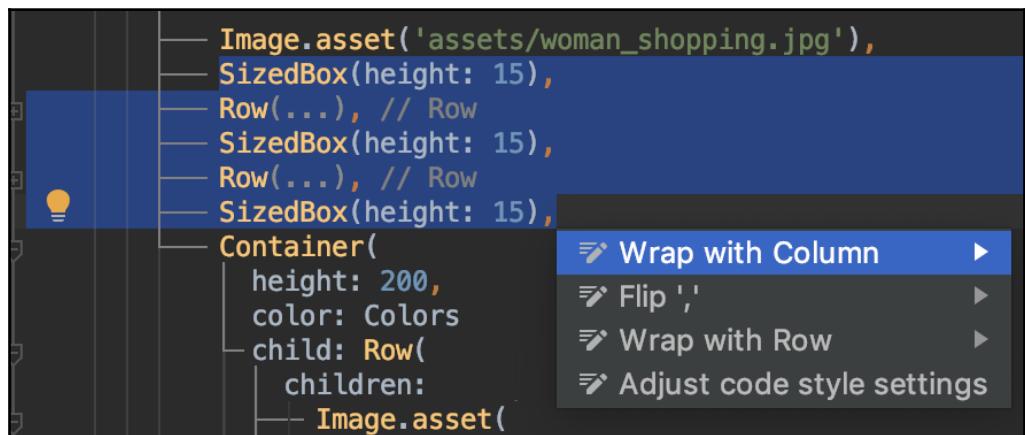
Move your cursor over `AppBar`, right-click and select **Refactor | Extract Method** on VS Code or **Refactor | Extract | Extract Method** on Android Studio to open the extract dialog. Call the method `_buildAppBar` and press `Enter`:



2. Perform an extract of the first `Row` widget in the `Column` widget using the same technique. Call this method `_buildToggleButton()`.
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:



4. With both rows collapsed, highlight the three `SizedBox` instances and the two rows and call the intentions dialog. Select **Wrap with Column**:



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`.

5. Finally, extract the last `Container` widget into a method called `_buildProductTile`. After all, this is done, the `build` method should be short enough to fit on the screen:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: _buildAppBar(),
    body: Padding(
      padding: const EdgeInsets.all(20.0),
      child: SingleChildScrollView(
```

```
        child: Column(
            children: <Widget>[
                _buildToggleBar(context),
                Image.asset('assets/woman_shopping.jpg'),
                DealButtons(),
                _buildProductTile(context),
            ],
        ),
    ),
),
);
}
```

6. 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.
7. Extract the third Expanded widget, the one that says **must buy in summer** into another Flutter widget called DealButton.
8. 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({
        Key key,
        this.text,
        this.color,
    }) : super(key: key);

    @override
    Widget build(BuildContext context) {
        return Expanded(
            child: Container(
                height: 80,
                decoration: BoxDecoration(
                    color: color,
                    borderRadius: BorderRadius.circular(20),
                ),
                child: Padding(
                    padding: const EdgeInsets.all(10.0),
                    child: Center(
                        child: Text(
                            text,
                            textAlign: TextAlign.center,
                            style: TextStyle(
                                color: Colors.white,

```

```
        fontSize: 20.0,
        fontWeight: FontWeight.bold,
    ),
),
),
),
),
);
}
}
```

9. The `build` method for the `DetailButtons` widget can now be significantly simplified. Replace that verbose repetitive method with this more efficient one:

```
@override
Widget build(BuildContext context) {
    return Column(
        children: <Widget>[
            SizedBox(height: 15),
            Row(
                children: <Widget>[
                    DealButton(
                        text: 'Best Sellers',
                        color: Colors.orangeAccent,
                    ),
                    SizedBox(width: 10),
                    DealButton(
                        text: 'Daily Deals',
                        color: Colors.blue,
                    )
                ],
            ),
            SizedBox(height: 15),
            Row(
                children: <Widget>[
                    DealButton(
                        text: 'Must buy in summer',
                        color: Colors.lightGreen,
                    ),
                    SizedBox(width: 10),
                    DealButton(
                        text: 'Last Chance',
                        color: Colors.redAccent,
                    )
                ],
            ),
            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:

- Extract method
- Extract widget

The first technique should be relatively straightforward. The Dart Analysis Server will inspect all the elements of your highlighted code 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: <Widget>[  
        Row(  
          children: <Widget>[  
            Text('Widget A'),  
            Text('Widget B'),  
          ],  
        ),  
        Row(  
          children: <Widget>[  
            Text('Widget C'),  
            Text('Widget D'),  
          ],  
        ),  
      ],  
    );  
  }  
}
```

You can go one step further beyond extracting and also 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: <Widget>[  
                buildRow('Widget A', 'Widget B'),  
                buildRow('Widget C', 'Widget D'),  
            ],  
        );  
    }  
  
    Widget buildRow(String textA, String textB) {  
        return Row(  
            children: <Widget>[  
                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 for refactoring 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.

When you invoke the `extract` method, your IDE will set the return to whatever top-level widget you are extracting. If you are extracting a `Row` widget, the return type of your method will be a row. It's usually considered best practice to change the return type to the `widget` superclass instead of something more specific. The benefit is that if you ever update the root widget in your `build` method to some other widget, you don't have to change the method signature. A `Row` is a widget. A `Column` is a widget. A `Padding` is a widget. Ensuring that your return type is always `Widget` will remove any blocks as your app grows.



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.

There are cases where both extraction methods make sense, but there aren't any hard rules for which one you should pick. Typically, if the widget tree that you are extracting is relatively simple and will not be reused, then a `build` method is fine. After the tree gets past a threshold of complexity, whatever that threshold may be, extracting to a widget instead of a method will become ideal.

See also

If you want to learn more about the pyramid of doom, see [https://en.wikipedia.org/wiki/Pyramid_of_doom_\(programming\)](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 a 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(  
                brightness: Brightness.dark,  
                primaryColor: 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,
```

When you hot reload, the `AppBar` will be green, respecting the `primaryColor` property of the app's theme.

4. The toggle bar could use a bit more refactoring, along with removing more styling information.
5. 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: <Widget>[
            _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.title.color
                        .withOpacity(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.



In a widget **class**, `BuildContext` is automatically available, so you don't need to pass it to the method.

```
Widget _buildToggleBar(BuildContext context) { ... }
```

9. Change the `build` method to pass down the context:

```
Column(
    children: <Widget>[
        _buildToggleBar(context),
```

```
        Image.asset('assets/woman_shopping.jpg'),
        DealButtons(),
        _buildProductTile(context),
    ],
),
```

10. 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.
11. Update `_buildProductTile` to make the text visible:

```
Widget _buildProductTile(BuildContext context) {
    return Container(
        height: 200,
        color: Theme.of(context).cardColor,
```

12. The screen should be fully responding to the app theme. But we can take it further.
13. 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,
        textTheme: TextTheme(
            headline6: TextStyle(
                fontFamily: 'LeckerliOne',
                fontSize: 24,
            ),
        ),
    ),
),
```

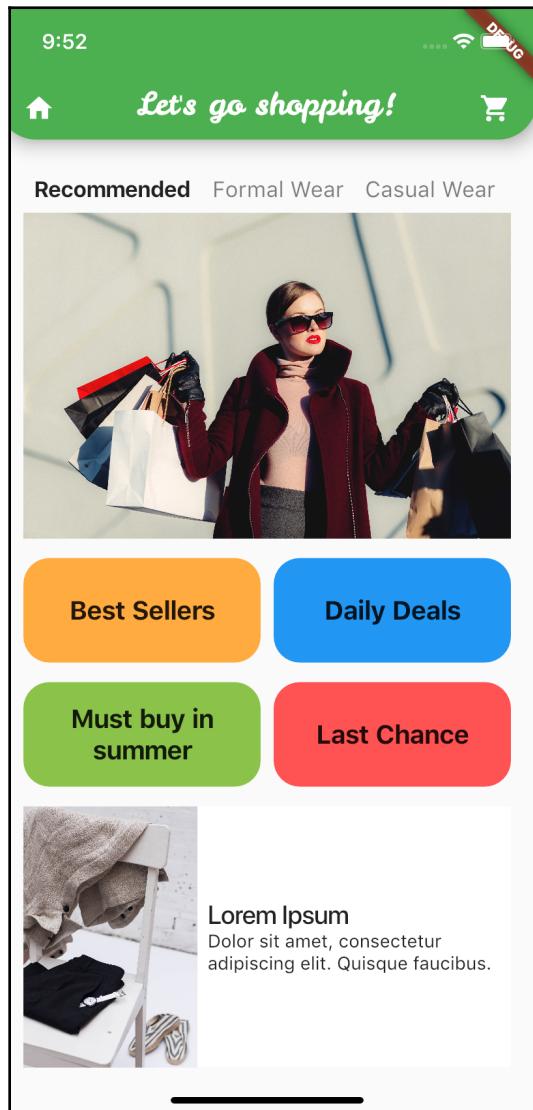
14. In order to use the `leckerlyOne` font, download it from <https://fonts.google.com/specimen/Leckerli+One>, add it to your `assets` folder and in your `pubspec.yaml` file, enable it with the following instructions:

```
fonts:
- family: LeckerliOne
  fonts:
    - asset: assets/LeckerliOne-Regular.ttf
```

15. Everything in the app is very dark, not so becoming of an e-commerce app.
Quickly adjust the theme's brightness to light:

```
brightness: Brightness.light,
```

16. Perform a hot reload and feast your eyes on the final themed product:



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.

If you *Ctrl + click* (or *Command + click* on a Mac) into the MaterialApp source code and find the Theme class, you'll see that it is also just a StatelessWidget:

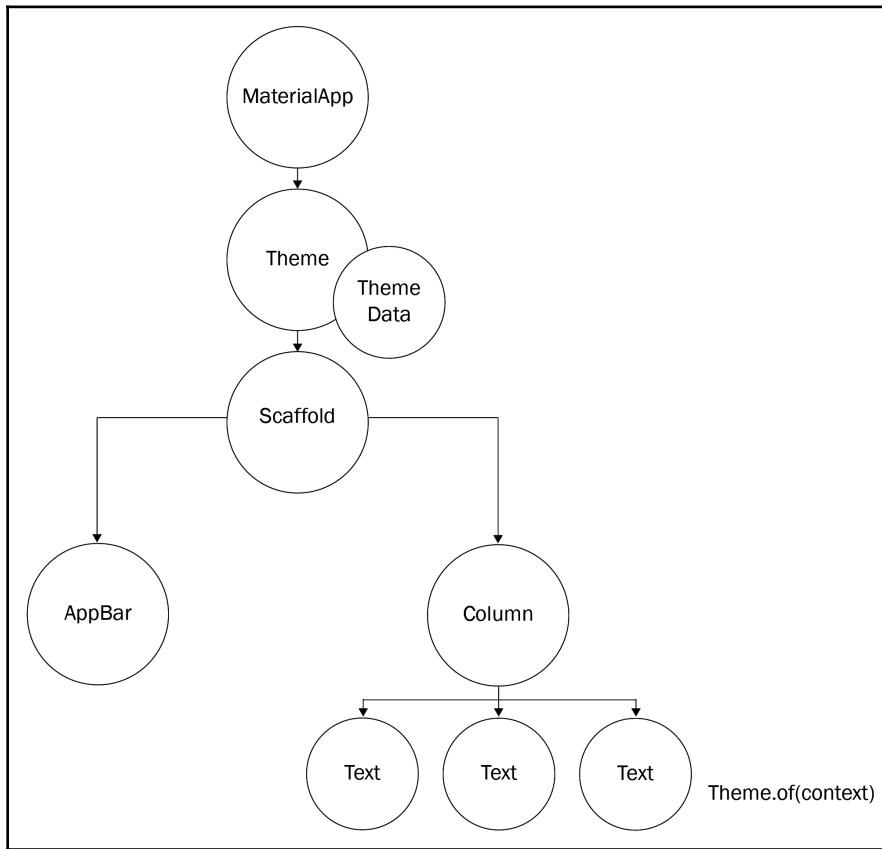
```
class Theme extends StatelessWidget {  
  /// Applies the given theme [data] to [child].  
  ///  
  /// The [data] and [child] arguments must not be null.  
  const Theme({  
    Key key,  
    @required this.data,  
    this.isMaterialAppTheme = false,  
    @required this.child,  
  }) : assert(child != null),  
        assert(data != null),  
        super(key: key);
```

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)
```

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` widget:



`BuildContext` is the key to unlock this tree traversal. It is the only object that is aware of the underlying widget tree and the node's parent/child relationships. This might seem very expensive to do, but after the `of` method is called once, a reference to the requested data is stored in the widget so that it can be retrieved instantly on subsequent calls.

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 now getting common. Apple introduced a toggle dark mode in iOS 13, and Google made it available in Android 10.

The `lightness` enum is how Flutter supports this feature. By toggling `lightness`, you have seen the background and text colors automatically get 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/>

5

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, TextFields, ScrollViews, and dialogs. You will also use the Navigator to create apps with multiple screens.

Throughout this chapter, you are going 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
- Invoking navigation routes by name
- Showing dialogs on the screen
- Presenting bottom sheets

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`. **Stateful widgets can keep information** and know how to recreate themselves whenever their state changes.

Compared to `StatelessWidgets`, `StatefulWidget`s 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 brand 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. Start off by adding 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(StopwatchApp());

class StopwatchApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: StopWatch(),
    );
  }
}
```

2. 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 'dart:async';
import 'package:flutter/material.dart';

class StopWatch extends StatefulWidget {
  @override
  State 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: Text('Stopwatch'),
      ),
      body: Center(
        child: Text(
          '0 seconds',
          style: Theme.of(context).textTheme.headline5,
        ),
      ),
    );
  }
}
```



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, but this text won't 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. Every 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;  
    Timer timer;  
  
    @override  
    void initState() {  
        super.initState();  
  
        seconds = 0;  
        timer = Timer.periodic(Duration(seconds: 1), _onTick);  
    }  
  
    void _onTick(Timer time) {  
        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';
```

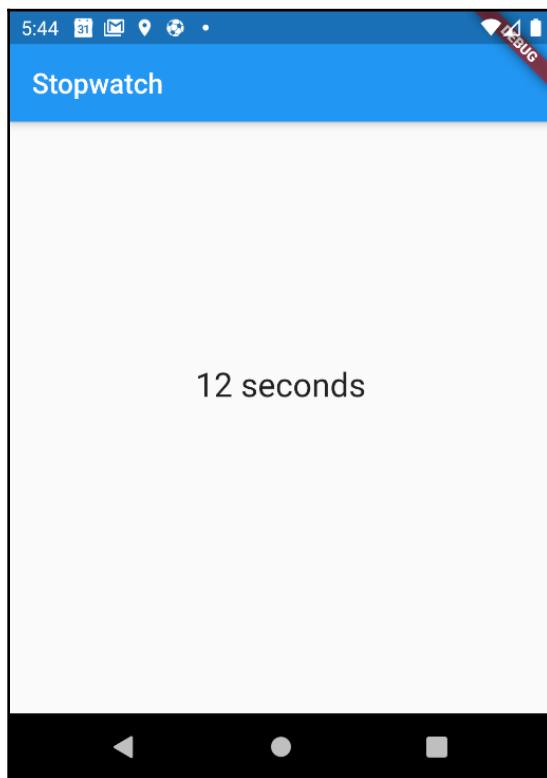
8. Update the `Text` widget in the `build` method so that it can now use the `seconds` property:

```
Text(  
    '$seconds ${_secondsText()}',  
    style: Theme.of(context).textTheme.headline5,  
) ,
```

- 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();  
}
```

Run the app. You should now see a counter incrementing once a second. Pretty fancy!



How it works...

StatefulWidgets are made up of two classes: the widget and the state. The *widget* part of StatefulWidget really doesn't do much, and all the properties that you store in it must be final; otherwise, you will get a compile error.



All widgets, whether they are stateless or stateful, are still 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:

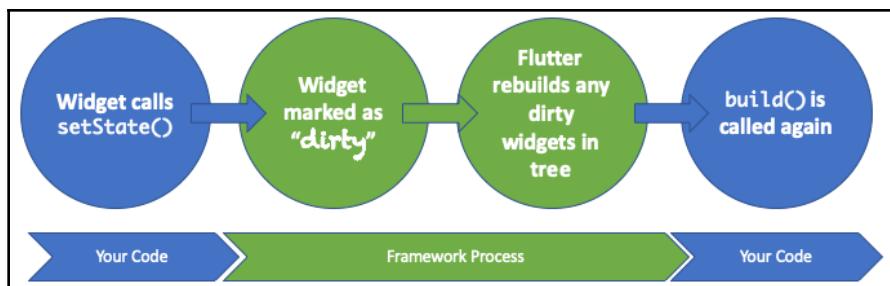
```
setState(() {  
    ++seconds;  
});
```

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.



Each time you call `setState`, the widget is repainted.

The following diagram summarizes how Flutter's render loop is impacted by `setState`:



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 written the following code and it would have had identical results:

```
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 is nothing more than a `build` method, `StatefulWidget`s have a few different 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 purpose:

- **`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 **before** the widget is added to the tree, so you do not have any access to the state's `BuildContext` property.

- **`didChangeDependencies`:**

This method is called immediately after `initState`, but unlike that method, the widget now has access to its `BuildContext`. If you need to do any setup work that requires context, then this is 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, you should create the UI.

- **dispose:**

This cleanup 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 and 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>
- State documentation: <https://api.flutter.dev/flutter/widgets/State-class.html>

Interacting with buttons

Buttons are one of the most important types of interactions in apps. 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.

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 are going to use two different styles of buttons – `ElevatedButton` and `TextButton` – for these different functions, but note that 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. Add some buttons to the screen.
2. Update the `build` method in `StopWatchState` to replace the `Center` widget with a `Column`. You should be able to use the intentions dialog that we discussed in the previous chapter to quickly wrap the `Text` widget in a `Column` and then remove the `Center` widget. When you are done, the `build` method should look like this:

```
return Scaffold(
    appBar: AppBar(
        title: Text('Stopwatch'),
    ),
    body: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
            Text(
                '${_seconds} ${_secondsText()}',
                style: Theme.of(context).textTheme.headline5,
            ),
            SizedBox(height: 20),
            Row(
                mainAxisAlignment: MainAxisAlignment.center,
                children: <Widget>[
                    ElevatedButton(
                        style: ButtonStyle(
                            backgroundColor: MaterialStateProperty
                                .all<Color>(Colors.green),
                            foregroundColor: MaterialStateProperty
                                .all<Color>(Colors.white),
                        ),
                        child: Text('Start'),
                        onPressed: null,
                    ),
                    SizedBox(width: 20),
                    TextButton(
                        style: ButtonStyle(
                            backgroundColor: MaterialStateProperty
                                .all<Color>(Colors.red),
                            foregroundColor: MaterialStateProperty
                                .all<Color>(Colors.white),
                        ),
                        child: Text('Stop'),
                        onPressed: null,
                    ),
                ],
            ),
        ],
    ),
);
```

```
) ,  
] ,  
)  
,  
) ;
```

3. If you hot reload, you'll see two buttons on the screen.
4. You can try tapping them, but nothing will happen. This is because you have to add an `onPressed` function, before the button will activate.
5. 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.
6. Add this line at the very top of `StopWatchState`:

```
bool isTicking = true;
```

7. 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;  
    });  
}
```

8. Now, you can hook these methods into the `onPressed` property.

9. Update your buttons so that they use these ternary operators in button declarations. Some of the setup code has been omitted for brevity:

```
ElevatedButton(  
    child: Text('Start'),  
    onPressed: isTicking ? null : _startTimer,  
    ...  
) ,  
...  
TextButton(  
    child: Text('Stop'),  
    onPressed: isTicking ? _stopTimer : null,  
    ...  
) ,
```

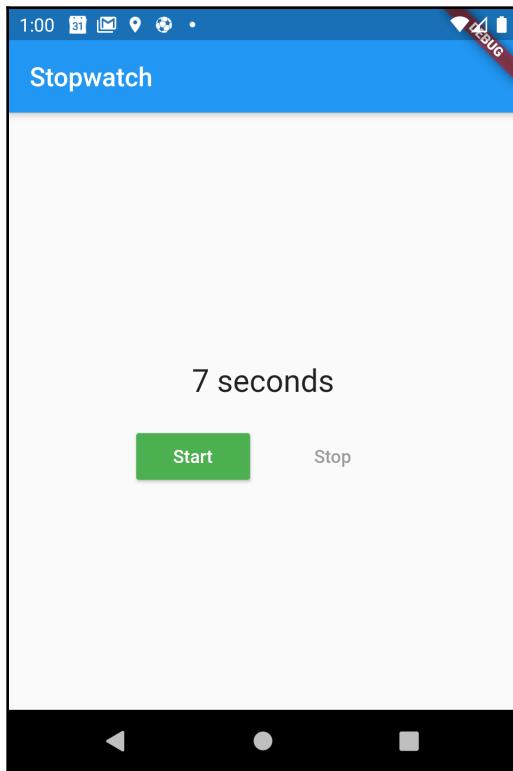
10. Now, it's time to add some logic to the start and stop methods to make the timer respond to our interactions.
11. Update those buttons so that they include the following code:

```
void _startTimer() {  
    timer = Timer.periodic(Duration(seconds: 1), _onTick);  
  
    setState(() {  
        seconds = 0;  
        isTicking = true;  
    });  
}  
  
void _stopTimer() {  
    timer.cancel();  
  
    setState(() {  
        isTicking = false;  
    });  
}
```

12. 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 and update the `seconds` property at the top of the class so that it is initialized with a value instead of `null`:

```
int seconds = 0;
```

Congratulations – you should now have a fully functioning timer app!



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. They are as follows:

- `ElevatedButton`
- `TextButton`
- `IconButton`
- `FloatingActionButton`

- DropDownButton
- CupertinoButton

You can play around with any of these widgets until you find a button that matches your desired look.

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 can extend beyond the height of a screen, you'll need to use 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...

Open up `stopwatch.dart` to get started:

1. The first thing you are going to do is make the timer a bit more precise. Seconds are not a very interesting value to use for stopwatches.
2. Use the refactoring tools to 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. 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 every time the user taps a lap button.
4. Add this property to the top of the `StopWatchState` class, just under the declaration of the timer:

```
final laps = <int>[];
```

5. Create a new `lap` method to increment the lap count and reset the current millisecond value:

```
void _lap() {
    setState(() {
        laps.add(milliseconds);
        milliseconds = 0;
    });
}
```

```
});  
}
```

6. We also need to tweak the `_startTimer` method in order to reset the lap list every time the user starts a new counter.
7. Add the following line inside the `setState` closure, in `_startTimer`:

```
laps.clear();
```

8. Now, we need to organize our widget code a bit to enable adding scrollable content. Let's start off by taking the existing `Column` in the `build` method and extracting it into its own method called `_buildCounter`. The refactoring tools should automatically add a `BuildContext` to that method. Don't forget to change the return type of this new method from `Column` to `Widget`.
9. 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.
10. 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: <Widget>[  
                Text(  
                    'Lap ${laps.length + 1}',  
                    style: Theme.of(context)  
                        .textTheme  
                        .subtitle1  
                        .copyWith(color: Colors.white),  
                ),  
                Text(  
                    _secondsText(milliseconds),  
                    style: Theme.of(context)  
                        .textTheme  
                        .headline5  
                        .copyWith(color: Colors.white),  
                ),
```

11. 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`.

12. 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:

```
SizedBox(width: 20),  
ElevatedButton(  
    style: ButtonStyle(  
        backgroundColor: MaterialStateProperty  
            .all<Color>(Colors.yellow),),  
    child: Text('Lap'),  
    onPressed: isTicking ? _lap : null,  
,  
SizedBox(width: 20),
```

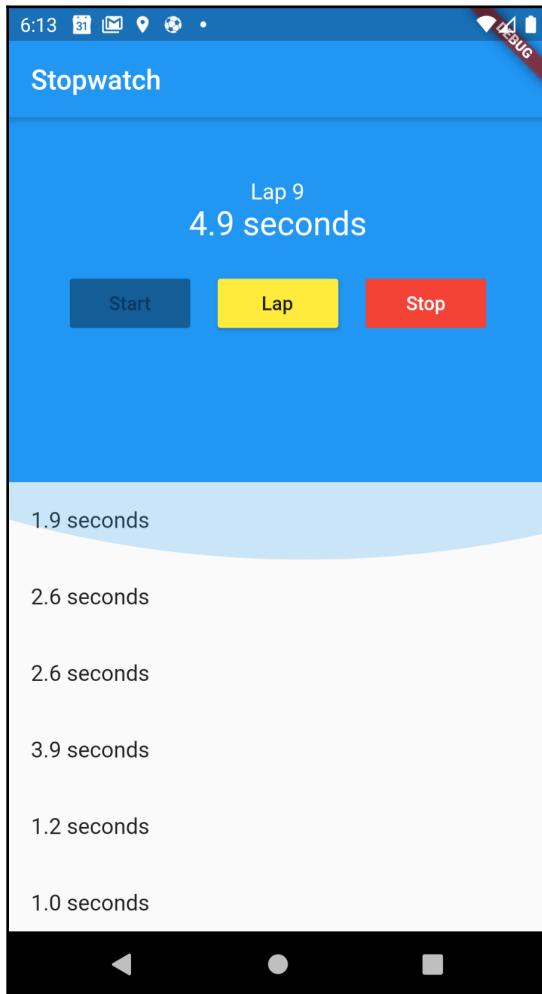
13. That's a lot of refactoring! But it was all for a good purpose. Now, you can finally add a `ListView`.
14. Add the following method before the `dispose()` method. This will create the scrollable content for the laps:

```
Widget _buildLapDisplay() {  
    return ListView(  
        children: [  
            for (int milliseconds in laps)  
                ListTile(  
                    title: Text(_secondsText(milliseconds)),  
                )  
        ],  
    );  
}
```

15. 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: <Widget>[  
                Expanded(child: _buildCounter(context)),  
                Expanded(child: _buildLapDisplay()),  
            ],  
        );  
}
```

16. 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:



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 to create the widgets for this list. This will essentially 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. 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 behavior if you want to make the app behave in a particular way, regardless of the platform:

```
ListView(
    physics: BouncingScrollPhysics(),
    children: [...],
);
```



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...

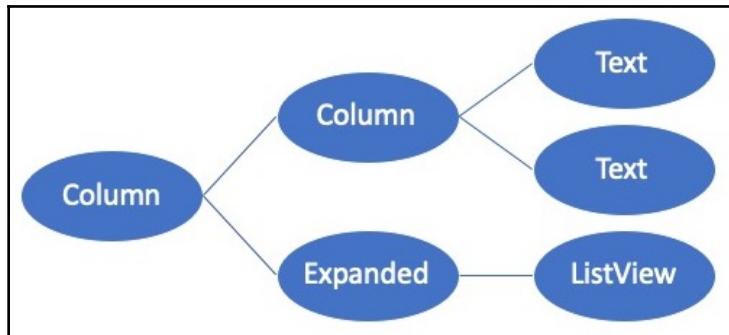
One final important thing to keep in mind about scrolling widgets is that because they need to know their parent's constraints to activate scrolling, putting scroll widgets inside widgets with unbounded constraints can cause Flutter to throw errors.

In our example, we placed `ListView` inside a `Column`, which is a flex widget that lays out its children based on their intrinsic size. This works fine for widgets such as `Containers`, `Buttons`, and `Text`, but it fails for `ListView`s. 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)
```

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, 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:



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 potentially 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. 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: 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:



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 key into `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.

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 will need to type something; for example, a form where users need to 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:

1. Create a new file called `login_screen.dart` and 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.
2. Name this class `LoginScreen`. The `State` class will automatically be named `_LoginScreenState`. Don't forget to fix your missing imports by bringing in the `material` library.
3. 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.

4. 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: Text('Login'),
        ),
        body: Center(
            child: loggedIn ? _buildSuccess() : _buildLoginForm(),
        ),
    );
}
```

5. The success widget is pretty simple – it's just a checkmark and a `Text` widget to print whatever the user typed:

```
Widget _buildSuccess() {
    return Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
            Icon(Icons.check, color: Colors.orangeAccent),
            Text('Hi $name')
        ],
    );
}
```

6. Now that that's done, we can get into the meat of the recipe: the login form.
7. Add some more properties at the top of the class; that is, two `TextEditingController`s to handle our `TextField` properties and `GlobalKey` to handle our Form:

```
final _nameController = TextEditingController();
final _emailController = TextEditingController();
final _formKey = GlobalKey<FormState>();
```

8. 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: InputDecoration(labelText: 'Runner'),
                    validator: (text) =>
                        text.isEmpty ? 'Enter the runner\'s
                            name.' : null,
                ),
            ],
        )));
}
```

9. Run the app. You will see a single `TextField` floating in the center of the screen.
Now, let's add one more field to manage the user's email address.
10. Add this widget inside `Column`, just after the first `TextField`. This widget uses a **regular expression** to validate its data:

```
TextField(
    controller: _emailController,
    keyboardType: TextInputType.emailAddress,
    decoration: 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;
    },
),
```



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>.

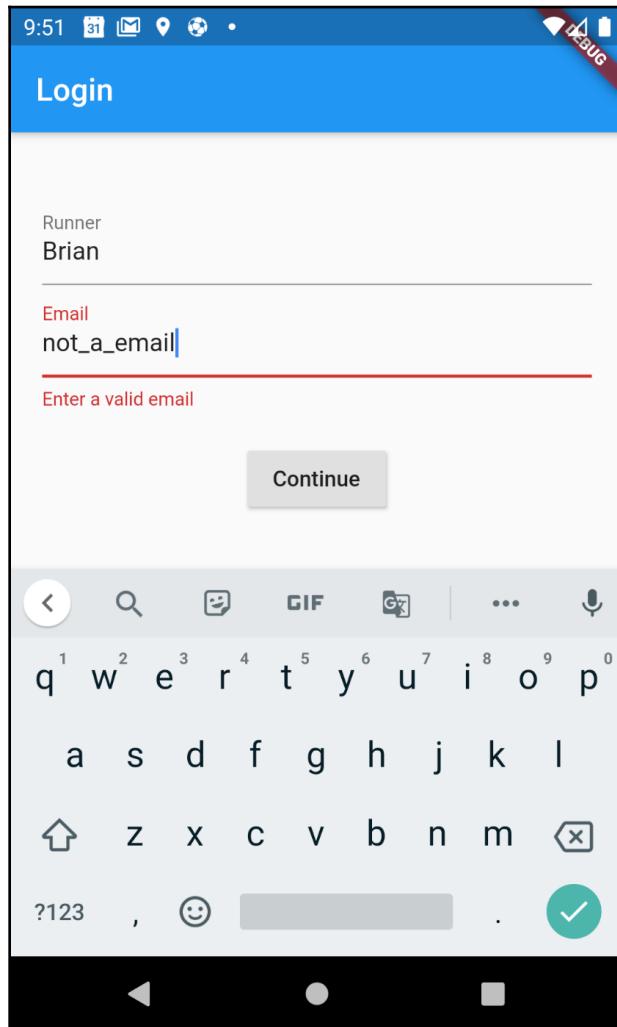
11. The form items set should be all set up; now, you just need a way to validate them. This can be accomplished with a button and function that calls the form's `validateAndSubmit` method.
12. Add these two widgets inside the same `Column`, just after the second `TextField`:

```
SizedBox(height: 20),  
ElevatedButton(  
    child: Text('Continue'),  
    onPressed: _validate,  
) ,
```

13. Now, implement the `validate()` method:

```
void _validate() {  
    final form = _formKey.currentState;  
    if (!form.validate()) {  
        return;  
    }  
  
    setState(() {  
        loggedIn = true;  
        name = _nameController.text;  
    });  
}
```

Perform a hot reload. As a bonus, try entering incorrect information into the form and see what happens:



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. 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 secondary 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. A simple way to explain `GlobalKeys` is that they do the opposite of `BuildContext`. `BuildContext` is an object that can find **parents** in the widget tree. Keys are objects that are used 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.



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, `TextEditingController`s 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 arsenal.

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 `TextEditingController`s, 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:

- Article and video about keys by Emily Fortuna. Don't miss this one!: <https://medium.com/flutter/keys-what-are-they-good-for-13cb51742e7d>
- Forms: <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 the widget we want to show and how we want to animate them 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 of the most enjoyable activities for a developer – deleting code.

2. 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.
3. Update the `build` method with the following snippet:

```
Widget build(BuildContext context) {  
    return Scaffold(  
        appBar: AppBar(  
            title: Text('Login'),  
        ),  
        body: Center(  
            child: _buildLoginForm(),  
        ),  
    );  
}
```

4. In the `_validate` method, we can kick off the navigation instead of calling `setState`:

```
void _validate() {  
    final form = _formKey.currentState;  
    if (!form.validate()) {  
        return;  
    }  
  
    final name = _nameController.text;  
    final email = _emailController.text;  
  
    Navigator.of(context).push(  
        MaterialPageRoute(  
            builder: (_) => Stopwatch(name: name, email: email),  
        ),  
    );  
}
```

5. 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({Key key, this.name, this.email}) : super(key:  
        key);
```

6. 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),  
) ,
```

7. 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. You can use the Navigator's pushReplacement method to achieve this:

```
Navigator.of(context).pushReplacement(  
    MaterialPageRoute(  
        builder: (_) => StopWatch(name: name, email: email),  
    ),  
) ;
```

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 yet 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.

Invoking navigation routes by name

Routing is a concept that is so engrained into the internet that we almost don't think about it anymore.

In Flutter, you can use **named routes**. This means you can assign a textual name to your screens and simply invoke them as if you were just going to another page on a website.

In this recipe, you are going to update the existing routing mechanism in the stopwatch project so that you can use named routes instead.

How to do it...

Open the existing stopwatch project to get started:

1. Named routes are referenced as strings. To reduce the potential for error, add some constants to the top of both `stopwatch.dart` and `login_screen.dart`:

```
class LoginScreen extends StatefulWidget {  
    static const route = '/login';  
  
    class Stopwatch extends StatefulWidget {  
        static const route = '/stopwatch';
```

2. These routes need to be wired up in `MaterialApp` so that they can be fed to the app's Navigator.
3. Open `main.dart` and update the `MaterialApp`'s constructor so that it includes these pages:

```
return MaterialApp(  
    routes: {  
        '/': (context) => LoginScreen(),  
        LoginScreen.route: (context) => LoginScreen(),
```

```
        StopWatch.route: (context) => Stopwatch(),
    },
    initialRoute: '/',
);
```

4. Now, we can invoke this route. Open `login_screen.dart` and scroll to the `_validate` method. Replace the existing navigator code at the bottom of the method by calling the `pushReplacementNamed` method:

```
Navigator.of(context).pushReplacementNamed(
    StopWatch.route,
);
```

5. There is one significant difference between named routes and manually constructed routes – you cannot use custom constructors to pass data to the next screen. Instead, you can use **route arguments**.
6. Update `Navigator` so that it uses the runner's name from the form in the optional argument property:

```
final name = _nameController.text;
Navigator.of(context).pushReplacementNamed(
    StopWatch.route,
    arguments: name,
);
```

7. To pull the data out of the route's argument, we need to retrieve the screen's route from its build method. Thankfully, this can be accomplished with the `of-context` pattern.
8. In the build method in `stopwatch.dart`, add the following code to the very top and update `AppBar`:

```
String name = ModalRoute.of(context).settings.arguments ?? "";
return Scaffold(
    appBar: AppBar(
        title: Text(name),
    ),
),
```

Hot reload the app. While you won't see any noticeable difference from the previous recipe, the code is slightly cleaner.

How it works...

If you look at the way we define the routes in `MaterialApp`, the **home** route is required. You can achieve this with the "/" symbol. Then, you can set up the other routes for your app:

```
routes: {  
  '/': (context) => LoginScreen(),  
  LoginScreen.route: (context) => LoginScreen(),  
  StopWatch.route: (context) => StopWatch(),  
,
```

We have a bit of redundancy here because there are only two screens in this app. Once the routes have been declared, `MaterialApp` needs to know which route to start with. This is just inputted as a string:

```
initialRoute: '/',
```

It is recommended that you define constants for your routes and use those instead of string literals. In this recipe, we put the constants as static elements for each screen. There is no real requirement to organize your code like that; you could also keep your constants in a single file if you prefer.



The decision to use named routes over manually constructed routes is not entirely clear-cut. If you decide to go with named routes, there is a bit more planning and setup that is required upfront, without any significant benefits. It could be argued that code for named routes is a bit cleaner, but it is also harder to change. Ultimately, it might be easier to start developing with manually constructed routes and then **refactor** toward named routes when the need arises.

Passing data between named routes also requires a bit more thought. You cannot use any custom constructor because `WidgetBuilder` is already defined and locked in `MaterialApp`. Instead, you can use arguments to add anything you want to pass to the next screen. If you take a look at the definition of the `pushNamed` function, you'll see that the type for arguments is simply `Object`:

```
pushNamed(  
  String routeName, {  
    Object arguments,  
  })
```

While flexible, this ignores any type of safety that we might have gotten by using generics. The responsibility is now on the programmer to make sure the correct objects are sent to the route.

We retrieved the arguments with the *of-context* pattern to get the route associated with this widget:

```
String name = ModalRoute.of(context).settings.arguments;
```

This code is not safe in itself. There is no guarantee that the value that was passed into the arguments is a string or even exists at all. If the object that created this route decided to put an integer or a Map into the arguments property, then this line would throw an exception, causing the red error screen to take over your whole app. Because of this, you need to be especially careful when working with route arguments.

Passing arguments through named routes requires some effort, especially if you want to do so safely. For these reasons, it's recommended that you use manually constructed routes when you need to pass data back and forth between your screens.

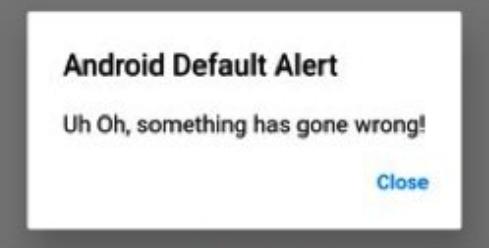
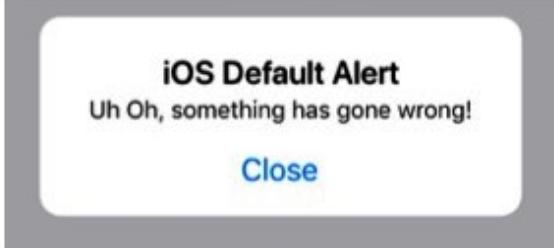
Showing dialogs on the screen

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.



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 necessary**.

The following are the default alerts for Android and iOS:

Android	iOS
 Material Design	 Cupertino Design

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.

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})
        : assert(title != null),
        assert(message != null);
}
```

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.



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: Text('Close'),  
                        onPressed: () => Navigator.of(context).pop()  
                ]);  
        });  
}
```

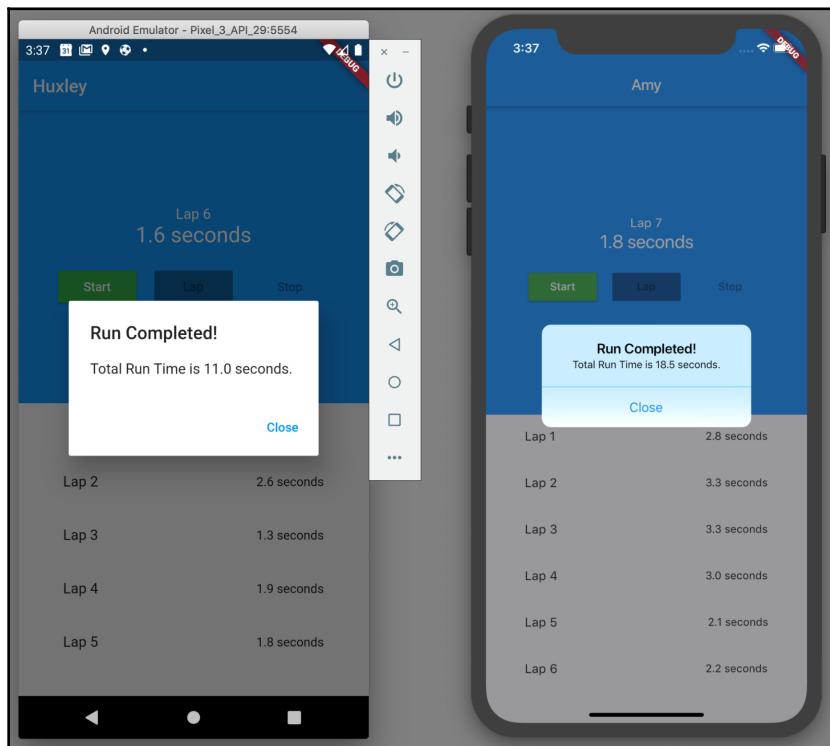
6. The iOS version is very similar – we just need to swap out the material components with their Cupertino counterparts. Add this method immediately after the material builder:

```
void _buildCupertinoAlert(BuildContext context) {  
    showCupertinoDialog(  
        context: context,  
        builder: (context) {  
            return CupertinoAlertDialog(  
                title: Text(title),  
                content: Text(message),  
                actions: [  
                    CupertinoButton(  
                        child: 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);
```

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:



How it works...

The way Flutter handles dialogs is fascinating in terms of its simplicity. 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 early days for this operating system and information is sparse, but this confirms that no matter what happens, the future of Flutter is 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. 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 the user to conveniently ignore this optional modal.

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.

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 `BuildContext` and a `WidgetBuilder`.
2. Let's create that builder as its own function.
3. Add the following code underneath 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: EdgeInsets.symmetric(vertical: 30.0),
                child: Column(mainAxisSize: MainAxisSize.min, children: [
                    Text('Run Finished!', style: textTheme.headline6),
                    Text('Total Run Time is
                        ${_secondsText(totalRuntime)}')
                ])),
        )
    );
}
```

4. 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);
}
```

5. Try running the code now and tap the stop button to present the sheet. Did it work? You are probably seeing a lot of nothing right now. In actuality, you might even be seeing the following error being printed to the console:

```
flutter: ======| EXCEPTION CAUGHT BY GESTURE |=====
flutter: The following assertion was thrown while handling a gesture:
flutter: No Scaffold widget found.
flutter: StopWatch widgets require a Scaffold widget ancestor.
flutter: The specific widget that could not find a Scaffold ancestor was:
flutter:   StopWatch
flutter: The ownership chain for the affected widget is:
flutter:   StopWatch ← Semantics ← Builder ← RepaintBoundary-[GlobalKey#a3f68] ← IgnorePointer ← Stack ←
flutter:   CupertinoBackGestureDetector<dynamic> ← DecoratedBox ← DecoratedBoxTransition ←
flutter:   FractionalTranslation ← ...
flutter: Typically, the Scaffold widget is introduced by the MaterialApp or WidgetsApp widget at the top of
flutter: your application widget tree.
flutter:
flutter: When the exception was thrown, this was the stack:
flutter: #0      debugCheckHasScaffold.<anonymous closure> (package:flutter/src/material/debug.dart:149:7)
flutter: #1      debugCheckHasScaffold (package:flutter/src/material/debug.dart:161:4)
flutter: #2      showBottomSheet (package:flutter/src/material/bottom_sheet.dart:481:10)
flutter: #3      StopWatchState._stopTimer (package:stopwatch/stopwatch.dart:142:9)
flutter: #4      StopWatchState._buildControls.<anonymous closure> (package:stopwatch/stopwatch.dart:101:40)
flutter: #5      _InkResponseState._handleTap (package:flutter/src/material/ink_well.dart:635:14)
flutter: #6      _InkResponseState.build.<anonymous closure> (package:flutter/src/material/ink_well.dart:711:32)
flutter: #7      GestureRecognizer.invokeCallback (package:flutter/src/gestures/recognizer.dart:182:24)
flutter: #8      TapGestureRecognizer._checkUp (package:flutter/src/gestures/tap.dart:365:11)
flutter: #9      TapGestureRecognizer.handlePrimaryPointer (package:flutter/src/gestures/tap.dart:275:7)
flutter: #10     PrimaryPointerGestureRecognizer.handleEvent (package:flutter/src/gestures/recognizer.dart:455:9)
flutter: #11     PointerRouter._dispatch (package:flutter/src/gestures/pointer_router.dart:75:13)
flutter: #12     PointerRouter.route (package:flutter/src/gestures/pointer_router.dart:102:11)
flutter: #13     _WidgetsFlutterBinding&BindingBase<GestureBinding>.handleEvent (package:flutter/src/gestures/binding.dart:218:19)
flutter: #14     _WidgetsFlutterBinding&BindingBase<GestureBinding>.dispatchEvent (package:flutter/src/gestures/binding.dart:198:22)
flutter: #15     _WidgetsFlutterBinding&BindingBase<GestureBinding>._handlePointerEvent (package:flutter/src/gestures/binding.dart:156:7)
flutter: #16     _WidgetsFlutterBinding&BindingBase<GestureBinding>._flushPointerEventQueue (package:flutter/src/gestures/binding.dart:102:7)
flutter: #17     _WidgetsFlutterBinding&BindingBase<GestureBinding>._handlePointerDataPacket (package:flutter/src/gestures/binding.dart:86:7)
flutter: #21     _invoke1 (dart:ui/hooks.dart:250:10)
flutter: #22     _dispatchPointerDataPacket (dart:ui/hooks.dart:159:5)
flutter: (elided 3 frames from package dart:async)
flutter:
flutter: Handler: "onTap"
flutter: Recognizer:
flutter:   TapGestureRecognizer#f56db
flutter: ======|
```

6. 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. 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,
  ...
)
```

We also have to update the `_stopTimer` method so that it accepts a `BuildContext` as a parameter:

```
void _stopTimer(BuildContext context) {...}
```

- 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:

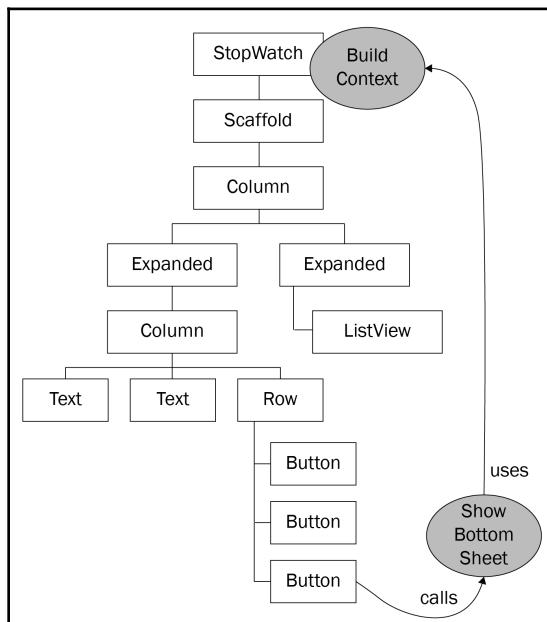
```
final controller =
    showBottomSheet(context: context, builder:
        _buildRunCompleteSheet);

Future.delayed(Duration(seconds: 5)).then((_) {
    controller.close();
});
```

Hot reload the app. The bottom sheet now politely excuses itself after 5 seconds.

How it works...

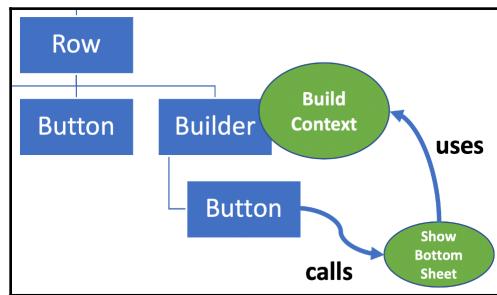
The bottom sheet part of this recipe should be pretty simple to understand, but what's going on with that error? Why did showing the bottom sheet initially fail? Take a look at how we organized the widget tree for the stopwatch screen:



Bottom sheets are a little different than 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 BuildContext 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:



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 buildBottomSheet 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 in their own chapter on asynchronous code later in this book.

See also

Take a look at these resources if you want to explore how Flutter manages state:

- Widgets 101 – StatefulWidgets: <https://www.youtube.com/watch?v=AqCMFXEmf3w>
- Flutter's Layered Design: <https://www.youtube.com/watch?v=dkY9WCGMi0>

6

Basic State Management

As apps grow, managing the flow of data through the app becomes a more complex and important issue. The Flutter community has struggled with this problem. Due to this, they have devised several solutions to deal with state management. All these solutions have one aspect they share: the separation of model and view.

Before diving into any state management solutions (BLoC, MVVM, Redux, and so on), we will explore the elements that they all share. Each of these patterns divides apps into *layers*. These are groups of classes that perform specific kinds of tasks. Layer strategies can be applied to almost any app architecture. Once you've mastered the basics, learning any of the more advanced patterns will be easy.

In this chapter, you are going to build a to-do note-taking application. In this application, users will be able to create to-do lists that contain many tasks. Users will be able to add, edit, delete, and complete their tasks.

We will cover the following recipes:

- Model-view separation
- Managing the data layer with `InheritedWidget`
- Making the app state visible across multiple screens
- Designing an *n*-tier architecture, part 1 – controllers
- Designing an *n*-tier architecture, part 2 – repositories
- Designing an *n*-tier architecture, part 3 – services

Technical requirements

Start off by creating a brand new Flutter project called `master_plan` in your favorite IDE. Once Flutter has generated the project, delete everything in the `lib` and `test` folders.

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.

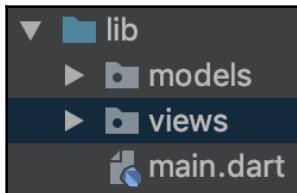
In Flutter, the closest analogy we have to views are `widgets`. Subsequently, a **model** would be a basic dart class that doesn't inherit from anything in the Flutter framework. Each one of these classes is responsible for one and only one job. Models are concerned with maintaining the data for your 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 work with.

In this recipe, we're going to build the start of our Todo app and create a model graph to go along with our views.

Getting ready

Any self-respecting app architecture must ensure it has the right folder structure set up and ready to go. 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:



You are now ready to start working on the project.

How to do it...

To implement separation of concerns for views 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 {  
    String description;  
    bool complete;  
  
    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 {  
    String name = '';  
    final List<Task> tasks = [];  
}
```

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 bloated 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 create a `StatelessWidget` that returns a `MaterialApp` that, in its home directory, calls a widget called `PlanScreen`. We will build this shortly:

```
import 'package:flutter/material.dart';
import './views/plan_screen.dart';

void main() => runApp(MasterPlanApp());

class MasterPlanApp extends StatelessWidget {
    @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 {
    @override
    State createState() => _PlanScreenState();
}

class _PlanScreenState extends State<PlanScreen> {
    final plan = Plan();

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(title: Text('Master Plan')),
            body: _buildList(),
            floatingActionButton: _buildAddTaskButton(),
        );
    }
}
```

6. This code won't compile because we're missing a few methods. 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: Icon(Icons.add),  
        onPressed: () {  
            setState(() {  
                plan.tasks.add(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]),  
    );  
}
```

8. We just need to return 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. Add the following code right after the `build list` method:

```
Widget _buildTaskTile(Task task) {  
    return ListTile(  
        leading: Checkbox(  
            value: task.complete,  
            onChanged: (selected) {  
                setState(() {  
                    task.complete = selected;  
                });  
            }),  
        title: TextField(  
            onChanged: (text) {  
                setState(() {  
                    task.description = text;  
                });  
            }),  
    ),
```

```
) ,  
);  
}
```

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. Add a scroll controller as a property of the `State` class, just after the `plan` property:

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

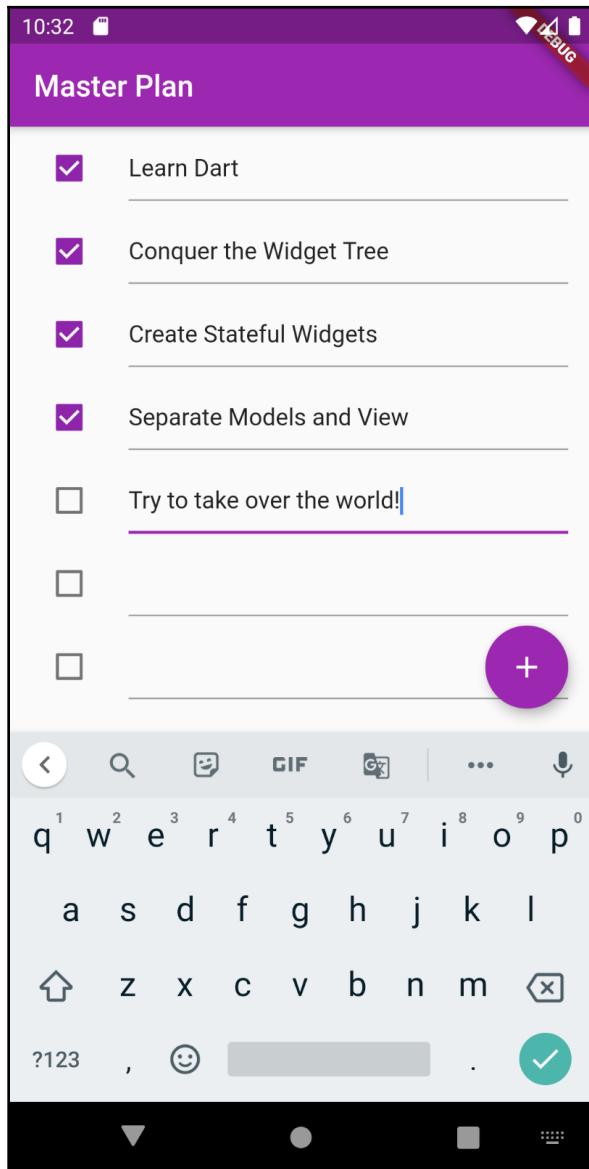
11. Add the controller to `ListView` in the `_buildList` method:

```
return ListView.builder(  
    controller: scrollController,
```

12. Finally, dispose of `scrollController` when the widget is removed from the tree:

```
@override  
void dispose() {  
    scrollController.dispose();  
    super.dispose();  
}
```

13. Hot restart (not reload) your app. You should see our plan coming together:



How it works...

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(  
    value: task.complete,  
    onChanged: (selected) {  
        setState(() {  
            task.complete = selected;  
        });  
    },
```

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 `onTapped` and `onChanged` callbacks, 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.



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. We will start exploring these components in the next recipe.

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. 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," which is a solid development pattern.

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>.
- **FocusManager**: <https://api.flutter.dev/flutter/widgets/FocusManager-class.html>.

Managing the data layer with InheritedWidget

How should you call the data classes in your app?

You could, in theory, set up a place in static memory where all your data classes will reside, but that won't play well with tools such as Hot Reload and could even introduce some undefined behavior down the road. The better 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 concerned with rendering widgets onto the screen; the only difference is that one can change and the other cannot. `InheritedWidget` is another beast entirely. 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 `InheritedWidget` 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 `InheritedWidget`:

```
import 'package:flutter/material.dart';
import './models/data_layer.dart';

class PlanProvider extends InheritedWidget {
    final _plan = Plan();

    PlanProvider({Key key, Widget child}) : super(key: key, child: child);

    @override
    bool updateShouldNotify(InheritedWidget oldWidget) => false;
}
```

2. To make the data accessible from anywhere in the app, we need to create our first *of-context* method. Add a static `Plan of` method that takes a `BuildContext` just after `updateShouldNotify`:

```
static Plan of(BuildContext context) {
    final provider =
```

```
context.dependOnInheritedWidgetOfExactType<PlanProvider>() ;  
    return provider._plan;  
}
```

3. 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 with a new PlanProvider class. Don't forget to fix any broken imports if needed:

```
return MaterialApp(  
    theme: ThemeData(primarySwatch: Colors.purple),  
    home: PlanProvider(child: PlanScreen()),  
) ;
```

4. Add two new get methods to the plan.dart file. These will be used to show the progress on every plan. Call the first one completeCount and the second completenessMessage:

```
int get completeCount => tasks  
    .where((task) => task.complete)  
    .length;  
  
String get completenessMessage =>  
    '$completeCount out of ${tasks.length} tasks';
```

5. 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).

6. To fix the errors that were raised in the previous step, add PlanProvider.of(context) to the _buildAddTaskButton and _buildList methods:

```
Widget _buildAddTaskButton() {  
    final plan = PlanProvider.of(context);  
    Widget _buildList() {  
        final plan = PlanProvider.of(context);
```

7. 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.

8. Finally, add a SafeArea widget with completenessMessage at the end of Column. The final result is shown here:

```
@override  
Widget build(BuildContext context) {  
    final plan = PlanProvider.of(context);  
    return Scaffold(
```

```
        appBar: AppBar(title: Text('Master Plan')),
        body: Column(children: <Widget>[
            Expanded(child: _buildList()),
            SafeArea(child: Text(plan.completenessMessage))
        ]),
        floatingActionButton: _buildAddTaskButton());
    }
```

9. Change `TextField` in `_buildTaskTile` to a `TextField`, to make it easier to provide initial data:

```
TextField(
    initialValue: task.description,
    onFieldSubmitted: (text) {
        setState(() {
            task.description = text;
        });
    },
},
```

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.

How it works...

InheritedWidgets are some of the most fascinating widgets in the whole Flutter framework. Their job isn't to render anything on the screen, but to **pass data down to lower widgets in the tree**. Just like any other widget in Flutter, InheritedWidgets can also have child widgets.

Let's break down the first portion of the `PlanProvider` class:

```
class PlanProvider extends InheritedWidget {
    final _plans = <Plan>[];

    PlanProvider({Key key, Widget child}) : super(key: key, child: child);

    @override
    bool updateShouldNotify(InheritedWidget oldWidget) => false;
```

First, we define an object that will store the plans (`_plans`). Then, we define a default unnamed constructor, which takes in a `key` and a `child`, and passes them to the superclass (`super`).

`InheritedWidget` is an abstract class, so you must implement the `updateShouldNotify` method. Flutter calls this method whenever the widget is rebuilt. In the `updateShouldNotify` method, you can look at the content of the old widget and determine if the child widgets need to be notified that the data has changed. In our case, we just return `false` and opt-out of this functionality. In most cases, it is rather unlikely that you need this method to return `true`.

Then, you must create your own implementation of the **of-context** pattern:

```
static Plan of(BuildContext context) {
  final provider = context.dependOnInheritedWidgetOfExactType
    <PlanProvider>();
  return provider._plan;
}
```

Here, you are using the context's `dependOnInheritedWidgetOfExactType` method to kick off the tree traversal process. Flutter will start from the widget that owns this context and travel upward until it finds a `PlanProvider`.

An interesting side effect of this method is that after it is called, the originating widget is registered as a dependency. This now creates a hard link between the child widget and `PlanProvider`. The next time this method is called, there is no need to travel up the tree anymore; the child already knows where the data is and can retrieve it immediately. This optimization makes it extremely fast, if not almost instant, to get the data from `InheritedWidgets`, no matter how deep the tree goes.

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, which originally came from React, 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, works perfectly for a single screen, but it is not ideal when you add a second. The higher in the tree your state object is stored, the easier it is for your children widgets to access it.

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 storage property from a single `plan` to a list of `plans`:

```
final _plans = <Plan>[];
```

2. We also need to update the `of-context` method so that it returns the correct type. This will temporarily break the project, but we will fix this in the next few steps:

```
static List<Plan> of(BuildContext context) {  
    final provider = context.dependOnInheritedWidgetOfExactType  
        <PlanProvider>();  
    return provider._plans;  
}
```

3. PlanProvider is also going to have a new home in the widget tree. Instead of sitting ***underneath*** MaterialApp, we actually want this global state widget to be placed ***above*** it. Update the build method in main.dart so that it looks like this:

```
return PlanProvider(  
    child: MaterialApp(  
        theme: ThemeData(primarySwatch: Colors.purple),
```

4. 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: PlanCreatorScreen(),
```

5. 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();  
}
```

6. 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: Text('Master Plans')),  
        body: Column(children: <Widget>[  
            _buildListCreator(),  
            Expanded(child: _buildMasterPlans())  
        ]),  
    );  
}
```

7. 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: InputDecoration(  
                    labelText: 'Add a plan',  
                    contentPadding: EdgeInsets.all(20)),  
                onEditingComplete: addPlan),  
        ));  
}
```

8. 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;  
    PlanProvider.of(context).add(plan);  
    textController.clear();  
    FocusScope.of(context).requestFocus(FocusNode());  
    setState(() {});  
}
```

9. 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() {  
    final plans = PlanProvider.of(context);  
  
    if (plans.isEmpty) {  
        return Column(  
            mainAxisAlignment: MainAxisAlignment.center,  
            children: <Widget>[  
                Icon(Icons.note, size: 100, color: Colors.grey),  
            ]);  
    }  
    else {  
        return ListView.builder(  
            itemCount: plans.length,  
            itemBuilder: (context, index) {  
                final plan = plans[index];  
                return Card(  
                    elevation: 5,  
                    child: ListTile(  
                        title: Text(plan.name),  
                        subtitle: Text(plan.description),  
                        trailing: IconButton(  
                            icon: Icon(Icons.delete),  
                            onPressed: () {  
                                plans.removeAt(index);  
                                PlanProvider.of(context).update(plans);  
                            }));  
                );  
            });  
    }  
}
```

```
        Text('You do not have any plans yet.',  
            style: Theme.of(context).textTheme.headline5)  
    ]);  
}  
  
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)));  
            };  
        );  
    };  
}
```

10. `PlanScreen` is going to need some small tweaks as well. We need to add a constructor where the specific plan can be injected and then update the build methods to read that value. Add this property and constructor to the widget in `plan_screen.dart`:

```
final Plan plan;  
const PlanScreen({Key key, this.plan}) : super(key: key);
```

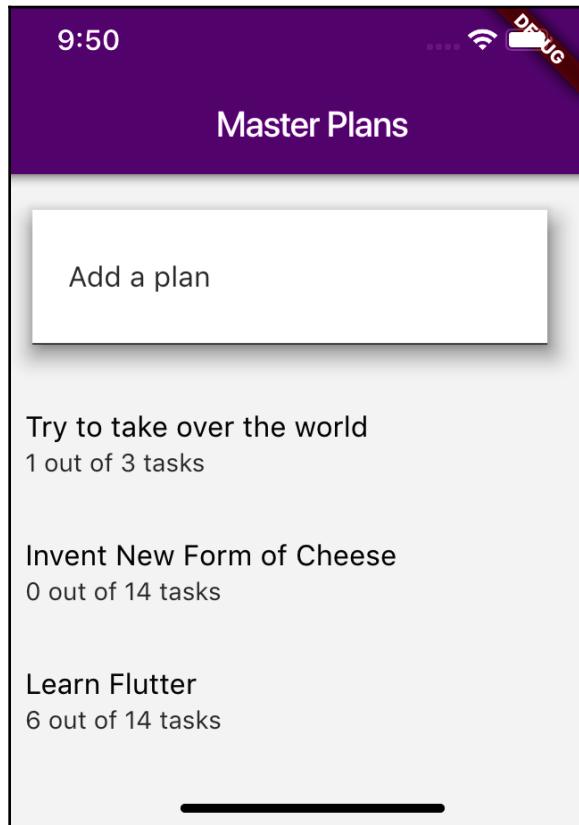
11. Finally, we just need to add an easy way to access the widget. Add this getter inside the state class:

```
Plan get plan => widget.plan;
```

12. Remove all the previous references to `PlanProvider`. You will need to skim through the class and **delete** this line everywhere it appears:

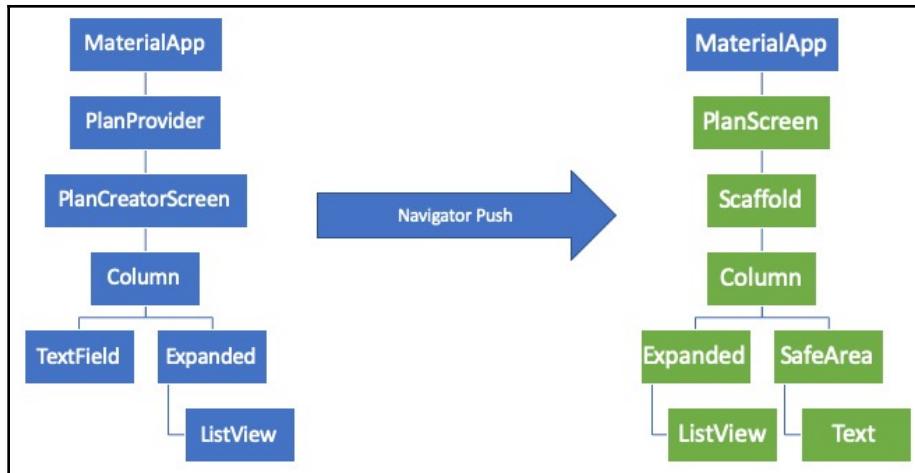
```
final plan = PlanProvider.of(context);
```

When you hot restart the app, you will be able to create multiple plans with different lists on each screen:



How it works...

The main takeaway from this recipe is the importance of proper widget tree construction. When you push a new route onto Navigator, you are essentially replacing every widget that lives underneath MaterialApp, as explained in this diagram:



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 automatically updated, like magic. This is one of the primary benefits of maintaining a clean architecture in our apps.

Designing an n-tier architecture, part 1 – controllers

There are many architectural patterns that have become popular in the last couple of years – **Model View Controller (MVC)**, **Model View ViewModel (MVVM)**, **Model View Presenter (MVP)**, Coordinator, and several others. These patterns have become so numerous that they are sometimes pejoratively referred to as `MV*` patterns. This essentially means that no matter which pattern you choose, you need some kind of intermediary object between your model and your view.

A concept that is shared by all the popular patterns is the idea of **tiers/layers** (we will be using the terms tier and layer interchangeably throughout this chapter). Each **tier** in your app is a section of the MV* classes that have a single responsibility. The term *n-tier* (sometimes called a multi-tier architecture) just means that you are not limited on the number of tiers in your app. You can have as many or as few as you need.

The top-most tier is one that you are already familiar with – the view/widget tier. This tier is only interested in setting up the user interface. All the data and logic for the app should be delegated to a lower tier. The first tier that typically sits underneath the view tier is the **controller** layer. These classes are responsible for handling business logic and providing a link between the views and the lower layers in our app.

In this recipe, we'll be moving the business logic for the Master Plan app from the view to a new controller class. We will also be adding the ability to **delete** notes from the list.

Getting ready

You should have completed the previous recipes in this chapter before following along with this one.

How to do it...

Let's start building an *n-tier* architecture, starting with the controller layer:

1. Since our controllers are supposed to represent a separate tier in the app, it's appropriate to put them in their own folder. Inside the `lib` directory, create a new folder called `controllers`.
2. Now, we can create a new dart file, `plan_controller.dart`. This class will be responsible for all the business logic in our app. Let's start with the class declaration:

```
import '../models/data_layer.dart';

class PlanController {
    final _plans = <Plan>[];

    // This public getter cannot be modified by any other object
    List<Plan> get plans => List.unmodifiable(_plans);
}
```

- Now, let's add the methods that will be responsible for creating and deleting plans. This is also an appropriate location where we can apply some business logic.

First, create a private method that will check a list of items and search for duplicate names. If it finds any, a number will be appended to the end to make sure the name is unique. Add this method to the `PlanController` class:

```
String _checkForDuplicates(Iterable<String> items, String text) {  
    final duplicatedCount = items  
        .where((item) => item.contains(text))  
        .length;  
    if (duplicatedCount > 0) {  
        text += ' ${duplicatedCount + 1}';  
    }  
    return text;  
}
```

- We can use our new business logic to check the input for new plans. Add the method to create a new plan, right after the public getter for the `plans` property and before the `_checkForDuplicates` method:

```
void addNewPlan(String name) {  
    if (name.isEmpty) {  
        return;  
    }  
  
    name = _checkForDuplicates(_plans.map((plan) => plan.name),  
        name);  
  
    final plan = Plan()..name = name;  
    _plans.add(plan);  
}
```

- Under the `addNewPlan` method, add the method for deleting a plan:

```
void deletePlan(Plan plan) {  
    _plans.remove(plan);  
}
```

- We can add similar methods for creating and deleting tasks inside the plan. Add the following method to add a new Task under the `deletePlan` method:

```
void createNewTask(Plan plan, [String description]) {  
    if (description == null || description.isEmpty) {  
        description = 'New Task';  
    }
```

```
        description = _checkForDuplicates(
            plan.tasks.map((task) => task.description), description);

        final Task task = Task()..description = description;
        plan.tasks.add(task);
    }
```

7. Add the method for deleting a task under the `createNewTask` method:

```
void deleteTask(Plan plan, Task task) {
    plan.tasks.remove(task);
}
```

8. With `PlanController` completed, we can now integrate it with the Flutter layer. Since we're going to follow proper separation of concerns, the only place `PlanController` is allowed to be instantiated is in the `PlanProvider` class. We need to update that class so that it can hold a `PlanController` instead of maintaining its own list. Also, update the property of the **of-context** method so that it returns the correct types (this will break the app, but don't worry – you'll fix this in the next few steps):

```
class PlanProvider extends InheritedWidget {
    final _controller = PlanController();
    /*...code elipted...*/

    static PlanController of(BuildContext context) {
        PlanProvider provider =
            context.dependOnInheritedWidgetOfExactType<PlanProvider>();
        return provider._controller;
    }
}
```

9. The previous step will have created some temporary compile errors that you can now address. Open `PlanCreatorScreen` and edit the `buildMasterPlans` method, as follows:

```
Widget _buildMasterPlans() {
    final plans = PlanProvider.of(context).plans;
```

10. Edit the `addPlan` method by removing the business logic from the view, as shown here:

```
void addPlan() {
    final text = textController.text;

    // All the business logic has been removed from this 'view'
    // method!
```

```
    final controller = PlanProvider.of(context);
    controller.addNewPlan(text);

    textController.clear();
    FocusScope.of(context).requestFocus(FocusNode());
    setState(() {});
}
```

11. Now that the errors have been addressed, we can add a feature for deleting plans. The business logic for this feature already exists. All we need to do is add a widget that can invoke the correct method in `PlanController`. You can wrap `ListTiles` in a `Dismissible` widget to create a nice swipe-to-delete gesture. Wrap `ListTile` in the `_buildMasterPlans()` method in a new widget and add the following code:

```
return Dismissible(
  key: ValueKey(plan),
  background: Container(color: Colors.red),
  direction: DismissDirection.endToStart,
  onDismissed: (_) {
    final controller = PlanProvider.of(context);
    controller.deletePlan(plan);
    setState(() {});
  },
  child: ListTile(...),
)
```

12. Hot reload the app. You can now create and delete plans.
13. The same updates can also be applied to the `PlanScreen` class. Make the same changes you made in the `PlanScreen` class to remove all traces of business logic from the view. In the `_buildAddTaskButton()` method, update the `onPressed` closure to create tasks via the controller:

```
onPressed: () {
  final controller = PlanProvider.of(context);
  controller.createNewTask(plan);
  setState(() {});
},
```

14. You can also use the same swipe-to-dismiss experience to delete tasks. Wrap `ListTile` in the `_buildTaskTile` method with another `Dismissible` widget:

```
Widget _buildTaskTile(Task task) {
  return Dismissible(
    key: ValueKey(task),
```

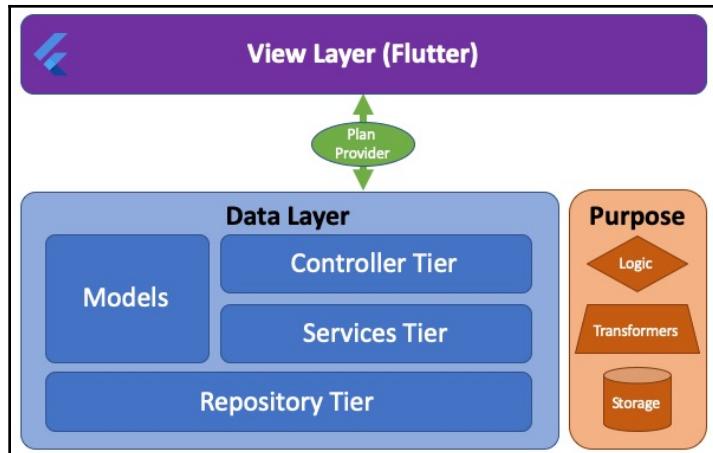
```
background: Container(color: Colors.red),  
direction: DismissDirection.endToStart,  
onDismissed: (_) {  
    final controller = PlanProvider.of(context);  
    controller.deleteTask(plan, task);  
    setState(() {});  
,  
child: ListTile(...),  
);  
}
```

15. Did you also notice that we get an annoying bug every time we hot reload? Every time we hot reload the app, all our lists disappear. This won't impact the final app, but it makes development more challenging than it should be. The solution to this bug is to simply lift the state up even higher. Remove PlanProvider from inside the build method of MasterPlanApp and add it as the topmost widget in the entire tree:

```
void main() => runApp(PlanProvider(child: MasterPlanApp()));
```

How it works...

The design that we will be aiming for by the end of this chapter can be summarized with this diagram:



We are beginning to divide the data layer into multiple components that will comprise the n-tier architecture – controllers, services, and repositories. While this diagram shows the full design, we will only be covering one tier at a time. In this recipe, we focused on the controller tier, which can communicate with the view layer via the `PlanProvider` interface.

To understand an n-tier architecture, it's helpful to think of your app as a cake. The topmost layer of the cake, where the icing and cherries are found, is known as the **view**. That's the first thing people see when they look at your cake.

Immediately underneath the view is where you put your **controllers**. For the purposes of this design, the job of the controllers is to **process business logic**. Business logic is defined as any rule in your app that is not related to presentation (view) or persistence (databases, web services, and so on). The Master Plan app is pretty thin on business logic, but one key functionality that we added was a check to make sure that duplicates aren't created. This procedure has nothing to do with the user interface; subsequently, it would be confusing to place it there. The correct place to store the `_checkForDuplicates` method is in the controller. You were able to reuse this exact same method for both plans and tasks. If this was placed inside the widgets, we would have to either write this code twice, once for each widget, or conjure some contrived inheritance structure. Either way, giving each class type a "job" and making sure that these roles are respected thins out the widgets and allows us to focus on one task at a time.

See also

The following resources explain the tiered architecture approach that we are striving for:

- What is Multi-Layered Software Architecture?: <https://hub.packtpub.com/what-is-multi-layered-software-architecture/>
- Multitiered architecture: https://en.wikipedia.org/wiki/Multitier_architecture
- Business logic: https://en.wikipedia.org/wiki/Business_logic

Designing an n-tier architecture, part 2 – repositories

The next stage of the *n*-tier architecture we are going to discuss in this recipe is the bottom-most layer: the **repositories**, or the *data layer*. The purpose of a repository is to store and retrieve data. This layer can be implemented as a database, web service, or in the case of the Master Plan project, a simple in-memory cache. Unlike the *controller* layer, which is business logic-aware, the repository layer is only concerned with getting and storing data in its most abstract form. These classes should not even know about the **model** files that we created earlier.

The reason why repositories are so purposefully ignorant is to keep them focused entirely on their task – persistence. Communicating with a database or a web service can become complicated if you have many small requirements. These concerns are typically beneath business logic and are easier to resolve when you're only focused on abstract objects.

Remember, the whole goal of an n-tier architecture is to strictly separate responsibilities; we let repositories do what they do best – store data – and let the higher layers handle the rest.

Getting ready

You should have completed the previous recipes in this chapter before following along with this one.

Get started with this tier by creating a new folder called `repositories` that will hold the code for this recipe

How to do it...

Let's define a repository interface and then implement a version of that interface as an in-memory cache:

1. In the `repositories` folder, create a new file called `repository.dart` and add the following interface:

```
import 'package:flutter/foundation.dart';

abstract class Repository {
    Model create();

    List<Model> getAll();
```

```
    Model get(int id);
    void update(Model item);

    void delete(Model item);
    void clear();
}
```

2. We also need to define a temporary storage class called `Model` that can be used in any implementation of our repository interface. Since this model is strongly coupled to the repository concept, we can add it to the same file:

```
class Model {
    final int id;
    final Map data;

    const Model({
        @required this.id,
        this.data = const {},
    });
}
```

3. The repository interface can be implemented in several ways, but for the sake of simplicity, we are just going to implement an *in-memory cache*. In the `repositories` folder, create a new file called `in_memory_cache.dart` and add the `InMemoryCache` class, which implements the `Repository` interface. This class will hold a private `Map` called `_storage` that will keep all the data:

```
import 'repository.dart';

class InMemoryCache implements Repository {
    final _storage = Map<int, Model>();
}
```



Once you've written the class declaration, you can use Android Studio/VS Code intentions dialog to automatically generate placeholders for all the required methods.

4. Now, we need to implement all the required functions from the repository interface. The most complex method is the `create` function, which needs to generate a unique identifier for every element in the storage:

```
@override
Model create() {
    final ids = _storage.keys.toList()..sort();
    final id = (ids.length == 0) ? 1 : ids.last + 1;
```

```
    final model = Model(id: id);
    _storage[id] = model;
    return model;
}
```

5. The remaining methods are simple wrappers of the map API. Write this code immediately after the `create` method:

```
@override
Model get(int id) {
    return _storage[id];
}

@Override
List<Model> getAll() {
    return _storage.values.toList(growable: false);
}

@Override
void update(Model item) {
    _storage[item.id] = item;
}

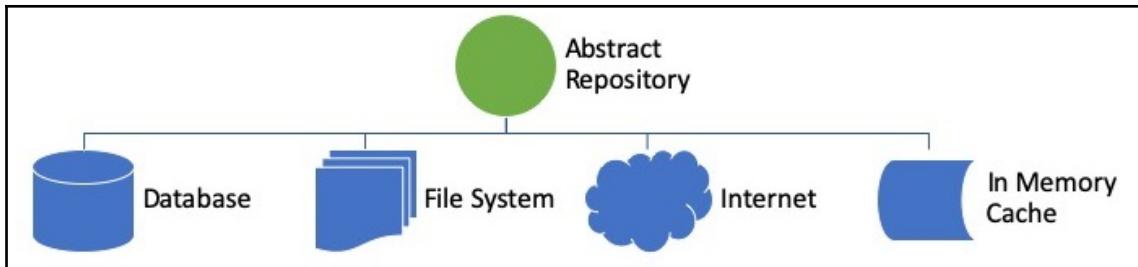
@Override
void delete(Model item) {
    _storage.remove(item.id);
}

@Override
void clear() {
    _storage.clear();
}
```

That completes the repository tier for this project. However, we won't be able to see it in action until we complete the next recipe on services.

How it works...

In this recipe, we chose to implement the repository tier as an **abstract** interface. Designing this sort of structure allows you to, in theory, have multiple implementations of the repository interface that can be swapped in and out effortlessly. The following diagram shows four of the most popular types of repositories that we could have configured:



In fact, this same abstract class could be used to store data in a database, in one or more files, or in a RESTful web service. In this recipe, we chose to implement the simplest option, `InMemoryCache`, which is nothing more than a fancy `Map`. The interesting thing about this class is that it represents a unified interface that can be used regardless of the actual destination of the data.

The higher layers in the n-tier architecture do not need to know *how* the data is stored, just that it is stored. Everything else, from their perspective, is an implementation detail.

Another requirement of the repository layer is that it cannot know anything about any domain-specific models that we have defined for our project, namely the `Plan` and `Task` models. Instead, we defined a transient `Model` class, which is just a map with an `id` property. The `id` property is used to retrieve the object from storage. It is critical that this value is unique; otherwise, your data will be overwritten.

We naively implemented a unique `id` formula in the `create` method:

```
final ids = _storage.keys.toList()..sort();
final id = (ids.length == 0) ? 1 : ids.last + 1;
```

Keys in `Maps` are not stored in any particular order. To find out what the largest value is, we need to sort all the keys in the map. Once we've retrieved the largest number, the function will just return that value incremented by one. This only works because the repository layer controls the `id` property. It should be a read-only value for any layer higher than the repository.

Designing an n-tier architecture, part 3 – services

The final tier that we will be talking about in this exploration of the n-tier architecture is the **services** tier. This tier serves as the glue between the controllers and repositories. Its primary job is to transform the data from the generic format used by the storage solution into the actual schema that is used by the controllers and the user interface.

In this recipe, we will be creating serialization and deserialization functions for our models, as well as stitching it all together with a services class.

How to do it...

This recipe will be divided into two components – serialization and integration. We're going to start with the serialization functions and then snap together all the pieces that we built over the last three recipes:

1. Open `task.dart` and add an `id` property and a default constructor. This will allow the `Task` model to be transformed into a generic `Model`:

```
import 'package:flutter/foundation.dart';
import '../repositories/repository.dart';

class Task {
    final int id;
    String description;
    bool complete;

    Task({@required this.id, this.complete = false, this.description
        = ''});
}
```

2. Now, we need to add the serialization and deserialization methods. These functions will take the data from a generic `Model` and return a more usable strongly typed object. Add the following code immediately after the constructor.

```
Task.fromModel(Model model)
: id = model.id,
description = model.data['description'],
complete = model.data['complete'];

Model toModel() =>
```

```
    Model(id: id, data: {'description': description, 'complete': complete});
```

3. Now, open `plan.dart` and add an immutable `id` property at the top of the class. We will also need to add the requisite constructor:

```
import 'package:flutter/foundation.dart';
import '../repositories/repository.dart';

final int id;
List<Task> tasks = [];
Plan({@required this.id, this.name = ''});
```

4. At the bottom of the class, add a deserialization constructor and a serialization method:

```
Plan.fromModel(Model model)
: id = model.id,
  name = model?.data['name'],
  tasks = model?.data['task']
    ?.map<Task>((task) => Task.fromModel(task))
    ?.toList() ?? <Task>[];

Model toModel() => Model(id: id, data: {
  'name': name,
  'tasks': tasks.map((task) => task.toModel()).toList()
});
```

5. Now, we can turn our attention to the services tier. As with all the other tiers in the n-tier architecture, we will need to create a folder. Create a new folder called `services` inside the `lib` folder.
6. Create a new file called `plan_services.dart`.
7. In the `plan_services.dart` file, instantiate a repository as a private property of the class:

```
import '../repositories/in_memory_cache.dart';
import '../repositories/repository.dart';
import '../models/data_layer.dart';

class PlanServices {
  final Repository _repository = InMemoryCache();
}
```

8. All the hard work of transforming plans has already been completed in the models. Now, we just need to expose the **Create, Read, Update, and Delete (CRUD)** methods in the services class. Add these four methods after the repository property:

```
Plan createPlan(String name) {
    final model = _repository.create();
    final plan = Plan.fromModel(model)..name = name;
    savePlan(plan);
    return plan;
}

void savePlan(Plan plan) {
    _repository.update(plan.toModel());
}

void delete(Plan plan) {
    _repository.delete(plan.toModel());
}

List<Plan> getAllPlans() {
    return _repository
        .getAll()
        .map<Plan>((model) => Plan.fromModel(model))
        .toList();
}
```

9. Integrating Tasks into the services layer is a bit easier. These objects are strongly coupled to their parent Plan objects. We just want to ensure that the tasks always have a unique id. Insert the following code after the plan CRUD methods:

```
void addTask(Plan plan, String description) {
    final id = plan.tasks.last?.id ?? 0 + 1;
    final task = Task(id: id, description: description);
    plan.tasks.add(task);
    savePlan(plan);
}

void deleteTask(Plan plan, Task task) {
    plan.tasks.remove(task);
    savePlan(plan);
}
```

10. Now, we need to tie the system into our existing functionality in `PlanController`. Remove the controller's private `_plan` property and replace it with an instance of the `PlanServices` class:

```
final services = PlanServices();
```

11. There will be few compile errors that need to be addressed. Go through all the red underlines and replace the code with the appropriate API in the services class. Start with the public `plan` getter and the `addNewPlan` method:

```
List<Plan> get plans => List.unmodifiable(services.getAllPlans());  
void addNewPlan(String name) {  
    if (name.isEmpty) {  
        return;  
    }  
  
    name = _checkForDuplicates(plans.map((plan) => plan.name), name);  
    services.createPlan(name);  
}
```

12. The `savePlan` and `deletePlan` methods can also be updated to simply delegate their functionality to the services tier. Update this code right after the `addNewPlan` method:

```
void savePlan(Plan plan) {  
    services.savePlan(plan);  
}  
  
void deletePlan(Plan plan) {  
    services.delete(plan);  
}
```

13. Wiring up the `Task` functionality is very similar to what we achieved for `Plans`. Simply replace the red errors that your IDE has highlighted with the appropriate method in the services class. The code that explicitly creates the task should be removed since that job is now being handled by a lower tier. To do this, update the `createNewTask` and `deleteTask` functions with the following code:

```
void createNewTask(Plan plan, [String description]) {  
    if (description == null || description.isEmpty) {  
        description = 'New Task';  
    }  
  
    description = _checkForDuplicates(  
        plan.tasks.map((task) => task.description), description);
```

```
        services.addTask(plan, description);
    }

    void deleteTask(Plan plan, Task task) {
        services.deleteTask(plan, task);
    }
}
```

14. There is one final change that needs to be made to the UI layer. When a user dismisses a `PlanScreen`, the data should be synchronized with our storage solution. Flutter has a widget called `WillPopScope` that allows you to run arbitrary code when routes are dismissed. Open `plan_screen.dart` and wrap the screen's `Scaffold` in a new widget. Then, include the following closure:

```
return WillPopScope(
    onWillPop: () {
        final controller = PlanProvider.of(context);
        controller.savePlan(plan);
        return Future.value(true);
    },
    child: Scaffold(...));
}
```

This concludes our **MasterPlan** app. Perform a hot reload and take a moment to play with the app. This architecture will allow you to accomplish great things in the future!

How it works...

This recipe is mostly about piping. The `service` class pipes the data from the controller down to the repository. This is a relatively simple job, which means that the `service` classes should be simple to understand. There is no business logic in these classes and they are not responsible for maintaining state; it's just about unidirectionally transforming data.

When we define the class, it's also important to note that we're referencing the repository layer by its abstract interface:

```
final Repository _repository = InMemoryCache();
```

By explicitly declaring this property as a `Repository`, we are saying that it doesn't matter that we're using an `InMemoryCache`. This repository could just as easily be a database connector or an HTTP client and it wouldn't change much. This style of coding is known as coding toward an interface instead of an implementation. It is usually preferable to reference your properties in this way when you have many modular components that you want to be able to swap in and out easily.

The primary job of the services tier, as we described in the *Designing an n-tier architecture, part 1 – controllers* recipe, is to transform data. This process can be broken down into two categories:

- Serialization
- Deserialization

Serialization is defined as the process of taking your data and transforming it into a type that's more appropriate for transportation. This could be a byte stream, JSON, XML, or in the case of this recipe, a Model. Serialization methods are typically named by prefacing them with the word *to* and then listing the type; for example, `toJson`, `toXml`, or `toModel`. We serialized the task model with the following function:

```
Model toModel() =>
    Model(id: id, data: {'description': description, 'complete': complete});
```

This function takes the content to Task and generates a Map with key-value pairs representing the content.

The opposite process, *deserialization*, takes the data coming **from** the transient structure and instantiates a strongly typed model. Deserialization methods are often implemented as constructors to make the API easier to work with:

```
Task.fromModel(Model model)
: id = model.id,
  description = model.data['description'],
  complete = model.data['complete'];
```

When writing these methods, we have to make sure that the loosely-typed keys are identical in both the serialization and deserialization methods. If we make a typo or use the incorrect keys, then these functions would fail. Tasks only have two keys we need to worry about – `'description'` and `'complete'`. Even here, there is a large opportunity for errors. When writing these sort of functions, especially on more complex models, you need to carefully double-check your keys, since this is unfortunately not something the compiler can catch for you. The only way to notice that you have an error is to experience it at runtime.

There's more...

This recipe also used some Dart language features that you might be unfamiliar with – null-aware operators and null coalescing operators. Let's look more carefully at this line of code:

```
tasks = model?.data['task']  
    ?.map<Task>((task) => Task.fromModel(task))  
    ?.toList() ?? <Task>[];
```

When you insert null-aware operators, a `?` is inserted after any variable that might be `null`. Here, you are telling Dart that if this variable is `null`, then just skip everything after the question mark. In this example, the `model` property might be `null`, so if we tried to access the `data` property on a `null`, Dart would throw an exception. In this case, if a `null` is ever encountered, this code will be gracefully skipped.

The null coalescing operator, `??`, is used almost like a ternary statement. This operator will return the value on the right-hand side of the question marks if the value on the left-hand side is `null`. You can think of null coalescing operators as a short form for this kind of statement:

```
String nothing;  
String something = nothing == null ? 'Something' : nothing
```

Null coalescing operators are often used as a fullback in combination with null-aware operators. This guarantees that you will have a value and not a `null` at the end of your statement.

See also

Take a look at these resources if you wish to dig deeper into this topic:

- **Interface-based programming:** https://en.wikipedia.org/wiki/Interface-based_programming
- **Null-aware operators:** <http://blog.sethladd.com/2015/07/null-aware-operators-in-dart.html>

7

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 9, *Advanced State Management with Streams*, we will also focus on streams, which are an alternative way to deal with asynchronous programming.

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 topics:

- 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 `StatefulWidget`

- 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 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.
- Your favorite code editor: Android Studio, Visual Studio Code, and IntelliJ IDEA. are recommended. All should have the Flutter/Dart extensions installed.

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:

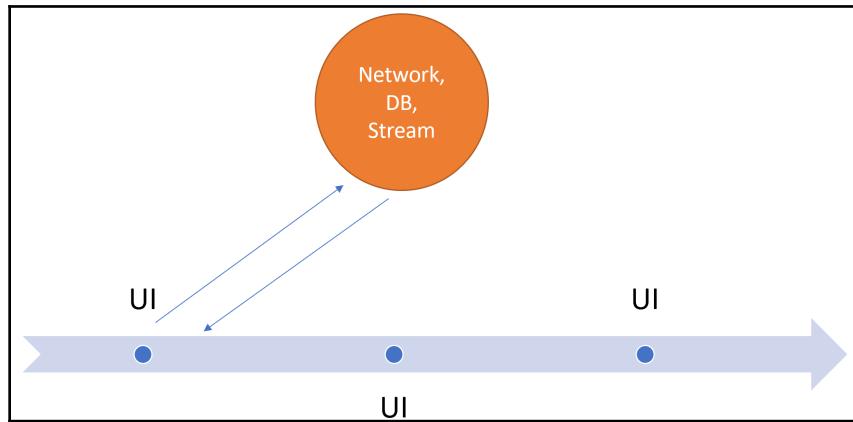
```
int x = 5;  
int y = x * 2;
```

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:



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 reactive 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, or reading data from a file on a device. Performing these tasks asynchronously will keep your app responsive.

In this recipe, you will create an app that reads some data from a web service using the `http` library. Specifically, we will read JSON data from the Google Books API.

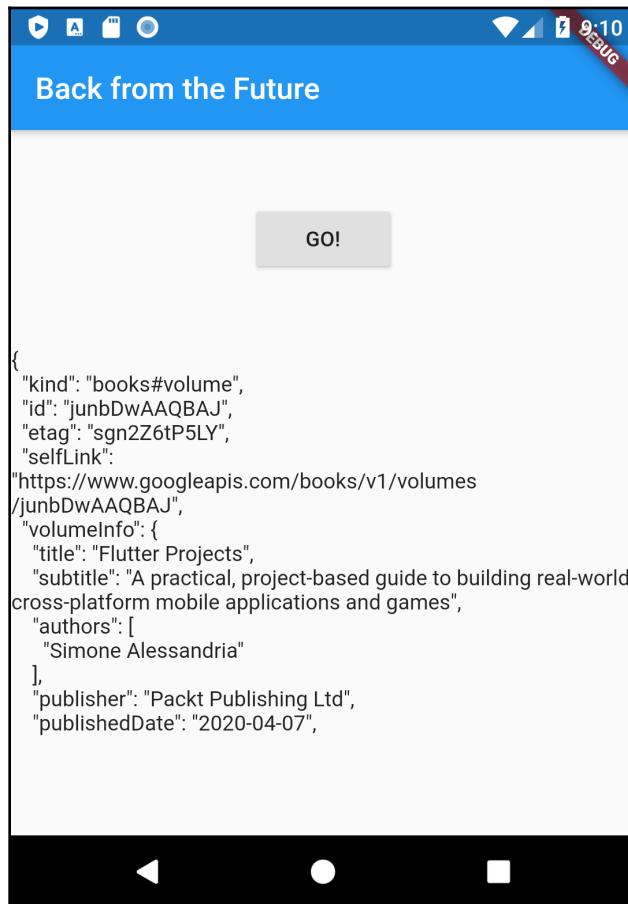
Getting ready

In order to follow along with this recipe, there are the following requirements:

- 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:



The steps required in order to retrieve and show the data using a Future are outlined here:

1. Create a new app and, in the `pubspec.yaml` file, add the `http` dependency (always make sure you are using the latest version, checking at <https://pub.dev/packages/http/install>):

```
dependencies:  
  flutter:
```

```
sdk: flutter
http: ^0.13.1
```

2. 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/tree/master/chapter_07. 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(MyApp());
}
class MyApp extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            title: 'Future Demo',
            theme: ThemeData(
                primarySwatch: Colors.blue,
                visualDensity: VisualDensity.adaptivePlatformDensity,
            ),
            home: FuturePage(),
        );
    }
}
class FuturePage extends StatefulWidget {
    @override
    _FuturePageState createState() => _FuturePageState();
}
class _FuturePageState extends State<FuturePage> {
    String result;
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                title: Text('Back from the Future'),
            ),
            body: Center(
                child: Column(children: [
                    Spacer(),
                    ElevatedButton(
                        child: Text('GO!'),
                        onPressed: () {
                            },
                    ),
                    Spacer(),
                    Text(result.toString()),
                    Spacer(),
                ],
            ),
        );
    }
}
```

```
        CircularProgressIndicator(),
        Spacer(),
    ],
),
),
);
}
}
```

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.

3. 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 {
    final String authority = 'www.googleapis.com';
    final String path = '/books/v1/volumes/junbDwAAQBAJ';
    Uri url = Uri.https(authority, path);
    return http.get(url);
}
```

4. 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: () {
        result = '';
        setState(() {
            result = result;
        });
        getData()
            .then((value) {
                result = value.body.toString().substring(0, 450);
                setState(() {
                    result = result;
                });
            })
            .catchError((_) {
                result = 'An error occurred';
                setState(() {
                    result = result;
                });
        });
    },
),
)
```

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` containing an integer number. 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 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`.



A **Uniform Resource Identifier (URI)** is a sequence of characters that universally identify a resource. Examples of URIs include a web address, an email address, 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.



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 in the main thread.
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>.

For Futures, in particular, I also recommend watching the official video 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 remove 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: 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.



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` 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 with async / await

Future<Response> getData() {    Future<Response> getData() {
  String url = https://myaddress.com';    String url = https://myaddress.com';
  return http.get(url);    return http.get(url);
}

void someMethod() {    Future someMethod() async {
  getData()    var value = await getData();
  .then((value) {    //do something with value
    //do something with value
  });
}

}
```

In the preceding code, you can see a comparison between the two approaches of dealing with Futures. Please note the following:

- The `then()` callback can be used in any method; `await` requires `async`.
- After the execution of the `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 use 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, there are the following requirements:

- 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/tree/master/chapter_07.

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<int>.delayed(const Duration(seconds: 3));
    return 1;
}

Future<int> returnTwoAsync() async {
    await Future<int>.delayed(const Duration(seconds: 3));
    return 2;
}

Future<int> returnThreeAsync() async {
    await Future<int>.delayed(const Duration(seconds: 3));
    return 3;
}
```

2. Under the three methods you just created, add the `count()` method leveraging the `async/await` pattern:

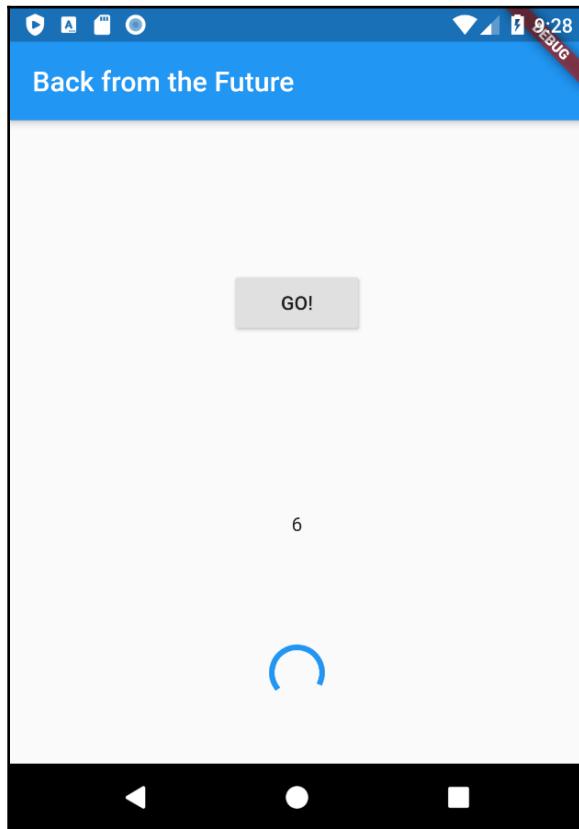
```
Future count() async {
    int total = 0;
    total = await returnOneAsync();
    total += await returnTwoAsync();
    total += await returnThreeAsync();
    setState(() {
        result = total.toString();
    });
}
```

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):



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. If you wanted to sum those numbers 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 very 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` will be 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. Add the following code in the `main.dart` file, in the `_FuturePageState` class:

```
Completer completer;
Future<int> getNumber() {
    completer = Completer<int>();
    calculate();
    return completer.future;
}
calculate() async {
    await new Future.delayed(const Duration(seconds : 5));
    completer.complete(42);
}
```

2. If you followed the previous recipes in the chapter, comment out the code in the `onPressed` function.
3. 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.

How it works...

A Completer creates Future objects that can be **completed** later. The Completer.future that's set in the getNumber method is the Future that will be completed once complete is called.

In the example in this recipe, 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.

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);
  }
  catch (_)
    completer.completeError(null);
}
```

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 added to your `pubspec.yaml`, and imported into your dart file as shown in the following code block:

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



Please note that `dart:async` and `async/async.dart` are different libraries: in many 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.



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;  
        value.forEach((element) {  
            total += element;  
        });  
        setState(() {  
            result = total.toString();  
        });  
    });  
}
```

2. In order to try this code, you can 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 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 add 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();  
    });  
});
```

```
});  
}
```

See also

FutureGroup is extremely useful and much less taken into account than it should be, so there's not much documentation for it at the time of writing. The official documentation page for this class is available at <https://api.flutter.dev/flutter/package-async-async/FutureGroup-class.html>.

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, you will need the following:

- 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/tree/master/chapter_07.

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:

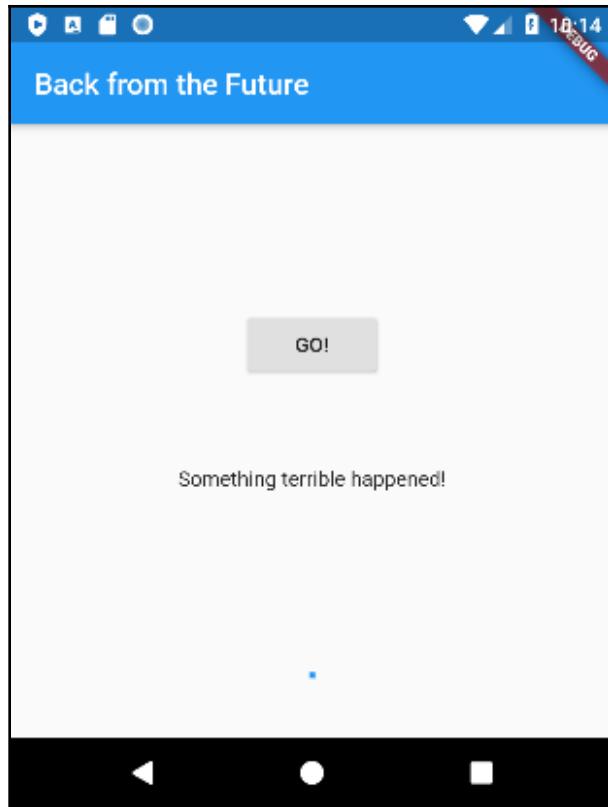
1. Add the following method to the `_FuturePageState` class in the `main.dart` file:

```
Future returnError() {  
    throw ('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;  
        });  
    })  
    .whenComplete(() => print('Complete'));
```

3. Run the app 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:



4. You should also see that the `whenComplete` callback was called as well. In **DEBUG CONSOLE**, you should see the `Complete` string:



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. Refactor the `handleError()` method as shown here:

```
Future handleError() async {  
  try {
```

```
        await returnError();
    }
    catch (error) {
        setState(() {
            result = error;
        });
    }
    finally {
        print('Complete');
    }
}
```

If you call `handleError()` in the `onPressed` method of the `ElevatedButton`, 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 Stateful Widgets

As mentioned previously, while Stateless widgets do not keep any state information, Stateful widgets can keep track of variables and properties, and in order to update the app, 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 Stateful widgets:

- `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 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/tree/master/chapter_07.
- For this recipe, we'll use the `geolocator` library, available at <https://pub.dev/packages/geolocator>. Add it to your `pubspec.yaml` file.

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 {
  @override
```

```
_LocationScreenState createState() => _LocationScreenState();  
}  
  
class _LocationScreenState extends State<LocationScreen> {  
  String myPosition = '';  
  @override  
  void initState() {  
    getPosition().then((Position myPos) {  
      myPosition = 'Latitude: ' + myPos.latitude.toString() + ' -'  
      'Longitude: ' + myPos.longitude.toString();  
      setState(() {  
        myPosition = myPosition;  
      });  
    });  
    super.initState();  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(title: Text('Current Location')),  
      body: Center(child: Text(myPosition)),  
    );  
  }  
  
  Future<Position> getPosition() async {  
    Position position = await  
    Geolocator().getLastKnownPosition(desiredAccuracy:  
    LocationAccuracy.high);  
    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.

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) {
    Widget myWidget;
    if (myPosition == '') {
        myWidget = CircularProgressIndicator();
    } else {
        myWidget = Text(myPosition);
    }
    return Scaffold(
        appBar: AppBar(title: Text('Current Location')),
        body: Center(child:myWidget),
    );
}
```



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<int>.delayed(const Duration(seconds: 3));`.

There is also an easier way to deal with Futures and stateful widgets: 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 complete the project in the previous recipe, *Using Futures with StatefulWidgets*, before starting this one.

How to do it...

To implement this, follow these steps:

1. Modify the `getPosition()` method. This will wait 3 seconds and then retrieve the current device's position (see the previous recipe for the complete code):

```
Future<Position> getPosition() async {
    await Future<int>.delayed(const Duration(seconds: 3));
    Position position = await
        Geolocator().getLastKnownPosition(desiredAccuracy:
            LocationAccuracy.high);
    return position;
}
```

2. 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: getPosition(),
            builder: (BuildContext context, AsyncSnapshot<dynamic>
                snapshot) {
                if (snapshot.connectionState ==
                    ConnectionState.waiting) {
                    return CircularProgressIndicator();
                }
                else if (snapshot.connectionState ==
                    ConnectionState.done) {
                    return Text(snapshot.data);
                }
                else {
                    return 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);  
}
```

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>.

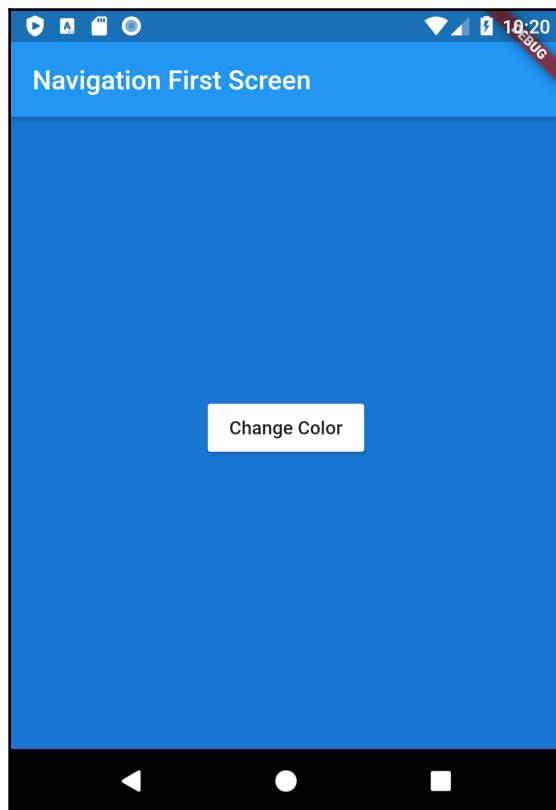
Turning navigation routes into asynchronous functions

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.

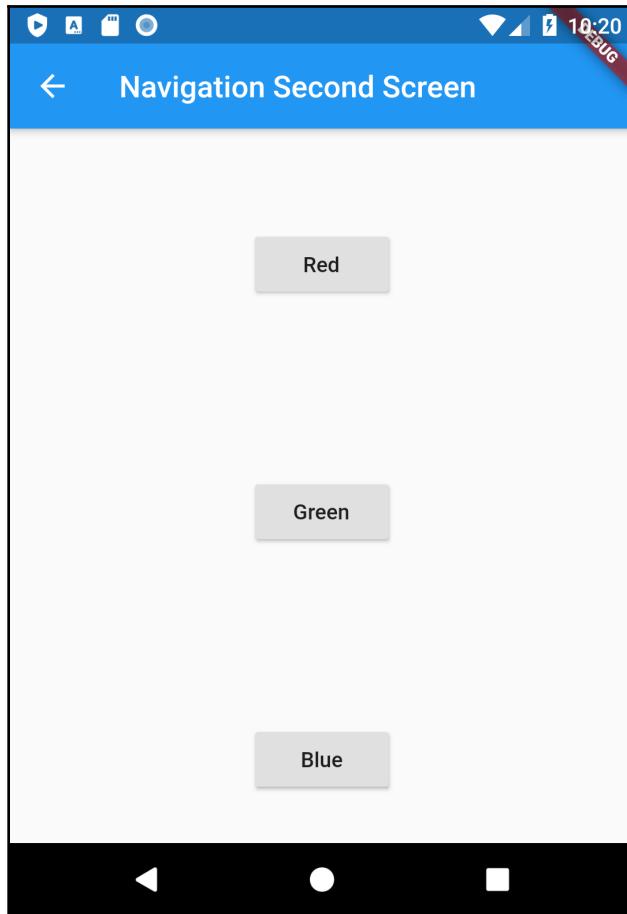
The steps we will follow are these:

- 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:



And here, the second screen, which allows choosing a color with three ElevatedButtons:



Getting ready

In order to follow along with this recipe, there is the following requirement:

- 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/tree/master/chapter_07.

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 {
    @override
    _NavigationFirstState createState() => _NavigationFirstState();
}

class _NavigationFirstState extends State<NavigationFirst> {
    Color color = Colors.blue[700];
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            backgroundColor: color,
            appBar: AppBar(
                title: Text('Navigation First Screen'),
            ),
            body: Center(
                child: ElevatedButton(
                    child: 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:

```
_navigateAndGetColor(BuildContext context) async {
    color = await Navigator.push(
        context,
        MaterialPageRoute(builder: (context) => NavigationSecond()),
    );
```

```
        setState(() {
            color = color;
        });
    }
}
```

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 {
    @override
    _NavigationSecondState createState() => _NavigationSecondState();
}

class _NavigationSecondState extends State<NavigationSecond> {
    @override
    Widget build(BuildContext context) {
        Color color;
        return Scaffold(
            appBar: AppBar(
                title: Text('Navigation Second Screen'),
            ),
            body: Center(
                child: Column(
                    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
                    children: <Widget>[
                        ElevatedButton(
                            child: Text('Red'),
                            onPressed: () {
                                color = Colors.red[700];
                                Navigator.pop(context, color);
                            }),
                        ElevatedButton(
                            child: Text('Green'),
                            onPressed: () {
                                color = Colors.green[700];
                                Navigator.pop(context, color);
                            }),
                        ElevatedButton(
                            child: Text('Blue'),
                            onPressed: () {
                                color = Colors.blue[700];
                                Navigator.pop(context, color);
                            })
                    ],
                )
            )
        );
    }
}
```

```
    } ,  
], , )); })}
```

7. In the `home` property of the `MaterialApp` in the `main.dart` method, call `NavigationFirst`:

```
home: 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

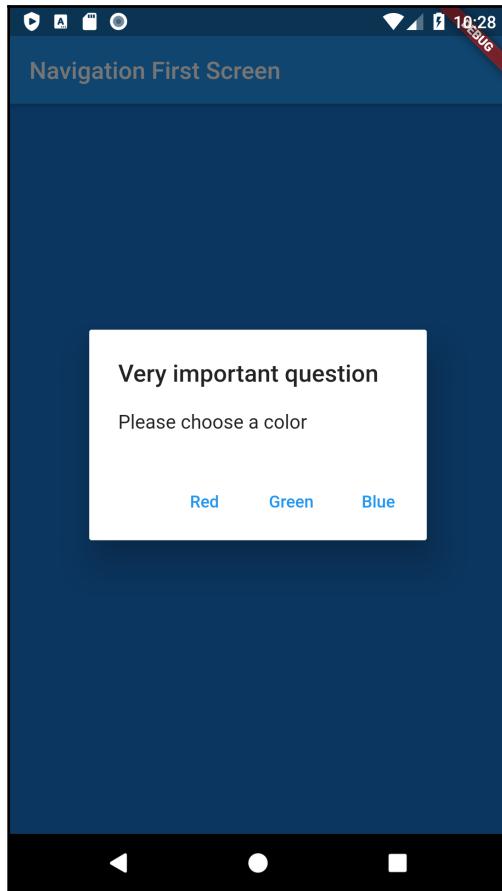
This recipe shows an alternative way of awaiting some data from another screen, which was shown in the previous recipe, but this time, instead of using a full-sized page, we will use a dialog box: actually dialogs behave just like routes that can be `await`-ed.

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, or 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:



Getting ready

In order to follow along with this recipe, there is the following requirement:

- The starting code for this recipe is available at https://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_07.

How to do it...

To implement this functionality, follow these steps:

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 NavigationDialog extends StatefulWidget {
    @override
    _NavigationDialogState createState() => _NavigationDialogState();
}

class _NavigationDialogState extends State<NavigationDialog> {
    Color color = Colors.blue[700];
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            backgroundColor: color,
            appBar: AppBar(
                title: Text('Navigation Dialog Screen'),
            ),
            body: Center(
                child: ElevatedButton(
                    child: 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 {
    color = null;
    await showDialog(
        barrierDismissible: false,
        context: context,
        builder: (_) {
            return AlertDialog(
                title: Text('Very important question'),
                content: Text('Please choose a color'),
                actions: <Widget>[
                    TextButton(
                        child: Text('Red'),
```

```
        onPressed: () {
            color = Colors.red[700];
            Navigator.pop(context, color);
        }),
    TextButton(
        child: Text('Green'),
        onPressed: () {
            color = Colors.green[700];
            Navigator.pop(context, color);
        }),
    TextButton(
        child: Text('Blue'),
        onPressed: () {
            color = Colors.blue[700];
            Navigator.pop(context, color);
        }),
),
),
),
);
);
setState(() {
    color = color;
});
}
}
```

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: 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).then((Color value) {  
         setState(() {  
             color = value;  
         });  
     });  
}
```

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>.

8

Data Persistence and Communicating with the Internet

Most applications, especially business applications, need to perform **CRUD** tasks: **Create**, **Read**, **Update**, or **Delete** data. In this chapter, we will cover recipes that explain how to perform CRUD operations in Flutter.

There are two ways that data can be persisted – locally and remotely. Regardless of the destination of your data, in many 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.

This will integrate with our previous discussion on Futures since interacting with data generally requires an asynchronous connection.

In this chapter, we will cover the following recipe:

- Converting Dart models into JSON
- Handling JSON schemas that are incompatible with your models
- Catching common JSON errors
- Saving data simply with Shared Preferences
- 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. 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/tree/master/chapter_08.

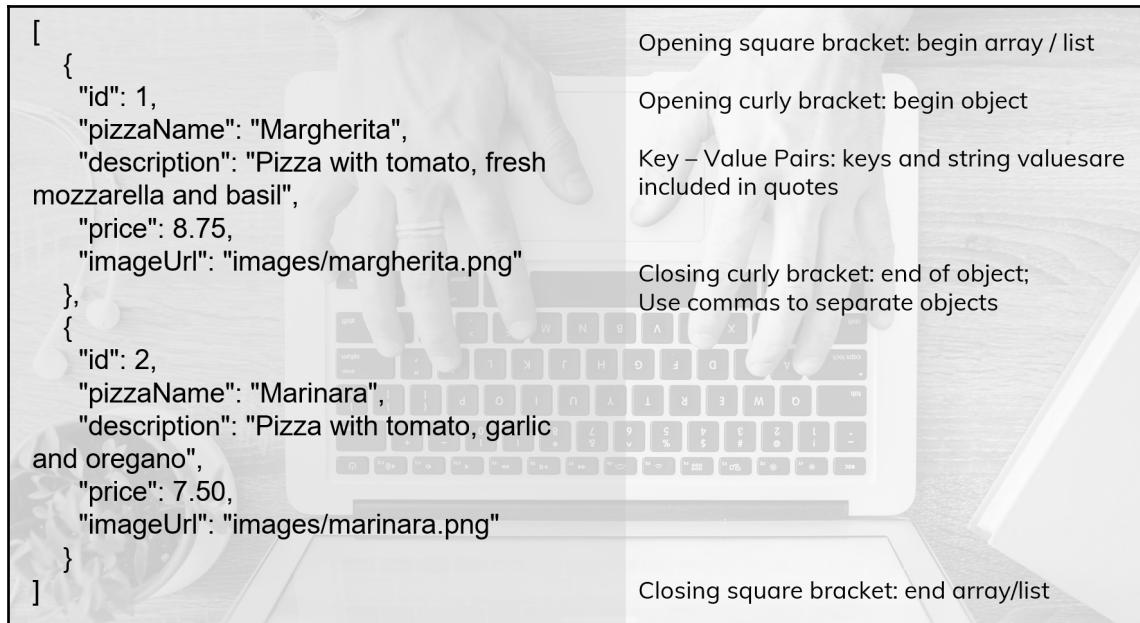
Converting Dart models into JSON

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:



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 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.

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://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_08.

How to do it...

For this recipe, we will create a new app that shows how to serialize and deserialize JSON Strings.

In your favorite editor, create a new Flutter project and call it `store_data`:

1. 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(MyApp());
}

class MyApp extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            title: 'Flutter JSON Demo',
            theme: ThemeData(
                primarySwatch: Colors.blue,
                visualDensity: VisualDensity
                    .adaptivePlatformDensity,
            ),
            home: MyHomePage(),
        );
    }
}
class MyHomePage extends StatefulWidget {
    @override
    _MyHomePageState createState() => _MyHomePageState();
}
class _MyHomePageState extends State<MyHomePage> {
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(title: Text('JSON')),
            body: Container(),
        );
    }
}
```

2. Add a new folder to the root of your project called `assets`.

3. 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.
4. In the `pubspec.yaml` file, add a reference to the new asset folder, as shown here:

```
assets:  
- assets/
```

5. In the `_MyHomePageState` class, in `main.dart`, add a state variable called `pizzaString`:

```
String pizzaString = '';
```

6. 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` that will set the `pizzaString` value, as shown here:

```
Future readJsonFile() async {  
    String myString = await DefaultAssetBundle.of(context)  
        .loadString('assets/pizzalist.json');  
    setState(() {  
        pizzaString = myString;  
    });  
}
```

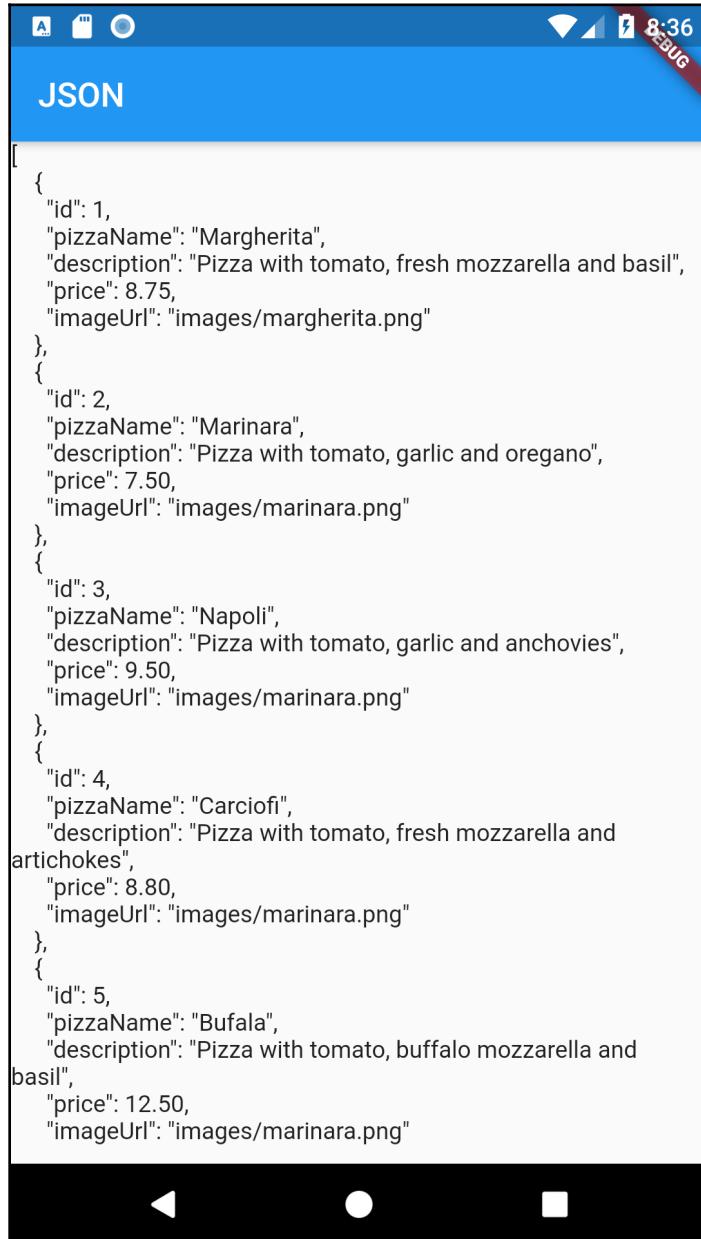
7. In the `_MyHomePageState` class, override the `initState` method, and inside it, call the `readJsonFile` method:

```
@override  
void initState() {  
    readJsonFile();  
    super.initState();  
}
```

8. Now, we want to show the retrieved JSON in `Container`, in the `body` property of `Scaffold`. To do that, add a `Text` widget as a child of our `Container`:

```
body: Container(  
    child: Text(pizzaString),  
) ,
```

9. 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 the following screenshot:



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 {  
    int id;  
    String pizzaName;  
    String description;  
    double price;  
    String imageUrl;  
}
```

13. Inside the `Pizza` class, define a named constructor called `fromJson` that will take a `Map` as a parameter and transform `Map` into an instance of a `Pizza`:

```
Pizza.fromJson(Map<String, dynamic> json) {  
    this.id = json['id'];  
    this.pizzaName = json['pizzaName'];  
    this.description = json['description'];  
    this.price = json['price'];  
    this.imageUrl = json['imageUrl'];  
}
```

14. Refactor the `readJsonFile()` method. The first step is transforming `String` into a `Map` by calling the `jsonDecode` method. In the `readJsonFile` method, add the following highlighted code:

```
Future readJsonFile() async {  
    String myString = await DefaultAssetBundle.of(context)  
        .loadString('assets/pizzalist.json');  
List myMap = jsonDecode(myString);  
    ...
```

15. Make sure that your editor automatically added the `import` statements from the `dart:convert` library at the top of the `main.dart` file; otherwise, just add them 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 the `Map` list and transforming it into `Pizza` objects. In the `readJsonFile` method, under the `jsonDecode` method, add the following code:

```
List<Pizza> myPizzas = [];
myMap.forEach((dynamic pizza) {
    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;
```

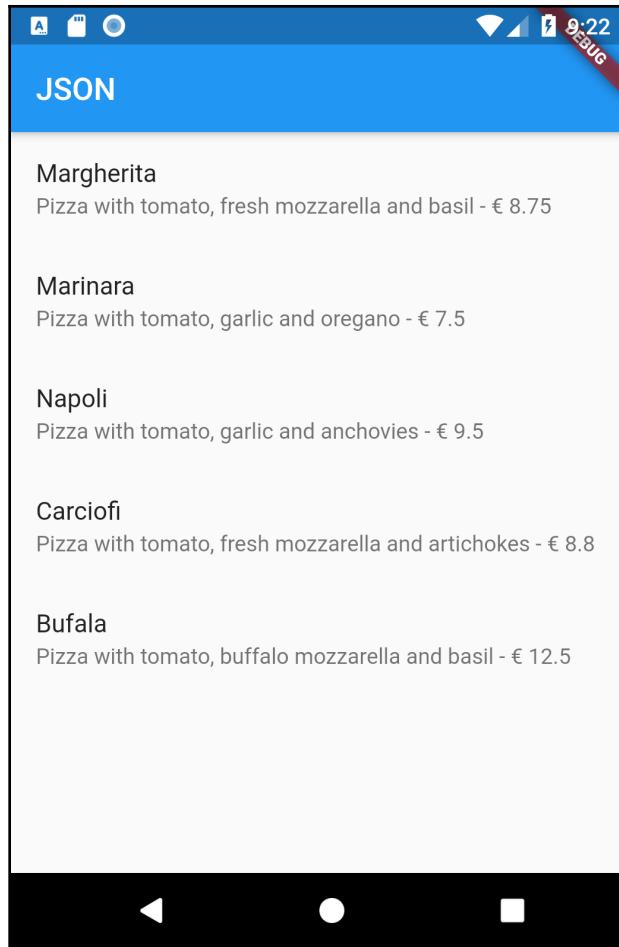
18. Change the signature of the method so that you show the return value explicitly:

```
Future<List<Pizza>> readJsonFile() async {
```

19. Now that we have a `List` of `Pizza` objects, we can leverage those so that instead of just showing our user a `Text`, we can show a `ListView` containing a set of `ListTile` widgets. Since `readJsonFile` is asynchronous, we can leverage a `FutureBuilder` to build the user interface. Add the following code inside our `Container` in the body of `Scaffold`, in the `build()` method:

```
body: Container(
    child: FutureBuilder(
        future: readJsonFile(),
        builder: (BuildContext context, AsyncSnapshot<List<Pizza>>
            pizzas) {
            return ListView.builder(
                itemCount: (pizzas.data == null) ? 0 :
                    pizzas.data.length,
                itemBuilder: (BuildContext context, int position) {
                    return ListTile(
                        title: Text(pizzas.data[
                            position].pizzaName),
                        subtitle: Text(pizzas.data[
                            position].description + ' -
                            € ' +
                            pizzas.data[position].price.toString()),
                );});}),),,
```

20. Run the app. The user interface should now be much friendlier and look as follows:



How it works...

The main features of the app we have built in this recipe are as follows:

- 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.json');
```



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 `asset` key:

```
assets:  
- assets/
```

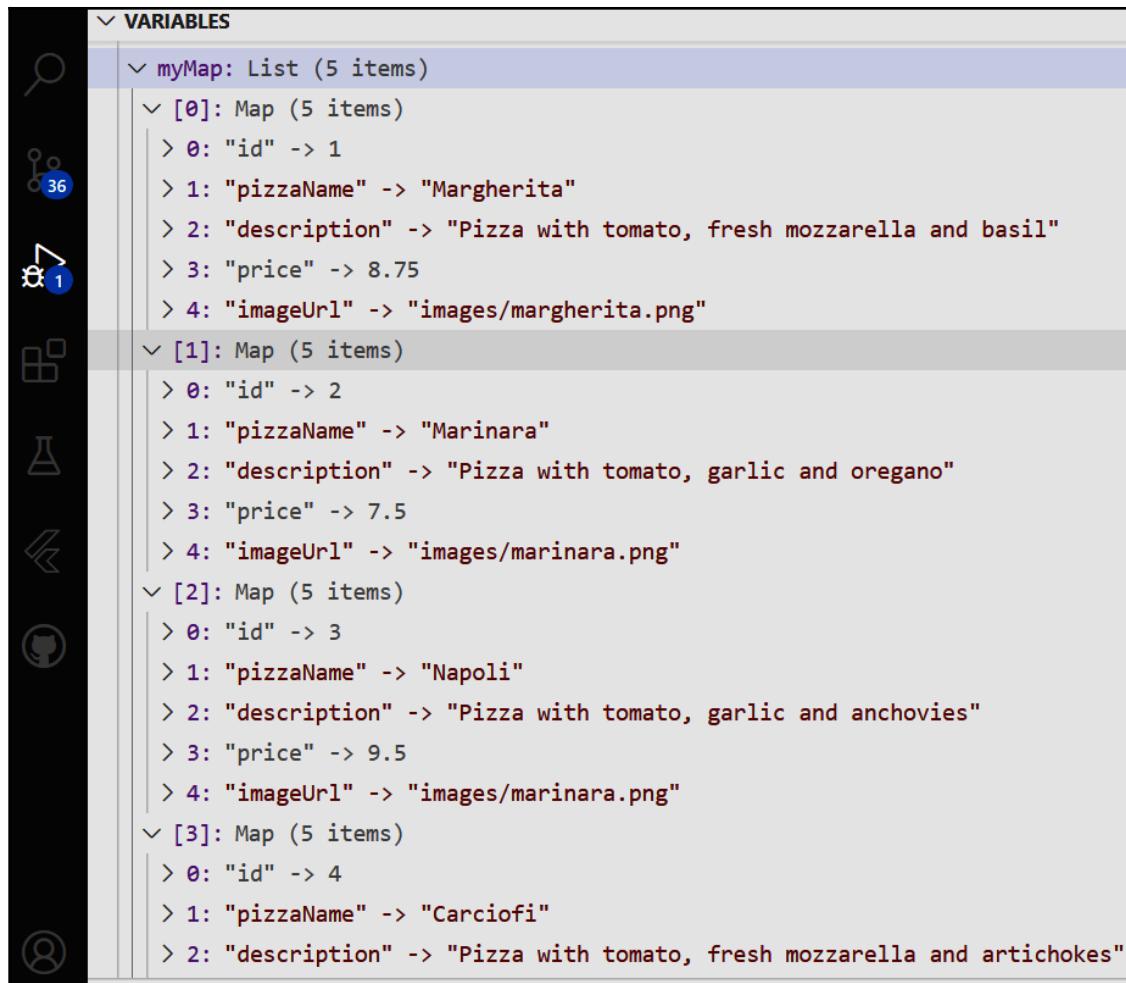
Transforming the JSON string into a list of Map objects

The `jsonDecode` method, available in the `dart:convert` library, decodes JSON. 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:



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 a `Map<String, dynamic>` and, from that, creates a new `Pizza` object:

```
Pizza.fromJson(Map<String, dynamic> json) {  
    this.id = json['id'];  
    this.pizzaName = json['pizzaName'];  
    this.description = json['description'];  
    this.price = json['price'];  
}
```

```
        this.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. To perform this task, 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) {
    String json = '[';
    pizzas.forEach((pizza) {
        json += jsonEncode(pizza);
    });
    json += ']';
    return json;
}
```

This method transforms our `List` of `Pizza` objects back into a `Json` string by calling the `jsonEncode` method again in the `dart_convert` library.

- 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:



The screenshot shows the Android Studio Debug Console interface. The tabs at the top are PROBLEMS, OUTPUT, DEBUG CONSOLE (which is selected), and TERMINAL. A search bar labeled "Filter (e.g. text, !exclude)" is present. The console output area contains the following text:

```
Restarted application in 1.863ms.
I/flutter ( 5353): [{"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: 7.5, imageUrl: images/marinara.png}, {"id": 3, "pizzaName": "Napoli", "description": "Pizza with tomato, garlic and anchovies, price: 9.5, imageUrl: images/marinara.png}, {"id": 4, "pizzaName": "Carciofi", "description": "Pizza with tomato, fresh mozzarella and artichokes, price: 8.8, imageUrl: images/marinara.png}, {"id": 5, "pizzaName": "Bufala", "description": "Pizza with tomato, buffalo mozzarella and basil, price: 12.5, imageUrl: images/marinara.png}]
```

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, the 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 ready for production.

Getting ready

To follow along with this recipe, you need to do the following:

- Complete the previous recipe.
- Download a more real-world `pizzalist.json` file at https://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_08.

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. Delete the content of the `pizzalist.json` file and paste the content available at https://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_08.

- Run the app. You should get the following error:

The screenshot shows a code editor with Dart code and an error message box. The code editor has two lines of code: line 16 and line 17. Line 17 contains the assignment `this.id = json['id'];`. A red arrow points from the error message box to the number 17 in the code editor. The error message box has a title "Exception has occurred. ×" and the message "_TypeError (type 'String' is not a subtype of type 'int')".

```
16 Pizza.fromJson(Map<String, dynamic> json) {  
17     this.id = json['id'];  
}  
  
Exception has occurred. ×  
_TypeError (type 'String' is not a subtype of type 'int')
```

- 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) {  
    this.id = json['id'];  
    this.pizzaName = json['pizzaName'].toString();  
    this.description = json['description'].toString();  
    this.price = json['price'];  
    this.imageUrl = json['imageUrl'].toString();  
}
```

- Run the app again. The error you can see should have changed, as shown here:

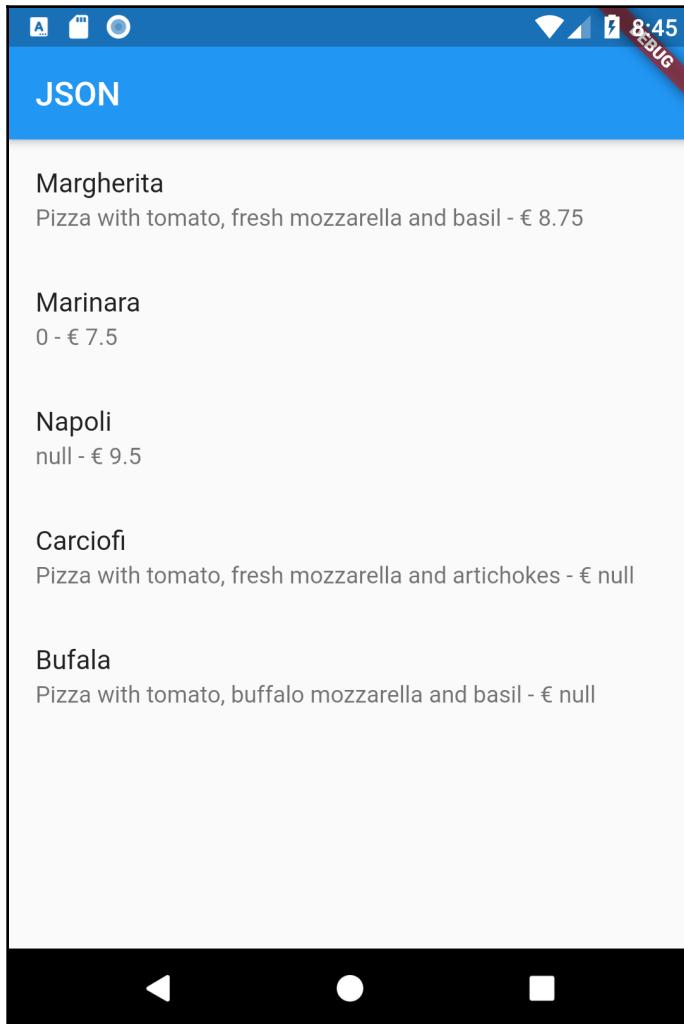
The screenshot shows a code editor with Dart code and an error message box. The code editor has one line of code: line 26. Line 26 contains the assignment `this.price = json[keyPrice];`. A red arrow points from the error message box to the number 26 in the code editor. The error message box has a title "Exception has occurred. ×" and the message "_TypeError (type 'String' is not a subtype of type 'double')".

```
26     this.price = json[keyPrice];  
  
Exception has occurred. ×  
_TypeError (type 'String' is not a subtype of type 'double')
```

- 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) {  
    this.id = int.tryParse(json['id'].toString());  
    this.pizzaName = json['pizzaName'].toString();  
    this.description = json['description'].toString();  
    this.price = double.tryParse(json['price'].toString());  
    this.imageUrl = json['imageUrl'].toString();  
}
```

6. 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:



7. Edit the `Pizza.fromJson` constructor one last time by adding ternary operators, as shown here:

```
Pizza.fromJson(Map<String, dynamic> json) {
    this.id = (json['id'] != null) ?
        int.tryParse(json['id'].toString()) : 0;
    this.pizzaName = (json['pizzaName'] != null) ?
```

```
        json['pizzaName'].toString() : '',
        this.description =(json['description'] != null) ?
            json['description'].toString() : '';
        this.price = (json['price'] != null &&
            double.tryParse(json['price'].toString()) != null) ?
            json['price'] : 0.0;
        this.imageUrl =(json['imageUrl'] != null) ?
            json['imageUrl'].toString() : '';
    }
```

9. 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 second pizza object in the file contains a `description` key, with a number instead of a String:

```
"description": 0,
```

That's why we received **Type Error: "Type int is not a subtype of type String"**, which basically means, "I was expecting a String, and I found an integer number. Not good."

Fortunately, 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 all the other strings in our class. The second error we received was a **Type Error** as well, but this depended on the opposite.

On 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 number, and that's causing a type error.

The solution here was adding a `double.tryParse` to the value:

```
double.tryParse(json['price'].toString());
```

Generally, when you want to transform some kind of data into another, you use the `parse` method. 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.

We repeated the same pattern with `id`:

```
int.tryParse(json[id].toString());
```

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 all our fields:

```
this.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 an empty string ('')
```

We use the same pattern with the numbers, but instead of an empty string, we return 0:

```
this.id = (json['id'] != null) ? int.tryParse(json['id'].toString()) : 0;
```

For the price, we also added a second check:

```
this.price = (json['price'] != null && double.tryParse(json['price'].toString()) != null) ? json['price']
```

In the preceding instruction, we checked that there was a price in the JSON data called `json['price'] != null`, AND (`&&`) that it was a real double number; that is, `double.tryParse(json['price'].toString()) != null`.

After applying those checks, we can use problematic JSON code and apply it successfully to our `Pizza` class.

There's more...

In the code shown in the previous section, even if the objects contained in the JSON file miss pieces of data that are extremely important for our app (such as the name of the pizza or its ID), we still create an instance of the pizza. But this is not always required, nor recommended. That's why we can leverage the `factory` constructor, and only create a new instance of a `Pizza` when its `id` and `pizzaName` are valid. Follow these steps:

1. Edit the `PizzaList.json` file so that the object with `"id": 2` has no `pizzaName`:

```
{  
  "id": 2,  
  "description": 0,  
  "price": 7.50,  
  "imageUrl": "images/marinara.png"  
,
```

2. In the `pizza.dart` file, in the `Pizza` class, add a new `factory` constructor method:

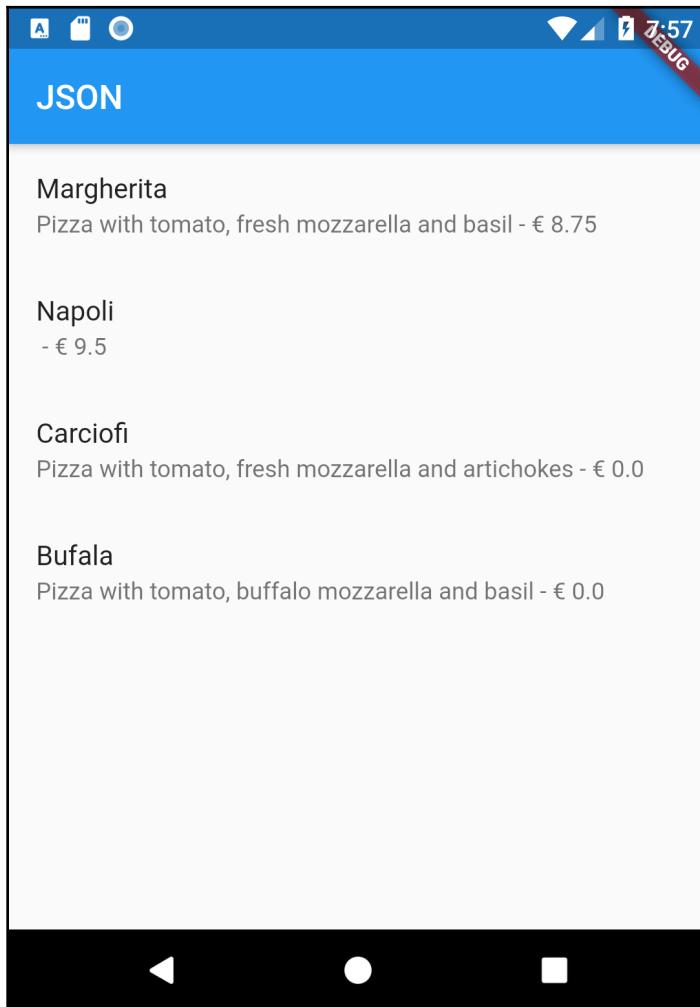
```
Pizza();  
factory Pizza.fromJsonOrNull(Map<String, dynamic> json) {  
  Pizza pizza = Pizza();  
  pizza.id = (json['id'] != null) ?  
    int.tryParse(json['id'].toString()) : 0;  
  pizza.pizzaName = (json['pizzaName'] != null) ?  
    json['pizzaName'].toString() : '';  
  pizza.description = (json['description'] != null) ?  
    json['description'].toString() : '';  
  pizza.price = (json['price'] != null &&  
    double.tryParse(json['price'].toString()) != null) ?  
    json['price'] : 0.0;  
  pizza.imageUrl = (json['imageUrl'] != null) ?  
    json['imageUrl'].toString() : '';  
  if (pizza.id == 0 || pizza.pizzaName.trim() == '') {  
    return null;  
  }  
  return pizza;  
}
```

3. In the `main.dart` file, edit the `readJsonFile` method so that the `myMap.forEach` method calls the new `factory` constructor, as shown here:

```
myMap.forEach((dynamic pizza) {  
  Pizza myPizza = Pizza.fromJsonOrNull(pizza);
```

```
    if (myPizza != null)
        myPizzas.add(myPizza);
});
```

4. Check the app. You should see that you now have four items on the list instead of five:



In Dart, a **factory constructor** is a constructor that does not always return a new instance of the object you are calling.

In this case, we only want to create a new `Pizza` object when the ID is not 0 and there is a `pizzaName` string that's not empty. The `trim()` method removes leading and trailing spaces from any `Strings`:

```
if (pizza.id == 0 || pizza.pizzaName.trim() == '') {  
    return null;  
}  
return pizza;  
}
```

What happens is that when our data is valid, we return an instance of `Pizza`; otherwise, we only return `null` and ignore the invalid item.

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, 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) {
  this.id = (json[keyId] != null) ?
    int.tryParse(json['id'].toString()) : 0;
  this.pizzaName =
    (json[keyName] != null) ? json[keyName].toString() : '';
  this.description = (json[keyDescription] != null) ?
    json[keyDescription].toString() : '';
  this.price = (json[keyPrice] != null &&
    double.tryParse(json[keyPrice].toString()) != null) ?
    json[keyPrice] : 0.0;
  this.imageUrl = (json[keyImage] != null) ?
    json[keyImage].toString() : '';
}
```

3. In the `Pizza.fromJsonOrNull` factory constructor, repeat the process of removing the strings and adding the constants:

```
factory Pizza.fromJsonOrNull(Map<String, dynamic> json) {
  Pizza pizza = Pizza();
  pizza.id = (json[keyId] != null) ?
    int.tryParse(json[keyId].toString()) : 0;
  pizza.pizzaName = (json[keyName] != null) ?
    json[keyName].toString() : '';
  pizza.description = (json[keyDescription] != null) ?
    json[keyDescription].toString() : '';
  pizza.price = (json[keyPrice] != null
    && double.tryParse(json[keyPrice].toString()) != null) ?
    json[keyPrice] : 0.0;
  pizza.imageUrl = (json[keyImage] != null) ?
    json[keyImage].toString() : '';
```

```
if (pizza.id == 0 || pizza.pizzaName.trim() == '') {
    return null;
}
return pizza;
}
```

4. 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_serializable and `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 `shared_preferences` for critical data as data stored there is **not** encrypted, and writes are not always guaranteed.

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, we must add a dependency to `shared_preferences`. Go to https://pub.dev/packages/shared_preferences/install and check the latest version of the library.
2. In the `pubspec.yaml` file of your project, add the `shared_preferences` dependency (with the version number you retrieved in step 1):

```
dependencies:  
  flutter:  
    sdk: flutter  
  shared_preferences: ^2.0.5
```

3. If necessary, run the `flutter pub get` command from your **Terminal** window.
4. At the top of the `main.dart` file, import `shared_preferences`:

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

5. At the top of the `_MyHomePageState` class, create a new integer state variable called `appCounter`:

```
int appCounter;
```

6. In the `_MyHomePageState` class, create a new asynchronous method called `readAndWritePreferences()`:

```
Future readAndWritePreference() async {}
```

7. Inside the `readAndWritePreference` method, create an instance of `SharedPreferences`:

```
SharedPreferences prefs = await SharedPreferences.getInstance();
```

8. After creating the `prefs` instance, try reading the value of the `appCounter` key. If its value is null, set it to 1; otherwise, increment it:

```
appCounter = prefs.getInt('appCounter');  
if (appCounter == null) {  
    appCounter = 1;  
} else {  
    appCounter++;  
}
```

9. After updating `appCounter`, set the value of the `appCounter` key in `prefs` to the new value:

```
await prefs.setInt('appCounter', appCounter);
```

10. Update the `appCounter` state value:

```
setState(() {  
    appCounter = appCounter;  
});
```

11. In the `initState` method in the `_MyHomePageState` class, call the `readAndWritePreference()` method:

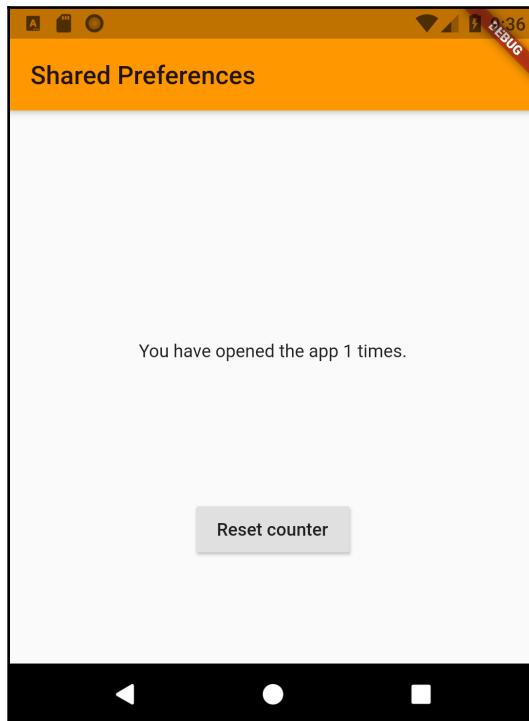
```
@override  
void initState() {
```

```
        readAndWritePreference();
        super.initState();
    }
```

12. In the build method, you'll create the user interface for the app. Add the following code inside the Container widget:

```
child: Center(
    child: Column(
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
        children: [
            Text(
                'You have opened the app ' + appCounter.toString() +
                ' times.'),
            ElevatedButton(
                onPressed: () {},
                child: Text('Reset counter'),
            ],
        ),
    ),
)
```

13. Run the app. The first time you open it, you should see a screen similar to the following:



14. Add a new method to the `_MyHomePageState` class called `deletePreference()` that will delete the saved value:

```
Future deletePreference() async {
    SharedPreferences prefs = await
    SharedPreferences.getInstance();
    await prefs.clear();
    setState(() {
        appCounter = 0;
    });
}
```

15. From the `onPressed` property of the `ElevatedButton` widget in the `build()` method, call `deletePreference()`:

```
ElevatedButton(
    onPressed: () {
        deletePreference();
    },
    child: Text('Reset counter'),
)
```

16. Run the app again. Now, when you press the reset 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	<code>getInt (key)</code>	<code>setInt (key, value)</code>
double	<code>getDouble (key)</code>	<code>setDouble (key, value)</code>
bool	<code>getBool (key)</code>	<code>setBool (key, value)</code>
String	<code>getString (key)</code>	<code>setString (key, value)</code>
stringList	<code>getStringList (key)</code>	<code>setStringList (key, listOfvalues)</code>



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 is different. By leveraging `path_provider`, you don't need to write two 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.
2. Check the current version of `path_provider` at https://pub.dev/packages/path_provider.
3. Add the dependency to the `pubspec.yaml` file:

```
path_provider: ^2.0.1
```

4. At the top of the `main.dart` file, import `path_provider`:

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

5. At the top of the `_MyHomePageState` class, add the State variables that we will use to update the user interface:

```
String documentsPath='';  
String tempPath='';
```

6. 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;  
    });  
}
```

7. In the `initState` method of the `_MyHomePageState` class, call the `getPaths` method:

```
@override  
void initState() {  
    getPaths();  
    super.initState();  
}
```

8. 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: Text('Path Provider')),  
        body: Container(  
            child: Column(  
                mainAxisSize: MainAxisSize.spaceEvenly,  
                children: [  
                    Text('Doc path: ' + documentsPath),  
                    Text('Temp path' + tempPath),  
                ],),  
        ),  
    );  
}
```

9. Run the app. You should see a screen that looks similar to the following:



This completes this recipe.

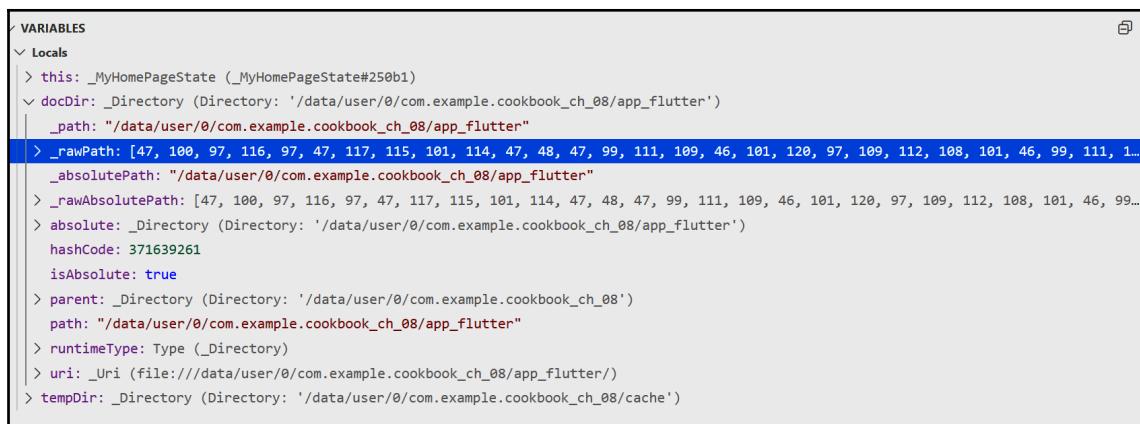
How it works...

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:



```
VARIABLES
Locals
> this: _MyHomePageState (_MyHomePageState#250b1)
> docDir: _Directory (Directory: '/data/user/0/com.example.cookbook_ch_08/app_flutter')
|   _path: "/data/user/0/com.example.cookbook_ch_08/app_flutter"
> _rawPath: [47, 100, 97, 116, 97, 47, 117, 115, 101, 114, 47, 48, 47, 99, 111, 109, 46, 101, 120, 97, 109, 112, 108, 101, 46, 99, 111, 1...
|   _absolutePath: "/data/user/0/com.example.cookbook_ch_08/app_flutter"
> _rawAbsolutePath: [47, 100, 97, 116, 97, 47, 117, 115, 101, 114, 47, 48, 47, 99, 111, 109, 46, 101, 120, 97, 109, 112, 108, 101, 46, 99...
> absolute: _Directory (Directory: '/data/user/0/com.example.cookbook_ch_08/app_flutter')
|   hashCode: 371639261
|   isAbsolute: true
> parent: _Directory (Directory: '/data/user/0/com.example.cookbook_ch_08')
|   path: "/data/user/0/com.example.cookbook_ch_08/app_flutter"
> runtimeType: Type (_Directory)
> uri: _Uri (file:///data/user/0/com.example.cookbook_ch_08/app_flutter/)
> tempDir: _Directory (Directory: '/data/user/0/com.example.cookbook_ch_08/cache')
```

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 temp 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 do the following:

- 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 beginning code for this recipe is the end of the previous one. 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:

```
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();  
}
```

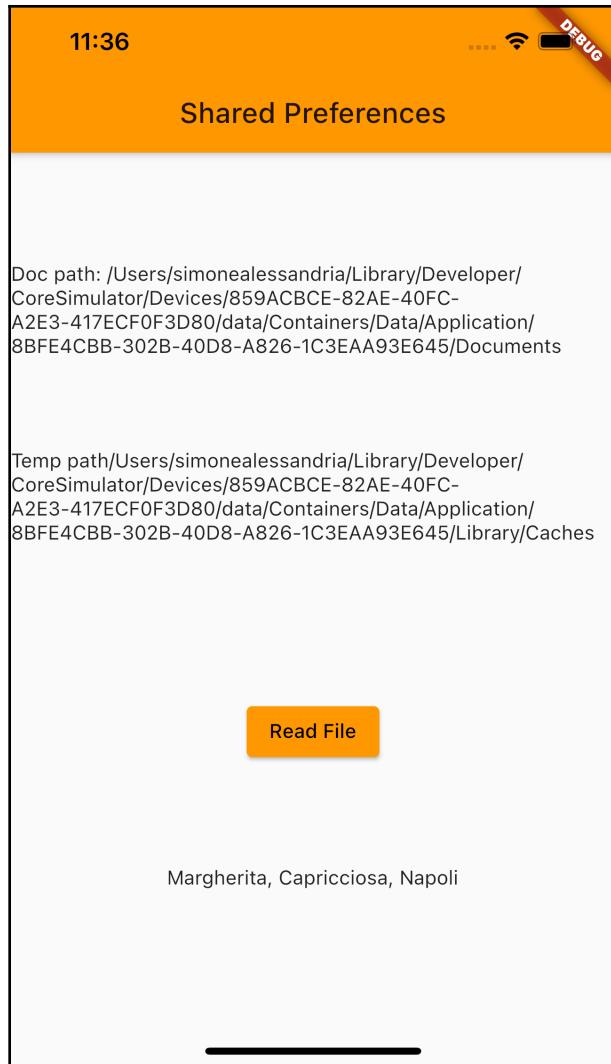
5. Create a method to read the file:

```
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: 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:



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.

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 `minSdkVersion` in your app's `build.gradle` file to 18. The file is located in your project folder, in `/android/app/build.gradle`. The `minSdkVersion` key is in the `defaultConfig` node.
- 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. Edit the `pubspec.yaml` file by adding the `flutter_secure_storage` dependency. You'll find the latest version at https://pub.dev/packages/flutter_secure_storage:

```
flutter_secure_storage: ^4.1.0
```

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.dart';
```

4. At the top of the `_myHomePageState` class, create secure storage:

```
final storage = 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: Text('Save Value'),  
    onPressed: () {  
        writeToSecureStorage();  
    }  
) ,
```

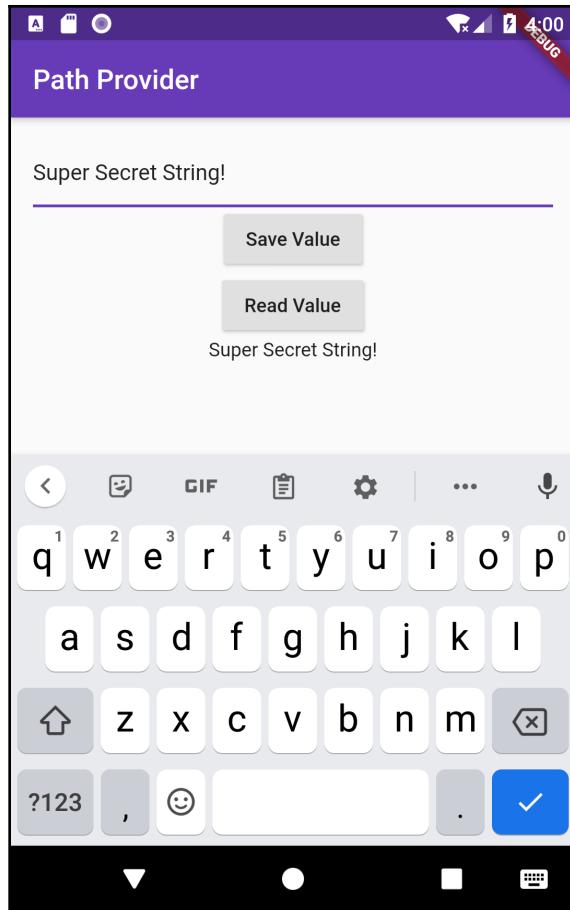
7. In the `_myHomePageState` class, add the method for **reading** data from the secure storage:

```
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:



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 = 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 app, we will create an app that reads and writes data from a web service. As creating a web API is well beyond the scope of this book, we will use a mock service, called **MockLab**, that 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`.

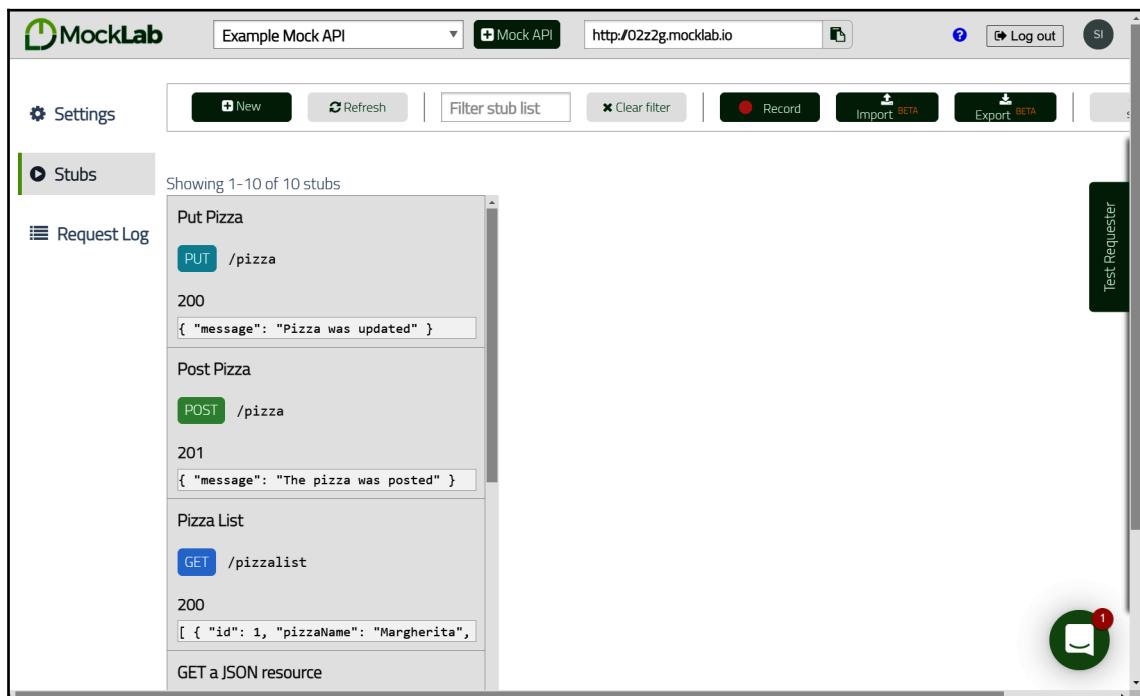
Getting ready

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 Mock Lab 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.mocklab.io/> and choose 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:



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, paste the JSON content available at <https://bit.ly/pizzalist>. The final result is shown here:

```
Request duration: 112ms

HTTP/1.1 200 OK
Matched-Stub-Id: f8f3616b-8926-42a9-b96afea790877a67
Matched-Stub-Name: Pizza List
Vary: Accept-Encoding, User-Agent

[
  {
    "id": 1,
    "pizzaName": "Margherita",
    "description": "Pizza with tomato, fresh mozzarella and basil",
  }
]
```

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. In the project, in the `pubspec.yaml` file, add the `http` dependency. Check the latest version at <https://pub.dev/packages/http>:

```
http: ^0.12.2
```

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 {
    Uri url = Uri.https(authority, path);
    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 null;
}
}
```

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 `initState` method of the `_MyHomePageState` class, call the `callPizzas()` method:

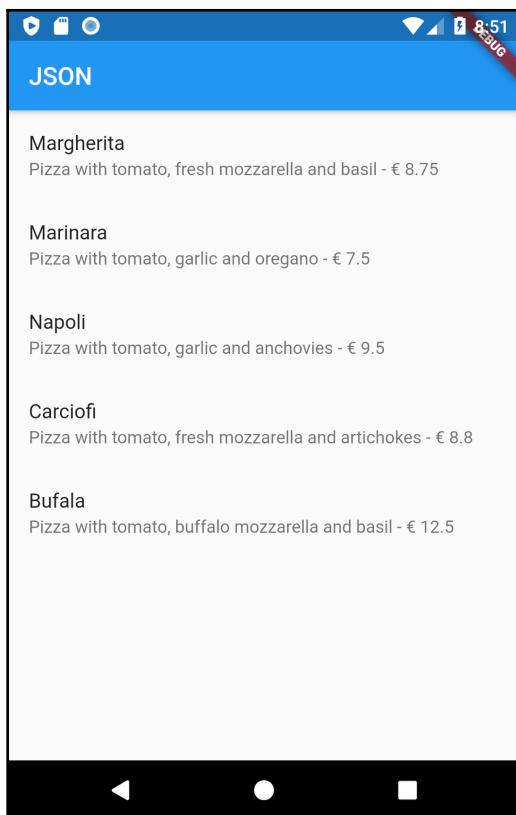
```
@override
void initState() {
    callPizzas();
    super.initState();
}
```

10. 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: Text('JSON')),
        body: Container(
            child: FutureBuilder(
                future: callPizzas(),
                builder: (BuildContext context,
                    AsyncSnapshot<List<Pizza>> pizzas) {
                    return ListView.builder(
                        itemCount: (pizzas.data == null) ? 0 :
                            pizzas.data.length,
```

```
        itemBuilder: (BuildContext context,
        int position) {
      return ListTile(
        title: Text(pizzas.data[position].pizzaName),
        subtitle: Text(pizzas.data[
          position].description +
          ' - € ' +
          pizzas.data[position].price.toString()),
      );
    },
  ),
}
}
```

11. Run the app. You should see a screen similar to the following:



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. **Mocklab** 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 creating 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.



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 MockLab service, have a look at <https://www.mocklab.io/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 into the Mock Lab service at <https://app.mocklab.io/> 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:** {"message": "The pizza was posted"}:

Request duration: 32ms

```
HTTP/1.1 201 Created
Matched-Stub-Id: 60af6f2b-3348-4353-91f1-eaa6278cadad
Matched-Stub-Name: Post Pizza
Vary: Accept-Encoding, User-Agent
Content-Type: application/json

{
  "message": "The pizza was posted"
}
```

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 {
  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 PizzaDetail:

```
class PizzaDetail extends StatefulWidget {
    @override
    _PizzaDetailState createState() => _PizzaDetailState();
}
class _PizzaDetailState extends State<PizzaDetail> {
    @override
    Widget build(BuildContext context) {
        return Container(
        );
    }
}
```

8. At the top of the `_PizzaDetailState` class, add five new TextEditingController widgets. These will contain the data for the Pizza object that will be posted later, and 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 postResult = '';
```

9. 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: Text('Pizza Detail'),
    ),
    body: Padding(
        padding: EdgeInsets.all(12),
        child: SingleChildScrollView(

```

```
        child: Column(
            children: [
        ]),
    );
}
```

10. 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(
    postResult,
    style: TextStyle(
        backgroundColor: Colors.green[200], color:
        Colors.black),
),
SizedBox(
    height: 24,
),
TextField(
    controller: txtId,
    decoration: InputDecoration(hintText: 'Insert ID'),
),
SizedBox(
    height: 24,
),
TextField(
    controller: txtName,
    decoration: InputDecoration(hintText: 'Insert Pizza
        Name'),
),
SizedBox(
    height: 24,
),
TextField(
    controller: txtDescription,
    decoration: InputDecoration(hintText: 'Insert
        Description'),
),
SizedBox(
    height: 24,
),
TextField(
    controller: txtPrice,
    decoration: InputDecoration(hintText: 'Insert
        Price'),
```

```
        ),
        SizedBox(
            height: 24,
        ),
        TextField(
            controller: txtImageUrl,
            decoration: InputDecoration(hintText: 'Insert Image
                Url'),
        ),
        SizedBox(
            height: 48,
        ),
        ElevatedButton(
            child: Text('Send Post'),
            onPressed: () {
                postPizza();
            }
        )
    ],
}
```

11. 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(() {
        postResult = result;
    });
    return result;
}
```

12. In the `main.dart` file, import the `pizza_detail.dart` file:

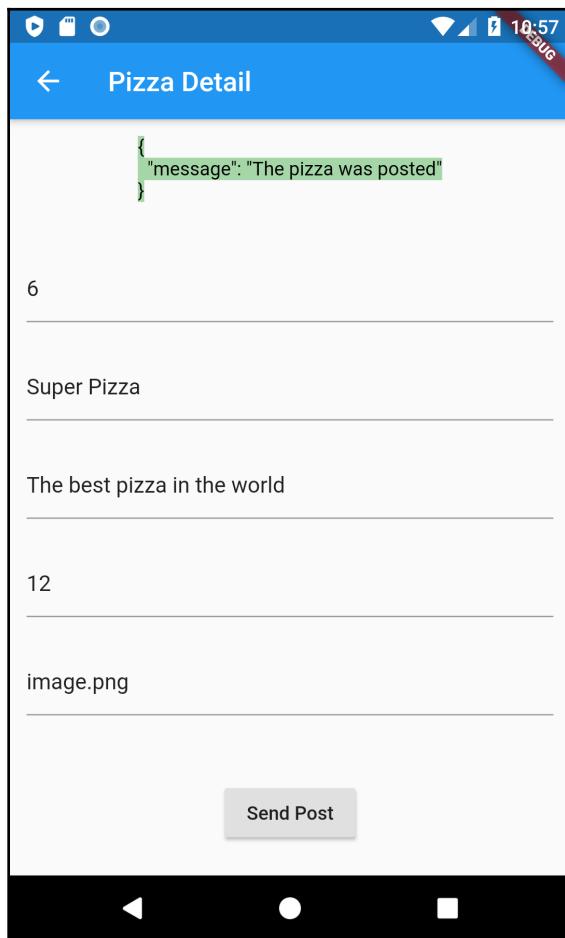
```
import 'pizza_detail.dart';
```

13. In `Scaffold` of the `build()` method of the `_MyHomePageState` class, add a `FloatingActionButton` that will navigate to the `PizzaDetail` route:

```
floatingActionButton: FloatingActionButton(
    child: Icon(Icons.add),
    onPressed: () {
```

```
        Navigator.push(
            context,
            MaterialPageRoute(builder: (context) =>
                PizzaDetail(),
            );
        ),
    ),
}
```

14. Run the app. On the main screen, press our **FloatingActionButton** to navigate to the **PizzaDetail** route.
15. Add pizza details in the text fields and press the **Send Post** button. You should see a successful result, as shown in the following screenshot:



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.

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 MockLab, 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.

Finally, we created the method that will call the `HttpHelper.postPizza` method. As this method requires an instance of `Pizza`, we had to read the values in the `TextFields`, using their `TextEditingController`s, and then create a new `Pizza` object and pass it to the `postPizza` method.

After receiving the response, we wrote it in the `postResult` `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 into the Mock Lab service at <https://app.mocklab.io/> 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:** {"message": "Pizza was updated"}:

Request duration: 136ms

```
HTTP/1.1 200 OK
Matched-Stub-Id: 0e64caf0-1574-4b32-8f77-63bda8177315
Matched-Stub-Name: Put Pizza
Vary: Accept-Encoding, User-Agent

{
  "message": "Pizza was updated"
}
```

3. In the Flutter project, add a `putPizza` method to the `HttpHelper` class in the `http_helper.dart` file:

```
Future<String> putPizza(Pizza pizza) async {
  String put = json.encode(pizza.toJson());
  Uri url = Uri.https(authority, putPath);
  http.Response r = await http.put(
    url,
    body: put,
  );
  return r.body;
}
```

4. In the `PizzaDetail` class in the `pizza_detail.dart` file, add two properties, a `Pizza` and a `boolean`, and a constructor that sets the two properties:

```
final Pizza pizza;
final bool isNew;
PizzaDetail(this.pizza, this.isNew);
```

5. In the `PizzaDetailState` 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();  
}
```

6. 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 {  
    ...  
  
    String result = '';  
    if (widget isNew) {  
        result = await helper.postPizza(pizza);  
    } else {  
        result = await helper.putPizza(pizza);  
    }  
    setState(() {  
        postResult = result;  
    });  
    return result;  
}
```

7. 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) =>
            PizzaDetail(pizzas.data[position], false)),
        );
    },
);
;
```

8. In `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) =>
                PizzaDetail(Pizza(), false)),
        );
    },
);
```

9. Run the app. On the main screen, tap on any `Pizza` to navigate to the `PizzaDetail` route.
10. 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. It would respond with a success message. 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 MockLab, 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 into the Mock Lab service at <https://app.mocklab.io/> and click on the Stubs section of the example API. Then, create a new Stub.
2. Complete the request, as follows:
 - **Name:** Delete Pizza
 - **Verb:** DELETE
 - **Address:** /pizza
 - **Status:** 200
 - **Body:** {"message": "Pizza was deleted"}:



3. In the Flutter project, add a `deletePizza` method to the `HttpHelper` class in the `http_helper.dart` file:

```
Future<String> deletePizza(int id) async {
    Uri url = Uri.https(authority, deletePath);
    http.Response r = await http.delete(
        url,
    );
    return r.body;
}
```

4. 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(
            onDismissed: (item) {
                HttpHelper helper = HttpHelper();
                pizzas.data.removeWhere((element) => element.id ==
                    pizzas.data[position].id);
                helper.deletePizza(pizzas.data[position].id);
            },
            key: Key(position.toString()),
            child: ListTile(
                ...
            )
        );
    }
);
```

5. 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 MockLab, 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.

9

Advanced State Management with Streams

In Dart and Flutter, futures and streams are the main tools for dealing 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.



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 see 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 transform 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 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
- 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/tree/master/chapter_09.

How to use Dart streams

In this recipe's demo, you will change the background color of the app each second. You will create a list of five colors, and each 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, which 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/tree/master/chapter_09.

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(MyApp());
}

class MyApp extends StatelessWidget {
    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            title: 'Stream',
            theme: ThemeData(
                primarySwatch: Colors.deepPurple,
                visualDensity: VisualDensity.adaptivePlatformDensity,
            ),
            home: StreamHomePage(),
        );
    }
}

class StreamHomePage extends StatefulWidget {
    @override
    _StreamHomePageState 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. In the ColorStream class, create a colorStream property and a method called getColors that returns a stream of the Color type and is marked as `async*` (note the asterisk at the end of `async`):

```
Stream colorStream;
Stream<Color> getColors() async* {  
}
```

6. At the beginning of the getColors() method, create a list of colors called colors, containing five colors:

```
final List<Color> colors = [
  Colors.blueGrey,
  Colors.amber,
  Colors.deepPurple,
  Colors.lightBlue,
  Colors.teal
];
```

7. After the colors declaration, add the `yield*` command, as shown here. This completes the ColorStream class:

```
yield* Stream.periodic(Duration(seconds: 1), (int t) {
  int index = t % 5;
  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;
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:

```
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:

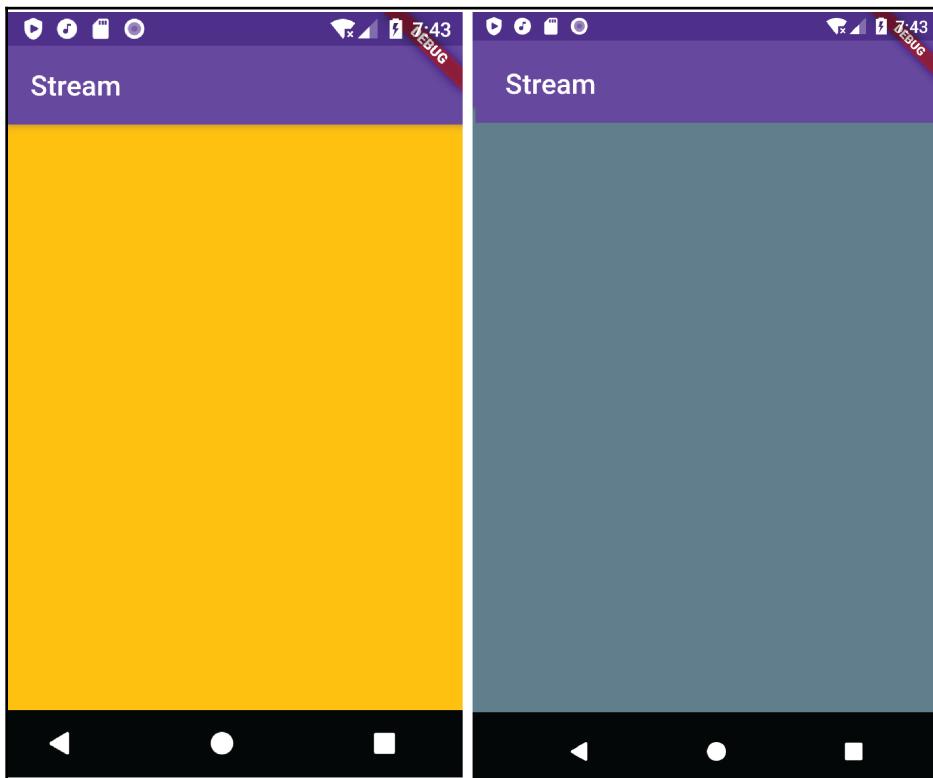
```
@override
void initState() {
    colorStream = ColorStream();
    changeColor();
    super.initState();
}
```

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: 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:



How it works...

The two core parts of the app that you have implemented in this recipe are creating a stream of data, and listening (or subscribing) 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 `async*`, you are creating 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(Duration(seconds: 1), (int t) {
    int index = t % 5;
    return colors[index];
});
```

In order to return a stream in an `async*` method, you use the `yield*` statement. Simplifying this slightly, you might think as `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 steam 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.
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.

See also

A great resource for obtaining information about streams is the official tutorial available at <https://dart.dev/tutorials/languagestreams>.

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.

A stream controller simplifies 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 show 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`:

```
StreamController<int> controller = StreamController<int>();
```

4. Still in the `NumberStream` class, add a method called `addNumberToSink`, with the code shown here:

```
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;
StreamController numberStreamController;
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. At the bottom of the `_StreamHomePageState` class, add a method called `addRandomNumber`, with the code shown here:

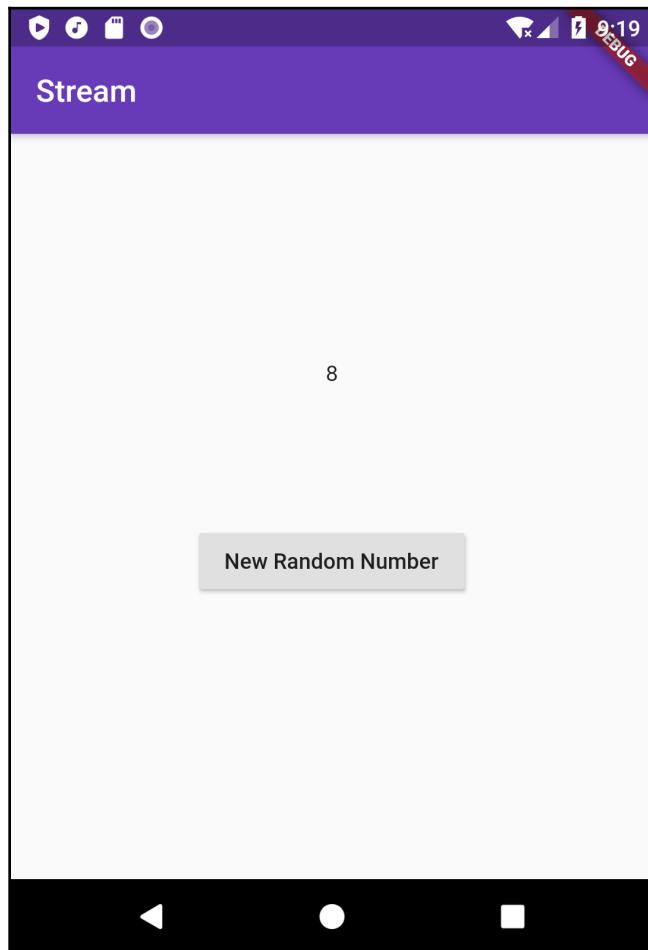
```
void addRandomNumber() {
    Random random = Random();
    int myNum = random.nextInt(10);
    numberStream.addNumberToSink(myNum);
}
```

10. In the `build` method, edit the body of the scaffold so that it contains a column with `Text` and `ElevatedButton`:

```
body: Container(
    width: double.infinity,
    child: Column(
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
        crossAxisAlignment: CrossAxisAlignment.center,
        children: [
```

```
        Text(lastNumber.toString()),  
        ElevatedButton(  
            onPressed: () => addRandomNumber(),  
            child: Text('New Random Number'),  
        ),  
    ],  
,  
)
```

11. Run the app. You'll notice that each time you press the button, a number appears on screen, as shown in the following screenshot:



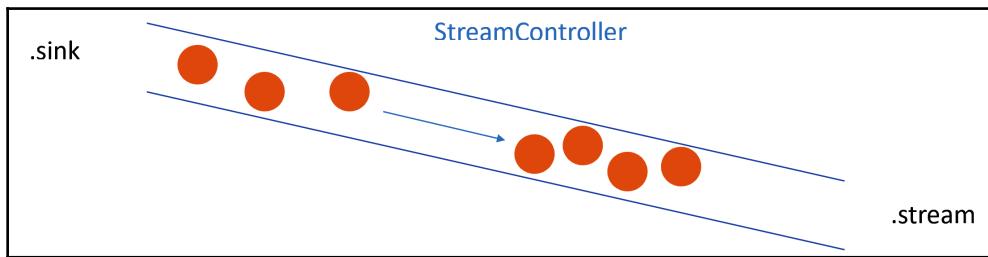
How it works...

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 do the following:

- You can use a stream controller to control a stream.
- A stream controller 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 diagram:



In the app you have built in this recipe, the first step involved the creation of a stream controller, 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 `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;
  });
})
```

So in this simple example, you used `StreamController`, `Stream`, and `StreamSink`. But there is another important feature that `StreamController` 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, append to the `listen` method in the `initState` function of `_StreamHomePageState` of the `onError` method, 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 button. You should see a **-1** in the center of the screen.
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 transform into streams

Sometimes, you may need to manipulate and transform the data that is being 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 data before showing it to your users, or process it to generate some new output.

Examples include converting a number into a string, or 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 have 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`:

```
StreamTransformer transformer;
```

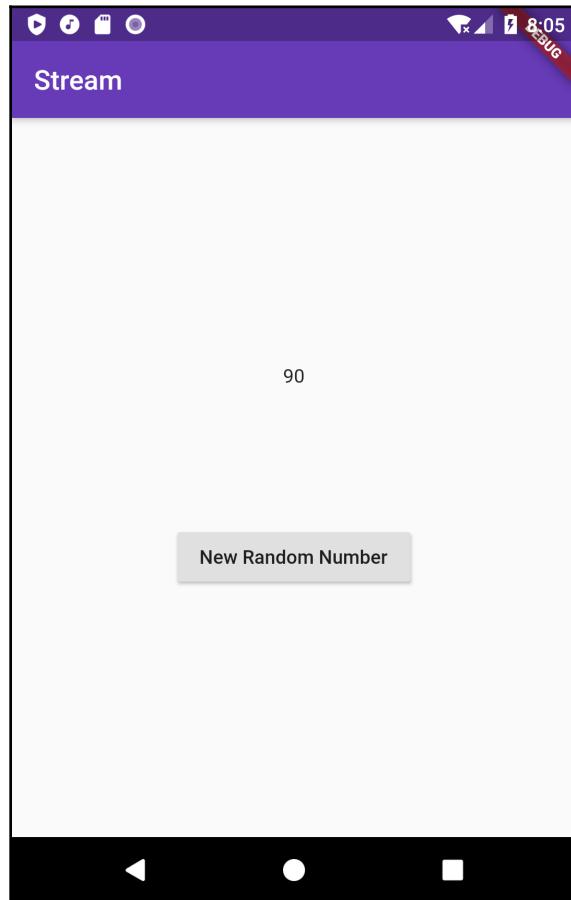
2. In the `initState` method, just after the declarations, create an instance of `StreamTransformer`, calling the `fromHandlers` constructor:

```
transformer = StreamTransformer<int, dynamic>.fromHandlers(  
    handleData: (value, sink) {  
        sink.add(value * 10);  
    },  
    handleError: (error, trace, sink) {  
        sink.add(-1);  
    },  
    handleDone: (sink) => sink.close());
```

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 10 to 100, instead of from 1 to 10, as shown in the following screenshot:



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 to a new value. For more information, have a look at <https://api.dart.dev/stable/1.10.1/dart-async/Stream/map.html> and <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, 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 transform 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`:

```
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();
}
```

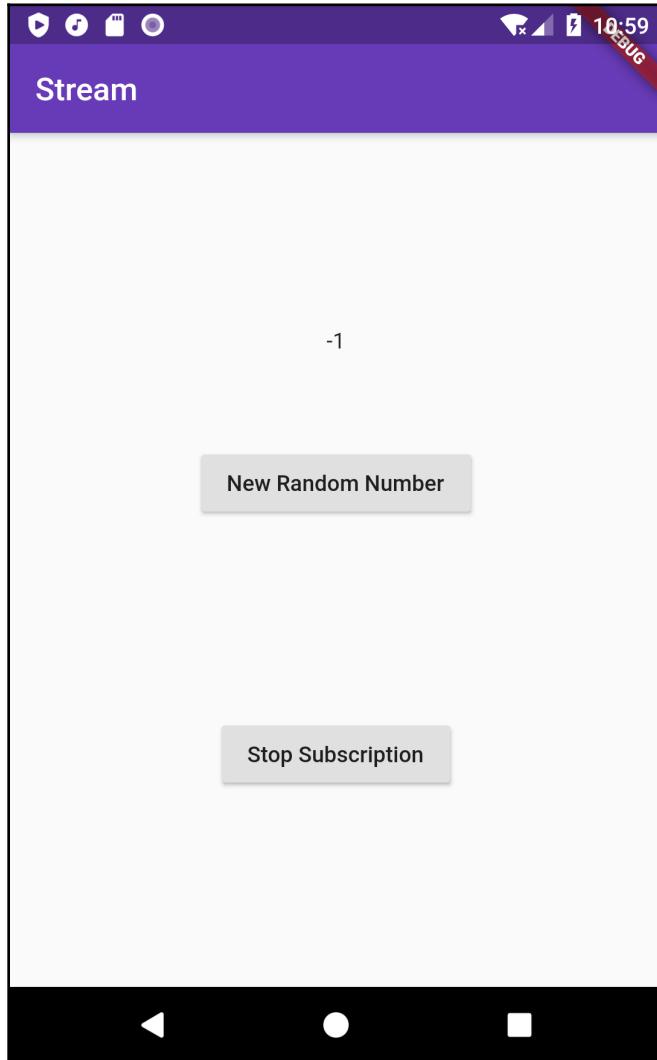
6. In the `build` method, at the end of the `Column` widget, add a second `ElevatedButton`, which calls the `stopStream` method:

```
ElevatedButton(
    onPressed: () => stopStream(),
    child: Text('Stop Stream'),
)
```

7. 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;
        });
    }
}
```

- Run the app. You should now see two buttons on the screen, as shown in the following screenshot:



9. Press the **Stop Stream** button, and then the **New Random Number** button. In the debug console, you should see the message **OnDone was called**, as shown in the following screenshot:



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
<code>onListen</code>	Required – Positional as first parameter	Function
<code>onDone</code>	Optional	Function
<code>onError</code>	Optional	Function
<code>cancelOnError</code>	Optional	Bool

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 have also seen 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 `StreamSubscription`, 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 pressed the **New Random Number** button, and the app tried to add a new number to the sink, an error would 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 have built in the previous recipes in this chapter:

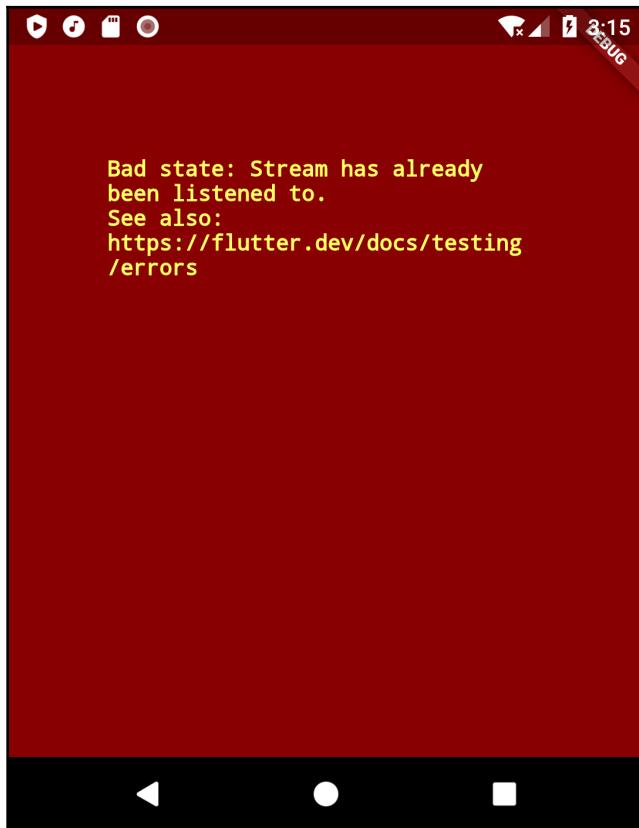
1. At the top of the `_StreamHomePageState` class, in the `main.dart` file, declare a second `StreamSubscription`, called `subscription2`, and a string, called `values`:

```
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.toString() + ' - ';
    });
});
subscription2 = stream.listen((event) {
    setState(() {
        values += event.toString() + ' - ';
    });
});
```

3. Run the app. You should see an error as shown in the following screenshot:



4. Still in the `initState` method, set the stream as a broadcast stream:

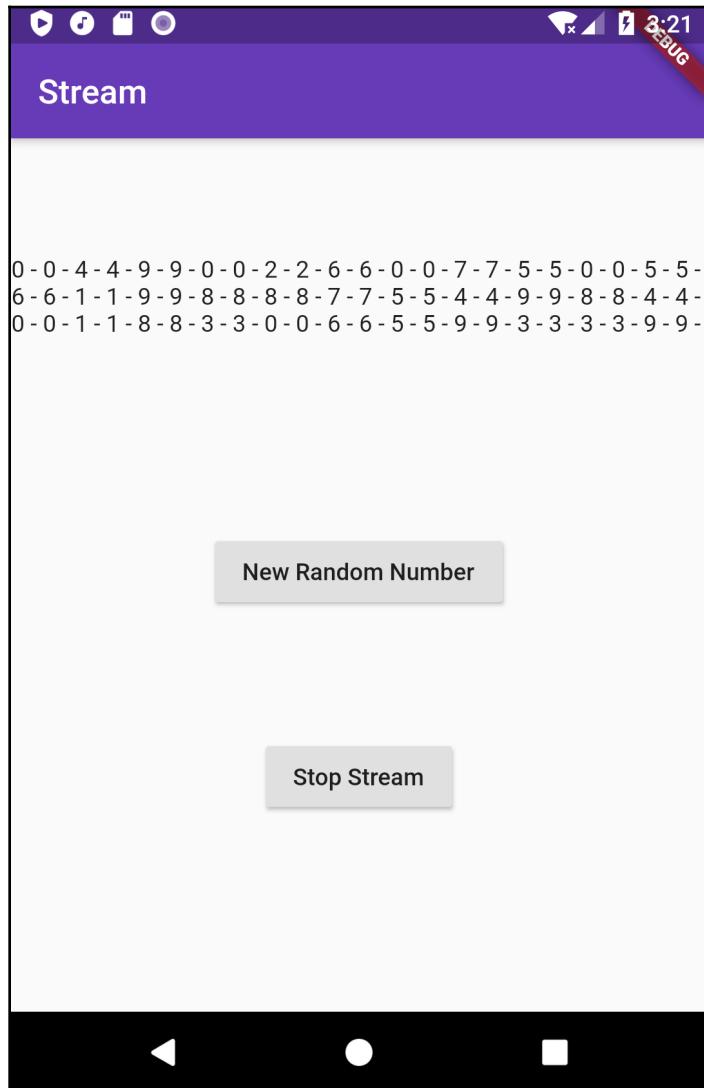
```
void initState() {  
    numberStream = NumberStream();  
    numberStreamController = numberStream.controller;  
    Stream stream =  
numberStreamController.stream.asBroadcastStream();  
    ...  
}
```

5. In the `build` method, edit the text in the column so that it prints the values string:

```
child: Column(  
    mainAxisSize: MainAxisSize.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 screenshot:



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 have seen 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, we will build an app from scratch. We recommend creating a new app to follow along.

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(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(MyApp());
}
class MyApp extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            title: 'Stream',
            theme: ThemeData(
                primarySwatch: Colors.deepPurple,
                visualDensity: VisualDensity.adaptivePlatformDensity,
            ),
            home: StreamHomePage(),
        );
    }
}
```

```
        );
    }
}

class StreamHomePage extends StatefulWidget {
    @override
    _StreamHomePageState createState() => _StreamHomePageState();
}

class _StreamHomePageState extends State<StreamHomePage> {
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                title: Text('Stream'),
            ),
            body: Container(
                ),
            );
    }
}
```

5. At the top of the `_StreamHomePageState` class, declare a stream of the `int` type called `numberStream`:

```
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: Container(
    child: StreamBuilder(
        stream: numberStream,
        initialData: 0,
        builder: (context, snapshot) {
            if (snapshot.hasError) {
                print('Error!');
            }
            if (snapshot.hasData) {
```

```
        return Center(
            child: Text(
                snapshot.data.toString(),
                style: TextStyle(fontSize: 96),
            )));
    } else {
        return Center();
    }
},
),
),
,
```

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:



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: 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. In 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

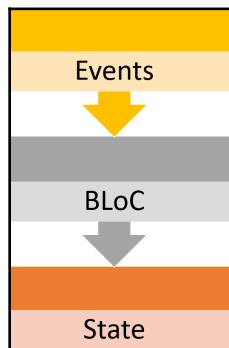
The documentation relating to `StreamBuilder` is complete and well laid out. Have a look at the official guide at <https://api.flutter.dev/flutter/widgets/StreamBuilder-class.html>.

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 here:



The main reason for using BLoCs is separating 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. The example we will build in this recipe will start from the previous one, and it's very simple, but it can be scaled as needed for bigger apps.

Getting ready

To follow along with this recipe, it is recommended to create a new app.

How to do it...

We will create a very simple countdown app, which goes from 60 to 0, using a BLoC:

1. Create a new file in the `lib` folder of your project and call it `countdown_bloc.dart`.
2. In the `countdown_bloc.dart` file, import the `dart:async` library:

```
import 'dart:async';
```

3. Still in the `countdown_bloc.dart` file, create a new class, called `TimerBLoC`:

```
class TimerBLoC {}
```

4. At the top of the `TimerBLoC` class, declare an integer, `StreamController`, a `Stream` getter, and `StreamSink`:

```
int seconds = 60;
StreamController _secondsStreamController = StreamController();
Stream get secondsStream =>
    _secondsStreamController.stream.asBroadcastStream();
StreamSink get secondsSink => _secondsStreamController.sink;
```

5. In the `TimerBLoC` class, create an `async` method called `decreaseSeconds`, which decreases the seconds and adds the new value to the sink:

```
Future decreaseSeconds() async {
    await Future.delayed(const Duration(seconds: 1));
    seconds--;
    secondsSink.add(seconds);
}
```

6. Still in the `TimerBLoC` class, create an `async` method called `countDown`, which calls the `decreaseSeconds` method until it reaches the value of 0:

```
countDown() async {
    for (var i = seconds; i > 0; i--) {
        await decreaseSeconds();
        returnSeconds(seconds);
    }
}
```

7. Create a method that returns the number of seconds:

```
int returnSeconds(seconds) {
    return seconds;
}
```

8. Create a `dispose` method to close the stream:

```
void dispose() {
    _secondsStreamController.close();
}
```

9. 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 'countdown_bloc.dart';

void main() {
    runApp(MyApp());
}
class MyApp extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            title: 'BLoC',
            theme: ThemeData(
                primarySwatch: Colors.deepPurple,
                visualDensity: VisualDensity.adaptivePlatformDensity,
            ),
            home: StreamHomePage(),
        );
    }
}
class StreamHomePage extends StatefulWidget {
    @override
```

```
_StreamHomePageState createState() => _StreamHomePageState();  
}  
  
class _StreamHomePageState extends State<StreamHomePage> {  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text('BLoC'),  
      ),  
      body: Container(  
    ),);  }}
```

10. At the top of the `_StreamHomePageState` class, declare a `TimerBLoC` and an integer containing the seconds:

```
TimerBLoC timerBloc;  
int seconds;
```

11. In the `_StreamHomePageState` class, override the `initState` method and set the initial values:

```
@override  
void initState() {  
  timerBloc = TimerBLoC();  
  seconds = timerBloc.seconds;  
  timerBloc.countDown();  
  super.initState();  
}
```

12. In the `build` method, add a `StreamBuilder` inside `Container`. This takes the stream created by the `BLoC` (`timerBloc.secondsStream`):

```
body: Container(  
  child: StreamBuilder(  
    stream: timerBloc.secondsStream,  
    initialData: seconds,  
    builder: (context, snapshot) {  
      if (snapshot.hasError) {  
        print('Error!');  
      }  
      if (snapshot.hasData) {  
        return Center(  
          child: Text(  
            snapshot.data.toString(),  
            style: TextStyle(fontSize: 96),
```

```
        );
    } else {
        return Center();
    }
},
),
),
,
```

Run the app. You should see the countdown running from 60 to 0.

How it works...

When you want to use a BLoC as a state management pattern, a number of steps need to be performed and these are as follows:

- Create a class that will serve as the BLoC.
- In the class, declare the data that needs to be updated in the app.
- Set StreamControllers.
- Create the getters for streams and sinks.
- Add the logic of the BLoC.
- Add a constructor in which you'll set the data.
- Listen to changes.
- Set a dispose method.
- From the UI, create an instance of the BLoC.
- Use StreamBuilder to build the widgets that will use the BLoC data.
- 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 countdown 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. Actually, the logic of this app is contained in the countDown method:

```
countDown() async {
    for (var i = seconds; i > 0; i--) {
        await decreaseSeconds();
        returnSeconds(seconds);
    }
}
```

Here, we await the result of the `decreaseSeconds` method (which returns the number of remaining seconds) for as many times as required (60 times in our example). The `decreaseSeconds` method waits for 1 second before decreasing the number of remaining seconds and adding the new value to `Sink`. By adding the value to the sink, any listener will receive the new value and update the user interface as required:

```
Future decreaseSeconds() async {
    await Future.delayed(const Duration(seconds: 1));
    seconds--;
    secondsSink.add(seconds);
}
```

See also

Implementing a BLoC manually requires some boilerplate code. While this is useful for understanding 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.

The BLoC pattern is one of the recommended 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>.

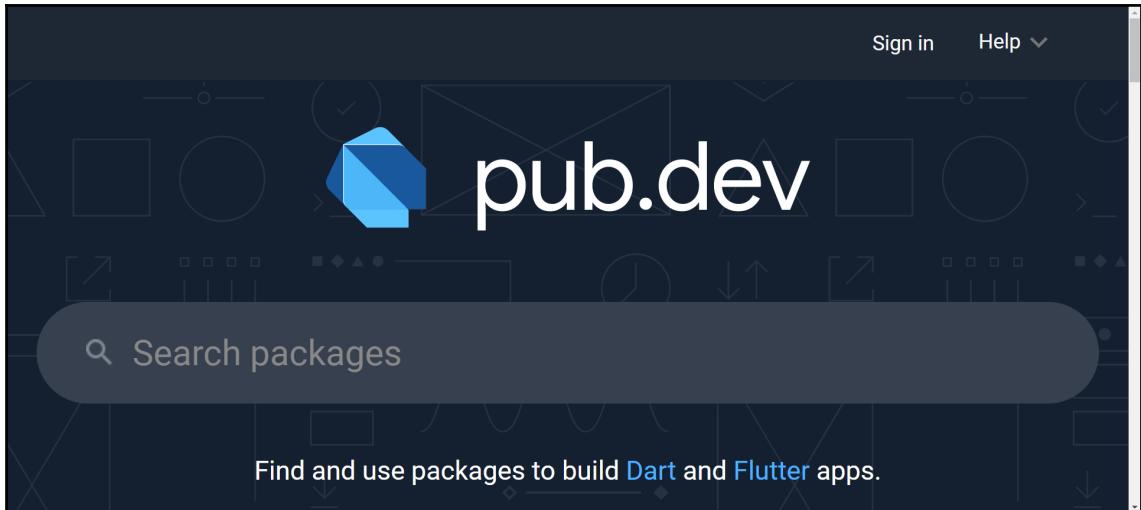
10

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 to 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 on the following website: <https://pub.dev>. This is the hub where you can go to search for packages, verify their platform compatibility (iOS, Android, web, and desktop), their 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:



In this chapter, we will cover the following topics:

- Importing packages and dependencies
- 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 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.
- 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/tree/master/chapter_10.

Importing packages and dependencies

This recipe shows how to get packages and plugins from <https://pub.dev> and integrate them into your app's `pubspec.yaml` file.

Specifically, you will retrieve the version and package name from `pub.dev`, import a package into the `pubspec.yaml` file in your project, download the package, 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...

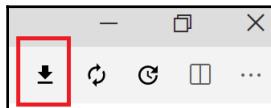
When you install a package from `pub.dev`, the process is very simple. For this example, we will install the `http` package, and connect to a Git repository:

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.
6. Copy the dependencies version in the **Depend on it** section at the top of the page.
7. Open the `pubspec.yaml` file in your project.
8. 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). Make sure that the alignment is precisely as shown here, where `http` is exactly under `flutter`:

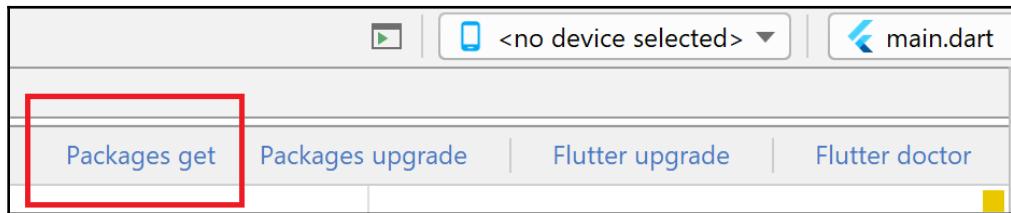
```
dependencies:  
  flutter:  
    sdk: flutter  
  http: ^0.13.1
```

9. To download the package, the actions depend on your system. In the **Terminal**, type `flutter pub get`. This will initiate downloading of the package.

10. 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):



11. In **Android Studio/IntelliJ Idea**, click the **Packages get** button at the top right of the window, as shown in the following screenshot:



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 are open source.
- Once published, packages cannot ever be removed.
- All packages can only depend on other published packages.

All these rules benefit the end user. If you use a package from `pub.dev`, you always know that your packages will remain available. They can be used without any limitations, and do not have hidden dependencies that follow different rules.

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: ^0.13.1
```

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 the package that you have added to the `pubspec.yaml` file, you also need to import it into the file where you are using the package with an `import` statement, such as the following:

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

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.

So, when you write `^0.13.1`, this means that any version is equal to or bigger than 0.13.1 and lower than 0.14.0.

When you write `^1.0.0`, this means that any version is equal to or bigger than 1.0.0 and lower than 2.0.0.

Tip: When you import packages, you usually need to find 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 are using 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 are using 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>.

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
- 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 just depend on it in your app. At the same time, code reuse 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, and other systems. 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.
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: ^0.17.0
```

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. In the `pubspec.yaml` file **of the main project**, add the dependency to the package you have just created:

```
area:  
  path: packages/area
```

9. At the top of the `main.dart` file of the main project, import the `area` package:

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

10. Remove the `MyHomePage` class created in the sample app.

11. Refactor the `MyApp` class, as shown here:

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'Packages Demo',  
      home: PackageScreen(),  
    );  
  }  
}
```

12. 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 {  
  @override  
  _PackageScreenState createState() => _PackageScreenState();  
}  
  
class _PackageScreenState extends State<PackageScreen> {  
  @override  
  Widget build(BuildContext context) {  
    return Container();  
  }  
}
```

13. 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 = '';
```

14. 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;  
  AppTextField(this.controller, this.label);
```

```
    @override
    Widget build(BuildContext context) {
        return Padding(
            padding: EdgeInsets.all(24),
            child: TextField(
                controller: controller,
                decoration: InputDecoration(hintText: label),
            ),
        );
    }
}
```

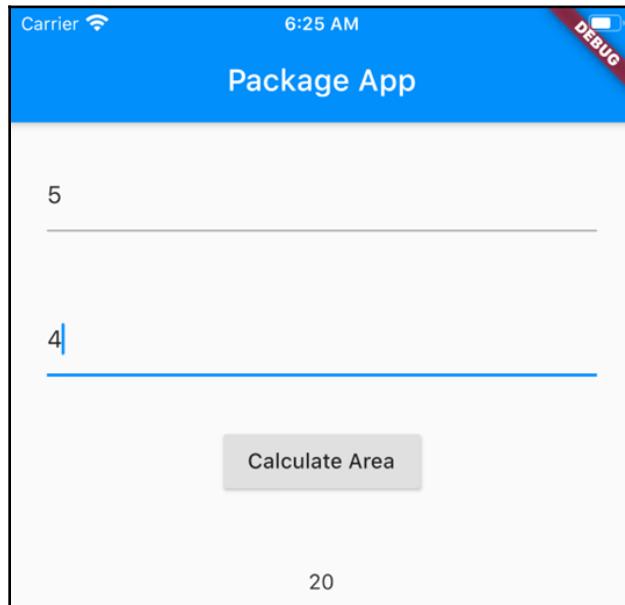
15. In the build method of the `_PackageScreenState` class, remove the existing code and return a `Scaffold`, containing an `appBar` and a `body`, as shown here:

```
return Scaffold(
    appBar: AppBar(
        title: Text('Package App'),
    ),
    body: Column(
        children: [
            AppTextField(txtWidth, 'Width'),
            AppTextField(txtHeight, 'Height'),
            Padding(
                padding: EdgeInsets.all(24),
            ),
            ElevatedButton(
                child: Text('Calculate Area'),
                onPressed: () {}),
            Padding(
                padding: EdgeInsets.all(24),
            ),
            Text(result),
        ],
    ),
);
```

16. In the `onPressed` function in `ElevatedButton`, add the code to call the `calculateAreaRect` method in the `area` package:

```
double width = double.tryParse(txtWidth.text);
double height = double.tryParse(txtHeight.text);
String res = calculateAreaRect(width, height);
setState(() {
    result = res;
});
```

17. Run the app.
18. 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:



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, containing the code of the package

A package can contain more than a single file in the `lib` folder, but must 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. It would make sense to have a single method that calculates both, but we need two methods for part 2 of this recipe, which follows.

In the `pubspec.yaml` file, we import the `intl` package:

```
intl: ^0.17.0
```

One of the great features of packages is that when you add a dependency there, 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.

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 options include a Git or the `pub.dev` repositories.

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:

```
class AppTextField extends StatelessWidget {  
  final TextEditingController controller;  
  final String label;  
  AppTextField(this.controller, this.label);  
  @override  
  Widget build(BuildContext context) {  
    return Padding(  
      padding: EdgeInsets.all(24),  
      child: TextField(  
        controller: controller,
```

```
        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 from 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 have 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. 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, remove the existing methods, and 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. 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
```

8. Add your own Git URL, or use the following git address in the dependencies:

```
area:  
  git: https://github.com/simoales/area.git
```

How it works...

In this recipe, you have seen how to create a package comprising 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, 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.



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.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 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.

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>.

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>.



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 2021 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
```

10. Run the flutter pub publish --dry-run command again and check that there are no more warnings.



Before running the command that follows, make sure your package adds real value to the Flutter community. **This specific package is probably 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 an open source 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 the parentheses (`youraddress.com`) and shows the text in the square brackets.

The `CHANGELOG.md` file is not required but highly recommended. It is also a Markdown file, which should contain all 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

This recipe shows how to add the Google Maps plugin 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. There are no prerequisites to follow along.

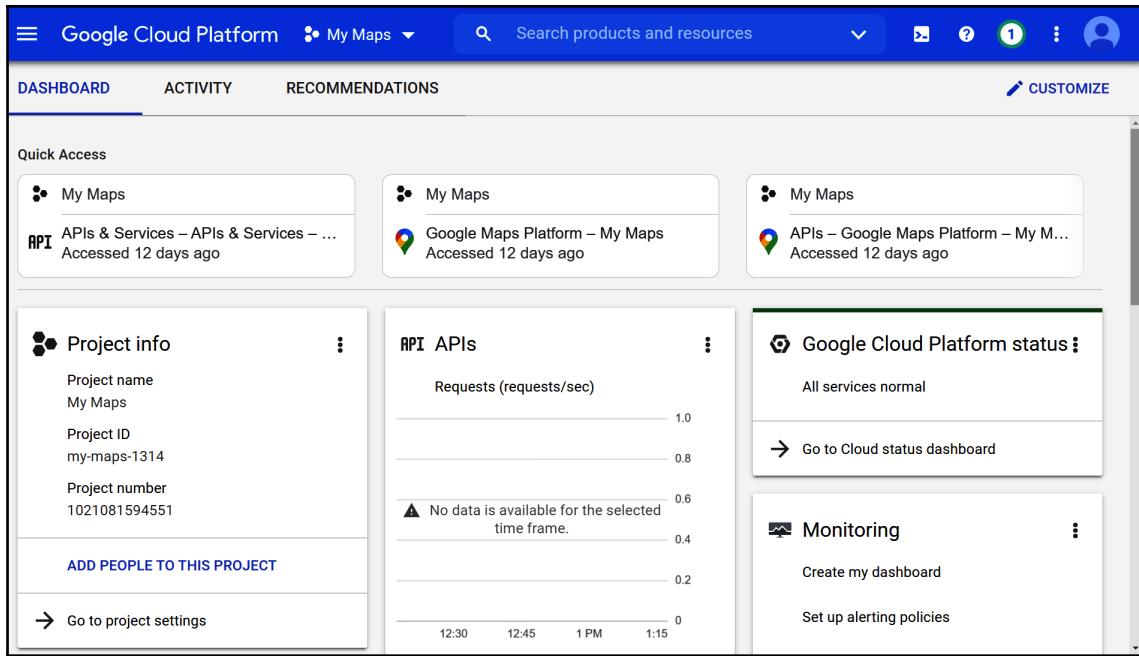
How to do it...

In this recipe, you will add Google Maps to your app, and the requisite steps to integrate it in your iOS or Android project:

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`:

```
dependencies:  
  google_maps_flutter: ^2.0.3
```

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:



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:

Enabled APIs				
Select an API to view details. Figures are for the last 30 days.				
API ↑	Requests	Errors	Avg latency (ms)	
Maps SDK for Android	0	0	-	Details
Maps SDK for iOS	0	0	-	Details

The following steps vary based on the platform you are using.

To add 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]" />
```

To add 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 {
  GMServices.provideAPIKey("YOUR API KEY HERE")
  GeneratedPluginRegistrant.register(with: self)
  return super.application(application,
    didFinishLaunchingWithOptions: launchOptions)
}
```

4. Opt into the preview of the embedded view. Open your project's ios/Runner/Info.plist file and add the following to the <dict> node:

```
<key>io.flutter.embedded_views_preview</key>
<true/>
```

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 {  
    @override  
    _MyMapState 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:

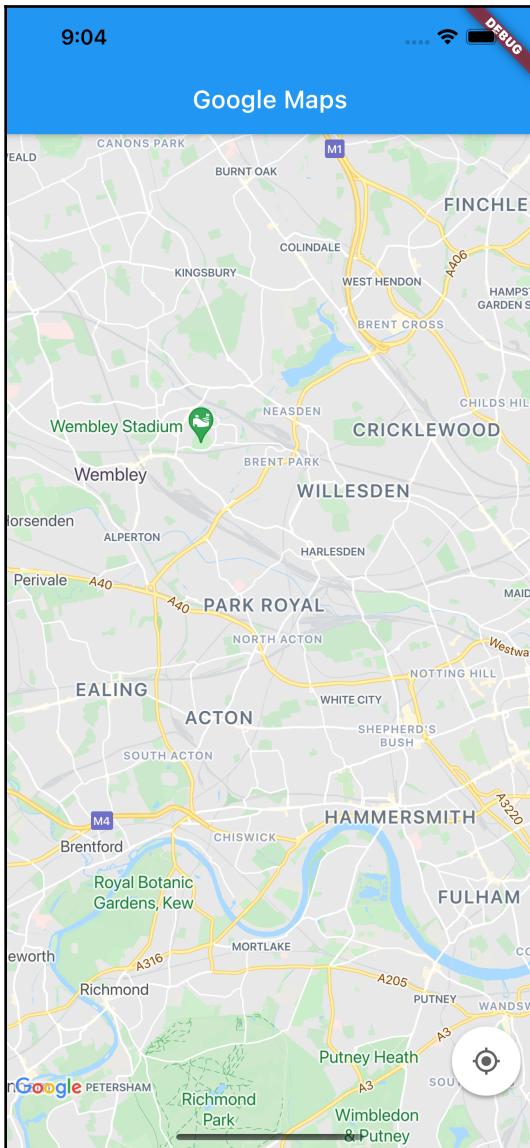
```
class MyApp extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            title: 'Map Demo',  
            theme: ThemeData(  
                primarySwatch: Colors.blue,  
            ),  
            home: MyMap(),  
        );  
    }  
}
```

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),  
        ),  
    );  
}  
}
```

Run the app. You should see a map covering the screen, showing the center of London:



How it works...

In this recipe, you have added Google Maps to your app. This requires three steps:

- Getting the Google Maps credentials and enabling them
- Configuring your app with the credentials
- 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 for using the API.



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 have seen, 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 in 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 `maps` plugin, available at <https://pub.dev/packages/maps>.

Using location services

In the previous recipe, you have seen how to show and position a map using Google Maps, with fixed coordinates. In this recipe, you will find the current position of the 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 into 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: ^4.1.1
```

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:

```
<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:

```
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 = LatLng(51.5285582, -0.24167);  
    }  
    return userPosition;  
}
```

6. In the `build` method of the `_MyMapState` class, enclose `GoogleMap` in a `FutureBuilder` object whose `future` property calls the `findUserLocation` method:

```
body: FutureBuilder(  
    future: findUserLocation(),  
    builder: (BuildContext context, AsyncSnapshot snapshot) {  
        return GoogleMap(  
            initialCameraPosition:  
                CameraPosition(target: snapshot.data, zoom: 12),
```

```
    );
},
)
```

7. Run the app. Now you should see your position on the map, or on an emulator, and the position set on the emulator itself.

How it works...

In order to find the current location of our user, you have 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 granted permission to retrieve their location. In the example in this recipe, you have used the following commands:

```
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 are not using in this recipe but might be useful for other apps.

In the `build` method, we are using 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 have seen 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 Map 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 `GooglePlaces` archive and point to any place in your maps with a marker.

Getting ready

You should have completed the previous two recipes, *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. Get back 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. 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';
```

3. At the top of the `_MyMapState` class, add a new `List` of `Marker` objects:

```
class _MyMapState extends State<MyMap> {  
  LatLng userPosition;  
  List<Marker> markers = [];
```

4. 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: Icon(Icons.map),
                    onPressed: () => findPlaces(),
                )
            ],
        ),
    );
}
```

5. 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),
);
```

6. At the bottom of the `_MyMapState` class, create a new asynchronous method, called `findPlaces`:

```
Future findPlaces() async {}
```

7. Inside the `findPlaces` method, add your Google Maps key and the base URL of the query:

```
final String key = '[Your Key Here]';
final String placesUrl =
'https://maps.googleapis.com/maps/api/place/nearbysearch/json?';
```

8. Under the declarations, add the dynamic part of the URL:

```
String url = placesUrl +
'key=$key&type=restaurant&location=${userPosition.latitude},${userP
osition.longitude}' + '&radius=1000';
```

9. 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 == 200) {
    final data = json.decode(response.body);
    showMarkers(data);
} else {
    throw Exception('Unable to retrieve places');
}
}
```

10. Create a new method, called `showMarkers`, that takes a `data` parameter:

```
showMarkers(data) {}
```

11. 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();
```

12. Create a `forEach` 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:

```
places.forEach((place) {
    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']));
});
```

13. Finally, update the `State` setting the markers:

```
setState(() {
    markers = markers;
});
```

14. Run the app. You should see a list of markers on the map, with all the restaurants near your position. If you 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 should look like this:

```
https://maps.googleapis.com/maps/api/place/nearbysearch/json?key=[YOUR  
KEY  
HERE]&type=restaurant&location=41.8999983,12.49639830000001&radius=1000  
0
```

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>.

11

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 but not as intuitive as building simple widgets. In this chapter, you will learn how to create effective animations and add them to your projects. This will allow you to create visually pleasing experiences for your users.

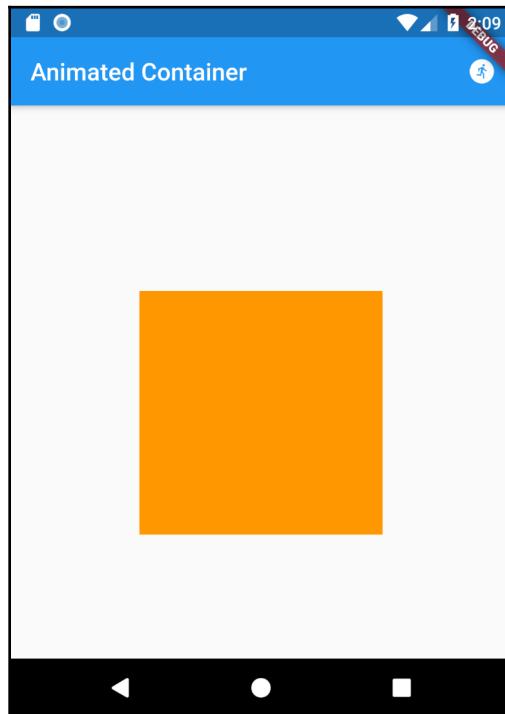
In this chapter, we will cover the following topics:

- Creating basic container animations
- Designing animations part 1 – VSync and 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 in the following screenshot will change color, size, and the top margin:



After following this recipe, you will understand how to work with the `AnimatedContainer` widget, which allows the creation of simple animations with a `Container`.

Getting ready

In this recipe, you will create a new project. There are no prerequisites to follow along.

How to do it...

In the following steps, you will create a new screen that shows an animation with an `AnimatedContainer` widget:

1. Create a new Flutter app and call it `myanimations`.
2. Remove the `MyHomePage` class created in the sample app.
3. Refactor the `MyApp` class as shown here:

```
import 'package:flutter/material.dart';

void main() {
    runApp(MyApp());
}

class MyApp extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            title: 'Animations Demo',
            theme: ThemeData(
                primarySwatch: Colors.blue,
            ),
            home: MyAnimation(),
        );
    }
}
```

4. Create a file called `my_animation.dart` and add a new stateful widget called `MyAnimation`, using the `stf` shortcut. The result is shown here:

```
class MyAnimation extends StatefulWidget {
    @override
    _MyAnimationState createState() => _MyAnimationState();
}

class _MyAnimationState extends State<MyAnimation> {
    @override
    Widget build(BuildContext context) {
        return Container();
    }
}
```

5. Remove the container in the `_MyAnimationStateClass`, 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 code is shown here:

```
Widget build(BuildContext context) {  
    return Scaffold(  
        appBar: AppBar(  
            title: Text('Animated Container'),  
            actions: []  
        ),  
        body: Center());
```

6. At the top of the `_MyAnimationState` class, add a `List` of colors:

```
final List<Color> colors = [  
    Colors.red,  
    Colors.green,  
    Colors.yellow,  
    Colors.blue,  
    Colors.orange  
];
```

7. Under the `List`, add an integer state variable, called `iteration`, which will start with a value of 0:

```
int iteration = 0;
```

8. In the body of the `Scaffold`, add a `Center` widget that contains 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: Duration(seconds: 1),  
        color: colors[iteration],  
    )),
```

9. In the `actions` property of the `AppBar`, add an `IconButton`, whose icon is the `run_circle` icon.

10. In the `onPressed` parameter, if the value of `iteration` is less than the length of the `colors` list, increment `iteration` by 1; otherwise, reset it to 0.
Then call the `setState` method, setting the new `iteration` value:

```
actions: [
    IconButton(
        icon: Icon(Icons.run_circle),
        onPressed: () {
            iteration < colors.length - 1 ? iteration++ : iteration = 0;
            setState(() {
                iteration = iteration;
            });
        },
    ),
],
```

11. 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 at 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.

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 have 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: 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.



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;  
  });  
,
```

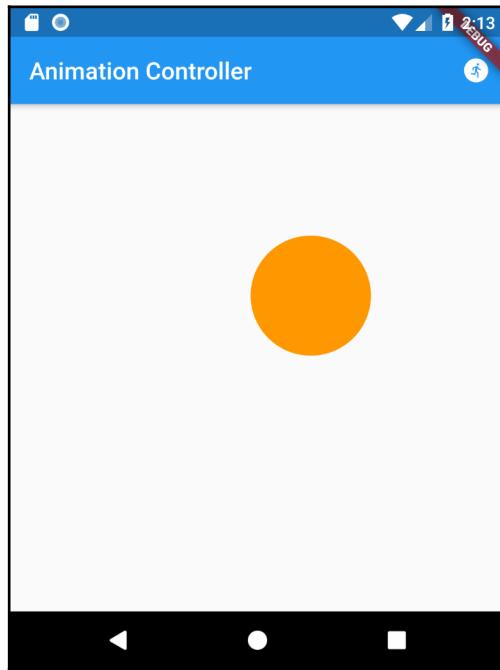
See also

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 the following screenshot:



Getting ready

To follow along with this recipe, you need any existing Flutter project already created, 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 through 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 `stfulWidget` shortcut, and call it `ShapeAnimation`. The result is shown here:

```
class ShapeAnimation extends StatefulWidget {  
    @override  
    _ShapeAnimationState 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 {  
    @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: BoxDecoration(color: Colors.orange, shape:  
        BoxShape.circle),  
);
```

6. At the top of the `_ShapeAnimationState` class, declare an `AnimationController` called `controller`:

```
AnimationController controller;
```

7. In the `_ShapeAnimationState` class, override the `initState` method:

```
@override  
void initState() {  
    super.initState();  
}
```

8. 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,  
>;
```

9. In the `_ShapeAnimationState` declaration, add the `with SingleTickerProviderStateMixin` instruction:

```
class _ShapeAnimationState extends State<ShapeAnimation> with  
SingleTickerProviderStateMixin {
```

10. At the top of the `_ShapeAnimationState` class, add the declaration of an `Animation` of type `double`:

```
Animation<double> animation;
```

11. Under the animation declaration, declare a `double` called `pos`:

```
double pos = 0;
```

12. 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 `right` properties to `pos`, as shown here:

```
return Scaffold(  
    appBar: AppBar(  
        title: Text('Animation Controller'),  
>,  
    body: Stack(<
```

```
        children: [
            Positioned(left: pos, top: pos, child: Ball()),
        ],);}
```

13. At the bottom of the `_ShapeAnimationState` class, create a new method, called `moveBall`, that will change the `pos` state value to the `value` property of animation:

```
void moveBall() {
    setState(() {
        pos = animation.value;
    });
}
```

14. In the `initState` method, set the animation with a `Tween` of type `double`, with 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();
});
```

15. 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: Icon(Icons.run_circle),
    ),
],
```

16. At the top of the `main.dart` file, import `shape_animation.dart`:

```
import './shape_animation.dart';
```

17. In the build method of the `MyApp` class, set the home of the `MaterialApp` to `ShapeAnimation`:

```
home: ShapeAnimation(),
```

18. Run the app, and observe the movement of the ball on the screen.

How it works...

This recipe contains the preparation for the next two recipes that follow, and you have added a few important features that make possible animating widgets in Flutter. In particular, there are three important classes that you have used:

- Animation
- Tween
- AnimationController

You have declared an `Animation` at the top of the `_ShapeAnimationState` class, with the following declaration:

```
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. `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-between") 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, 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 could 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 that 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 towards the bottom and right of the screen.

See also

The Flutter team invested a lot of resources in creating and documenting animations, as they are one of the features that make Flutter stand out compared to other frameworks. The first place to start understanding the internals of how animations work is the [animations tutorial](https://flutter.dev/docs/development/ui/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 moving 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;
Animation<double> animationTop;
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:
200).animate(controller);
animationTop = Tween<double>(begin: 0, end:
400).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.

How it works...

An `AnimationController` controls one or more animations. In this recipe, you've seen 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 times.

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

The linear movement exists only in theoretical physics. Why not make the movement a bit more realistic with the ease in and ease out curves? 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 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 the maximum left coordinate, a new `Animation` of type `double` (animation may already exist if you didn't delete it in the previous recipe), and a `final int` containing the size of the ball:

```
double maxTop = 0;
double maxLeft = 0;
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: 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,
);
```

- Under the animation setting, add the listener, as shown here:

```
animation
..addListener(() {
moveBall();
});
```

- In the moveBall method, set posTop and posLeft as shown here:

```
void moveBall() {
setState(() {
posTop = animation.value * maxTop;
posLeft = animation.value * maxLeft;
});
}
```

- Run the app and observe the movement of the ball.

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 intrusions by the operating system, 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've 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, when the animation completed with a value of `1`.

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" 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: Positioned(left: posLeft, top: posTop, child: Ball()),
    builder: (BuildContext context, Widget child) {
        moveBall();
        return Positioned(left: posLeft, top: posTop, child: Ball());
    }]]);
}
```

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 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.

When 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" animation. This is a recent addition to the Flutter framework, and as you have seen in this recipe, it makes it very easy to create this kind of animation.

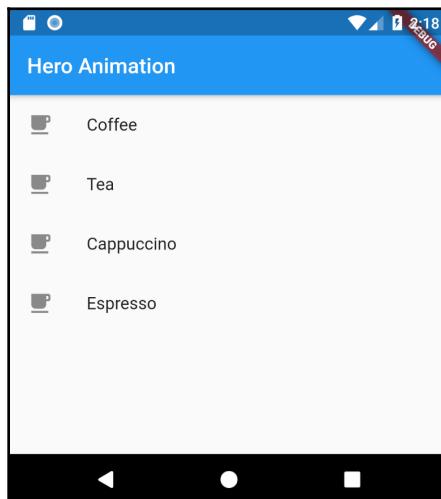
See also

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-use-animatedbuilder-or-animatedwidget-57ecae0959e8>.

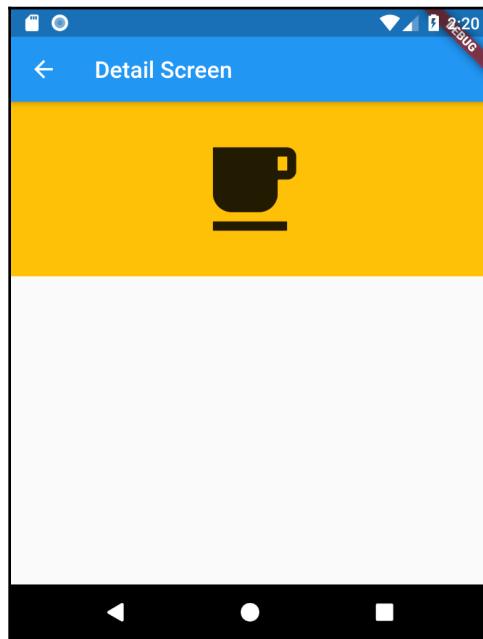
Using Hero animations

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 screenshot. The first screen contains a List:



When the user clicks on one of the items in the List, the second screen appears with an 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 `listscreen.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 {  
  @override  
  Widget build(BuildContext context) {  
    return Container();  
  }  
}
```

4. At the top of the `ListScreen` widget, add a new `List` of strings, called `drinks`:

```
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: 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:  
    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 `DetailScreen.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 {  
    @override  
    Widget build(BuildContext context) {  
        return Container();  
    }  
}
```

12. At the top of the `DetailScreen` widget, add a `final int` called `index`, and a constructor that allows setting the index:

```
final int index;  
DetailScreen(this.index);
```

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: 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: BoxDecoration(color: Colors.amber),
            child: Hero(
                tag: 'cup$index',
                child: Icon(Icons.free_breakfast, size: 96, ),
            ),
        ),
    ),
    Expanded(
        flex: 3,
        child: Container(),
    ),
],
```

15. At the top of the listscreen.dart file, import detailscreen.dart:

```
import './detailscreen.dart';
```

16. In the main.dart file, in the home of the MaterialApp, call ListScreen:

```
home: ListScreen(),
```

17. Run the app, and tap on any item in the ListView, and observe the animation to get to the DetailScreen.

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: Icon(Icons.free_breakfast)),
```

By concatenating the index to the "cup" string, we achieved a unique tag.



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>.

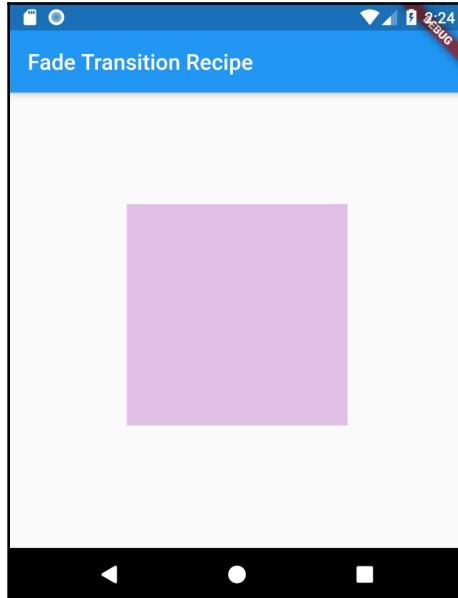
Using premade animation transitions

You can use transition widgets to create animations in an easier way than using traditional animations. The Flutter framework contains several pre-made transitions that make animating objects extremely straightforward. These include the following:

- `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:



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...

By taking the following steps, you will implement the `FadeTransition` widget in your app to make a square 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 {
  @override
  _FadeTransitionScreenState 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`:

```
AnimationController controller;  
Animation 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() {  
    controller = AnimationController(vsync: this, duration:  
        Duration(seconds: 3));  
    super.initState();  
}
```

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 top of the `build` method, call the `forward` method on the `AnimationController`:

```
controller.forward();
```

9. Still 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 the color purple, as shown here:

```
@override  
Widget build(BuildContext context) {  
    controller.forward();  
    return Scaffold(  
        appBar: AppBar(  
            title: 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`, call the `FadeTransitionScreen` widget:

```
home: 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: 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.



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 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 controller `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.

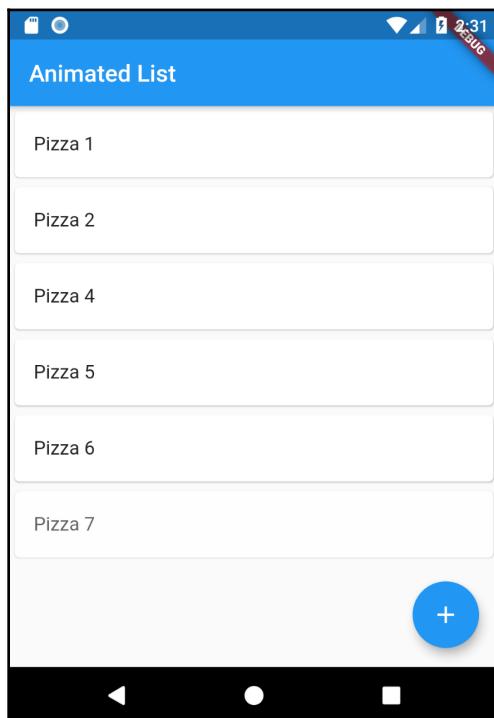
See also

`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 to show 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:



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, we will use a `FadeTransition` into an `AnimatedList`:

1. Create a new file in the `lib` folder, called `animatedlist.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 {
  @override
  _AnimatedListScreenState 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` that 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 = 0;
```

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 on the list through the index that was passed. Then return a `FadeTransition` that 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.
In the `onTap` parameter of the `ListTile`, call the `removePizza` method, which you will create in the next step (this will show an error that you can ignore).
The full code of the `fadeListTile` method is shown here:

```
fadeListTile(BuildContext context, int index, Animation animation)
{
    int item = _items[index];
    return FadeTransition(
        opacity: Tween(begin: 0.0, end: 1.0).animate(animation),
        child: Card(
            child: ListTile(
                title: Text('Pizza ' + item.toString()),
                onTap: () {
                    removePizza(index);
                },
            ),
        ),
    );
}
```

8. Add another method, called `removePizza`, which takes an integer containing the item that will be removed.
9. 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.

At the bottom of the method, also remove the deleted item from the `_items` List.
The code for the `removePizza` method is shown here:

```
removePizza(int index) {
    int animationIndex = index;
    if (index == _items.length - 1) animationIndex--;
    listKey.currentState.removeItem(
        index,
        (context, animation) => fadeListTile(context, animationIndex,
            animation),
        duration: Duration(seconds: 1),
    );
    _items.removeAt(index);
}
```

10. Add a new method in the `_AnimatedListScreenState` class, called `insertPizza`.
11. Inside the function, call the `insertItem` method using the `GlobalKey` that was created at the top of the class.
This method takes an integer with the position of the item that should be added, and an optional duration.
12. At the end of the method, also add the item in the `_itemsList`, incrementing the counter.

The code of the `insertPizza` method is shown here:

```
insertPizza() {  
    listKey.currentState.insertItem(  
        _items.length,  
        duration: Duration(seconds: 1),  
    );  
    _items.add(++counter);  
}
```

13. In the `build` method of the `_AnimatedListScreenState` class, return a `Scaffold`.
14. 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: Text('Animated List'),  
        ),  
        body: AnimatedList(  
            key: listKey,  
            initialItemCount: _items.length,  
            itemBuilder: (BuildContext context, int index, Animation  
                animation) {  
                return fadeListTile(context, index, animation);  
            },  
        ),  
    ),
```

15. Add a floatingActionButton to the Scaffold. This will show an add icon, and when pressed will call the insertPizza method:

```
floatingActionButton: FloatingActionButton(  
    child: Icon(Icons.add),  
    onPressed: () {  
        insertPizza();  
    },  
,
```

16. In the main.dart file, in the home of the MaterialApp, call the AnimatedListScreen widget:

```
home: AnimatedListScreen(),
```

17. Run the app and try adding and removing items from the list, and notice the animation showing with each action you perform.

How it works...

The layout for this recipe was very simple, only containing the list view, and a FloatingActionButton to add new items in the AnimatedList.

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 animation) {  
    return fadeListTile(context, index, animation);  
  },  
)
```

Specifically, in the `itemBuilder` parameter, we called the `fadeListTile` method:

```
fadeListTile(BuildContext context, int index, Animation animation) {  
  int item = _items[index];  
  return FadeTransition(  
    opacity: Tween(begin: 0.0, end: 1.0).animate(animation),  
    child: Card(  
      child: ListTile(  
        title: Text('Pizza ' + item.toString())  
      ),  
    ),
```

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`.



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: 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.



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 animationIndex = index;
    if (index == _items.length - 1) animationIndex--;
    listKey.currentState.removeItem(
        index,
        (context, animation) => fadeListTile(context, animationIndex,
            animation),
        duration: 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 on a Fade transition.

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`, like when you swipe away an email to delete it.

There's a very useful widget in Flutter designed to remove an item responding 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, 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 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 {
  @override
  _DismissibleScreenState 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: Text('Dismissible Example'),
        ),
        body: ListView.builder();
    );
}
```

6. In the `ListView.builder`, set the `itemCount` and `itemBuilder` parameters, as shown here:

```
itemCount: sweets.length,
itemBuilder: (context, index) {
    return Dismissible(
        key: Key(sweets[index]),
        child: ListTile(
            title: Text(sweets[index]),
        ),
        onDismissed: (direction) {
            sweets.removeAt(index);
        },
    );
},
```

7. In the home of the `MaterialApp` in the `main.dart` file, 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 of 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 for both directions.
- Inside the function, you just call the `removeAt` method on the `sweets` list to remove the item from the `List`.



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`.

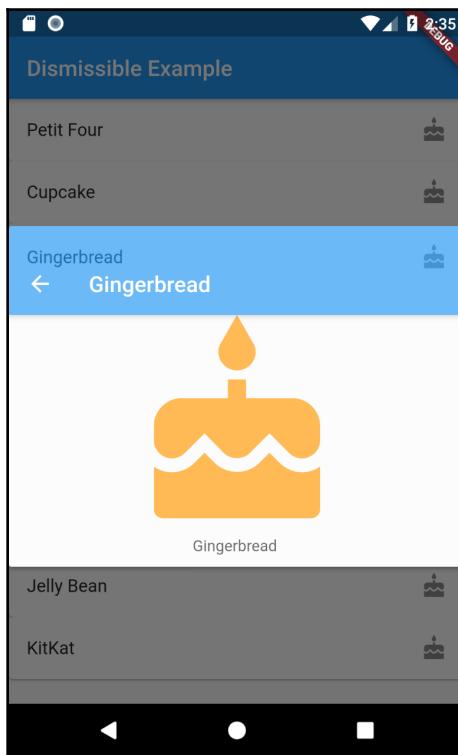
See also

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-class.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 that 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** that 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:



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 the pubspec.yaml file, import the animations dependency:

```
dependencies:  
  flutter:  
    sdk: flutter  
  animations: ^2.0.0
```

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: Duration(seconds: 3),  
transitionType: ContainerTransitionType.fade,
```

5. Still in the `OpenContainer` object, add the `openBuilder` parameter, containing a `Scaffold` with the detailed view of the selected `ListTile`:

```
openBuilder: (context, closeContainer) {  
    return Scaffold(  
        appBar: AppBar(  
            title: Text(sweets[index]),  
        body: Center(  
            child: Column(  
                children: [  
                    Container(  
                        width: 200,  
                        height: 200,  
                        child: Icon(  
                            Icons.cake,  
                            size: 200,  
                            color: Colors.orange,  
                        ),  
                    ),  
                    Text(sweets[index])  
                ],  
            ),  
        );  
    },  
},
```

7. Run the app and observe the transition from the `ListTile` to the `Scaffold` and vice versa.

How it works...

In this recipe, you have used a **Container Transform** transition. Basically, you have transformed a `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) has been completely managed by the `OpenContainer` widget.

There are two important properties that we have 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: [
                    Container(
                        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>.

12

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 app, 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 in 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, and saving 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 topics:

- Configuring a Firebase app
- Creating a login form
- Adding Google 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. Following configuration, you can add all the other Firebase services, including data and file storage, sign-in, and notifications.

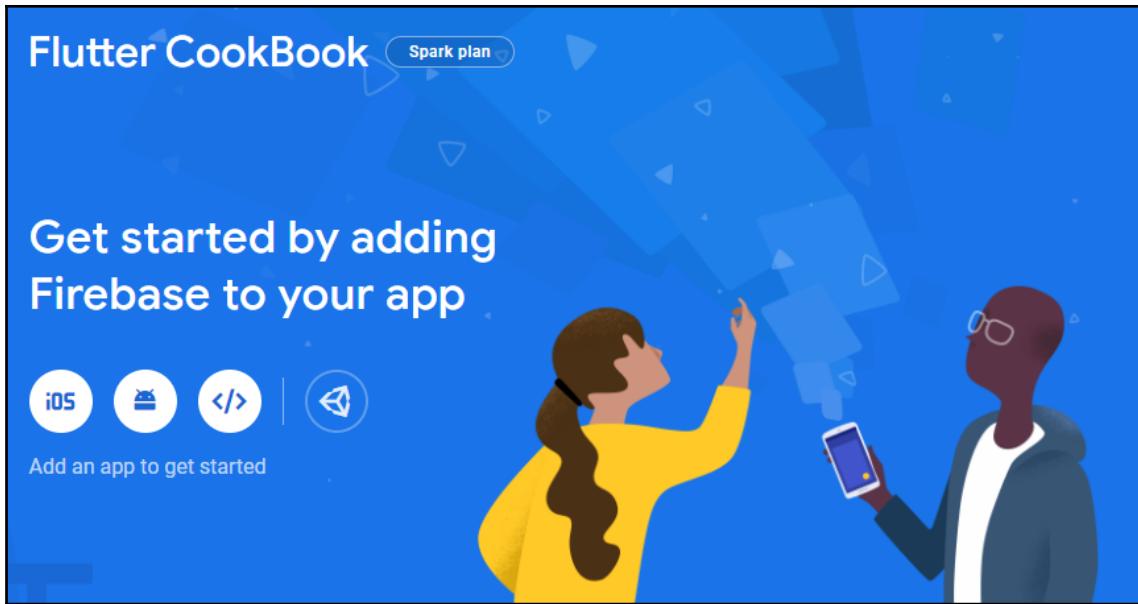
Getting ready

In this recipe, you will create a new project. A Google account is required to use Firebase, and this is a prerequisite for the whole chapter.

How to do it...

In this first recipe in this chapter, 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. Perform the following steps:

1. Open your browser at the Firebase console at <https://console.firebaseio.google.com/>.
2. Enter your Google credentials, or create a new account.
3. From the Firebase console, click the `Add Project` (or `New Project`) link.
4. In the **Create a Project** page, insert the title, `Flutter CookBook`.
5. Click the **Continue** button.
6. On the *Step 2* page, enable the **Google Analytics** feature.
7. On the *Step 3* page, select an analytics account or create a new one.
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 the following screenshot:



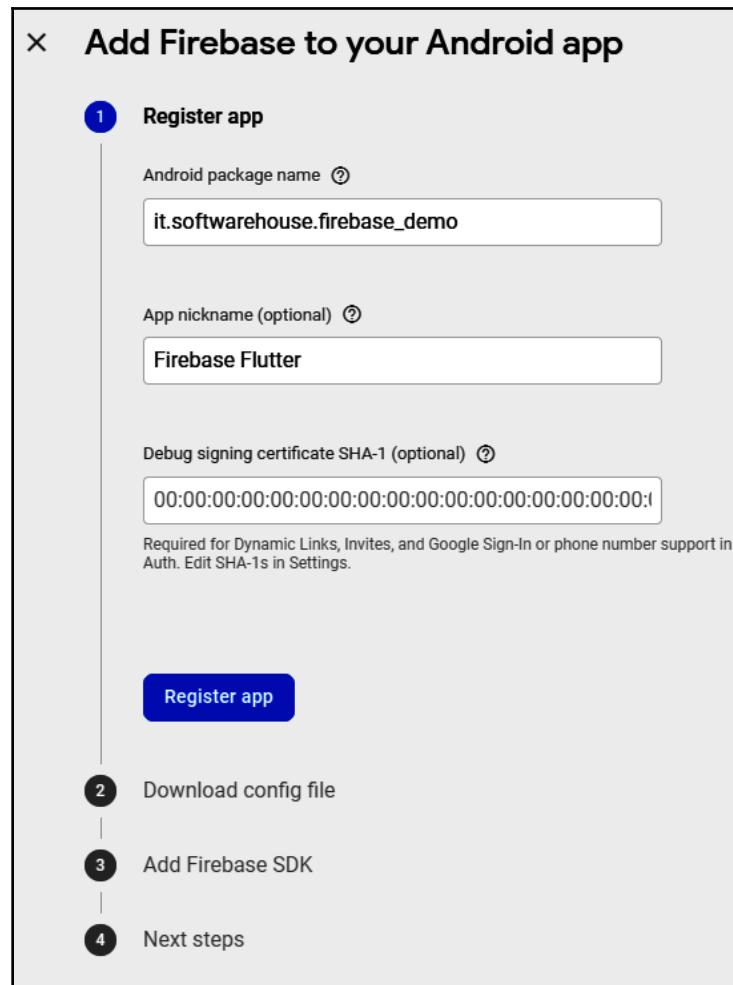
Your Firebase project is now ready. Now you need to configure a Flutter app so that it can connect to the new project. The process is different for Android and iOS, and **you need to complete both if you want to create an app for both Android and iOS**. Create a new Flutter app and call it `firebase_demo`.

Android configuration

Let's now see the steps required to configure Firebase for an **Android** device. Perform the following steps:

1. Open the file at the path `<project-name>/android/app/build.gradle`.
2. Retrieve the `applicationId` key in the `defaultConfig` node, delete the `com.example.firebaseio_demo` value, and add your own domain (or your name and surname); for example `it.softwarehouse.firebaseio_demo` (you will require a name that **uniquely identifies your app**).
3. From the Firebase project's overview page, click the **Android** button under the **Get started by adding Firebase to your app** button at the top of the screen.

4. Set the package name to the applicationId you have chosen and the nickname to `Firebase Flutter`, as shown in the following screenshot:



5. Click on the **Register app** button.
6. Download the `google-services.json` file.
7. Move the file into the `android/app` folder of your project.

8. In the `build.gradle` file in the app folder, add the following command under the `apply plugin: com.android.application` command:

```
apply plugin: 'com.google.gms.google-services'
```

9. In your project-level Gradle file (`android/build.gradle`), add the following rule (please check the latest version of the `google-services` plugin at the following address: <https://developers.google.com/android/guides/google-services-plugin>):

```
dependencies {  
    ...  
  
    classpath 'com.google.gms:google-services:4.3.4'  
}
```

10. Add the following dependencies to the `pubspec.yaml` file (please check the latest versions at <https://firebase.flutter.dev/>):

```
firebase_core: ^0.5.3  
firebase_auth: ^0.18.4  
cloud_firestore: ^0.14.4
```



In Android, you may need to update the `minSdkVersion` property in the app's `build.gradle` file. Currently, the minimum supported version is 21, so in the `defaultConfig` node, you should add the following:

```
minSdkVersion 21
```

Your Android app is now configured to use Firebase.

iOS configuration

Now, let's see the steps required to configure Firebase for an iOS device. In order to configure your app for iOS, you will need to change the `bundle Id` of the flutter project. This is the value that identifies an iOS app:

1. Open your app with Xcode (you may just open the app's iOS directory).
2. Get to the **General** tab in the top-level `Runner` directory.

3. Set the **Bundle Identifier** value to a string that uniquely identifies your project.
4. Save your project and get back to the Firebase console.
5. From the Firebase Project Overview page, click the **Add App** button and then choose **iOS**.
6. Insert the iOS bundle ID that you have chosen previously.
7. Click the **Register app** button.
8. Click **Download GoogleService-Info.plist** to get the Firebase iOS configuration file, named `GoogleService-Info.plist`.
9. In Xcode, move the downloaded file into the app's `Runner` directory.
10. Back at the Firebase console, click **Next**.
11. You can now skip the remaining steps of the configuration.

You have now completed the iOS configuration.

Adding Firebase dependencies

This final part should be completed in the app, regardless of the system you are using. Perform the following steps to add the Firebase dependencies:

- Add the following dependencies to `pubspec.yaml` (please check the latest versions at <https://firebase.flutter.dev/>):

```
firebase_core: ^1.1.0
firebase_auth: ^1.1.2
cloud_firestore: ^1.0.7
```

You now have completed the configuration of both an iOS and an Android app to use the Firebase platform.

How it works...

FlutterFire is a set of Flutter plugins that enable Flutter apps to use Firebase services.

The steps required to integrate FlutterFire into a Flutter app are as follows:

- **Creating a Firebase project:** This is the top-level entity for Firebase. Each feature you add to your apps belongs to a Firebase project. You completed this task in points 1 to 9 in the *How to do it...* section of this recipe.
When you create a new project in Firebase, you'll need to choose a **project name**. This is your project's identifier and is from 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.
- **Registering your android, iOS, web, and desktop apps in the Firebase console:** If you are planning to release your app on multiple platforms, **use the same project to register all your apps**. You completed this task in the *Android configuration* and *iOS configuration* sections.
Firebase needs to identify your apps: for **Android**, the app's `build.gradle` file contains several settings, including `applicationId`, that uniquely identify your app. For **iOS**, you achieve the same by accessing the **General** tab in the top-level `Runner` directory and changing the **Bundle Identifier** value.
In both cases, you need to download a configuration file from the Firebase console.
- **Adding the configuration file to your project(s):** This is platform-specific: `google-services.json` for Android and `GoogleService-Info.plist` for iOS. Both contain the project setup information you added in the Firebase console, and later changes you make to the project may require the file(s) to be downloaded again.
- **Adding the required dependencies to the `pubspec.yaml` file:** This is `firebase_core` and the other services you require in your app.

Enabling the analytics feature is a requirement for the Firebase Analytics integration, which will be completed in a later recipe in this chapter.



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 of 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 the 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. For a full and updated list of the available services and packages, take a look at the following page: <https://firebase.flutter.dev>.

See also

Some Firebase services are now available for web and desktop as well. The updated list and compatibility information is available on the official FlutterFire page at <https://firebase.flutter.dev>.

Creating a login form

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.

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, get to the **Authentication** option inside the **Build** section of the Project's dashboard and click on the **Get Started** button.
2. Click on the **Sign-in** method tab.
3. Enable the **Email/Password** authentication method.
4. Get back to your Flutter project, and create a new file in the `lib` folder and call it `login_screen.dart`.
5. At the top of the `login_screen.dart` file, import the `material.dart` library:

```
import 'package:flutter/material.dart';
```

6. Create a new stateful widget using the `stful` shortcut and call it `LoginScreen`.
7. At the top of the `_LoginScreenState` class, add the following variables:

```
String _message = '';
bool _isLogin = true;
final TextEditingController txtUserName = TextEditingController();
final TextEditingController txtPassword = TextEditingController();
```

8. Create a method that returns a custom widget for `UserName`, as shown here:

```
Widget userInput() {
    return Padding(
        padding: EdgeInsets.only(top: 128),
        child: TextFormField(
            controller: txtUserName,
            keyboardType: TextInputType.emailAddress,
            decoration: InputDecoration(
                hintText: 'User Name', icon: Icon(Icons.verified_user)),
            validator: (text) => text.isEmpty ? 'User Name is required'
                : '',
        )));
}
```

9. Follow the same pattern to create a method that returns a custom widget for the password, as shown here:

```
Widget passwordInput() {
    return Padding(
        padding: EdgeInsets.only(top: 24),
        child: TextFormField(
            controller: txtPassword,
            keyboardType: TextInputType.emailAddress,
            obscureText: true,
            decoration: InputDecoration(
                hintText: 'password', icon:
                    Icon(Icons.enhanced_encryption)),
            validator: (text) => text.isEmpty ? 'Password is required'
                : '',
        ));
}
```

10. Create a method that returns a custom widget for the main button of the screen, as shown here:

```
Widget btnMain() {
    String btnText = _isLogin ? 'Log in' : 'Sign up';
    return Padding(
        padding: EdgeInsets.only(top: 128),
        child: Container(
            height: 60,
            child: ElevatedButton(
                style: ButtonStyle(
                    backgroundColor: MaterialStateProperty.all
                        (Theme.of(context).primaryColorLight),
                    shape: MaterialStateProperty.all
                        <RoundedRectangleBorder>(
                            RoundedRectangleBorder(
                                borderRadius: BorderRadius.circular(24.0),
                                side: BorderSide(color: Colors.red)),
                        ),
                ),
                child: Text(
                    btnText,
                    style: TextStyle(
                        fontSize: 18, color:
                            Theme.of(context).primaryColorLight),
                ),
                onPressed: () {}
            )));
}
```

11. Create a method that returns a custom widget for the secondary button of the screen, as shown here:

```
Widget btnSecondary() {
    String buttonText = _isLogin ? 'Sign up' : 'Log In';
    return TextButton(
        child: Text(buttonText),
        onPressed: () {
            setState(() {
                _isLogin = !_isLogin;
            });
        });
}
```

12. Create one more method for this screen that returns `Text` containing the validation message that the user will receive in the event of an input error:

```
Widget txtMessage() {  
    return Text(  
        _message,  
        style: TextStyle(  
            fontSize: 16, color: Theme.of(context).primaryColorDark,  
            fontWeight: FontWeight.bold),  
    );}
```

13. Edit the build method of the `State` class, placing the widgets returned by the methods you have created in a `ListView` widget, which is contained in a `Container` with some padding, as shown in the following code:

```
@override  
Widget build(BuildContext context) {  
    return Scaffold(  
        appBar: AppBar(  
            title: Text('Login Screen'),  
        ),  
        body: Container(  
            padding: EdgeInsets.all(36),  
            child: ListView(  
                children: [  
                    userInput(),  
                    passwordInput(),  
                    btnMain(),  
                    btnSecondary(),  
                    txtMessage(),  
                ],  
            ),  
        ),  
    );  
}
```

14. At the top of the `_LoginScreenState` class, add another declaration, as shown here (the `FirebaseAuthentication` class does not exist yet, so this will raise an error, but you will create it in the next steps):

```
FirebaseAuthentication auth;
```

15. Override the `initState` method by calling the Firebase `initializeApp` asynchronous method, which creates an instance of a Firebase app. When the method completes, set `auth` `FirebaseAuthentication` so that it takes a new instance of `FirebaseAuthentication`:

```
@override  
void initState() {  
    Firebase.initializeApp().whenComplete(() {  
        auth = FirebaseAuth();  
        setState(() {});  
    });  
    super.initState();  
}
```

16. In the `main.dart` file, import `loginscreen.dart` at the top of the file, and edit the `MyApp` class so that `primarySwatch` is `deepPurple` and the `home` calls the `LoginScreen` class:

```
class MyApp extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            title: 'Flutter Demo',  
            theme: ThemeData(  
                primarySwatch: Colors.deepPurple,  
            ),  
            home: LoginScreen(),  
        );  
    }  
}
```

17. Create a new directory in the `lib` folder of your app and call it `shared`.
18. In the `shared` folder, add a new file and call it `firebase_authentication.dart`.
19. At the top of the new file, import the `firebase_auth` package and `dart:async`:

```
import 'package:firebase_auth/firebase_auth.dart';  
import 'dart:async';
```

20. Under the `import` statements, create a class and call it `FirebaseAuthentication`:

```
class FirebaseAuth {}
```

21. At the top of the class, create an instance of FirebaseAuth:

```
final FirebaseAuth _firebaseAuth = FirebaseAuth.instance;
```

22. In the FirebaseAuth class, create a method to create a new user, leveraging the createUserWithEmailAndPassword method. Also include the method in a try block and return null when an error occurs. The code for the method is shown here:

```
Future<String> createUser(String email, String password) async {
    try {
        UserCredential credential = await _firebaseAuth
            .createUserWithEmailAndPassword(email: email, password:
                password);
        return credential.user.uid;
    } on FirebaseAuthException {
        return null;
    }
}
```

23. In the same way, create a method to perform the login action, as shown here:

```
Future<String> login(String email, String password) async {
    try {
        UserCredential credential = await _firebaseAuth
            .signInWithEmailAndPassword(email: email, password:
                password);
        return credential.user.uid;
    } on FirebaseAuthException {
        return null;
    }
}
```

24. Finally, add a method to log out:

```
Future<bool> logout() async {
    try {
        _firebaseAuth.signOut();
        return true;
    } on FirebaseAuthException {
        return false;
    }
}
```

25. Back to the `login_screen.dart` file, in the `onPressed` method inside the `btnMain` method, let's call the methods to create and log in a user, and set the message for the user:

```
 onPressed: () {
    String userId = '';
    if (_isLogin) {
        auth.login(txtUserName.text, txtPassword.text).then((value) {
            if (value == null) {
                setState(() {
                    _message = 'Login Error';
                });
            } else {
                userId = value;
                setState(() {
                    _message = 'User $userId successfully logged in';
                });
            }
        });
    } else {
        auth.createUser(txtUserName.text,
            txtPassword.text).then((value) {
            if (value == null) {
                setState(() {
                    _message = 'Registration Error';
                });
            } else {
                userId = value;
                setState(() {
                    _message = 'User $userId successfully signed in';
                });
            }
        });
    }
},
```

26. In `AppBar` in the `build` method, set the `actions` parameter so that it contains an `IconButton` to perform user logout, as shown in the following code snippet:

```
 actions: [
    IconButton(
        icon: Icon(Icons.logout),
        onPressed: () {
            auth.logout().then((value) {
                if (value) {
                    setState(() {
                        _message = 'User Logged Out';
                    });
                } else {
                    _message = 'Unable to Log Out';
                }
            });
        }
),],
```

27. Run the app and try creating a new user. Then, log in as this newly created user, and finally log out.

How it works...

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 others. In this recipe, you have created a screen that leverages Firebase authentication to provide login, logout, and signup features.

In order to leverage Firebase authentication, you need to enable the method or methods you want to use first, and you do that 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. This provider allows you to sign up and log in with an email and a password.

Once the authentication method was enabled, you designed the user interface in `login_screen.dart`. There, you imported `firebase_auth`, which is the package used for the Firebase authentication service.



Make sure that you always use the latest version of `firebase_auth` in your `pubspec.yaml` file in order to avoid conflicts in your dependencies. Refer to the official page at https://pub.dev/packages/firebase_auth.

You use the login screen for three actions: users can log in, sign up and create a new identity, and log out. As we are using email and password authentication, we designed a screen allowing the input of a username and a password.

The `userInput` method returns a `Padding` widget that creates some space at the top, and contains a `TextField` widget as a child. We also set `InputDecoration` with `hintText` and an `Icon` to make the required input easier to understand for the user.

You followed the same pattern for `passwordInput`, this time leveraging the `obscureText` parameter to hide the content of the field.

Then you added the two buttons, one for the "Primary" action, which is the main action of the screen (log in or sign up), and a "Secondary" button, to change the user action. Based on the value of the `_isLogin` variable, users see different main and secondary buttons: when `_isLogin` is true, users will log in with the primary button, while the secondary button will facilitate getting to the signup action. When `_isLogin` is false, users will sign up with the primary button and will get back to the login action with the secondary button.

The last widget in `Column` is a `Text` widget that contains the message for the user, based on the result of the actions. This may contain a validation error, or an error returned by the Firebase authentication service, or just a success message.

After completing the UI design, you created the `firebase_authentication.dart` file, which contains the methods that directly interact with the Firebase authentication service. In this file, you created the three methods that allow users to log in, log out, and sign up. **All interactions with Firebase are asynchronous.**

In particular, both the `login` and `createUser` methods take two strings, one for the username and one for the password, and return a `Future` of the `String` type, containing the ID of the user.

In the `createUser` function, you called the `createUserWithEmailAndPassword` method. This creates a new user, and if the action is successful, it returns a `UserCredential` object, which contains several properties, including the user identification (`uid`):

```
UserCredential credential = await _firebaseAuth  
    .createUserWithEmailAndPassword(email: email, password: password);  
return credential.user.uid;
```

Similarly, in the `login` method, we called the `signInWithEmailAndPassword` method, which tries to log in the user with the email and password that you include in the method. If this task is successful, it also returns `UserCredential`:

```
UserCredential credential = await _firebaseAuth  
    .signInWithEmailAndPassword(email: email, password: password);  
return credential.user.uid;
```



For a full list of properties for the `UserCredential` class, see <https://firebase.flutter.dev/docs/auth/usage/>.

The last method in the `firebase_authentication.dart` file allows users to log out, and the method to do that is the `signOut` method, which returns a `Future` of the `void` type:

```
await _firebaseAuth.signOut();
```

Now, after calling these methods from the user interface, you might wonder where user data is stored in Firebase. To see the user data you have inserted, 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 the following screenshot:

The screenshot shows the Firebase Authentication screen. At the top, there are navigation links for 'Users', 'Sign-in method', 'Templates', and 'Usage'. Below this is a search bar with placeholder text 'Search by email address, phone number, or user UID'. To the right of the search bar are buttons for 'Add user' and a three-dot menu icon. A table below lists two users with columns for Identifier, Providers, Created, Signed In, and User UID. The first user is 's.alessandria@inwin...', and the second is 'simone@softwarehou...'. Both users have email provider icons and dates indicating when they were created and signed in.

Identifier	Providers	Created	Signed In	User UID
s.alessandria@inwin...	✉	Apr 29, 2...	Apr 29, 2...	EAuC05N1Sm0DxbVD4a6B...
simone@softwarehou...	✉	Jan 4, 20...	Jan 12, 2...	UgZlobabFtKfmHsnh8rTfgnE...

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 the login screen you designed in the previous section.

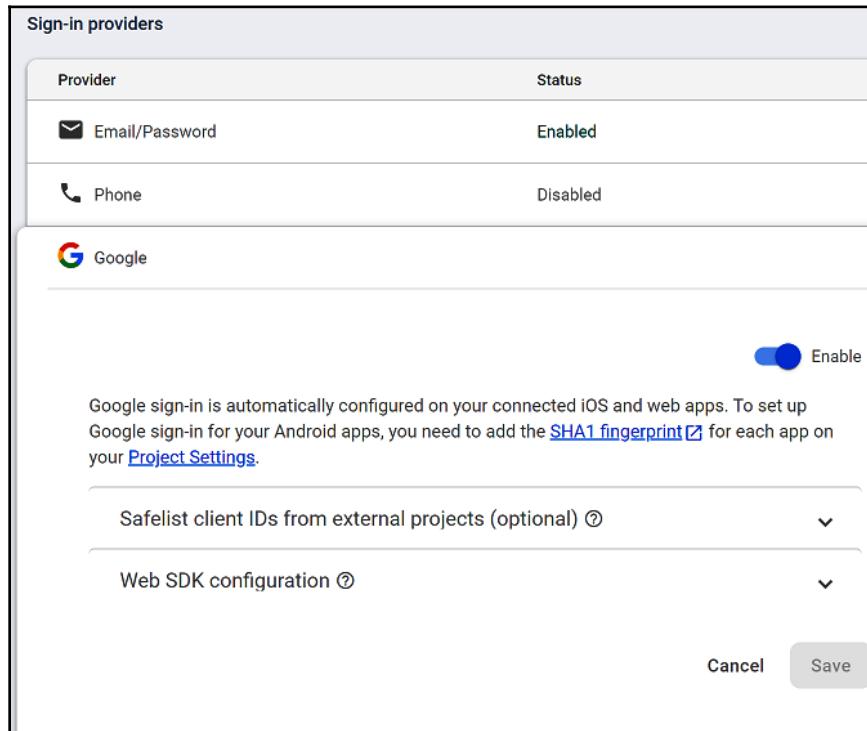
Getting ready

To follow along with this recipe, you need to complete the previous two recipes, *Configuring a Firebase app* and *Creating a login form*.

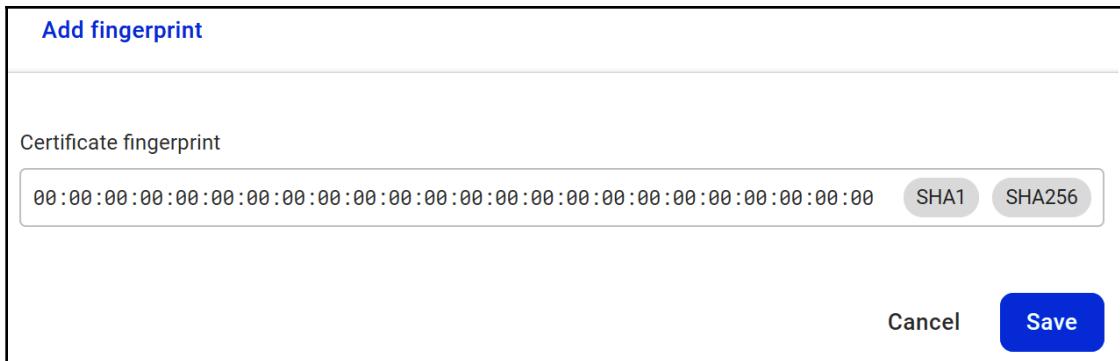
How to do it...

In this recipe, you will extend the previous sign-in form to add social sign-in through the Google Sign-in service, starting from the tasks required to enable Google Sign-in in your Firebase project. Perform the following steps:

1. From the Firebase console, in the **Authentication** page on your project, get to the **Sign In method** page and then enable the Google Sign-in feature, as shown in the following screenshot:



2. Notice that there is 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.
3. Once you have the SHA-1 key, get back to the **Project Overview** page, enter the apps settings page, and then click the **Add fingerprint** button, as shown in the following screenshot:



4. Insert the SHA1 fingerprint and then click the **Save** button.



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>.

You need to repeat the process for each app in your project (both iOS and Android, if you are targeting both systems).

5. Download the `google-services.json` or `GoogleService-Info.plist` file again and paste it into your project, as you did in the *Configuring a Firebase app* recipe in this chapter.
6. Back to the Flutter project, add the `google_sign_in` package in your `pubspec.yaml` file (make sure you check for the latest version at https://pub.dev/packages/google_sign_in/install):

```
google_sign_in: ^5.0.2
```

7. If you are using iOS, you will need to update your `info.plist` file. Follow the instructions in the iOS integration section at https://pub.dev/packages/google_sign_in.
8. In the `_LoginScreenState` class, in `login_screen.dart`, create a method that returns the design of a button for the Google Sign-in process:

```
Widget btnGoogle() {  
    return Padding(  
        padding: EdgeInsets.only(top: 128),  
        child: Container(  
            height: 60,  
            child: ElevatedButton(  
                style: ButtonStyle(  
                    backgroundColor: MaterialStateProperty.all(  
                        Theme.of(context).primaryColorLight),  
                    shape: MaterialStateProperty.all  
                        <RoundedRectangleBorder>(  
                            RoundedRectangleBorder(  
                                borderRadius: BorderRadius.circular(24.0),  
                                side: BorderSide(color: Colors.red)),  
                        ),  
                    ),  
                    onPressed: () {},  
                    child: Text(  
                        'Log in with Google',  
                        style: TextStyle(  
                            fontSize: 18, color:  
                                Theme.of(context).primaryColorDark),  
                        ),  
                    ),  
                )),  
            );  
}
```



If you want to add a Google logo to Google Sign-in, please refer to the *See also* section at the end of this recipe.

9. In `ListView` in the `build` method, add the new widget under the secondary button:

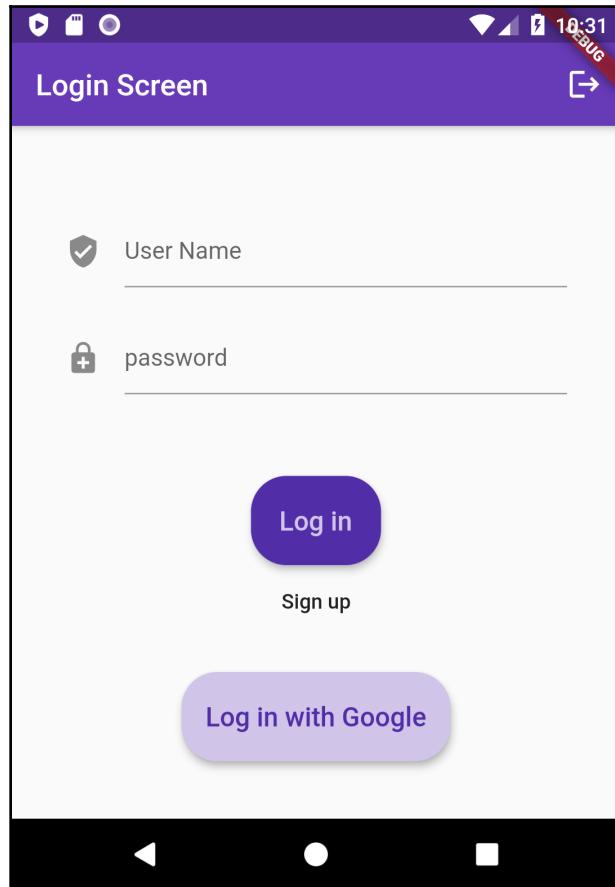
```
child: ListView(  
    children: [  
        userInput(),  
        passwordInput(),
```

```
        btnMain(),  
        btnSecondary(),  
        btnGoogle(),  
        txtMessage(),  
    ],
```

10. In the `userInput` method, reduce the padding to 24:

```
Widget userInput() {  
    return Padding(  
        padding: EdgeInsets.only(top: 24),  
    );
```

11. Run the app. You should see a screen similar to the following screenshot:



12. In the `firebase_authentication.dart` file, in the shared folder in your project, add the Google Sign-in import statement at the top of the file:

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

13. In the `FirebaseAuthentication` class, add the a `GoogleSignIn` instance under the `FirebaseAuth` instance you have already created:

```
class FirebaseAuthentication {  
    final FirebaseAuth _firebaseAuth = FirebaseAuth.instance;  
    final GoogleSignIn googleSignIn = GoogleSignIn();
```

14. Still in the `FirebaseAuthentication` class, create a new asynchronous method, called `loginWithGoogle`, that returns a `Future` of the `String` type:

```
Future<String> loginWithGoogle() async {}
```

15. In the `loginWithGoogle` method, add the following code:

```
final GoogleSignInAccount googleSignInAccount = await  
googleSignIn.signIn();  
final GoogleSignInAuthentication googleSignInAuthentication =  
    await googleSignInAccount.authentication;  
final AuthCredential authCredential =  
    GoogleAuthProvider.credential(  
        accessToken: googleSignInAuthentication.accessToken,  
        idToken: googleSignInAuthentication.idToken,  
    );  
final UserCredential authResult =  
    await _firebaseAuth.signInWithCredential(authCredential);  
final User user = authResult.user;  
if (user != null) {  
    return '$user';  
}  
return null;
```

16. Back to the `_LoginScreenState` class, in the `onPressed` property of `ElevatedButton` in the `btnGoogle` method, add the call to the `loginWithGoogle` method, as shown here:

```
onPressed: () {  
    auth.loginWithGoogle().then((value) {  
        if (value == null) {  
            setState(() {  
                _message = 'Google Login Error';  
            });  
        } else {
```

```
        setState(() {
            _message =
        'User $value successfully logged in with Google';
    });
}
),
},
},
```

17. Run the app and then press the **Login with Google** button. You should be able to use your Google account and see the `value` variable on the screen, containing the user information retrieved from Google.

How it works...

In this recipe, you performed two tasks; the configuration of the app to enable Google Sign-in, and the actual code for using 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).

Once you get your SHA1 fingerprint, you need to add it to the Firebase apps you have configured and download the updated `google-services.json` or `GoogleService-Info.plist` files again.

This completed the configuration tasks for the **Firebase** project.

In the **Flutter** project, the first step you performed was importing the `google_sign_in` package in the `pubspec.yaml` file with the following command:

```
google_sign_in: ^5.0.2
```

As the name implies, this package enables Google Sign-in into your app. In order to log in with Google, you simply need to write the following commands:

```
final GoogleSignIn googleSignIn = GoogleSignIn();  
  
final GoogleSignInAccount googleSignInAccount = await  
googleSignIn.signIn();
```

`GoogleSignIn` is the class that allows you to authenticate Google users and contains the `signIn` method, which actually performs the sign-in process. This returns a `Future` that contains an instance of `GoogleSignInAccount` when the sign-in process was successful. In case the login is **not** successful, this returns `null`.

In case there is already a logged-in user, the method will just return the account of the logged-in user. `GoogleSignIn` includes several useful properties that you can leverage in your apps, including `email`, `id`, and `displayName` of the logged-in user.

Once we got `GoogleSignInAccount`, we also retrieved `GoogleSignInAuthentication`, with the following command:

```
final GoogleSignInAuthentication googleSignInAuthentication = await  
googleSignInAccount.authentication;
```

`GoogleSignInAccount` contains an `accessToken` and an `idToken`. These are required to retrieve an `AuthCredential` object, which you can then use to log in to Firebase:

```
final AuthCredential authCredential = GoogleAuthProvider.credential(  
    accessToken: googleSignInAuthentication.accessToken,  
    idToken: googleSignInAuthentication.idToken,  
) ;
```

An `AuthCredential` object contains an `accessToken`, the `id` of the provider (in this case "google.com"), and the sign-in method that was used. Once you obtain an `AuthCredential` object, you can then log in to Firebase:

```
final UserCredential authResult = await  
_firebaseAuth.signInWithCredential(authCredential);
```

`UserCredential` is part of the `firebase_auth` package and contains the `user` property, which includes all user information, such as `email`, `displayName`, `uid`, and `photoUrl`.

Once you get `UserCredential`, you are successfully logged in to Firebase using Google authentication.

See also

It would be a nice touch to add a Google logo to Google Sign-in. There are rules you need to follow in order to legally incorporate third-party images into your app, but the Google policy is rather accommodating in this case. Have a look at <https://developers.google.com/identity/branding-guidelines> for more information and examples on using Google logos to authenticate your users

Integrating Firebase Analytics

One of the most important features of Firebase is for you to get real feedback about how your app is being used, and the best way to do that is by using Firebase Analytics. In this recipe, we will add screen and event tracking and show how that gets reported to the Firebase dashboard.

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 to Firebase Analytics.

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 it works...

You will now add `firebase_analytics` to your app and activate a custom event:

1. In the dependencies section of your project's `pubspec.yaml` file, add the `firebase_analytics` package:

```
firebase_analytics: ^8.0.1
```

2. In the `lib` folder of 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 {
    @override
    _HappyScreenState createState() => _HappyScreenState();
}

class _HappyScreenState extends State<HappyScreen> {
    @override
    Widget build(BuildContext context) {
        return Container();
    }
}
```

5. In the `build` method of the `_HappyScreenState` class, return `Scaffold`, with an `appBar` and a `body`. `body` contains a `Center` widget, and its child is `ElevatedButton`. The code is shown here:

```
return Scaffold(
    appBar: AppBar(title: Text('Happy Happy!')),
    body: Center(
        child: ElevatedButton(
            child: Text('I\'m happy!'),
            onPressed: () {},
        ),
    );
)
```

6. In the `onPressed` parameter of `ElevatedButton`, add the `FirebaseAnalytics` `logEvent` method, passing 'Happy' as the name of the event:

```
onPressed: () {
    FirebaseAnalytics().logEvent(name: 'Happy', parameters: null);
},
```

7. Open the `login_screen.dart` file and, at the top of the file, import `happy_screen.dart`:

```
import './happy_screen.dart';
```

8. At the bottom of the `_LoginScreenState` class, create a new method and call it `changeScreen`. Inside the method, call the `Navigator.push` method to reach the `HappyScreen` widget:

```
void changeScreen() {  
    Navigator.push(  
        context, MaterialPageRoute(builder: (context) =>  
            HappyScreen()));  
}
```

9. In the `btnMain` method, following a successful login, add a call to the `changeScreen` method:

```
userId = value;  
setState(() {  
    _message = 'User $userId successfully logged in';  
});  
changeScreen();
```

10. In the `btnGoogle` method, following a successful login, add a call to the `changeScreen` method:

```
setState(() {  
    _message = 'User $value successfully logged in with Google';  
});  
changeScreen();
```

11. Open a terminal window and run the following command:

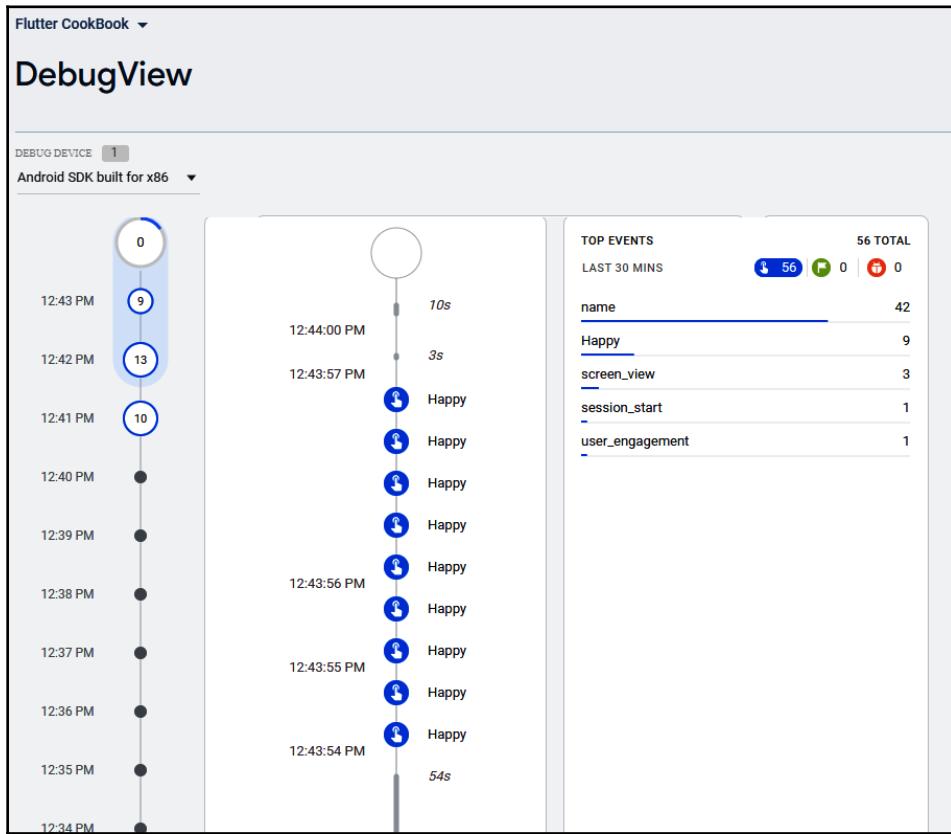
```
adb shell setprop debug.firebaseio.analytics.app [your app name here]
```



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`).

12. Get to the Firebase console, click the **Analytics** menu, and then select **DebugView**. Leave this page open.
13. Run the app, log in with any method, and then tap a few times on the **I'm happy** button.

13. Wait a few seconds, and then get back to the Firebase Analytics debug view. You should see that the event has been logged, as shown in the following screenshot:



How it works...

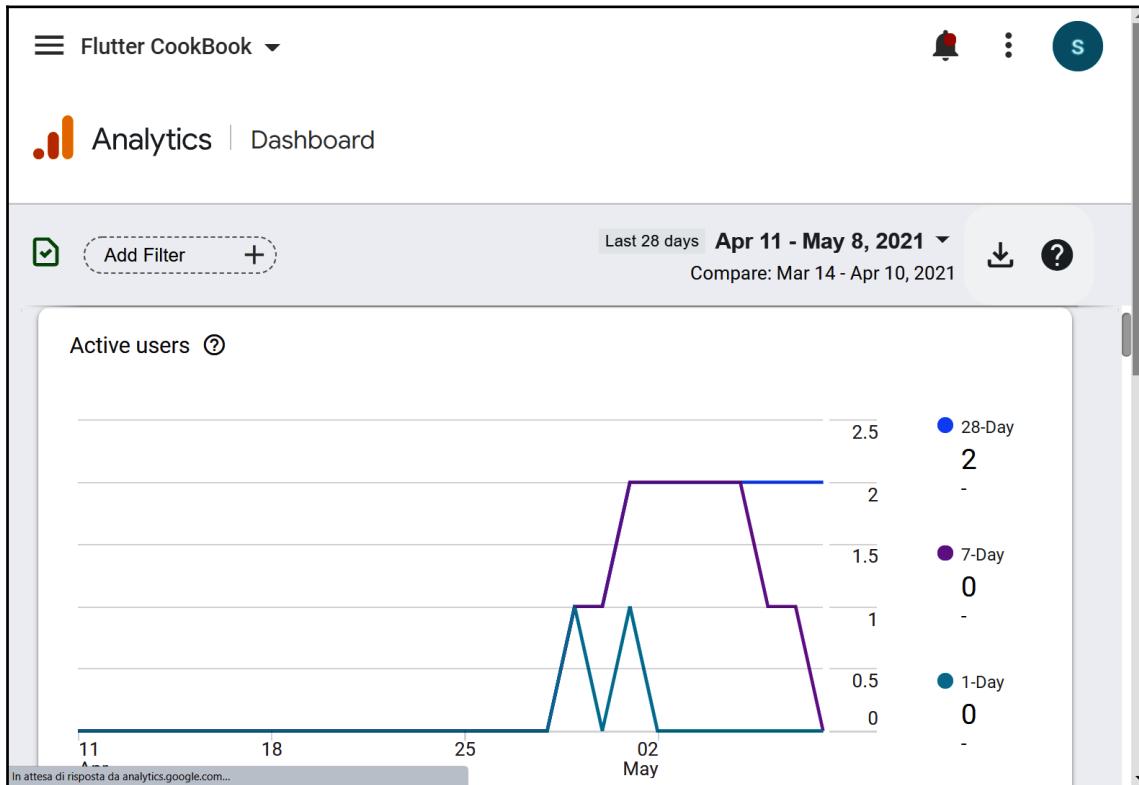
Google Analytics for Firebase can give you invaluable information on how your users behave in your app. The great news is that when using Firebase, if you enable Analytics, **several statistics are generated automatically** for you. These are called **Default Events** and include logging errors, sessions (when your app starts), notifications, and several others.

If you get 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.



It may require **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:



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 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 as follows:

```
FirebaseAnalytics().logEvent(name: 'Happy', parameters:null);
```

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 by opening the terminal window and running the following command:

```
adb shell setprop debug.firebaseio.analytics.app [your app name here]
```

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. 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.

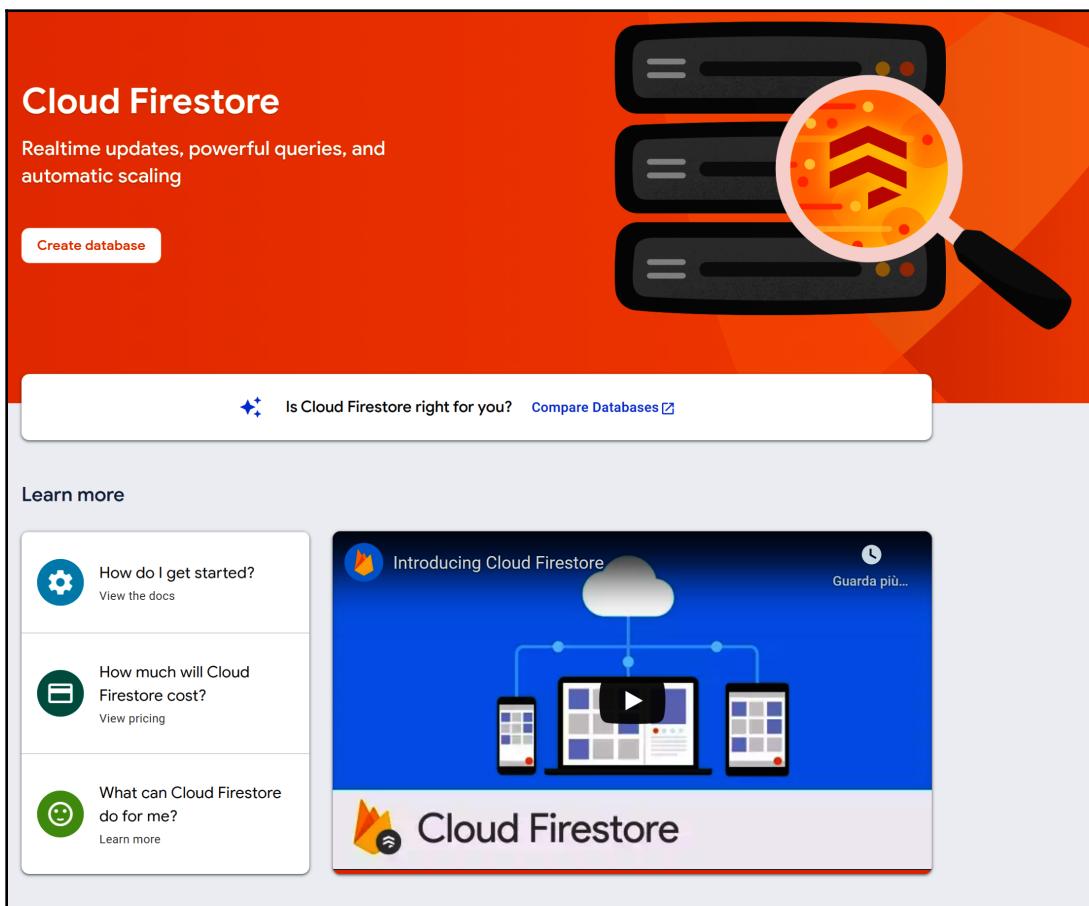
Getting ready

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, 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 the following screenshot:



2. Click on the **Create Database** button, choose the **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 here:

The screenshot shows the Cloud Firestore console interface. At the top, there's a navigation bar with 'Flutter CookBook' and a dropdown arrow. Below it is a large header 'Cloud Firestore'. Underneath the header is a navigation bar with tabs: 'Data' (which is selected and underlined), 'Rules', 'Indexes', and 'Usage'. To the right of the tabs, there's a note: 'Prototype and test end-to-end with the Local Emulator Suite,' followed by a blue starburst icon. The main area contains a table with three rows. The first row has a house icon and a blank column. The second row has a file icon and the text 'flutter-cookbook-43242'. The third row has a plus sign icon and the text 'Start collection'. The entire screenshot is enclosed in a light gray border.

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 the following screenshot:

The screenshot shows the 'Start a collection' interface in the Cloud Firestore console. At the top, there are two steps: 'Give the collection an ID' (marked with a checkmark) and 'Add its first document' (step 2). Below the steps, the 'Document parent path' is set to '/poll'. Under 'Document ID', the ID 'jRp7PQcFhOVUMICIJg03' is entered. A tooltip appears stating: 'A collection must contain at least one document, Cloud Firestore's unit of storage. Use an auto-generated ID or enter a custom ID if needed. Documents store your data as fields.' Below the ID, there are two data entries in a table:

Field	Type	Value
icecream	= number	0
pizza	= number	0

At the bottom right, there are 'Cancel' and 'Save' buttons.

9. Click the **Save** button.
10. In the dependencies node in your `pubspec.yaml` file, add the latest version of the `cloud_firestore` package:

```
cloud_firestore: ^1.0.7
```

11. Create a new file in the `lib` folder of 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 {
  @override
  _PollScreenState createState() => _PollScreenState();
}
class _PollScreenState extends State<PollScreen> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

14. In the build method of the `_PollScreenState` class, return `Scaffold`. In the body of `Scaffold`, place a `Column`, with a `mainAxisAlignment` value of `spaceAround`, as shown in the following code block:

```
return Scaffold(
  appBar: AppBar(
    title: 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: [Icon(Icons.icecream), Text('Ice-cream')]),
  onPressed: () {},
),
ElevatedButton(
  child: Row(
    mainAxisAlignment: MainAxisAlignment.spaceAround,
    children: [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 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. Run the app, and after logging in, try pressing the two buttons a few times each.
26. 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 the following screenshot:

The screenshot shows the Cloud Firestore interface. At the top, there are tabs for Data, Rules, Indexes, and Usage. Below the tabs, a banner says "Prototype and test end-to-end with the Local Emulator Suite, now with Firebase Authentication" and has a "Get started" button. The main area shows a navigation path: Home > poll > jRp7PQcFhOVU... . On the left, there's a sidebar with a "Start collection" button and a "poll" collection listed under "flutter-cookbook-43242". The "poll" collection has a "Start document" button. A specific document, "jRp7PQcFhOVUMICIJg03", is selected and expanded. This document contains fields "icecream" with the value 7 and "pizza" with the value 10.

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, which 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 document ID: document ID 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



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:

```
Firestore db = FirebaseFirestore.instance;
```

Then, to get to the 'poll' collection, which is the only collection in the database, you used the `collection` method over 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-vsfirestore>.

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:

1. Add the latest version of the `firebase_messaging` plugin to the `pubspec.yaml` file of your app.

2. At the top of the login_screen.dart file in your app, import the firebase_messaging.dart file:

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

3. At the top of the _LoginScreenState class, add the following declaration:

```
final FirebaseMessaging messaging = FirebaseMessaging.instance;
```

4. In the initState method, edit the code in the whenComplete callback of the Firebase initializeApp method as follows (you will get an error in _firebaseBackgroundMessageReceived that we will fix in the next step):

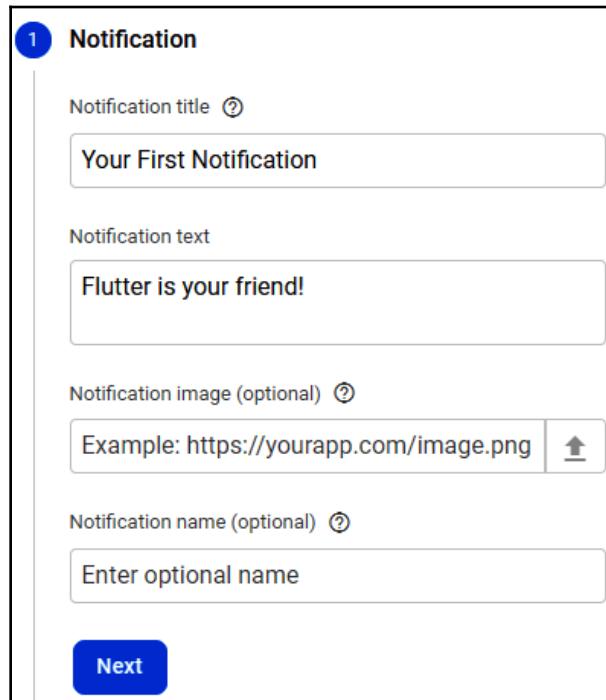
```
Firebase.initializeApp().whenComplete(() {  
    auth = FirebaseAuth();  
    FirebaseMessaging.onBackgroundMessage(  
        _firebaseBackgroundMessageReceived);  
    setState(() {});  
});
```

5. Outside of the _LoginScreenState class, add a method, called _firebaseBackgroundMessageReceived, that prints the notification information:

```
Future _firebaseBackgroundMessageReceived(RemoteMessage message)  
async {  
    print(  
        "Notification: ${message.notification.title} -  
        ${message.notification.body}");  
}
```

6. Run the app and then close it.
7. Get to the Firebase console and click on the **Engage** section. Then, click on the **Cloud Messaging** link.
8. Click on the **Send your first message** button at the top of the page.

9. In the **Compose Notification** page, insert a title and text, as shown in the following screenshot, and then click the **Next** button:



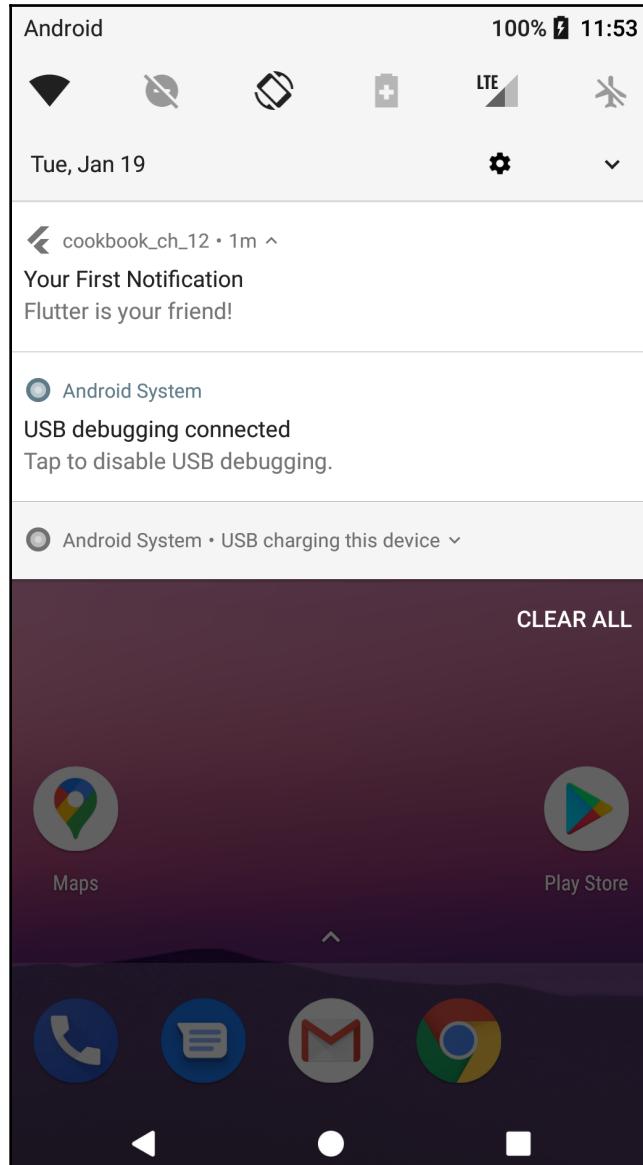
10. On the target page, select your app and then click **Next**.



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.

11. On the **Scheduling** page, leave **Now** and then click **Next**.
12. On the **Conversion Events** page, click **Review**.
13. Make sure your app is NOT in the foreground and then, from the Review message, click **Publish**.
14. In your device or emulator, you should see a small flutter icon at the top of the screen.

15. Open the notifications tab, and there you should see the message that you just added from the **Firebase Cloud Messaging** page, as shown in the following screenshot:



How it works...

Firebase Cloud Messaging (FCM) is a service that allows you to send notifications to your users. In Flutter, where you use the FlutterFire tools, 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 in the `Firebase.initializeApp().whenComplete` callback.

The `onBackgroundMessage` callback is triggered when a new notification message is received and takes the method that deals with the notification:

```
FirebaseMessaging.onBackgroundMessage(_firebaseBackgroundMessageReceived);
```

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 have seen 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) are 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/products/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 pubspec.yaml file, add the latest versions of the firebase_storage and image_picker packages:

```
image_picker: ^0.7.4  
firebase_storage: ^8.0.5
```

2. In the lib folder of your project, create a new file, and call it upload_file.dart.
3. At the top of the upload_file.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';
```

4. Create a new stateful widget, and call it UploadFileScreen:

```
class UploadFileScreen extends StatefulWidget {  
    @override  
    _UploadFileScreenState createState() => _UploadFileScreenState();  
}  
class _UploadFileScreenState extends State<UploadFileScreen> {  
    @override
```

```
Widget build(BuildContext context) {  
    return Container();  
}
```

5. 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();
```

6. In the `build` method of the `_UploadFileScreenState` class, return `Scaffold`, with an `appBar` and a `body`, as shown in the following code block:

```
return Scaffold(  
    appBar: AppBar(title: Text('Upload To FireStore')),  
    body: Container(  
        padding: EdgeInsets.all(24),  
        child: Column(  
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
            crossAxisAlignment: CrossAxisAlignment.stretch,  
            children: [],  
        ),);
```

7. In the `children` parameter of the `Column` widget, add four widgets – `ElevatedButton`, `Image`, another `ElevatedButton`, and `Text`, as shown in the following code block:

```
ElevatedButton(  
    child: Text('Choose Image'),  
    onPressed: () {},  
) ,  
(_image == null) ? Container(height: 200) : Container(height: 200,  
    child: Image.file(_image)),  
ElevatedButton(  
    child: Text('Upload Image'),  
    onPressed: () {},  
) ,  
Text(_message),
```

8. 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.getImage(source:  
        ImageSource.gallery);
```

```
        setState(() {
            if (pickedFile != null) {
                _image = File(pickedFile.path);
            } else {
                print('No image selected.');
            });
        });
    }
}
```

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` (**Choose image**), add a call to the `getImage` method:

```
getImage();
```

15. In the `onPressed` parameter of the second `ElevatedButton` (**Upload image**), add a call to the `uploadImage` method:

```
uploadImage();
```

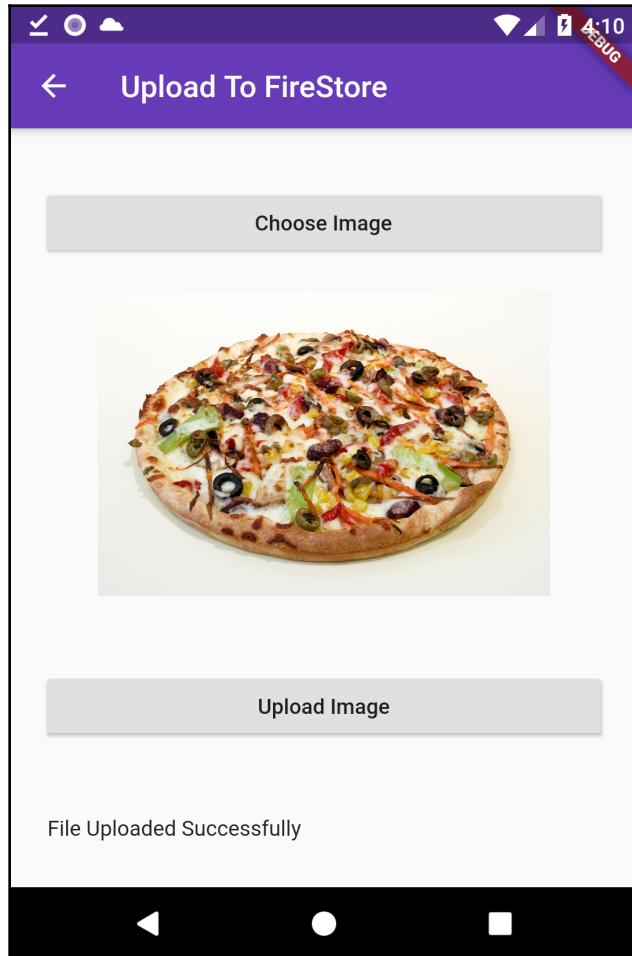
16. At the top of the `login_screen.dart` file, import the `upload_file.dart` file:

```
import './upload_file.dart';
```

17. In the `changeScreen` method, edit `Navigator.push` so that it calls the `UploadFileScreen` widget:

```
Navigator.push(  
  context, MaterialPageRoute(builder: (context) =>  
    UploadFileScreen()));
```

18. Run the app, and after logging in, pick an image and upload it to Firebase. The end result should look similar to the following screenshot:



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 to add the `firebase_storage` dependency to your `pubspec.yaml` file and import it into your files. In 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:

```
final 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:

```
final 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 completed successfully. This allowed 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.getImage(source: ImageSource.gallery);
```

First, you retrieve an instance of `ImagePicker`, and then you call the `getImage` method, selecting the source where you want to retrieve the images.

13

Machine Learning with Firebase

ML Kit

Machine Learning (ML) has become a critical topic in any application's development. In a nutshell, ML means that you import data that "trains" an algorithm, and use it to generate 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 ML Kit, 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 ML Kit, you can create your own custom model with TensorFlow Lite. Most of the services outlined in this chapter can run both on the cloud or on your device.

In this chapter, you will start by taking a picture with your device, then you will use ML Kit 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 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 create full-stack apps without any server-side code.

Using the device 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 exclusive to ML but is one of the prerequisites for ML visual functions that you will use in the following recipes in this chapter.

By the end of this recipe, you will be able to use the device 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 12, *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. The 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:

```
camera: ^0.8.1  
path_provider: ^2.0.1
```

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 {
    @override
    _CameraScreenState createState() => _CameraScreenState();
}

class _CameraScreenState extends State<CameraScreen> {
    @override
    Widget build(BuildContext context) {
        return Container(
    );
}
```

7. At the top of the `_CameraScreenState` class, declare the following variables:

```
List<CameraDescription> cameras;
List<Widget> cameraButtons;
CameraDescription activeCamera;
CameraController cameraController;
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 `ElevatedButton` widgets with the name of the camera, as shown:

```
List<Widget> buttons = [];
cameras = await availableCameras();
if (cameras == null) return null;
if (activeCamera == null) activeCamera = cameras.first;
if (cameras.length > 0) {
```

```
        for (CameraDescription camera in cameras) {
            buttons.add(ElevatedButton(
                onPressed: () {
                    setState(() {
                        activeCamera = camera;
                        setCameraController();
                    });
                },
                child: Row(
                    children: [
                        Icon(Icons.camera_alt),
                        Text(camera == null ? '' : camera.name)
                    ],
                )));
        }
        return buttons;
    } else {
        return [];
    }
}
```

10. In the `_CameraScreenState` class, create a new asynchronous method, called `setCameraController`, that based on the value of `activeCamera` will set the `CameraPreview` `preview` variable, as shown:

```
Future setCameraController() async {
    if (activeCamera == null) return;
    cameraController = CameraController(activeCamera,
        ResolutionPreset.high,);
    await cameraController.initialize();
    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.value.isInitialized) {
        return null;
    }
    if (cameraController.value.isTakingPicture) {
        return null;
    }
}
```

```
        }
    try {
        await cameraController.setFlashMode(FlashMode.off);
        XFile picture = await cameraController.takePicture();
        Navigator.push(context,
            MaterialPageRoute(builder: (context) =>
                PictureScreen(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() {
    listCameras().then((result) {
        setState(() {
            cameraButtons = result;
            setCameraController();
        });
    });
    super.initState();
}
```

13. Override the `dispose` method and inside it, dispose of the `cameraController` widget:

```
@override
void dispose() {
    if (cameraController != null) {
        cameraController.dispose();
    }
    super.dispose();
}
```

14. In the `build` method, return a `Scaffold` with an `AppBar` whose title is `Camera View`, and a body with a `Container` widget:

```
return Scaffold(
    appBar: AppBar(
        title: Text('Camera View'),
    ),
    body: Container());
```

15. In Container, insert a Padding with an EdgeInsets.all value of 24 and a child of Column, as shown:

```
Container(  
    padding: 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 ?? [Container(child: Text('No cameras  
available'))],  
)  
Container(height: size.height / 2, child: preview) ?? Container(),  
Row(  
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
    children: [  
        ElevatedButton(  
            child: Text('Take Picture'),  
            onPressed: () {  
                if (cameraController != null) {  
                    takePicture().then((dynamic picture) {  
                        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 stateful widget called `PictureScreen`:

```
class PictureScreen extends StatefulWidget {
  @override
  _PictureScreenState createState() => _PictureScreenState();
}

class _PictureScreenState extends State<PictureScreen> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

20. At the top of the `PictureScreen` class, add a final `XFile` called `picture` and create a constructor method that sets its value:

```
final XFile picture;
PictureScreen(this.picture);
```

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:

```
double deviceHeight = MediaQuery.of(context).size.height;
return Scaffold(
  appBar: AppBar(
    title: Text('Picture'),
  ),
  body: Column(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    children: [
      Text(widget.picture.path),
      Container(height: deviceHeight / 1.5, child:
        Image.file(File(widget.picture.path))),
      Row(
        children: [
          ElevatedButton(
            child: Text('Text Recognition'),
            onPressed: () {}),
        ],
      ),
    ],
  );
)
```

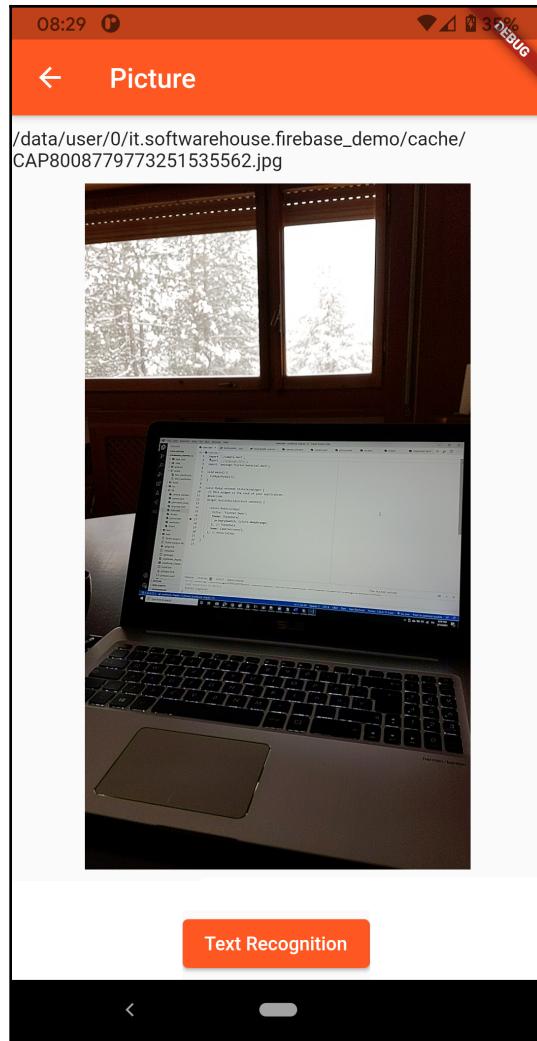
22. Get back to the `camera.dart` file. In the `onPressed` function, in the **Take Picture** button, navigate to the `PictureScreen` widget, passing the picture that was taken, as shown. If the `picture.dart` file is not automatically imported by your IDE, also import the `picture.dart` file:

```
ElevatedButton(  
    child: Text('Take Picture'),  
    onPressed: () {  
        if (cameraController != null) {  
            takePicture().then((dynamic picture) {  
                Navigator.push(  
                    context,  
                    MaterialPageRoute(  
                        builder: (context) => PictureScreen(picture)));  
            });  
        } }, )
```

23. 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: CameraScreen(),  
        ); } }
```

24. 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:



How it works...

Being able to use the camera and adding 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 to use ML algorithms).

The `CameraController` asynchronous `takePicture` method actually takes a picture; this will save the picture in a default path that we later show in 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 the `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 obtaining 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 `firebase_ml_vision` package in your `pubspec.yaml` file:

```
firebase_ml_vision: ^0.10.0
```

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 `firebase_ml_vision` packages:

```
import 'dart:io';
import 'package:firebase_ml_vision/firebase_ml_vision.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 FirebaseVision vision = FirebaseVision.instance;
final FirebaseVisionImage visionImage =
    FirebaseVisionImage.fromFile(image);
TextRecognizer recognizer = vision.textRecognizer();
final results = await recognizer.processImage(visionImage);
return results.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 {
  @override
  _ResultScreenState createState() => _ResultScreenState();
}

class _ResultScreenState extends State<ResultScreen> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

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;  
ResultScreen(this.result);
```

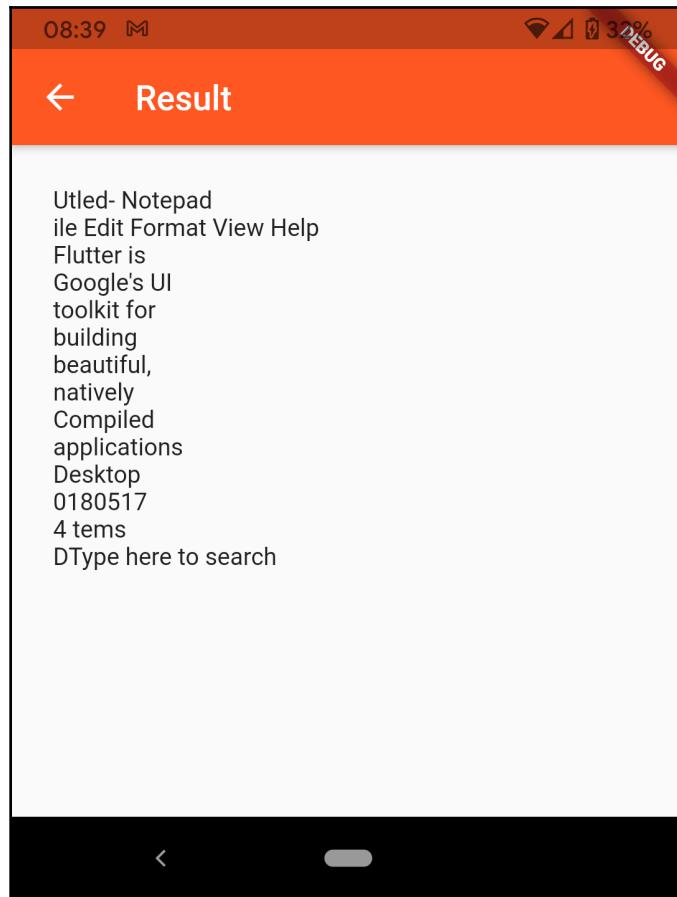
11. In the `build` method of the `_ResultScreenState` class, return a `Scaffold`, and in its body, add a `SelectableText`, as shown:

```
return Scaffold(  
    appBar: AppBar(  
        title: Text('Result'),  
    ),  
    body: Container(  
        child: Padding(  
            padding: EdgeInsets.all(24),  
            child: SelectableText(widget.result,  
                showCursor: true,  
                cursorColor: Theme.of(context).accentColor,  
                cursorWidth: 5,  
                toolbarOptions: ToolbarOptions(copy: true, selectAll:  
                    true),  
                scrollPhysics: ClampingScrollPhysics(),  
                onTap: (){},  
            )),  
    ),  
) ;
```

12. In the `picture.dart` file, in the `onPressed` function, in the `Text Recognition` button, add the following code:

```
onPressed: () {  
    MLHelper helper = MLHelper();  
    helper.textFromImage(image).then((result) {  
        Navigator.push(  
            context,  
            MaterialPageRoute(  
                builder: (context) => ResultScreen(result))),  
    });  
},
```

13. 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:



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 use it as necessary.

The first step is getting an instance of the Firebase ML Vision API. In this recipe, we got it with the following instruction:

```
final FirebaseVision vision = FirebaseVision.instance;
```

The next step is creating a `FirebaseVisionImage`, which is the image object used for the API detector. In our example, you created it with the following instruction:

```
final FirebaseVisionImage visionImage =
    FirebaseVisionImage.fromFile(image);
```

Once the `FirebaseVision` instance is available and `FirebaseImageVision` is available, you call a detector; in this case, you called a `TextRecognizer` detector with the following instruction:

```
TextRecognizer recognizer = vision.textRecognizer();
```

To get the text from the image, you need to call the `processImage` method on `TextRecognizer`. This asynchronous method returns a `VisionText` 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 results = await recognizer.processImage(visionImage);
return results.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; you can actually choose which options should be shown to the user with the `ToolBarOptions` enum.

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 a `DocumentTextBlock`, which is a single element of text recognized by 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 to your projects. It is a fast and convenient way to take some user input and return relevant information.

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 two previous ones: *Using the device camera* and *Recognizing text from an image*.

How to do it...

By following these steps, you will now add the barcode reading feature to the existing app:

1. 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 {}
```

2. At the top of the `readBarcode` method, declare a `String`, a `FirebaseVision`, a `FirebaseVisionImage`, and a `BarcodeDetector`, as shown:

```
String result = '';
final FirebaseVision vision = FirebaseVision.instance;
final FirebaseVisionImage visionImage =
    FirebaseVisionImage.fromFile(image);
BarcodeDetector detector = vision.barcodeDetector();
```

3. After the declarations, use a `try/catch`. In the `try` block, call the `BarcodeDetector detectInImage` method to retrieve a list of `Barcode` objects. Then, for each barcode in the list, add its display value to the result string. In the `catch` block, just print the error that was returned:

```
try {
    List<Barcode> results = await
        detector.detectInImage(visionImage);
    results.forEach((Barcode barcode) {
        result += barcode.displayValue + '\n';
    });
} catch (error) {
    print(error.toString());
}
```

4. At the bottom of the `readBarCode` method, return the `result` string:

```
return result;
```

5. 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: Text('Barcode Reader'),
    onPressed: () {
        MLHelper helper = MLHelper();
        helper.readBarcode(image).then((result) {
            Navigator.push(
                context,
                MaterialPageRoute(
                    builder: (context) => ResultScreen(result)));
        });
    },
)
```

6. 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.

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.



The detection of a barcode happens on the device, so it doesn't require a network connection.

A `Barcode` object contains several properties, including URLs, emails, geopoints, and calendar events, but most barcodes have a `displayValue` (the number you find under the barcode on a book, for instance).

The steps to scan a barcode are similar to those required when recognizing text from an image: you need to get an instance of `FirebaseVision`, then retrieve a `FirebaseVisionImage`, and then obtain a `BarcodeDetector` from the `FirebaseVision` instance. You did this with the following instructions:

```
final FirebaseVision vision = FirebaseVision.instance;
final FirebaseVisionImage visionImage =
    FirebaseVisionImage.fromFile(image);
BarcodeDetector detector = vision.barcodeDetector();
```

Next, you need to call the `detectInImage` method on `BarcodeDetector`, passing the `VisionImage` you want to analyze: this returns a list of strings, one for each barcode that was recognized. In this recipe, you read the barcodes with the following instruction:

```
List<Barcode> results = await detector.detectInImage(visionImage);
```

From the results, you wrote a string containing each `displayValue` of the barcodes that were recognized:

```
results.forEach((Barcode barcode) {
    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...

By following these steps, you will add an image labeling feature to the existing app:

1. In the `ml.dart` file, 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 {}
```

2. At the top of the `labelImage` method, declare four variables:

`String`, `FirebaseVision`, `FirebaseVisionImage`, and `ImageLabeler`, as shown:

```
String result = '';
final FirebaseVision vision = FirebaseVision.instance;
final FirebaseVisionImage visionImage =
    FirebaseVisionImage.fromFile(image);
ImageLabeler labeler = vision.imageLabeler();
```

3. After the declarations, insert a try/catch: in the try block, call the `ImageLabeler processImage` method to retrieve a list of `ImageLabel` objects. For each label in the list, add its `text` and `confidenceLevel` into the result string. In the catch block, just print the error that was returned:

```
try {  
    List<ImageLabel> labels = await  
        labeler.processImage(visionImage);  
    labels.forEach((label) {  
        result += label.text + ' - Confidence '  
        + (label.confidence * 100).toString() + '%\n';  
    });  
} catch (error) {  
    print(error.toString());  
}
```

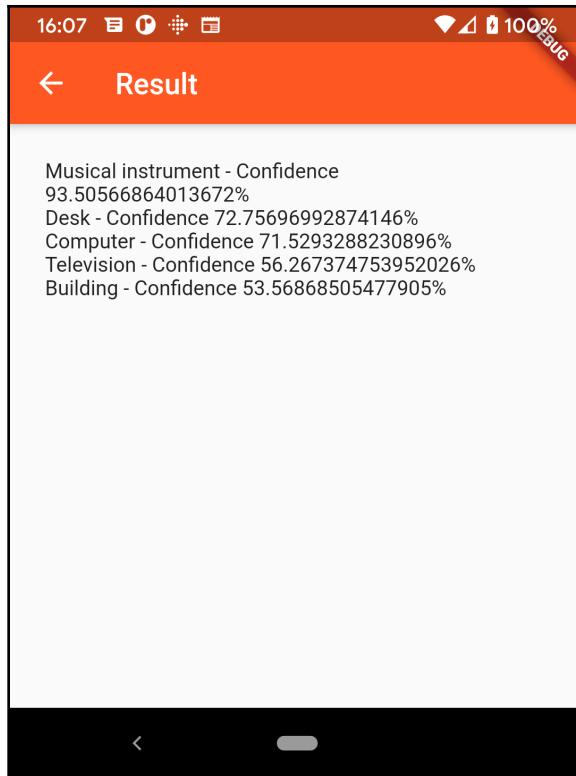
4. At the bottom of the `labelImage` method, return the `result` string:

```
return result;
```

5. 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: Text('Image Labeler'),  
    onPressed: () {  
        MLHelper helper = MLHelper();  
        helper.labelImage(image).then((result) {  
            Navigator.push(  
                context,  
                MaterialPageRoute(  
                    builder: (context) => ResultScreen(result)));  
        });  
    },  
,
```

6. Run the app and take a picture of your environment. You should see a list of objects as shown in the following screenshot:



How it works...

The Image Labeling service allows recognizing different objects in an image. These include people, places, animals, and plants. There are two kinds of `ImageLabeler`: one is on the device, and we used it in this recipe. Or, if you want to connect to the Firebase cloud services, you can use `CloudImageLabeler` instead.

When using an on-device image labeler you can recognize over 400 labels, but with the cloud services, there are over 10,000 available labels.

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.

In this case, the object you used was `ImageLabeler`:

```
ImageLabeler labeler = vision.imageLabeler();
```

The method to call to get the labels of the objects in a picture is `processImage`, which returns a list of `ImageLabel` objects:

```
List<ImageLabel> labels = await labeler.processImage(visionImage);
```

`ImageLabel` contains a `text` property, which is the name of the object that was retrieved, and a `confidence` double value. This is a number between 0 and 1, where 1 is the highest confidence level and 0 is the lowest.

See also

While you can use the Image Labeling API to describe the full image, for classifying one or more objects in an image, you can use ML Kit's on-device Object Detection and Tracking API. With it, you can detect and track single objects in your images. For more information, have a look at <https://developers.google.com/ml-kit/vision/object-detection>.

Building a face detector and detecting facial gestures

This recipe will explore ML Kit's face detector. This also includes a model that predicts the probability of a face smiling. 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 *Recognising 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. In the `ml.dart` file, add a new `async` method, called `faceRecognition`. The method takes `File` as a parameter, and returns a future of type `String`:

```
Future<String> faceRecognition(File image) async {}
```

2. At the top of the `faceRecognition` method, declare three variables: `String`, `FirebaseVision`, and `FirebaseVisionImage`:

```
String result = '';
final FirebaseVision vision = FirebaseVision.instance;
final FirebaseVisionImage visionImage =
    FirebaseVisionImage.fromFile(image);
```

3. Also declare a `FaceDetector` enabling `Classification`, `Landmark`, and `Tracking`:

```
FaceDetector detector = vision.faceDetector(const
    FaceDetectorOptions(
        enableClassification: true,
        enableLandmarks: true,
        enableTracking: true,
        mode: FaceDetectorMode.accurate));
```

4. After the declarations, insert a `try/catch` block.
5. Call the `detector processImage` method to retrieve a list of `Face` objects.
6. Retrieve the number of faces that were found, then for each item in the list, print the probability of the face smiling and having the left and right eye open.
7. In the `catch` block, just print the error that was returned:

```
try {
    List<Face> results = await
        detector.processImage(visionImage);
    int count = results.length;
    result = 'There are $count face(s) in your picture \n';
```

```
        int num = 1;
        results.forEach((Face face) {
            result += 'Face #$num: \n';
            result += 'Smiling: ${face.smilingProbability * 100}% \n';
            result += 'Left Eye Open: ${face.leftEyeOpenProbability *
                100}% \n';
            result += 'Right Eye Open: ${face.rightEyeOpenProbability *
                100}% \n';
        });
    } catch (error) {
        print(error.toString());
    }
}
```

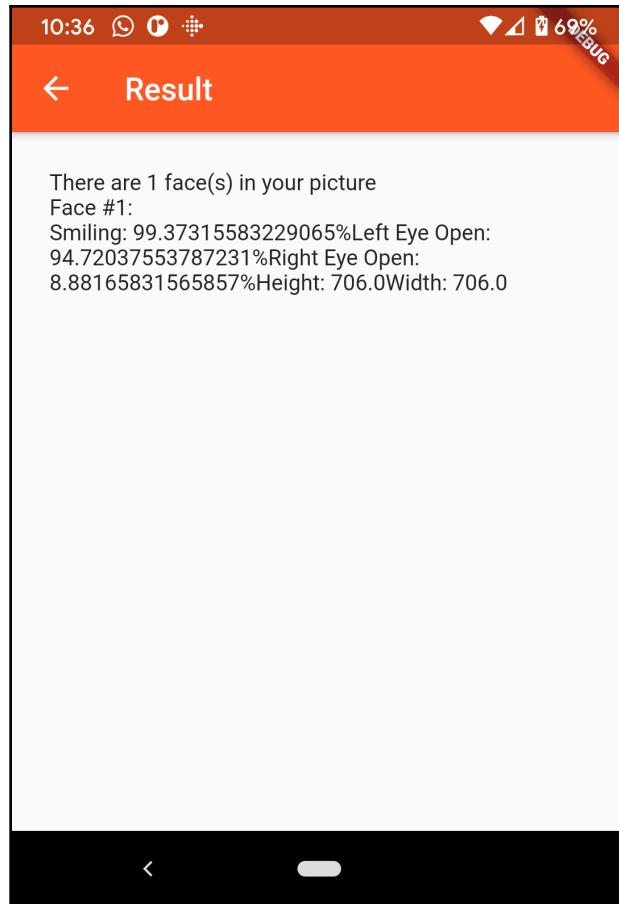
8. At the bottom of the `faceRecognition` method, return the `result` string:

```
return result;
```

9. 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 `faceRecognition` method and shows the results in the results screen:

```
Row(mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    children: [
        ElevatedButton(
            child: Text('Face Recognition'),
            onPressed: () {
                MLHelper helper = MLHelper();
                helper.faceRecognition(image).then((result) {
                    Navigator.push(
                        context,
                        MaterialPageRoute(
                            builder: (context) => ResultScreen(result)));
                });
            },
        ),
    ],
)
```

10. Run the app, smile, keep your eyes open, and take a selfie. The results should look similar to the following screenshot:



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.



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 the 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`:

```
FaceDetector detector = vision.faceDetector(const FaceDetectorOptions(  
    enableClassification: true,  
    enableLandmarks: true,  
    enableTracking: true,  
    mode: FaceDetectorMode.accurate));
```

Note that the `faceDetector` method allows specifying a few options:

- `enableClassification` specifies whether adding 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 detector.processImage(visionImage);
```

`Face` contains several properties: the ones you used in this recipe are `smilingProbability`, `leftEyeOpenProbability`, and `rightEyeOpenProbability`. They all can contain a value between 0 and 1, where 1 is the highest probability level and 0 the lowest.



`Face` also contains `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 will be 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 *Recognising 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. In your pubspec.yaml file, add the latest version of the firebase_mlkit_language package:

```
firebase_mlkit_language: ^1.1.3
```

2. In the ml.dart file, 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 {}
```

3. At the top of the `identifyLanguage` method, declare four variables: `String`, a `FirebaseLanguage` instance, `LanguageIdentifier`, and a list of `LanguageLabel` objects:

```
String result = '';
final language = FirebaseLanguage.instance;
final identifier = language.languageIdentifier();
List<LanguageLabel> languages;
```

4. After the declarations, insert a `try/catch` block: call the `identifier.processText` method to retrieve a list of `LanguageLabel` objects.
5. Run a `forEach` method over the list, and for each `LanguageLabel` that was identified, add to the `result` string the language code and the confidence level. In the `catch` block, just print the error that was returned:

```
try {
  languages = await identifier.processText(text);
  languages.forEach((LanguageLabel label) {
    result +=
      Language: ${label.languageCode} - Confidence:
      ${label.confidence * 100}%
      \n';
  });
} catch (error) {
  print(error.toString());
}
```

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 {
  @override
  _LanguageScreenState createState() => _LanguageScreenState();
}
```

```
class _LanguageScreenState extends State<LanguageScreen> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

10. At the top of the `_LanguageScreenState` class, declare a `TextEditingController` and call it `txtLanguage`:

```
final TextEditingController txtLanguage = TextEditingController();
```

11. 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: Text('Language Detection'),
  ),
  body: Container(
    padding: EdgeInsets.all(32),
    child: Column(
      mainAxisSize: MainAxisSize.spaceEvenly,
      children: [
        TextField(
          controller: txtLanguage,
          maxLines: 5,
          decoration: InputDecoration(
            labelText: 'Enter some text in any language',
          ),
        ),
        Center(child: ElevatedButton(
          child: Text('Detect Language'),
          onPressed: () {
            MLHelper helper = MLHelper();
            helper.identifyLanguage(txtLanguage.
              text).then((result) {
              Navigator.push(
                context,
                MaterialPageRoute(
                  builder: (context) =>
                    ResultScreen(result)));
            });
          },
        )),
      ],
    ),
  ),
);
```

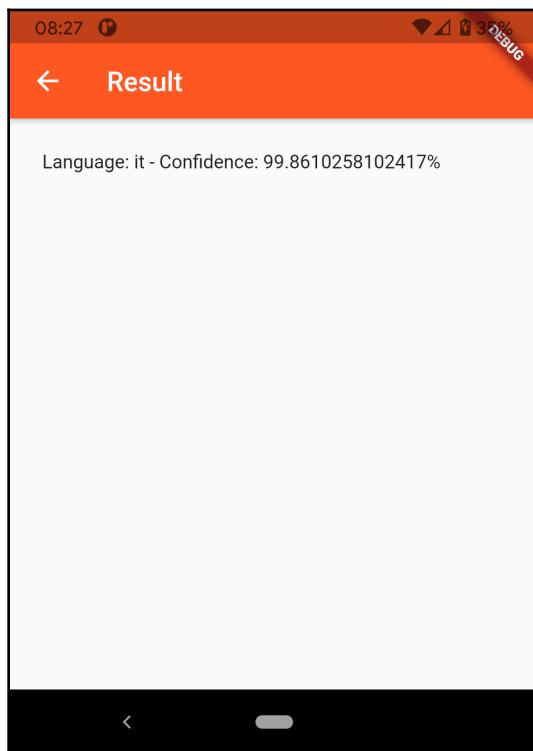
12. In the main.dart file, import language.dart:

```
import './language.dart';
```

13. In the home parameter of MaterialApp, set LanguageScreen:

```
home: LanguageScreen(),
```

14. 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:



How it works...

ML Kit provides several tools that deal with language: **language identification**, text translation, 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.

To get access to the ML Kit language tools in Flutter, you need to add the dependency in the `pubspec.yaml` file:

```
firebase_mlkit_language: ^1.1.3
```

In particular, in this recipe, you have implemented language recognition on a user-provided text.



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.

There are two objects you need to implement language recognition: an instance of the `FirebaseLanguage` class, which is the starting point of all language services, and an instance of `LanguageIdentifier`. In this recipe, you declared them with first the following instruction:

```
final language = FirebaseLanguage.instance;
```

You also used the following instruction:

```
final identifier = language.languageIdentifier();
```

To identify the language, you called the `LanguageIdentifier processText` method; this takes a string, which is the text you want to analyze, and returns a list of `LanguageLabel` 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.

You called the `processText` method with the following instruction:

```
languages = await identifier.processText(text);
```

A `LanguageLabel` object contains two properties that you used in the code in this recipe: `languageCode`, 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 language code and confidence level to the `result` string with the following instruction:

```
result +=     'Language: ${label.languageCode} - Confidence:  
${label.confidence * 100}% \n';
```

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 to create and use ML models, and **TensorFlow Lite** is a lightweight platform specifically designed to be used on mobile and IoT devices.



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*, *Recognising 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_lite` 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. In the `pubspec.yaml` file, download the latest version of the `tflite_lite` package:

```
tflite_flutter: ^0.5.0
```

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 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;
    }
```

8. 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;
    TFHelper helper = TFHelper();
    int value = await helper.classify(message);
    if (value > 0) {
        result = 'Positive sentiment';
    } else {
        result = 'Negative sentiment';
    }
    return result;
}
```

9. 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: Text('Classify Text'),
    onPressed: () {
        MLHelper helper = MLHelper();
        helper.classifyText(txtLanguage.text).then((result) {
            Navigator.push(
                context,
                MaterialPageRoute(
                    builder: (context) => ResultScreen(result)));
        });
    },
),
])
```

10. Run the app and write `I like pizza`. This should return a positive sentiment.
11. 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).



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 contains 10,000 words that are used by the model to retrieve the positive and negative sentiments.



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>.

It should be noted that there are also other great ML frameworks, including scikit-learn (https://scikit-learn.org/stable/getting_started.html) and Caffe (<https://caffe.berkeleyvision.org/>). Another great product with detailed guides and tutorials for beginners and advanced users is the Microsoft Cognitive Toolkit, available at <https://docs.microsoft.com/en-us/cognitive-toolkit/>.

14

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 developed by Google. At its core, `fastlane` is a set of Ruby scripts that automate the deployment of iOS and Android apps. Some scripts are only available for iOS, and some are not currently available for Flutter yet, 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 signs your apps
- `deliver` to upload your apps, screenshots, and metadata to the stores
- `pilot` to upload and manage your projects to TestFlight
- `scan` to run automated tests

In this chapter, we will cover the following topics:

- 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
- Auto-incrementing your Android build number
- 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.



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.

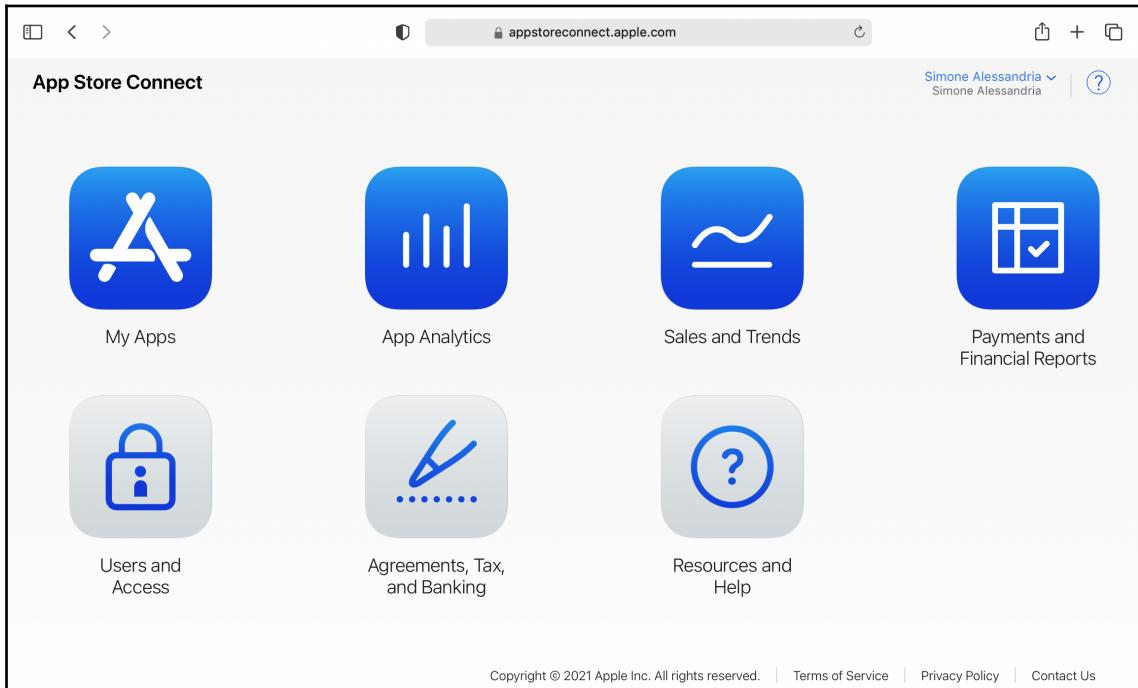
In order to follow the App Store Connect publication steps, you should have a Mac with Xcode installed.

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:



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:

The screenshot shows a web interface titled "Register a new identifier". At the top left is a link "[All Identifiers](#)". On the right is a blue "Continue" button. Below the title, there is a list of options with radio buttons:

- App IDs**
Register an App ID to enable your app, app extensions, or App Clip to access available services and identify your app in a provisioning profile. You can enable app services when you create an App ID or modify these settings later.
- Services IDs**
For each website that uses Sign in with Apple, register a services identifier (Services ID), configure your domain and return URL, and create an associated private key.
- Pass Type IDs**
Register a pass type identifier (Pass Type ID) for each kind of pass you create (i.e. gift cards). Registering your Pass Type IDs lets you generate Apple-issued certificates which are used to digitally sign and send updates to your passes, and allow your passes to be recognized by Wallet.
- Website Push IDs**
Register a Website Push Identifier (Website Push ID). Registering your Website Push IDs lets you generate Apple-issued certificates which are used to digitally sign and send push notifications from your website to macOS.
- iCloud Containers**
Registering your iCloud Container lets you use the iCloud Storage APIs to enable your apps to store data and documents in iCloud, keeping your apps up to date automatically.

4. On the screen you get 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** boxes. 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**:

Certificates, Identifiers & Profiles

[« All Identifiers](#)

Register an App ID

[Back](#)

[Continue](#)

Platform

iOS, macOS, tvOS, watchOS

App ID Prefix

MRHET26894 (Team ID)

Description

BMI Calculator

You cannot use special characters such as @, &, *, ', ", -, .

Bundle ID Explicit Wildcard

it.softwarehouse.bmicalculator

We recommend using a reverse-domain name style string (i.e., com.domainname.appname). It cannot contain an asterisk (*).

Capabilities

ENABLED NAME

- | | |
|--------------------------|--|
| <input type="checkbox"/> |  Access WiFi Information i |
| <input type="checkbox"/> |  App Attest i |
| <input type="checkbox"/> |  App Groups i |
| <input type="checkbox"/> |  Apple Pay Payment Processing i |

- At the end of the process, note that an app ID has been created.
- Go back to the apps page at <https://appstoreconnect.apple.com/apps>.
- 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 publish your **existing** app there.



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.

10. On the **New App** page, complete the required fields. Choose the available platforms for your new app, the name, and its primary language, and select the bundle ID you created in the previous steps, then select an SKU (a unique ID for your app that will not be visible on the App Store). An example of the data you can insert is as follows:

New App

Platforms ?

iOS macOS tvOS

Name ?

BMI Calculator

16

Primary Language ?

English (U.S.)

Bundle ID ?

BMI Calculator - it.softwarehouse.bmicalculator

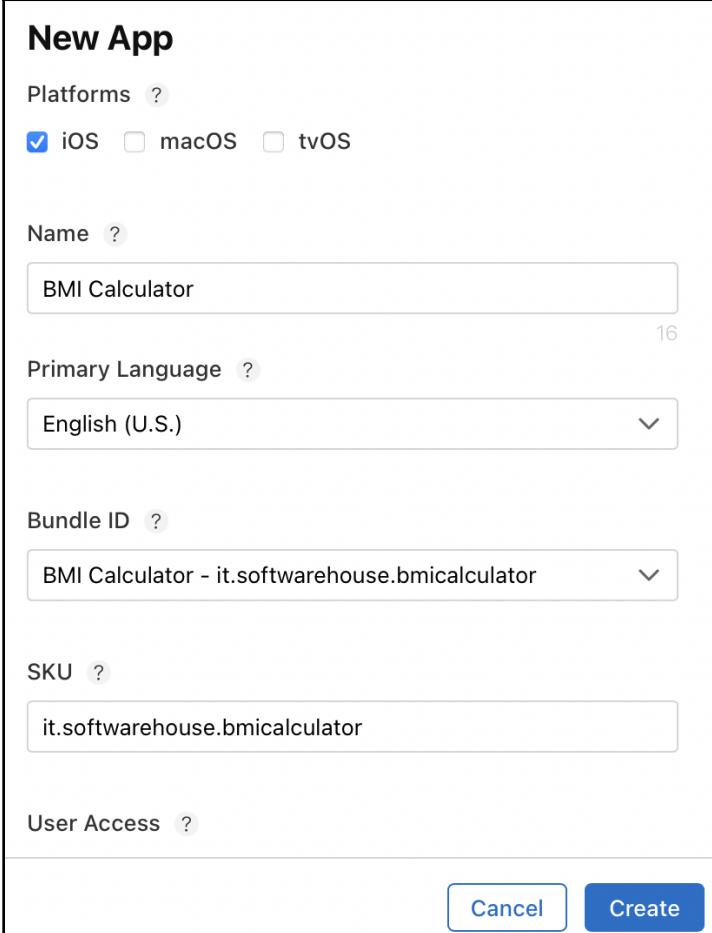
SKU ?

it.softwarehouse.bmicalculator

User Access ?

Create

Cancel



11. 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.



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.

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 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 finally register an app. This requires choosing a name, a primary language that you can select among the supported languages, the bundle ID that you created previously, and an **SKU**. The SKU is another identifier.



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

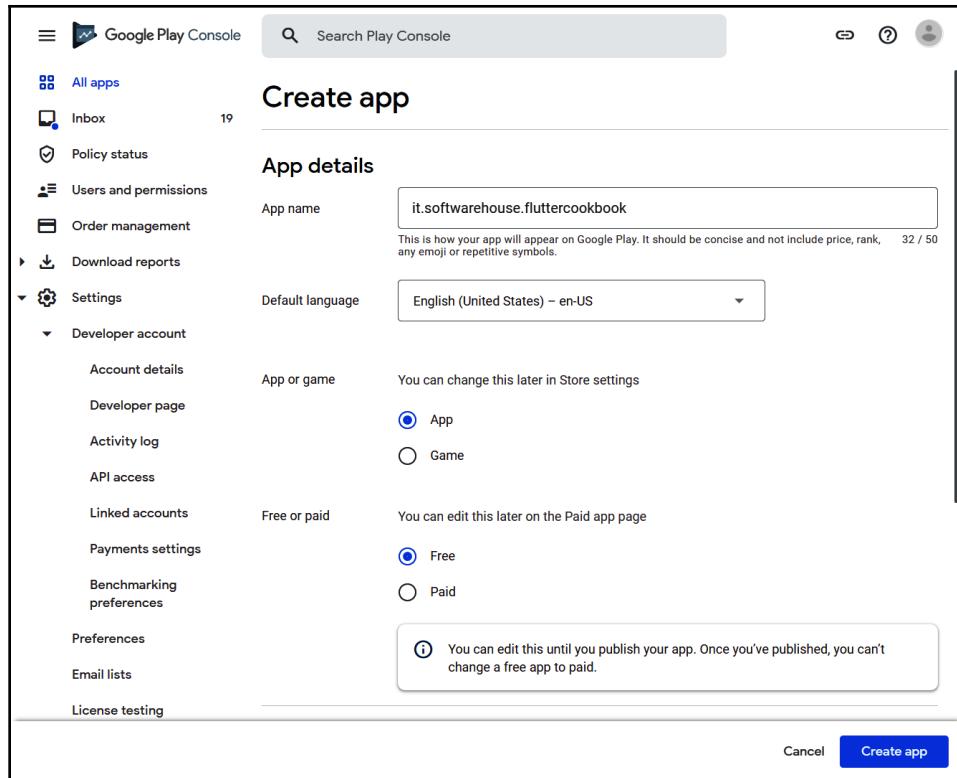
Before publishing an app to the Google Play Store, you should have a Google Developer account; you can get one for a one-time fee, and it is a prerequisite for this recipe and all the other recipes in this chapter that target Android.

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 the following screenshot:



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.



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.



Currently, in order to develop for Google Play, you only need 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, but when you choose whether your app is free or paid, currently **you cannot change this feature after your app is published.**

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 the stores is a long and cumbersome process; it may well take a full day of work for the first 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.



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 (2.7.2 at the time this chapter is being written). Then, double-click on the file and follow the installation process, leaving the default options.
2. Open Command Prompt and install `fastlane` with the following command:

```
gem install fastlane
```

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`. 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 **Settings** menu, then **API access**.

3. In the **Service accounts** section, select the **Create new Service Account** 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, a name for the service account, and a description, as shown in the following screenshot. Then, click the **Create** button:

The screenshot shows the Google Cloud Platform IAM & Admin Service Accounts page. The left sidebar lists various services: IAM, Identity & Organization, Policy Troubleshooter, Policy Analyzer, Organization Policies, Service Accounts (which is selected and highlighted in blue), Labels, Settings, Privacy & Security, Identity-Aware Proxy, Roles, and Audit Logs. The main content area has a header 'publish_sa' with a back arrow. It is divided into 'DETAILS' and 'PERMISSIONS' tabs, with 'DETAILS' being active. The 'Service account details' section contains fields for Name (set to 'publish_sa'), Description (set to 'Publishing Service Account'), Email (set to 'publish-sa@firebase-demo-305209.iam.gserviceaccount.com'), and Unique ID (set to '108784229501087015862'). The 'Service account status' section shows 'Account currently active' with a checked checkbox. At the bottom is a 'DISABLE SERVICE ACCOUNT' button.

7. In the **Grant this service account access to project** section, choose **Service Accounts | Service Account User** and click **Continue**.
8. In the **Grant users access to this service account** section, leave the boxes blank and click **Done**.

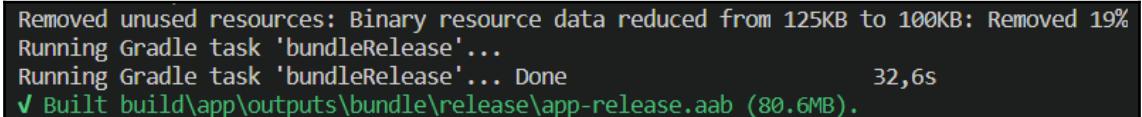
9. On the service account page, click on the **actions** button of the service account, and select the **Create key** menu option.
10. Leave **JSON** as the key format and click **Create**. This will automatically download a JSON file on your PC or Mac. *You will need this file later.*
11. Go back to 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 **Grant Access** link near your service account.
13. On the **Account Permissions** page, enable all the permissions in the **Release** 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 Command Prompt, run the following command:

```
flutter build appbundle;
```

16. Note the success message, as shown in the following screenshot:



```
Removed unused resources: Binary resource data reduced from 125KB to 100KB: Removed 19%
Running Gradle task 'bundleRelease'...
Running Gradle task 'bundleRelease'... Done                                32,6s
✓ Built build\app\outputs\bundle\release\app-release.aab (80.6MB).
```

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. If `brew` 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)"
```

3. Install `fastlane` with the following command:

```
brew install fastlane
```

4. Initialize `fastlane` with the following command:

```
fastlane init
```

5. When prompted, choose the **Automate app store distribution** option.
6. When prompted, insert your developer Apple ID username and password.
7. If appropriate, answer yes (y) at the prompt to create the App Store Connect app.
8. 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 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.



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 in your prompt:

```
Successfully installed certificate [YOUR CERTIFICATE ID]
Installed Provisioning Profile
```

8. In 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 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 (GitHub or other platforms).
- It installs the certificates from the repository in 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
```



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=/Users/sales/key.jks
```

4. In the `app build.gradle` file, 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  
    }  
}
```

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:

- You generate a new private key with the `-genkey` option.
- You create an alias for this key named `key` with the `-alias` option.
- You store the private key in a file called `key.jks` with the `-keystore` option.
- You specify 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 perform this task with the following instruction:

```
signingConfig signingConfigs.release
```

Once you add the correct configuration in your app's `build.gradle` file, each 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>.

Auto-incrementing your Android build number

Some of the `fastlane` features are only available for iOS, but it's very easy to create scripts that can also automate Android deployment when some features are not available.

In this recipe, you will learn how to write a simple script to increment the Android version of your app. This will increase your productivity and help you prevent a version duplicate issue when publishing your app.

Getting ready

Before following along with this recipe, you should have completed the previous recipes in this chapter that target Android: *Registering your Android app on Google Play*, *Installing and configuring fastlane*, and *Generating Android release certificates*.

How to do it...

You will now write a new script, or **lane**, to automatically increment the version of your app in Android:

1. Open the `android/fastlane/Fastfile` file in your project.
2. In the file, identify the `platform :android do` section.
3. At the end of the section, before the `end` instruction, insert a new lane, called `IncrementVersion`:

```
lane :IncrementVersion do
end
```

4. In the `IncrementVersion` lane, add the following instructions:

```
path = '../.../pubspec.yaml'
re = /version:\s+(\d+)/
s = File.read(path)
versionCode = s[re, 1].to_i
s[re, 1] = (versionCode + 1).to_s
f = File.new(path, 'w')
f.write(s)
f.close
```

5. Open a terminal window and go to the `android` folder of your project.
6. From the terminal, run the following command:

```
fastlane IncrementBuildNumber
```

7. Note the success message in the terminal: `fastlane.tools` finished successfully.

How it works...

At its core, `fastlane` is a set of Ruby scripts that automates the deployment of iOS and Android apps. In many cases, the scripts already available within the `fastlane` toolset are enough to publish your apps, but in some cases, you might find it useful to add your own scripts.



Writing a script that increments the build version is *not* necessary for iOS, as there is already a lane called `increment_build_number` that performs this same task.

In the `fastlane` world, a script is called a **lane**, and in this recipe, you created a new lane with the following instructions:

```
lane :IncrementVersion do
end
```

You write lanes using **Ruby**, an open source language that is beyond the scope of this book but should be simple enough to read.

The following instruction sets the `path` variable, which points to the position of the `pubspec.yaml` file in your project, which is the file that will be updated by the script:

```
path = '../../.pubspec.yaml'
```

With the following instruction, you create a regular expression that finds the version string, space, and digit that will be incremented:

```
re = /version:\s+(\d+)/
```

You open the file with the following instruction:

```
s = File.read(path)
```

You use the `to_i` method to retrieve the number from the leading characters in the string that you pass.

You then increment the number, write to the file, and then close it with the following instructions:

```
s[re, 1] = (versionCode + 1).to_s
f = File.new(path, 'w')
f.write(s)
f.close
```

See also

Creating new lanes for your apps can boost your productivity and automate almost every task for your publishing workflow. In order to get started with creating new lanes, you can see the guide at <https://docs.fastlane.tools/advanced/lanes/>.

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/android"  
    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 in 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.



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:

- 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.



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:

```
flutter_launcher_icons: ^0.8.1
```

4. At the same level as `dependencies` and `dev_dependencies`, add the `flutter_icons` node, as shown:

```
flutter_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.

For Android, check that the icons are available in the `android/main/res` folder, under the "drawable" directories.

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 to publish your app to the stores.



`fastlane snapshot` allows automatically taking 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 is not currently working at the time this chapter is being written.

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:

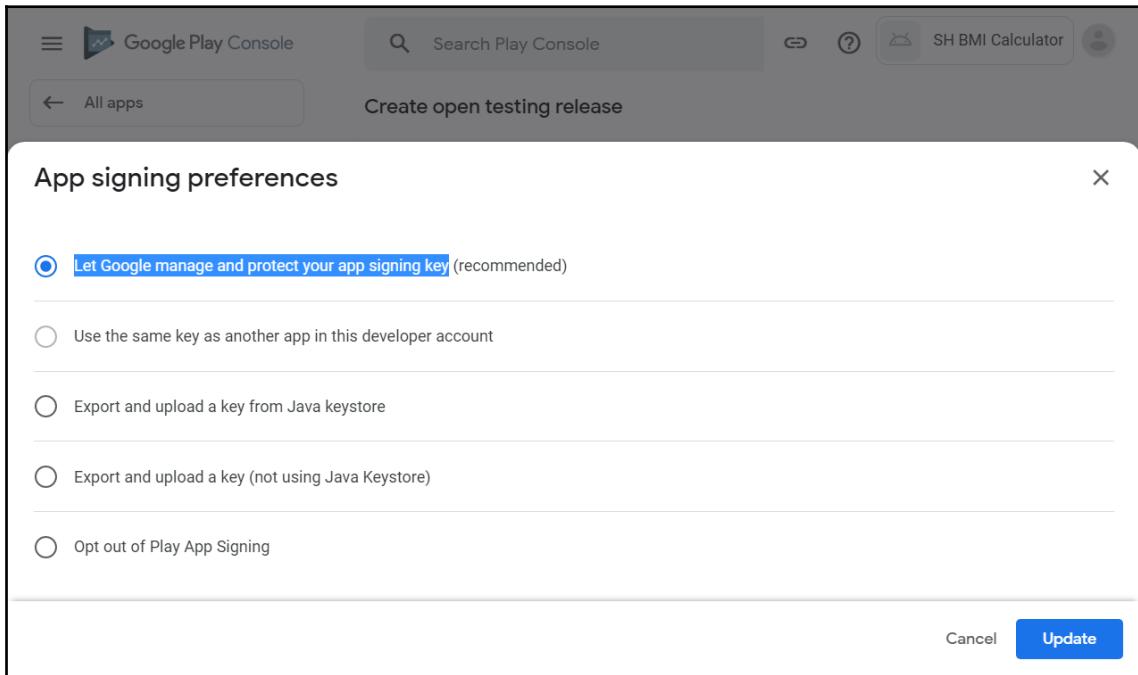
```
fastlane beta
```



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 press the **Update** button:



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:

App signing preferences

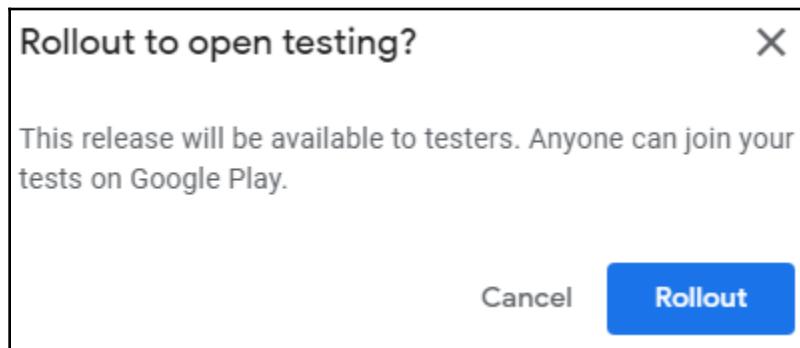
Let Google manage and protect your app signing key (recommended)

Use the same key as another app in this developer account

Export and upload a key from Java keystore

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:



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.



Currently, the first time you publish an Android release to the Google Play Store you must do it manually. For the second time, you can leverage the `fastlane supply` command to upload your builds.

The first time you publish an app, you need to provide 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 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>ITSSAppUsesNonExemptEncryption</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.



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.



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.



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 each 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 just do everything else. This can save you hours of work each 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).

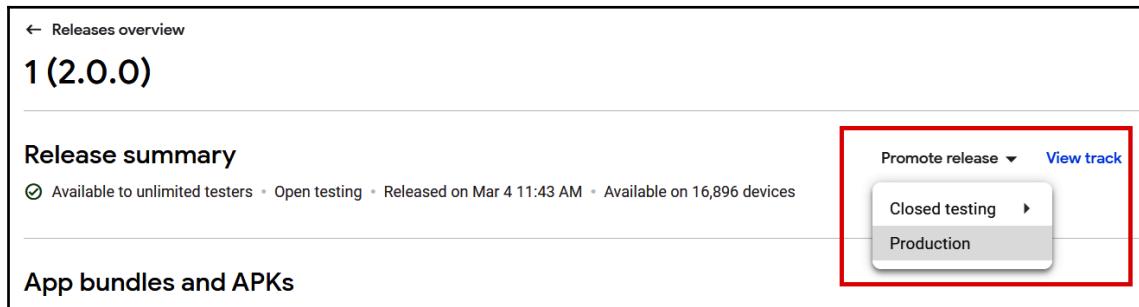
How to do it...

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. In 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:



5. Add or edit the release notes in the text field, then click the **Review Release** button.
6. Check the release summary 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 the **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 pressing the **Start rollout to production** button, you can expect to see your app in the Play Store within 48 hours. In my experience, this time is much shorter after you first publish your app (even 2 hours in some cases). For iOS, you can expect to have your app published within 4 days.

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.

15

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 go 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. Now Flutter is returning to its roots and once again running on the web. There are many other places where people are experimenting with Flutter, including Desktop, ChromeOS, and even the **Internet of Things (IoT)**. Eventually, it will get to a point where if your device has a screen, it will be able to run Flutter.

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, or Linux **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.



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

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 this recipe, you will also see how to make your layout **responsive**, and build your app so that it can be later published to any web server. You will also see how to solve a CORS issue when loading images.

In this recipe, 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. In the project's `pubspec.yaml` file, in the `dependencies` section, add the latest version of the `http` package:

```
http: ^0.13.0
```

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 {  
    String id;  
    String title;  
    String authors;  
    String thumbnail;  
    String description;  
}
```

6. In the `Book` class, create a constructor that sets all the fields:

```
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:

```
final String authority = 'www.googleapis.com';
final String path = '/books/v1/volumes';
Map<String, dynamic> params = {'q':'flutter dart', 'maxResults':
'40', };
```

12. Still in the `HttpHelper` class, create a new asynchronous method, called `getFlutterBooks`, that returns a `Future` of a `List` of `Book` objects, as shown in the following code:

```
Future<List<Book>> getFlutterBooks() async {
    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 {
    @override
    _BookListScreenState createState() => _BookListScreenState();
}
class _BookListScreenState extends State<BookListScreen> {
    @override
    Widget build(BuildContext context) {
        return Container();
    }
}
```

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 = [];
bool isLargeScreen;
```

17. In the `BookListScreenState` class, override the `initState` method, and there call the `getFlutterBooks` method to set the value of the `books` variable, as shown here:

```
@override
void initState() {
    HttpHelper helper = HttpHelper();
    helper.getBooks('flutter').then((List<Book> value) {
        setState(() {
            books = value;
        });
    });
    super.initState();
}
```

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:

```
if (MediaQuery.of(context).size.width > 600) {  
    isLargeScreen = true;  
} else {  
    isLargeScreen = false;  
}
```

19. Still in the `build` method, instead of returning a `Container`, 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:

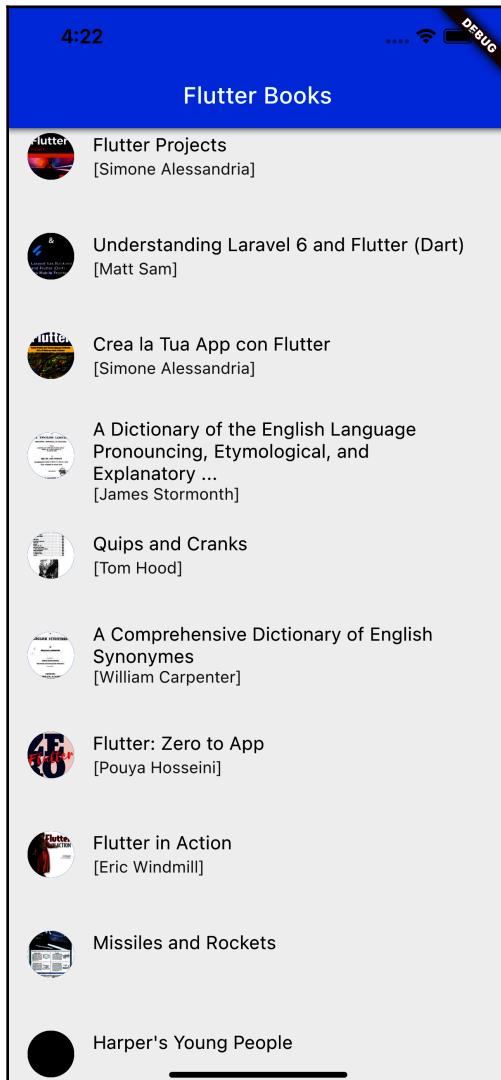
```
return Scaffold(  
    appBar: AppBar(title: Text('Flutter Books')),  
    body: GridView.count(  
        childAspectRatio: isLargeScreen ? 8 : 5,  
        crossAxisCount: isLargeScreen ? 2 : 1,  
        children: List.generate(books.length, (index) {  
            return ListTile(  
                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 {  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            title: 'Flutter Demo',  
            theme: ThemeData(  
                primarySwatch: Colors.blue,
```

```
        primarySwatch: Colors.blue,  
    ),  
    home: BookListScreen(),  
);  
}  
}
```

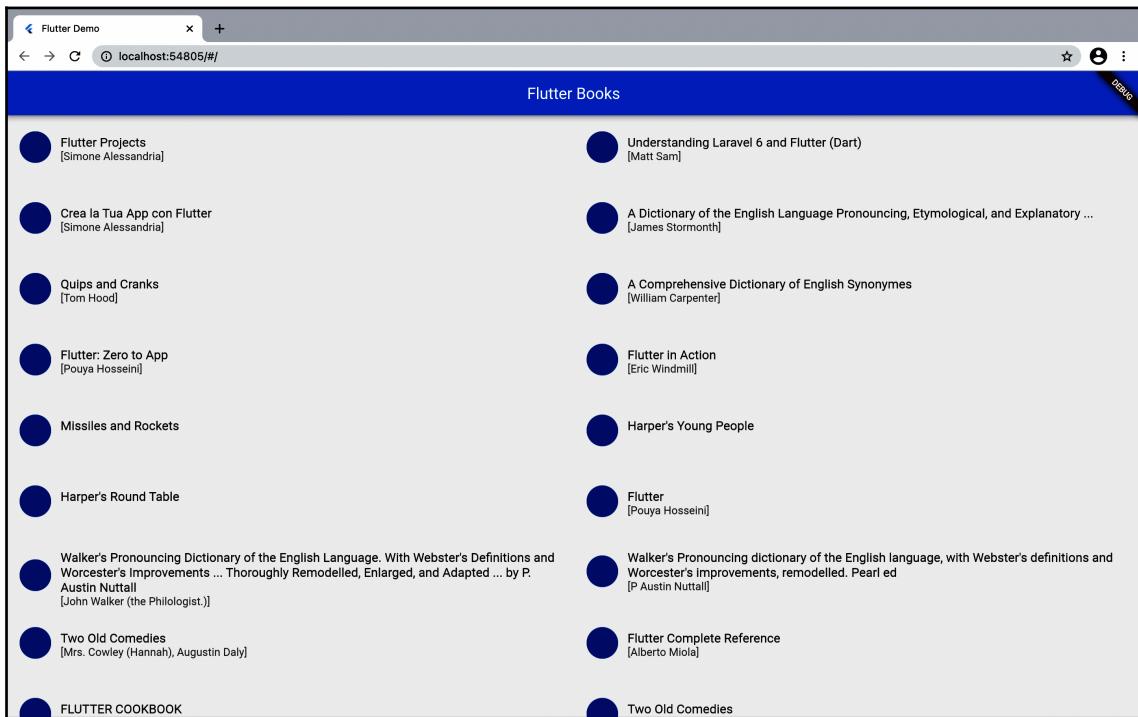
21. Run the app on your mobile device. You should see an app screen similar to the screenshot:



22. Stop the app, then in the Terminal window in your project's directory, run this command:

```
flutter run -d chrome
```

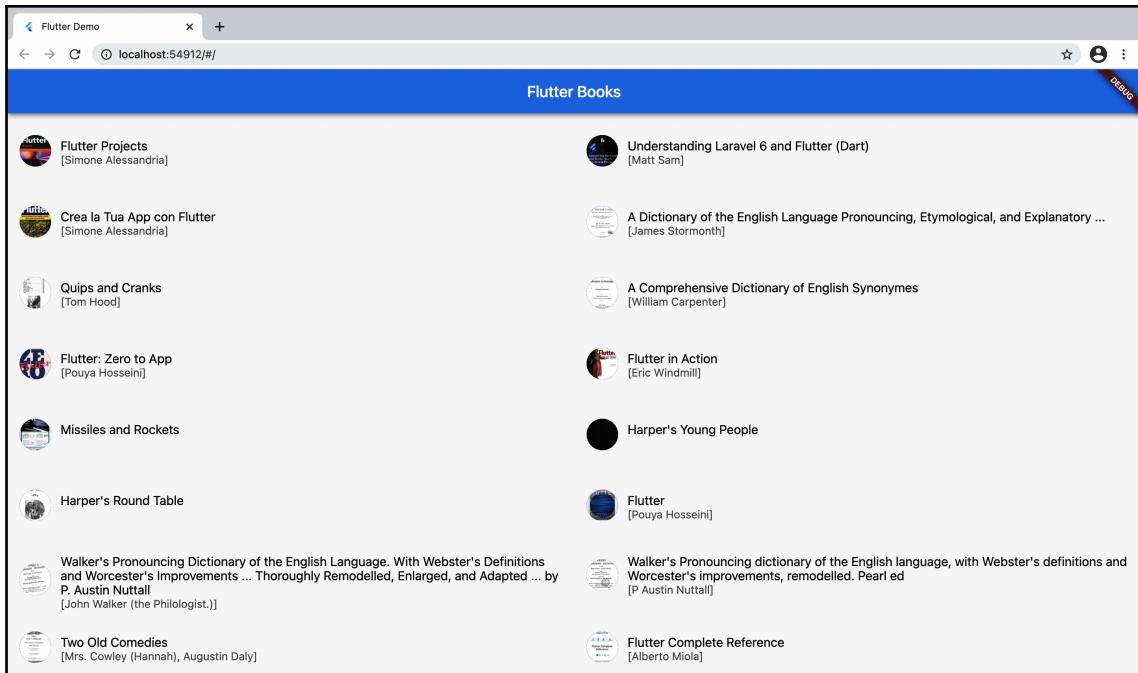
Note that the app is running on your web browser, and showing two columns, but the images are not showing:



23. Close the browser window, or press *Ctrl + C* on the Terminal to stop the debug.
24. In the Terminal, run the following command:

```
flutter run -d chrome --web-renderer html
```

25. This time, note that the images are showing correctly:



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 it's null.

In the case of the `imagelinks` field, at this time, 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.



You can also enable retrieving data through an `http` connection: the procedure depends on your target system. See <https://flutter.dev/docs/release/breaking-changes/network-policy-ios-android> for more information.

Another breaking change happened in the `http` package from version 0.13: the `http.get` method now requires a `Uri` object instead of a string. 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.



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 on your designs, or on 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`.

Another very interesting aspect of this recipe is the last part, where you run the app in your Chrome browser with this instruction:

```
flutter run -d chrome
```

The `-d` option allows specifying 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` are not showing. 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 `` 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.



CORS stands for **Cross-Origin Resource Sharing**. For security reasons, browsers block JavaScript requests that originate from an origin different from the 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 `` element, cross-origin requests are allowed.

See also...

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

With version 2, desktop support has been added to the already rich Flutter framework. This means that 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.

How to do it...

In order to run the app you have built on your Mac, implement the following steps:

1. In the Terminal, get to the project you have completed in the previous recipe, and run this command:

```
flutter config --enable-macos-desktop
```

2. In the same Terminal window, run this command:

```
flutter devices
```

3. In the list of devices returned by the command, note that **macOS (desktop)** is now showing among the available devices.



If you don't see the **macOS (desktop)** device in the list, try closing and reopening your editor and the Terminal, and then run the `flutter devices` command again.

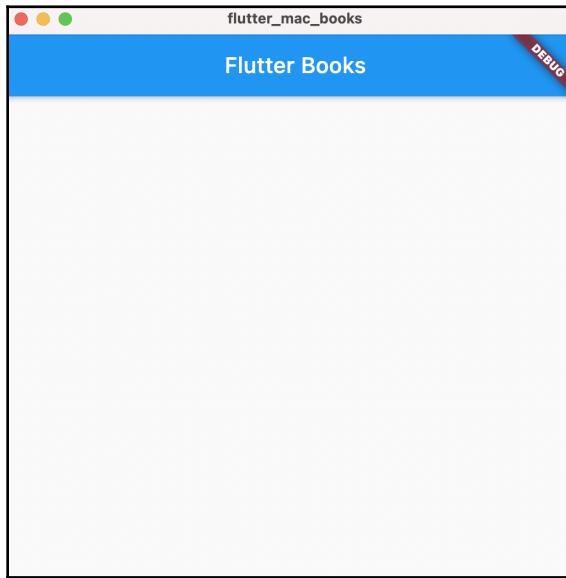
4. In the Terminal, in order to create the macOS app run the command (the dot . is part of the command):

```
flutter create .
```

5. Still in the Terminal window, run this command:

```
flutter run -d macos
```

6. Note that the app is running, but the books and images are not showing on the screen:



7. Run the app in debug mode from your editor, choosing `macos` as the device, and note that you get a `SocketException` (`Operation not permitted`) error as shown in the screenshot:

```
13 Future<List<Book>> getFlutterBooks() async {  
14  
15     Uri uri = Uri.https(authority, path, params);  
D 16     Response result = await http.get(uri);  
  
Exception has occurred.  
SocketException (SocketException: Connection failed (OS Error: Operation not permitted, errno = 1), address =  
www.googleapis.com, port = 443)
```

8. Open the `macos/Runner/DebugProfile.entitlements` file in your project and add this key:

```
<key>com.apple.security.network.client</key>  
<true/>
```

9. Open the `macos/Runner/Release.entitlements` file and add the same key you added in the `DebugProfile.entitlements` file.
10. Run the app again, and note that the books' data and images are showing correctly.

How it works...

Before running your app on a desktop, you should enable your platform. In this recipe, you enabled macOS with this command:

```
flutter config --enable-macos-desktop
```

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 macos
```

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 at 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/>
```



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://developer.apple.com/macos/submit/>.

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 up on 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 (**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>.

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:

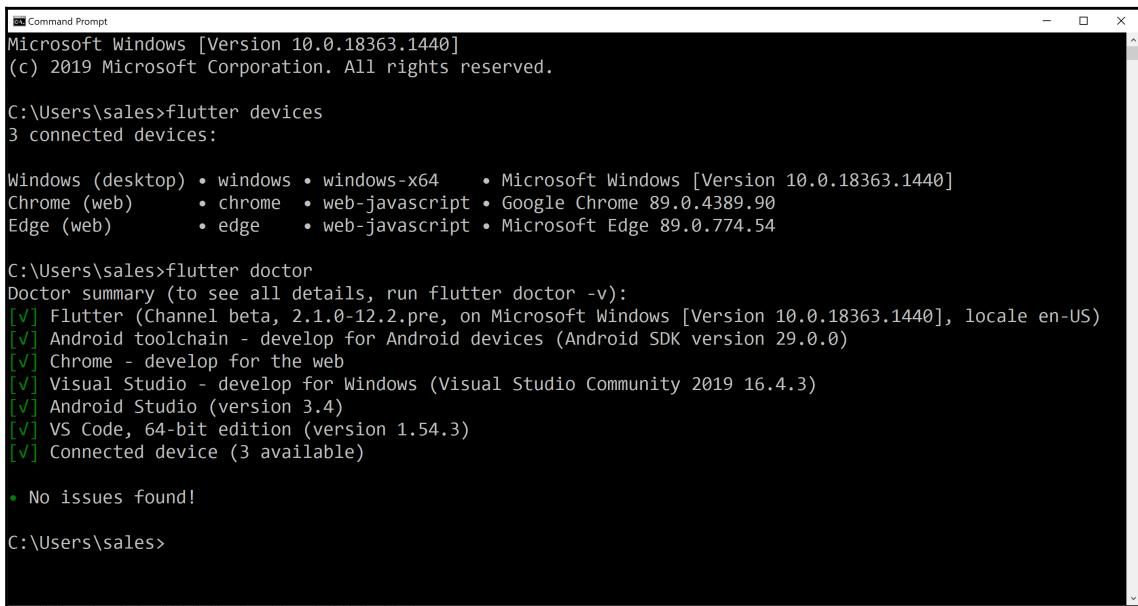
1. In a Command Prompt window, get to the project you completed in the first recipe in this chapter, and run this command:

```
flutter config --enable-windows-desktop
```

2. In the same window, run this command:

```
flutter devices
```

3. In the list of devices returned by the command, note that **Windows (desktop)** is now showing as a device.
4. Run the `flutter doctor` command, and note that **Visual Studio – develop for Windows** is showing, as shown in the screenshot:



```
Microsoft Windows [Version 10.0.18363.1440]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\sales>flutter devices
3 connected devices:

Windows (desktop) • windows • windows-x64      • Microsoft Windows [Version 10.0.18363.1440]
Chrome (web)       • chrome   • web-javascript • Google Chrome 89.0.4389.90
Edge (web)         • edge     • web-javascript • Microsoft Edge 89.0.774.54

C:\Users\sales>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel beta, 2.1.0-12.2.pre, on Microsoft Windows [Version 10.0.18363.1440], locale en-US)
[✓] Android toolchain - develop for Android devices (Android SDK version 29.0.0)
[✓] Chrome - develop for the web
[✓] Visual Studio - develop for Windows (Visual Studio Community 2019 16.4.3)
[✓] Android Studio (version 3.4)
[✓] VS Code, 64-bit edition (version 1.54.3)
[✓] Connected device (3 available)

• No issues found!

C:\Users\sales>
```



If you don't see the **Windows (desktop)** device in the list, try closing and reopening your Command Prompt, then run the `flutter devices` command again.

5. In Command Prompt, in order to create the Windows app, run this command:

`flutter create .`
6. Run the app with your editor, or by calling this command:

`flutter run -d Windows`
7. Note that the app runs correctly, and the books and images are shown on the screen.

How it works...

Before running your app on a desktop, you should enable your platform. In this recipe, this was performed with this command:

```
flutter config --enable-windows-desktop
```

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 at the top-right corner of the screen.

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 developing 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 up on 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 the address <https://console.firebaseio.google.com>.
4. Create a new Firebase project (you can also use an existing one).



For a refresher on Firebase projects, see the *Configuring a Firebase app* recipe in Chapter 12, *Using Firebase*.

5. Download the Firebase CLI. You will find the link for your OS at <https://firebase.google.com/docs/cli>.
6. In your Terminal, type the following command:

```
firebase login
```

7. From the browser window that asks for your login, confirm your credentials and allow the required permissions. At the end of the process, note the success message in your browser:

Woohoo!

Firebase CLI Login Successful

You are logged in to the Firebase Command-Line interface. You can immediately close this window and continue using the CLI.

8. In your terminal, note the success message: ✓ Success! Logged in as [yourusername].
9. Type this command:

```
firebase init
```

- When prompted, choose the hosting option.
 - When prompted, choose the Use an existing project option and choose the project you have created above.
 - When prompted, type build/web to choose the files that will be deployed.
 - When prompted, answer "no" when asked whether you want to overwrite the index.html file.
 - When prompted, confirm that you want to configure a single-page application.
 - When prompted, choose whether you want to automatically deploy your changes to GitHub. A screenshot of the full configuration process is shown here:

```
#####
##   ##   ##   ##   #####
##   ##   ##   ##   ##   ##
#####   ##   #####   #####   #####   #####
##   ##   ##   ##   ##   ##   ##   ##   ##
##   ##   ##   ##   ##   ##   ##   ##   ##

You're about to initialize a Firebase project in this directory:

/Users/simonealessandria/Documents/flutter_mac_books

Before we get started, keep in mind:

* You are initializing in an existing Firebase project directory

? Which Firebase CLI features do you want to set up for this folder? Press Space to select features, then Enter to confirm your choices. Hosting: Configure and deploy Firebase Hosting sites

== Project Setup

First, let's associate this project directory with a Firebase project.
You can create multiple project aliases by running firebase use --add,
but for now we'll just set up a default project.

i .firebaserc already has a default project, using flutter-cookbook-43242.

== Hosting Setup

Your public directory is the folder (relative to your project directory) that
will contain Hosting assets to be uploaded with firebase deploy. If you
have a build process for your assets, use your build's output directory.

? What do you want to use as your public directory? build/web
? Configure as a single-page app (rewrite all urls to /index.html)? Yes
? Set up automatic builds and deploys with GitHub? No
? File build/web/index.html already exists. Overwrite? No
i Skipping write of build/web/index.html

i Writing configuration info to firebase.json...
i Writing project information to .firebaserc...

✓ Firebase initialization complete!
```

16. In your Terminal, type the following:

```
firebase deploy
```

- 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.

Flutter developers 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 Firebase Command Line Interface makes publishing a web app with Flutter extremely easy. Here are the steps that you should take, and you followed in this recipe:

- You create a Firebase project.
- You install the Firebase CLI (**only once** for each developing machine).
- You run `firebase login` to log in to your Firebase account.
- You run `firebase init`. For this step, you need to provide Firebase with some information, including your target project and the position of your files that will be published.



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 with an FTP client 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 option to publish your Flutter web apps is 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 than 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 up 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 the next steps:

1. Open the `book_list_screen.dart` file in your project.
2. At the top of the `_BookListScreenState` class, add a new `List` of `Color`, called `bgColors`, and set it to an empty `List`:

```
List<Color> bgColors = [];
```

3. In the `initState` method, in the `then` callback of the `getFlutterBook` 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 `Container` widget, and the `Container` 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: Container(  
                child: ListTile(  
                    [...]  
    ])
```

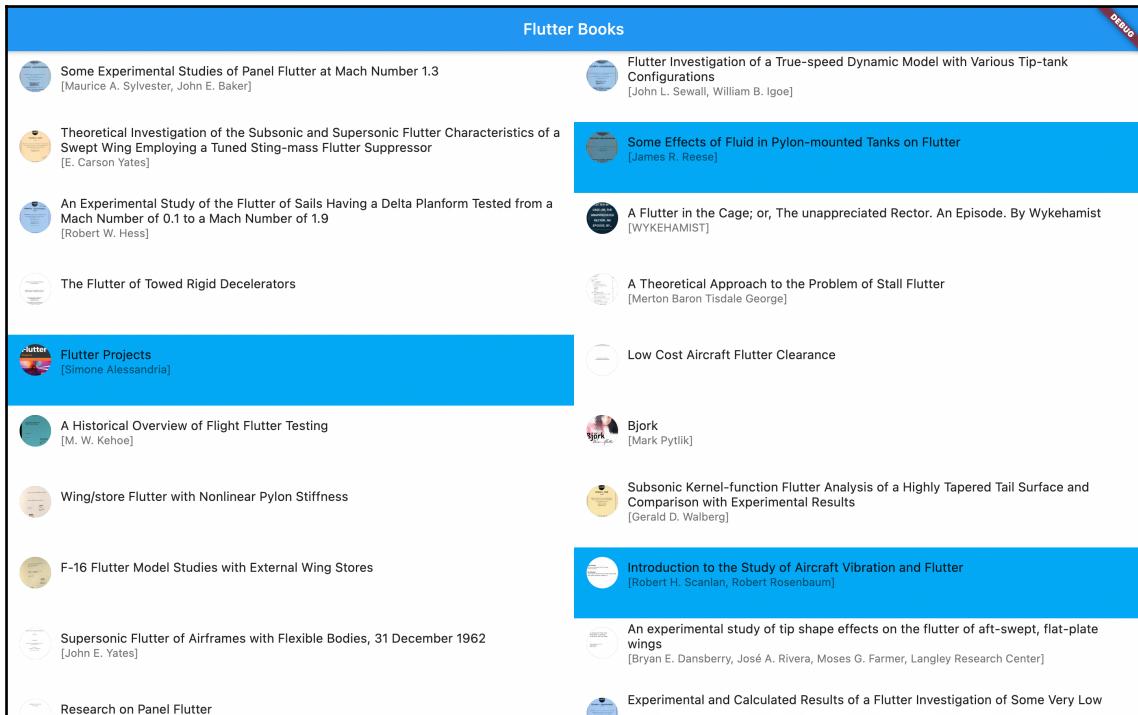
6. In the `Container`, set the `color` property based on the value of the `bgColors` list at the `index` position:

```
color: bgColors.length > 0 ? 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 screenshot:



9. Right-click on one or more of the items that you selected previously. The background color will get 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 go back to white.

How it works...

When you design an app that works both on mobile and on 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 select/deselect items in a `GridView`. You can use a `GestureDetector` both for touch screen gestures, such as swipes and long-presses, and for mouse gestures, such as right-click and scroll.

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 touch screen and when they click on the main button of a mouse or 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 a graphic tablet.

`onLongPress` is mainly targeted at mobile devices or touch screens 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.getFlutterBooks().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 up on 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, open the pubspec.yaml file and add the following dependency:

```
menubar:  
  git:  
    url: git://github.com/google/flutter-desktop-embedding.git  
    path: plugins/menubar
```

2. In the http_helper.dart file in the data folder, edit the getFlutterBooks method, so that it becomes a more generic getBooks method that takes a String as a parameter, and move the params declaration inside the method, as shown here:

```
class HttpHelper {  
  final String authority = 'www.googleapis.com';  
  final String path = '/books/v1/volumes';  
  Future<List<Book>> getBooks(String query) async {  
    Map<String, dynamic> params = {'q':query, 'maxResults': '40', };  
    ...  
}
```

3. At the top of the book_list_screen.dart file, add the menubar import:

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

4. Move the HttpHelper helper declaration to the top of the _BookListScreenState class, and in the initState method, set helper to be a new instance of HttpHelper:

```
class _BookListScreenState extends State<BookListScreen> {  
  List<Book> books = [];  
  bool isLargeScreen;  
  HttpHelper helper;  
  @override  
  void initState() {  
    helper = HttpHelper();  
  [...]
```

5. 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:

```
updateBooks(String key) {  
  helper.getBooks(key).then((List<Book> value) {
```

```
        setState(() {
            books = value;
        });
    });
}
}
```

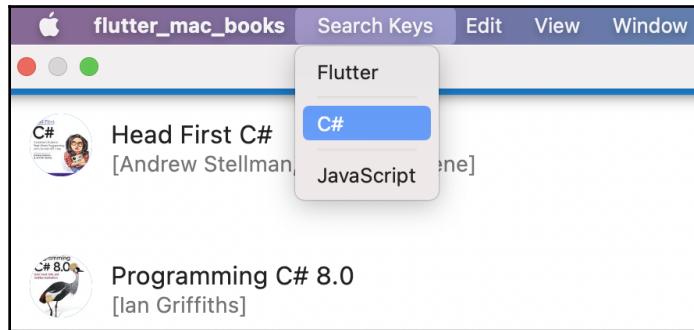
6. At the bottom of the `_BookListScreenState` class, add another new method, called `addMenuBar`, that adds three search terms as menu items in the menu bar, as shown here:

```
void addMenuBar() {
    setApplicationMenu([
        Submenu(label: 'Search Keys', children: [
            MenuItem(
                label: 'Flutter',
                enabled: true,
                onClicked: () => updateBooks('flutter')),
            MenuDivider(),
            MenuItem(
                label: 'C#',
                enabled: true,
                onClicked: () => updateBooks('c#')),
            MenuDivider(),
            MenuItem(
                label: 'JavaScript',
                enabled: true,
                onClicked: () => updateBooks('javascript')),
        ])
    ]);
}
```

7. In the `initState` method, call the `addMenuBar` method and update the call to the `helper` method so that it gets Flutter books as soon as the screen loads. The updated `initState` method should look like the following code:

```
@override
void initState() {
    addMenuBar();
    helper = HttpHelper();
    helper.getBooks('flutter').then((List<Book> value) {
        setState(() {
            books = value;
        });
    });
    super.initState();
}
```

- Run the app. In the menu bar, you should see a new menu, called **Search Keys**. 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:



How it works...

At the time of writing, the `menubar` plugin is still in experimental mode, which means it will probably be included in Flutter or in an official package soon. Meanwhile, you can use it through its GitHub URL. This is why in the `pubspec.yaml` file, you added the Git URL instead of a package dependency:

```
menubar:  
  git:  
    url: git://github.com/google/flutter-desktop-embedding.git  
    path: plugins/menubar
```

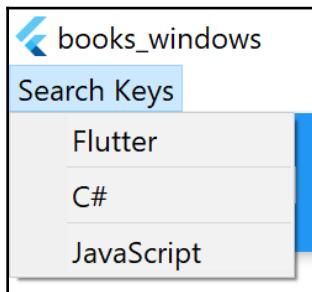
When you want to add a menu item in your app, you need to call the `setApplicationMenu` method, which is asynchronous and takes a `List` of `SubMenu` objects.

A `SubMenu` in turn is a class that requires a label (the text you see on the menu) and a list of children, which can be `MenuItem` or `MenuDivider` objects. When you insert a `MenuDivider` in a `SubMenu`, a horizontal line separator will be added to the list of items in the menu.

A `MenuItem` is what your users will actually click to perform the action you have provided in your app. In this recipe, you set its `label`, the `enabled` Boolean, and the `onClicked` callback that calls the `updateBooks` method:

```
MenuItem(  
  label: 'Flutter',  
  enabled: true,  
  onClicked: () => updateBooks('flutter')),
```

The great thing is that depending on the system you are using, the menus will change accordingly. Here, you can see an example of the menu as it appears on a Windows desktop:



See also...

At the time of writing, the menu bar plugin is still a prototype. In the future, it might be published as an official package in `pub.dev`, or embedded in Flutter itself. To stay updated on this project, have a look at <https://github.com/google/flutter-desktop-embedding/tree/master/plugins/menubar>.

You can also add menus to Linux systems, and the process is quite similar to what was described in this recipe: see https://pub.dev/packages/flutter_menu for more details.



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Index

A

abstract interface 239
AlertDialog 280
Android app
 registering, on Google Play Console 548, 549, 550
Android build number
 auto-incrementing 560, 561, 562
Android emulator
 creating 22, 23, 24, 25
Android metadata
 adding 563, 564
Android release certificates
 generating 557, 558, 559
Android SDK setup
 configuring 20
Android Studio
 about 26, 27
 installing 20, 21, 22
Android
 fastlane, configuring for 551, 552, 553
animated widgets, Flutter
 reference link 443
AnimatedContainer class
 reference link 417
AnimatedList widget
 using 443, 444, 445, 446, 447
 working 448, 449
animation package
 reference link 457
animations Flutter package
 using 453, 454, 455, 456, 457
animations tutorial
 reference link 424
animations
 design, by adding multiple animations 424, 425

designing, with AnimationController 418, 419, 420, 422, 423, 424
designing, with curves 426, 427, 428, 429
optimizing 430, 431
app metadata
 configuring 563, 564
app state visibility
 making, across multiple screens 225, 226, 227, 228, 229, 230
App Store Connect
 about 542
 iOS app, registering on 542, 543, 544, 545, 547
App Store
 app, moving to production 577
app
 fonts, importing into 115, 116, 117, 119, 120
 Google Maps, adding to 398, 399, 400, 401, 403
 images, importing into 115, 116, 117, 119, 120
 moving, to production in App Store 577
 moving, to production in Google Play Store 577
 publishing, to Google Play Store 576
 running, on macOS 590, 591, 592, 593
 running, on Windows 594, 595, 596
 state, adding 167, 168, 170, 171
Apple Developer Program
 reference link 542
async/await
 used, for dealing with errors 268
 used, for removing callbacks 256, 257, 258, 259, 260
asynchronous code
 errors, resolving 266
asynchronous functions
 navigation routes, turning into 276, 277, 278, 280
asynchronous pattern, Flutter

reference link 255
asynchronous task 294
authentication 472

B

Backend as a Service (BAAS) 458
barcode
 reading 520, 521, 522
beta testing options, Google Play Console
 reference link 573
beta version, of app
 publishing, in Google Play Store 568, 569, 570, 572
Big-O notation
 reference link 70
BLoC pattern
 about 219
 references 380
 using 375, 376, 377, 378, 379
bottom sheets
 presenting 206, 207, 208, 209, 210
box constraints
 reference link 137
builder pattern
 reference link 82
built-in variable type
 reference link 48
built_value libraries
 reference link 307
Business Logic Component (BLoC) 375
buttons
 interacting with 173, 174, 175, 176, 177, 178

C

Caffe
 reference link 540
callbacks
 removing, with `async/await` 256, 257, 258, 259, 260
cascade operator
 using 79, 80, 81, 82
channel
 selecting 30
class constructor shorthand
 using 60, 61, 62, 63
classes
 creating 60, 61, 62, 63
Clean Code
 reference link 220
closed testing 572
closures
 about 58
 functions, using as variables 58, 59, 60
Cloud Firestore, versus Realtime Database
 reference link 495
cloud
 files, storing in 500, 501, 503, 504
cocoapods 16, 17
code
 writing, with higher-order functions 70, 71, 72, 73
collection library
 reference link 69
collection-for syntax 183
collections
 data, grouping 64, 65, 66, 67
 data, manipulating 64, 65, 66, 67
command line
 setting up 10
Completer object
 reference link 263
const
 versus final 42, 43, 44, 45, 46, 47
 versus var 42, 43, 44, 45, 46, 47
constants 305
constrained spacing 136
container animations
 creating 413, 414, 415, 416, 417
Container Transform transition 455
Container widget
 using 102, 103, 104, 105, 107, 108
controller layer 231
Create, Read, Update, and Delete (CRUD) 70, 243
crypto
 reference link 324
curves
 reference link 429
 used, for designing animations 426, 427, 428, 429
CustomPaint

reference link 144
used, for drawing shapes 137, 138, 139, 140,
141, 143

D

Dart 2.3
reference link 70
Dart Analysis Server
about 150
reference link 151
Dart class
transforming, into JSON string 296, 297
Dart models
converting, into JSON 286, 287, 288, 289, 290,
291, 292, 293
Dart streams
using 346, 347, 348, 349, 350, 351
DartPad
URL 42, 43
data layer
managing, with InheritedWidget 220, 221, 222,
223, 224
data transform
injecting, into streams 358, 359, 360, 361
data
grouping, with collections 64, 65, 66, 67
manipulating, with collections 64, 65, 66, 67
saving, with SharedPreferences 308, 309, 310,
311, 312
storing, with secure storage 321, 322, 323
datasets
handling, with list builders 185, 186, 187, 188
de-serialization (decoding) 287
deferred rendering 187
DELETE action
performing, on web service 342, 343, 344
dependencies
importing 382
design specifications, for iOS and Android
reference link 165
desktop menus
interacting with 604, 605, 606, 607, 608
development provisioning profile 556
device camera
using 506, 507, 508, 510, 511, 512, 513, 514

DevTools
reference link 93
dialog
reference link 284
showing, on screen 201, 202, 203, 204, 205
using 280, 281, 282, 283
directories
working with 317, 318, 320
DismissDirection class
reference link 453
Dismissible widget
swipe, implementing 450, 451
working 452
distribution profile 556

E

encrypt
reference link 324
environment
checking, with Flutter Doctor 14, 15
error handling, Dart
reference link 269
error handling
enabling 357, 358
errors
resolving, in asynchronous code 266
escape character 51

F

face detection API 529
face detector
building 526, 527, 528, 529, 530
facial gestures
detecting 526, 527, 528, 529, 530
fastlane
configuring 550
configuring, for Android 551, 552, 553
installing 550
installing, for iOS 554
installing, on Mac 551
installing, on Windows 551
reference link 550
file access, guide
reference link 320
files

accessing, with directories 317
storing, in cloud 500, 501, 503, 504

filesystem
accessing, with `path_provider` library 313

filter functions 75

final
versus `const` 42, 43, 44, 45, 46, 47
versus `var` 42, 43, 44, 45, 46, 47

Firebase 458

Firebase Analytics
integrating 482, 483, 484, 485, 486

Firebase app
Android configuration 460, 461, 462
configuring 459, 460
iOS configuration 462, 463

Firebase Cloud Firestore
using 487, 488, 489, 490, 491, 492, 493, 495

Firebase Cloud Messaging (FCM)
about 499
Push Notifications, sending 495, 496, 497, 498, 499
reference link 499

Firebase Cloud Storage 500

Firebase console
URL 459

Firebase dependencies
adding 463

first-class functions 58, 77

flattening 76

Flexible and Expanded widgets
used, for proportional spacing 130, 133, 134, 135, 136, 137

Flutter app
creating 31, 32, 33

Flutter Assets Guide
reference link 120

Flutter Canvas class
reference link 144

Flutter Desktop
responding, to mouse events 600, 601, 602, 604

Flutter Doctor
environment, checking 14, 15
running 18, 19

Flutter Favorite program
reference link 386

Flutter SDK
cloning 10
configuring 9
managing, with Git 9
reference link 10

Flutter website
deploying 596, 597, 598, 599

Flutter
URL 9

FlutterFire
integrating, into Flutter app 463, 464

FocusManager
reference link 220

fonts
importing, into app 115, 116, 117, 119, 120

functions
using, as variables with closures 58, 59, 60
writing 54, 55, 56, 57

Future
completing 260, 261, 263
states 255
using 249, 250, 251, 252, 253, 254
using, with `StatefulWidget` 269, 270, 271

FutureBuilder
reference link 275
using 273, 274, 275

FutureGroup
reference link 266

G

GestureDetector widget
reference link 604

Git
installing 9
URL 9
using, to manage Flutter SDK 9

GitHub Desktop
URL 9

global themes
applying 158, 159, 160, 162, 163, 164

Google Maps
adding, to Flutter app 398, 399, 400, 401, 403

Google Places API
URL 410

Google Play Console
about 548
Android app, registering on 548, 549, 550
Google Play Store
app, moving to production 577
app, publishing to 576
beta version of app, publishing 568, 569, 570, 572
Google Sign-in
adding 474, 475, 476, 477, 479, 480, 481
graphics processing unit (GPU) 143

H

Hardware Accelerated Execution Manager (HAXM)
installer 21
Hero animations
reference link 437
using 432, 433, 434, 435, 436, 437
higher-order functions
chaining 78
code, writing with 70, 71, 72, 73
iterables 78
reference link 78
home route 200
Homebrew
about 18
URL 18
HTTP client
data, obtaining from 324, 326, 329
designing 324, 326, 329

I

icons
adding, to app 565, 567
IDE
selecting 26
image labeling
about 523
adding, to app 523, 524, 525, 526
image
importing, into app 115, 116, 117, 119, 120
text, recognizing from 515, 516, 517, 519
immutable data
benefits 48
immutable widgets

creating 90, 91, 92, 93, 94, 95
InheritedWidget
reference link 224
used, for managing data layer 220, 221, 222, 223, 224
IntelliJ IDEA
about 29
URL 29
interface-based programming
reference link 247
internal testing 572
iOS app, beta version
publishing, with TestFlight 573, 574, 575, 576
iOS app
registering, on App Store Connect 542, 543, 544, 545, 547
iOS code signing certificates
generating 555, 556
iOS SDK
configuring 15
iOS
fastlane, installing for 554
metadata, adding for 564
itemExtent 188

J

JavaScript Object Notation (JSON)
about 287
Dart models, converting into 286, 287, 288, 289, 290, 291, 292, 293
JSON data structure
example 287
JSON errors
catching 305, 306, 307
JSON file
reading 294
JSON schemas
handling 298, 299, 300, 301, 302
JSON string
Dart class, transforming into 296, 297
transforming, into list of Map objects 294
json_serializable
reference link 307, 590

K

keytool 557

L

lane script 562

language

identifying 531, 533, 534, 535, 536

list builders

used, for handling large datasets 185, 186, 187, 188

location services

using 404, 405, 406

login form

creating 465, 466, 467, 468, 469, 470, 471, 472, 473, 474

M

Mac

fastlane, installing on 551

macOS command-line

setting up 10, 11

macOS

app, running on 590, 591, 592, 593

map function 74

Map objects

transforming, into Pizza objects 295

markers

adding, to map 407, 408, 409, 410, 411

metadata

adding, for iOS 564

method cascades

reference link 82

Microsoft Cognitive Toolkit

reference link 540

ML Kit face identification feature

reference link 531

MockLab

about 324

reference link 325, 331

Model View Controller (MVC) 230

Model View Presenter (MVP) 230

Model View ViewModel (MVVM) 230

model-view separation

about 213

implementing 214, 215, 216, 218

working 219, 220

models 213

mouse events

responding to 600, 601, 602, 604

multiple Futures

firing 263, 264, 265

multiple stream subscriptions

allowing 367, 368, 369, 370

N

n-tier architecture

controllers, designing 231, 232, 233, 234, 235, 236

repositories, designing 237, 238, 239, 240

services, designing 241, 242, 243, 244, 245, 246, 247

named routes 198

navigation routes

invoking, by name 198, 199, 200, 201

turning, into asynchronous functions 276, 277, 278, 280

next screen

navigating to 195, 196, 197, 198

null safety

about 82, 83, 84, 85, 86, 88

reference link 88

null-aware operators

reference link 247

O

object-oriented programming (OOP)

about 60

fundamentals 63

of-context pattern 210

open testing 572

P

package managers 15

packages, versus plugins

reference link 392

packages

about 381

creating 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397

importing 382, 383
path variables
 saving 10
path_provider library
 reference link 313
 using 313, 314
 working 315, 316
payload 337
permission_handler
 reference link 407
platform language
 selecting, for app 33
plugin 386
POST action
 performing, on web service 331, 332, 333, 336, 338
Postman 337
premade animation transitions
 using 438, 439, 440, 441
 working 442
pricing and thresholds, Google Maps
 reference link 403
project
 main folders 34, 35
proportional spacing
 with Flexible and Expanded widgets 129, 133, 134, 135, 136, 137
provisioning profile
 about 556
 generating 555, 556
pub.dev 384
Push Notifications
 about 495
 sending, with Firebase Cloud Messaging (FCM) 495, 496, 497, 498, 499
PUT action
 performing, on web service 338, 339, 340, 342
pyramid of doom
 about 151
 reference link 158

R

reactive user interfaces
 creating, with StreamBuilder 370, 371, 372, 374
reduce function 76

Redux patterns 219
refactoring 151
regular expression 191
regular expression, in Dart
 reference link 192
responsive app leveraging Flutter Web
 creating 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590
RESTful web services 337
reverse domain name notation 547
route arguments 199
routing 198

S

Scaffold
 using 96, 97, 98, 99, 101, 102
Scalable Vector Graphics (SVG) 143
scikit-learn
 reference link 540
screen
 dialog, showing on 201, 202, 203, 204, 205
ScrollController 188
scrolling content
 creating 178, 179, 180, 181, 182, 183
scrolling widgets
 about 184
 designing 185
secure storage
 used, for storing data 321, 322, 323
serialization (encoding) 287
shapes
 drawing, with CustomPaint 137, 138, 139, 140, 141, 143
shared_preferences plugin
 reference link 312
SharedPreferences
 used, for saving data 308, 309, 310, 311, 312
single and broadcast subscriptions, Dart
 reference link 370
singleton pattern 330
sinks
 using 352, 353, 354, 356, 357
SliverAnimatedList
 reference link 449
sort function 75

Sourcetree
 URL 9

Spacer class
 reference link 136

State class, life cycle methods
 build 172
 didChangeDependencies 172
 dispose 173
 initState 172

State class
 life cycle 172

state
 adding, to app 167, 168, 170, 171

stateful hot reload 36, 37, 38, 39, 40

StatefulWidget
 Futures, using with 269, 270, 271
 references 173

stateless widget 90

stock-keeping unit (SKU)
 about 547
 reference link 547

stopwatch app
 laps, displaying in scrollable list 178, 179, 180, 181, 182, 183

stream controllers
 using 352, 353, 354, 356, 357

stream events
 subscribing to 361, 362, 363, 364, 365, 366

StreamBuilder
 reference link 375
 used, for creating reactive user interfaces 370, 371, 372, 374

streams
 data transform, injecting into 358, 359, 360, 361
 reference link 352

StreamSubscription
 reference link 366

string interpolation 48, 49, 50, 51, 52

strings
 about 48, 49, 50, 51, 52
 declaring 52, 53
 references 53

stylish text
 printing, on screen 108, 109, 111, 112, 113, 114

subscript syntax 68

swiping
 implementing, with Dismissible widget 450, 451, 452

T

TensorFlow Lite
 using 536, 537, 538, 539, 540

TensorFlow
 about 536
 reference link 540

TestFlight
 reference link 576
 using, to publish beta version of iOS app 573, 574, 575, 576

testing, Flutter
 reference link 35

text
 recognizing, from image 515, 517, 518

TextFields
 working with 189, 190, 191, 192, 193, 194, 195

ThemeData object 205, 206

themes
 reference link 115

then() callback
 used, for dealing with errors 267

tier 231

tiered architecture approach
 references 236

translation features, ML Kit
 reference link 536

type safety
 reference link 305

typedef keyword 59

U

Uniform Resource Identifier (URI) 254

Unsplash
 URL 115

V

var
 versus const 42, 43, 44, 45, 46, 47
 versus final 42, 43, 44, 45, 46, 47

variables

declaring 42, 43, 44, 45, 46, 47
views 213

VS Code
about 28, 29
URL 28

W

web service

DELETE action, performing on 342, 343, 344
POST action, performing on 331, 332, 333,
336, 338
PUT action, performing on 338, 339, 340, 342

widget lifecycle, Flutter
reference link 273

widget trees
nesting 144, 145, 146, 147, 148, 150
refactoring 151, 152, 153, 154, 155, 156, 157

widgets

placing 122, 123, 124, 125, 126
working 128, 129

Windows command-line
setting up 12, 13, 14

Windows
app, running on 594, 595, 596
fastlane, installing on 551
writeAsString method 320

X

Xcode
command-line tools 17, 18
downloading 15, 16

Y

YAML Ain't Markup Language (YAML)
URL 35