Google Flutter Mobile Development Quick Start Guide

Get up and running with iOS and Android mobile app development

Prajyot Mainkar and Salvatore Giordano

Packt
www.packt.com

Google Flutter Mobile Development Quick Start Guide

Get up and running with iOS and Android mobile app development

Prajyot Mainkar
Salvatore Giordano

BIRMINGHAM - MUMBAI

Google Flutter Mobile Development Quick Start Guide

Acquisition Editor: Reshma Raman
Content Development Editor: Mohammed Yusuf Imaratwale
Technical Editor: Diksha Wakode
Copy Editor: Safis Editing
Project Coordinator: Kinjal Bari
Proofreader: Safis Editing
Indexer: Manju Arasan
Graphics: Alishon Mendonsa
Production Coordinator: Melwyn D'sa

First published: March 2019

Production reference: 1290319

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78934-496-7

www.packtpub.com

To my mother, Eda Schwartz, and to the memory of my father, Zeev, for their sacrifices and for exemplifying the power of determination. To my wife, Orit, for being my loving partner throughout our joint life-journey

– Stefan Rosca

To my sons, Ben and Yoav for showing me how talent and creativity evolve.
To Tsippi and Shlomo Bobbe for their love, support, and inspiration.

– Den Patin

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 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
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

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.

Contributors

About the authors

Prajyot Mainkar is the director of Androcid, a mobile app development company based in India. The company builds UI/UX and mobile apps for clients. He has been recognized as an Intel Innovator. Prajyot has been an avid programmer and speaker at over 300 mobile developer conferences across the globe. including Android Developer Days in Turkey, Droidcon Greece, Droidcon India, and many more. He is the chairman of the IT & Young entrepreneurship forum at the Goa Chamber of Commerce and Industry. He has been awarded the title of Young Entrepreneur of the Year by Business Goa and the GEMS Trailblazer award for his contributions to the field of information technology. He is on board as an adviser to many incubation centers across India.

To my parents – Shital (Aai) and Prakash (Baba) Mainkar, and brother, Pramay for keeping faith in me always and trusting in my work. Their constant support and inspiration have been the driving fuel all my life. I thank the Almighty for the blessings and the teachers and inspirational minds in my life whose lessons have always helped me to grow.

Salvatore Giordano is a 23 year-old software engineer from Italy. He currently works as a mobile and backend developer in Turin, where he attained a bachelor's degree in computer engineering. He is member of the Google Developer Group of Turin, where he often gives talks regarding his experiences. He is really passionate about cutting-edge technologies, always staying up to date with the latest trends. He has written many articles on Flutter and contributed to the development of a number of plugins and libraries for the framework.

Thanks to everyone on the Packt team, who helped me so much. It wasn't an easy journey, but with the right people, you can achieve anything. Also, thanks to my team at Iakta, who supported me, and my girlfriend, Beatrice, who pushed me to always do better.

A special thanks to my family, who gave me this lucky, happy life.

Thank you very much Mohammed. Let me know if there is anything else I can do.

About the reviewer

Luka Knezić was an Android developer for five years before discovering Flutter. He has been using Flutter since the early alpha release and hasn't returned to Android since. Now, he holds monthly Flutter meetups in Zagreb.

> *I was using Flutter when it was in its infancy, so I have had the opportunity to see how it has changed and have contributed to it by submitting issues, and publishing new packages and plugins. I had learned enough to be able to initiate Flutter meetups in Zagreb, Croatia.*

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

Flutter is a cross-platform application development framework developed by Google. It uses the Dart programming language for its development needs. This book is going to be your guide to getting started on your cross-platform application development journey, by helping you understand the basic concepts of Flutter.

Who this book is for

This book is intended for readers who are interested in learning the basic concepts of Flutter and in learning how to build cross-platform applications.

What this book covers

`Chapter 1`, *Introducting Flutter*, covers a brief introduction to Flutter and how this book is going to serve you as a guide for learning cross-platform application development with Flutter. We will then move on to where and how Flutter originated. Then, before moving on to why Flutter is a good option, we will take a look at where Flutter fits in with the existing world of mobile application development.

`Chapter 2`, *Getting Started with Flutter*, covers the installation of the tools needed to use Flutter and gets readers familiar with IDE, as well as looking at Hot reload, one of the best features in Flutter. We will then learn about two principal concepts that are required in every application development workflow—debugging and testing.

`Chapter 3`, *Widgets, Widgets Everywhere*, goes through the widget catalog and explains how to create custom widgets. We will then learn how to route and navigate through these widgets.

`Chapter 4`, *Exploiting the Widgets Variety*, explores the constraints in Flutter and provides an introduction to animations in Flutter. We will then learn how to use Listview and scrolling widgets and, at the end of the chapter, will be introduced to silvers.

`Chapter 5`, *Widening our Flutter Horizons*, explains how networking plays an important role in the apps, along with sample code for setting up and running a server for fetching JSON code. This section will be followed by an understanding of why accessibility is important and what improvements can developers bring to support accessibility in the app. The following section is about app support internationalization.

`Chapter 6`, *Using a Platform to Power Flutter Apps*, explains how to include packages in the Flutter code, followed by how to include platform-specific channels to support Flutter code. We will then use the BatteryManager API to understand the battery state of the Android phone. We will cover some of the best tips to consider before building our own plugin, followed by how to publish your own plugin on the Flutter Pub site.

`Chapter 7`, *Firebase - Flutter's Best Friend*, examines how Firebase can help us build apps quicker using the Firestore Cloud database. We will also take a look at an example that captured ListView using the Firestore Cloud database. And finally, we will discuss some of the use cases as regards using the remote config for your apps.

`Chapter 8`, *Deploying Flutter Apps*, covers how to deploy and publish the android and iOS app on the respective stores.

To get the most out of this book

Before you start reading, some experience of the Dart language will be beneficial, along with experience in working on Android, iOS, or any mobile development framework. Finally, a familiarity with object-oriented languages, such as Java and C++, and some knowledge of OOPS would be extremely useful.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packt.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Google-Flutter-Mobile-Development-Quick-Start-Guide`. 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!

Conventions used

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

`CodeInText`: Indicates code words in the text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
void main() {
debugPaintSizeEnabled=true;
runApp(MyApp());
}
```

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

```
 Center(
child: Container(
decoration: BoxDecoration(border: Border.all()),
height: 200.0,
width: 200.0,
),
),
```

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

```
$ flutter packages get
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

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.packt.com/submit-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
Introducing Flutter

Flutter is a application development framework from Google for creating cross-platform mobile applications (in iOS and Android). As mentioned on the official website (`https://flutter.io/`), it aims to make the development as easy, quick, and productive as possible. Things such as **Hot Reload**, a vast widget catalog, very good performance, and a solid community contribute to meeting that objective and makes Flutter a pretty good framework.

This book is going to be a guide for you in your journey from getting started with Flutter to eventually deploying your applications on it. But, before that, let's have a quick introduction to Flutter.

In this chapter, we will cover the following:

- The origin of Flutter
- What is a widget?
- Comparing Flutter to existing frameworks

The origin of Flutter

The origin of Flutter was similar to that of a lot of famous software. It was developed at Google. Initially, Flutter was an experiment, as the developers at Google were trying to remove a few compatibility supports from Chrome, to try to make it run smoother. After a few weeks, and after many of the compatibility supports were removed, the developers found that they had something that rendered 20 times faster than Chrome did and saw that it had the potential to be something great.

Google had created a layered framework that communicated directly with the CPU and the GPU in order to allow the developer to customize the applications as much as possible.

What is a widget?

Everything in Flutter can be created using widgets. Orientation, layout, opacity, animation... everything is just a widget. It is the main feature of Flutter, and everything from a simple button to an animation or gesture is done using widgets. And this is great because it allows the users to choose composition over inheritance, making the construction of an app as simple as building a Lego tower. All you do is just pick up widgets and put them together to create an application.

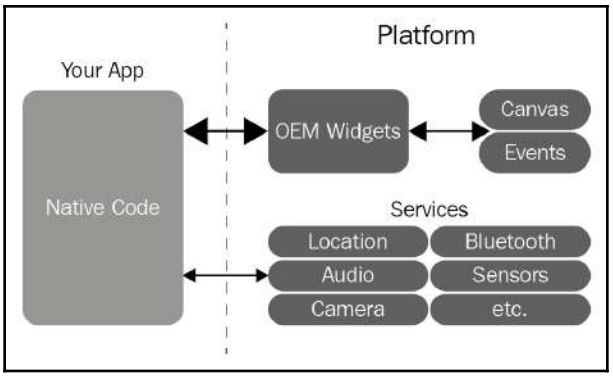
There are a number of fundamental widgets that will help you build an application with Flutter. All these widgets are cataloged in the **Flutter Widget Catalog**. Because everything in Flutter is made up of widgets, the more you learn how to use, create, and compose them, the better and faster you become at using Flutter. We will be going into much more detail about widgets and the widget catalog in `Chapter 3`, *Widgets*, *Widgets Everywhere*.

Comparing Flutter to existing frameworks

When speaking of mobile application development, there are many different approaches that we can find, but, in the end, everything comes down to either a native or a cross-platform approach. Let's see how different approaches look and work when compared to Flutter. We will first take a look at the native platforms, and then, before looking at the cross-platform approach, we will take a look at the **WebView** system, and finally we will see where Flutter fits into this mix.

Native platforms

Native frameworks such as Android and iOS SDKs are rock solid. They are the most stable choice for mobile application development. They have lots of available apps that are deeply tested and have a large community and openly available tutorials. The following diagram displays the working of native mobile application development frameworks:



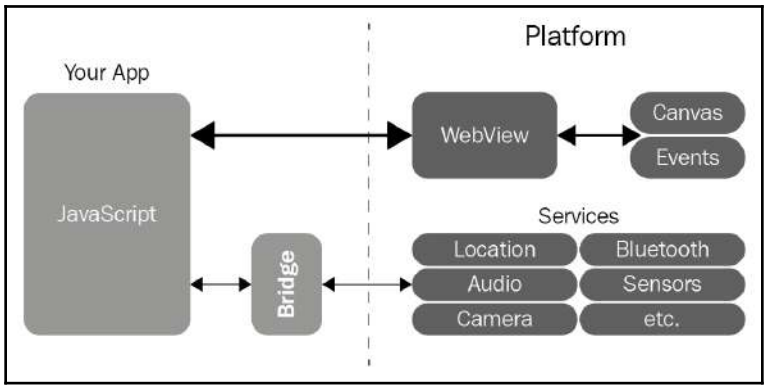
As we can see in the preceding diagram, the **app** in this framework talks directly to the system. This makes the native framework the most powerful choice in terms of functionality. However, it does have a drawback: you need to learn two different languages, Kotlin or Java for Android, Obj-C or Swift for iOS and the SDKs. These languages are used to write two different apps with the same functionalities. Every modification must be duplicated on both platforms, and the process might not be that smooth. It is not a good choice for a small team or for someone who needs speed in their development process.

WebView systems

On the other hand, we have the cross-platform approach, which is famous for being productive. In this approach, we can get the application for both Android and iOS from a single code base, just like in Flutter. But every framework has some drawbacks.

Cordova-, Ionic-, PhoneGap-, and WebView-based frameworks in general are good examples of cross-platform frameworks, and they are especially good solutions for frontend developers. But these lack in performance, and the app view in these approaches is composed by a WebView rendering HTML; this means that the app is basically a website.

The following diagram shows how a WebView-based framework works:



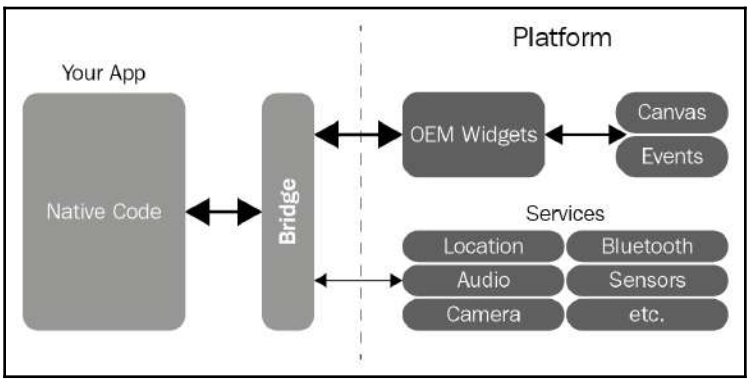
The system uses a bridge to make the switch between JavaScript to the native system. This process will be too slow, depending on the features you need, which adds another drawback to this system.

Other cross-platform approaches

Let's take an example of another cross-platform approach to see what could be the shortcomings of it. **Xamarin** is the Windows answer to cross-platform development, which in my opinion is not so convenient, especially in terms of productivity and compiling time.

When looking at other platforms, **React Native** could be considered as one of the best of the cross-platform frameworks, but it heavily relies on OEM components.

Lets take a look at the workings of React Native:



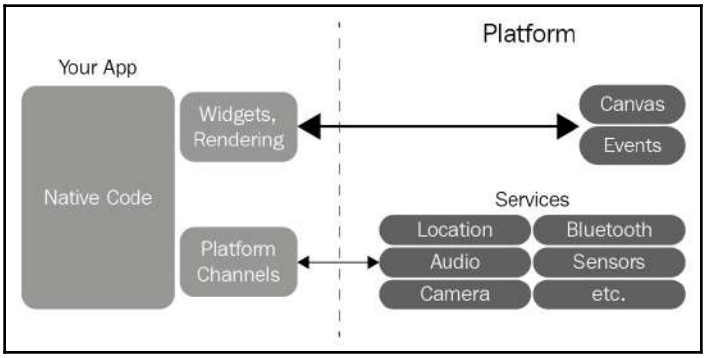
React Native expands the bridge concept in the WebView systems, and uses it not only for services, but also to build widgets. This is really dangerous in terms of performance; for example, a component may be built hundreds of times during an animation, but due to the expanded concept of the bridge, this component may slow down to a great extent. This could also lead to other problems, especially on Android, which is the most fragmented operating system.

Flutter's approach

In the previous sections, we took a look at different approaches to mobile application development. We have briefly seen how these approaches work and their drawbacks. Now let's take a look at Flutter.

Flutter performs much better in comparison to other solutions, because the application is compiled **AOT** (**Ahead Of Time**) instead of **JIT** (**Just In Time**) like the JavaScript solutions. It also eliminates the concept of the bridge and does not rely on the OEM platform. It does allow custom components to use all the pixels in the screen. What does this mean? It basically means that the app displays the same on every version of Android and iOS.

We did take a look at the workings of other approaches, so let's take a look at the workings of Flutter as well. You can see the way the Flutter framework works as shown in the following diagram:



Now you can see the difference between other cross-platform approaches and Flutter. As stated before, Flutter eliminated the bridge and the OEM platform and uses **Widgets Rendering** instead to work with the canvas and events. And it uses **Platform Channels** to use the services. In addition, it is not difficult to use platform APIs with an asynchronous messaging system, which means if you need to use a specific Android or iOS feature, you can do it easily.

Flutter also makes it possible to create plugins using channels that can be used by every new developer. So, to put it simply: code once, and use it everywhere!

Why use Flutter?

Flutter is a good option for cross-platform development due to its many features and a few things that it does differently than other approaches, as we have seen. It is not only a good option for the developers, but also for users and designers; let's take a look at why this is:

- For users, Flutter makes attractive user interfaces for apps, and this enhances the usage of these apps by the users.
- For developers, Flutter makes it easy for the new developers to enter the world of building mobile apps, as it is very easy to build apps with Flutter. Flutter not only reduces the time for development of applications, but it also reduces the cost and complexity of creating an application.
- For designers using Flutter, an application can be created using the original design that was conceived for the application, without compromising on any aspect of it. Therefore, the original vision of the designer is not changed at the time of development.

Most important, Flutter is a very useful tool to create mockups and prototypes, which is a pro, as it is a good point of contact for both designers and developers, two roles often very distant from each other.

Summary

In this chapter, we first had a quick introduction to Flutter and how this book was going to serve as a guide for learning cross-platform application development with Flutter. We then moved on to discussing the origin of Flutter. Then, before moving on to why Flutter is a good option, we took a look at where Flutter fits in with the existing world of mobile application development.

Nowadays, mobile development is not really a new world, but Flutter makes it possible to make it more fun and much quicker. And, by improving the developer workflow, it brings mobile application development closer to a gameplay.

In the next chapter, we will install the Flutter framework and try to learn as much as possible from the sample app.

2
Getting Started with Flutter

Before developing any applications, it is ideal to understand the installation process for that system. In this chapter, we will first look at how to install Flutter on your system and choose the right IDE. We will then move on to exploring a sample app that displays the basic `Hello World` on our screen. Before we look at how to debug and test our application, we will take a quick look at what **Hot Reload** is.

To Develop iOS applications, I would recommend using a Mac. We can always use and test applications only on Android and use macOS when deploying those applications. However, problems are always around the corner, so testing the application on the respective platform during building will be highly recommended.

In this chapter, we will will cover the following topics:

- Installing Flutter
- Choosing a suitable IDE
- Exploring a sample application
- Hot Reload
- Looking at the Flutter tools and how to use them
- Writing and executing tests in Flutter

Installing Flutter

Let's get started with our main application and get Flutter installed onto your system. Depending on the operating system you are using, you can follow the given steps to install Flutter on your system. We will take a look at installing Flutter on Windows, Mac, and Linux.

Installing Flutter on Windows

To install Flutter on Windows, follow these steps:

1. Download Flutter from `https://storage.googleapis.com/flutter_infra/releases/stable/windows/flutter_windows_v1.2.1-stable.zip`.
2. Extract the downloaded file and place it in your desired folder on your system.
3. Locate and run `flutter_console.bat` to start the installation.
4. We will then need to download and set up Node.js; you can download it from `https://nodejs.org/en/download/`.
5. Finally, we will need to download and install Git For Windows 2.x: `https://gitforwindows.org/`.

Installing Flutter on Mac

To install Flutter on Mac, follow these steps:

1. Download Flutter for Mac from `https://storage.googleapis.com/flutter_infra/releases/stable/macos/flutter_macos_v1.2.1-stable.zip`.
2. Extract the downloaded file and place it in your desired folder on your system using the `$ export PATH=`pwd`/flutter/bin:$PATH` command.
3. Run `$ flutter doctor` to verify that everything is set up in the right way.
4. We then need to download and set up Node.js; you can download it from `https://www.npmjs.com/get-npm`.

We will use the following commands: `bash`, `curl`, `git 2.x`, `mkdir`, `rm`, `unzip`, and `which`.

5. Finally, we will need to download and install Git: `https://git-scm.com/download/mac`

Installing Flutter on Linux

To install Flutter on Linux, follow these steps:

1. Download Flutter from `https://storage.googleapis.com/flutter_infra/releases/stable/linux/flutter_linux_v1.2.1-stable.tar.xz`.
2. Extract the downloaded file and place it in your chosen folder on your system using `$ tar xf ~/Downloads/flutter_linux_v1.2.1-stable.tar.xz`.
3. Then, add Flutter to your path using `$ export PATH="$PATH:`pwd`/flutter/bin"`.
4. We will then need to download and set up Node.js; you can download it from `https://git-scm.com/download/linux`.

Here are the commands we will use: `bash`, `curl`, `git 2.x`, `mkdir`, `rm`, `unzip`, `which`, and `xz-utils`.

Getting familiar with IDEs

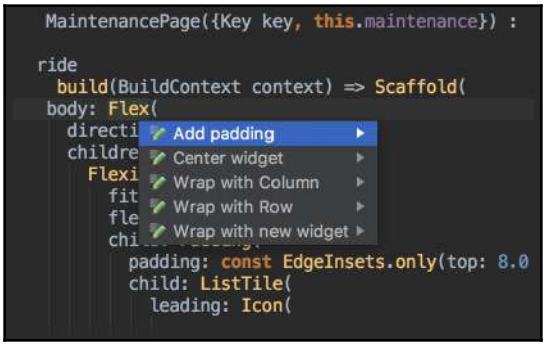
For Flutter, it's best to use Android Studio/IntelliJ or **Visual Studio** (**VS**) code with Mac/Windows as your operating systems. These IDEs are the best you can find for developing mobile applications. But to use these with Flutter, we will need to use a few plugins.

We will need a plugin for the Dart compiler, another for code analysis, and another for the Flutter developer workflow (building, running, and debugging).

These plugins can be installed on both Android studio and VS code. All you need to do is search for them in the corresponding plugin sections. The IDEs not only provide the option of these fantastic plugins to support your development. Let's take a look at some quick tips you can use when developing your application.

Some quick tips for using your desired IDE

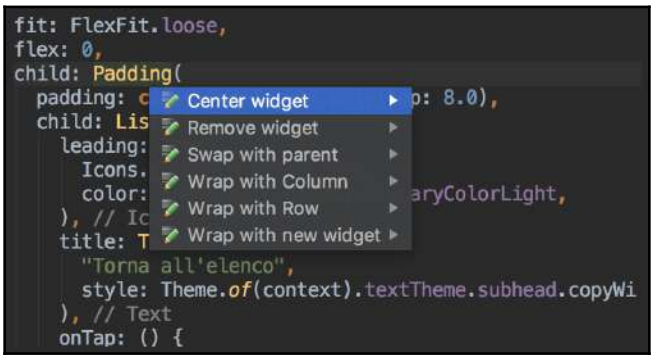
When using the Flutter plugin, there is a very good option that can be used while developing your application; it is called the **quickfix** option. To use this, press *Alt + Enter* (*Ctrl + .* on VS code) and a popup will be displayed with some quickfixes. Lets take a look at how it looks on screen. The following screenshot shows what it looks like:



In the preceding screenshot, the quickfix gives an option to add a padding: a center widget, wrapping it using a column or a row, and wrapping it with a new widget.

This is a very useful option as it will help you save a lot of time during the development of the application, considering you will be nesting several widgets. While you do this, keeping the code clean is not an easy task.

Another great thing that can be done using the quickfix option is that you can order the children in a column to swap a widget with their parents or remove a widget completely but quickly. The following screenshot shows these options:



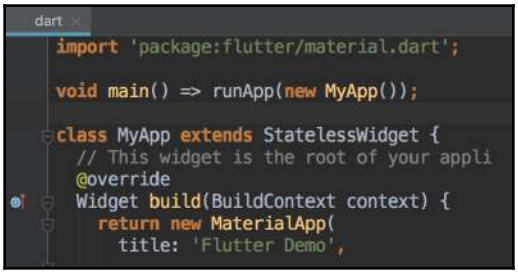
Speaking of nesting, a very useful option from the plugin is the presence of some fake comments at the end of each widget. This helps you understand the tree of the widgets you are composing at a single glance. The following screenshot shows what those fake comments look like:

These few tips may not seem very useful at first, but once you start developing applications with Flutter, they will be essential and will help you work more quickly.

Exploring a sample app

Let's take a look at how the code in Flutter looks and explore its elements. First, let's create a new project; this way, Flutter-cli will create a sample app for us to explore. Before we start looking at the code, here is the GitHub repository dedicated to this book: `https://github.com/PacktPublishing/Flutter-Quick-Start-Guide/tree/master/sample_app`.

The following screenshot shows how the Flutter code looks; let's explore its elements:



As you can see, the entry point of the application is the `main` function in which you can see the call to the `runApp`. This is the first line that is executed; its task is to set up the Flutter framework and run the selected application. When we set up the application, initially, it is a normal stateless widget.

Next, we come to the `Build` method. It is displayed in the previous screenshot as `Widget build(BuildContext context)`. The `Build` method is the one that returns the `MaterialAPP`, sets the title, and sets a general theme. In addition to this, the `Build` method also sets the routing of an application and the home screen.

Moving on, let's take a look at the following screenshot:

```
44
45  class _MyHomePageState extends State<MyHomePage> {
46    int _counter = 0;
47
48    void _incrementCounter() {
49      setState(() {
50        // This call to setState tells the Flutter frame
51        // changed in this State, which causes it to
52        // so that the display can reflect the updated
53        // _counter without calling setState(), then the
54        // called again, and so nothing would appear to
55        _counter++;
56      });
57    }
```

The sample application that we are working on is composed of a Scaffold with a counter that is incremented with the pressure of a **Floating Action Button** (**FAB**). As we can see in the preceding screenshot, there is no `setText` here. The counter is described by just one variable that is updated by the handler of the `onPressed` action of the FAB.

Let's move on and look at the most important line in the code: line 49 in the previous screenshot. In Flutter, you use the `setState()` method to update the UI and sync it with the underlying variables. In this case, we are incrementing the `_counter` variable, and, at the same time, we also want the application to render the text showing the number. These are a few of the elements you will be able to use when you create a sample project in Flutter. We'll take a look at one of the best features of Flutter in the next section: Hot Reload.

Hot Reload

Before you begin the actual development of your application in Flutter, it is good to know what features of Flutter you can use to make life easier. Hot Reload is one such feature; it will make development much easier. How is it going to do that? Let's take a look.

To understand how Hot Reload is a blessing, let's consider a normal development flow, where you are building a tab for setting up a page of your application. When you navigate to your tab, you find out that a certain text is too small. Usually, you would have to go back and change the font size in the code and then navigate back to the point and check whether the size is now correct. If not, you do the same thing again: you go back to change the font size in the code, come back to the point, and check whether it is now correct. You will have to do this again and again until you figure out the right font size.

This is very time-consuming and frustrating, right? But in Flutter, we won't have to do this, because we have the Hot Reload feature. In this particular situation, all you will have to do is edit the font size and press ⌘+S. Once you do this, your app will show the updated version of the code! We will not have to recompile or navigate to that specific screen again and again.

How is that even possible? Hot Reload uses the JIT compiling feature of Dart. The edited code is injected into the application running in debug mode in a matter of milliseconds, keeping the state in its memory.

This is an amazing feature for developers, as it changes the way the development workflow works. You will have the opportunity here to write the code in a different way, and this helps you make more modifications in your UI code, without being afraid of rigorous work.

Debugging an application

Debugging an application is one of the most important things to learn about when learning to develop any application. Debugging will help you identify and work on errors in your code. Errors are always around the corner, and knowing how to deal with them is essential. To understand debugging in Flutter, we will have to understand these three concepts:

- **Dart analyzer**
- **Dart observatory**
- **Visual debugging**

We will see in detail what they are and how they help with debugging in detail in the following sections.

Dart analyzer

Dart analyzer checks your Dart code for errors. It is essentially a linter of Dart, a simple wrapper around the `dartanalyzer` tool. Dart analyzer is also included in the Flutter plugin for Android Studio and VS code, so you won't have to worry about including it separately in your IDE.

We can also create a file named `analysis_options.yaml` and specify some additional options that will raise errors/warnings and will help you write better Flutter code.

Dart observatory

The dart observatory is a tool dedicated to debugging and analyzing Flutter apps. To put a breakpoint and run the app step by step, you can use the help of an IDE. An alternative is the `debugger()` statement. This line will break the execution in the point where you put it. It's also possible to specify a condition, and the app will stop only if the condition is true:

```
void function(int aNumber) {
 debugger(when: aNumber < 10);
 // ...
 }
```

When you are running a Flutter application, you will see a line in the console, specifying the observatory URL. The line will look as follows:

```
Observatory listening on http://127.0.0.1:8100/
```

You can perform a number of things by navigating to this URL. You can open the obeservatory, use it to profile the app, examine the heap, allocate memory, and so on. This is a really powerful tool; you can find more information on this at `https://dart-lang.github.io/observatory/`.

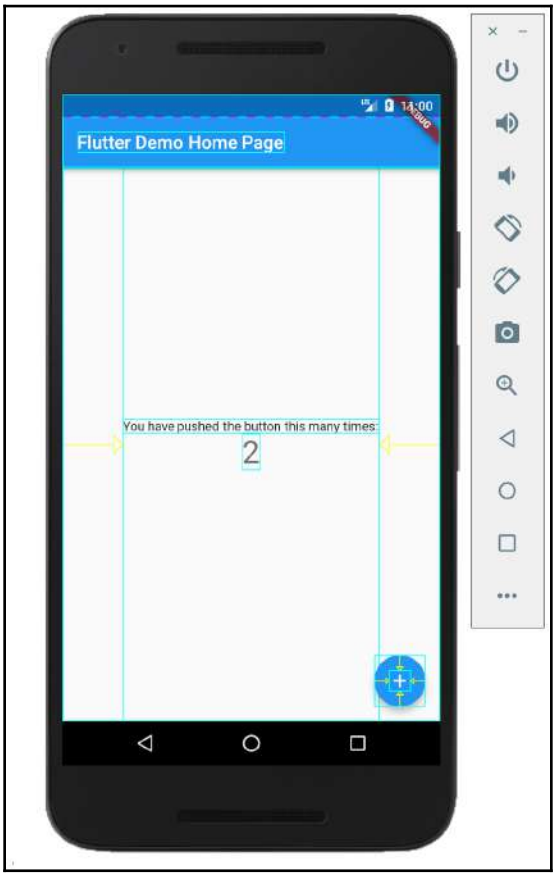
Visual debugging

There are going to be cases where we will need to debug the layout of our application. We might need to align some widgets in a particular way, or sometimes we might not know whether the space between widgets is a margin or padding. In such instances, we will need to visually debug our application. To debug in such instances, enable the `debugPaintSize` option.

To do so, set the `debugPaintSizeEnabled` variable to `true` as follows:

```
void main() {
 debugPaintSizeEnabled=true;
 runApp(MyApp());
 }
```

The following output will be displayed:



As you can see in the previous screenshot, every widget gets colored in and can be easily distinguished now.

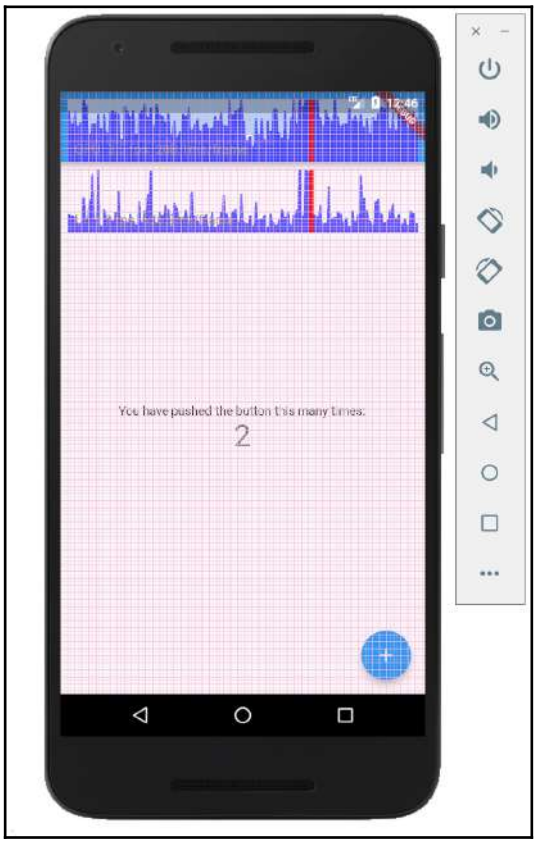
This is a very powerful feature and can help you with visual debugging, especially if you are not that "pixel perfect" frontend developer.

Material grid variable

Let's take a look at another visual debugging variable: it's called the **material grid**. Here, you will declare your MaterialApp by setting `debugShowMaterialGrid` to `true`! Your application will be overlayed by the material pixel grid—which is perfect to study the app layout. The following is how your application would then look:

The showPerformanceOverlay variable

The next useful option is `showPerformanceOverlay`. By setting it to `true`, you will see the performance of your application displayed in the form of a graph on the upper part of the graph. There will be two graphs displayed on your screen, as shown in the following screenshot:

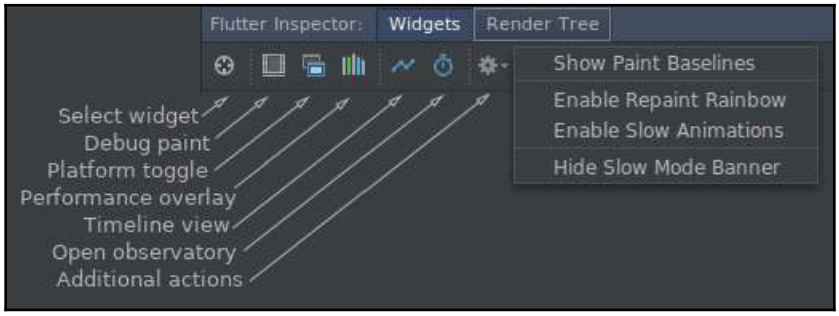


The upper graph shows the time spent by the GPU thread, and the bottom one shows the time spent by the CPU thread. They will also display whether the app is running at less than 60Hz; in this case, you might have some performance issues. This feature will help you understand the performance of your application and to verify whether it is running as expected.

Be sure to use this feature only in **release** mode. In debug mode, the performance is intentionally reduced to have Hot Reload available and raise more warnings.

Flutter widget inspector

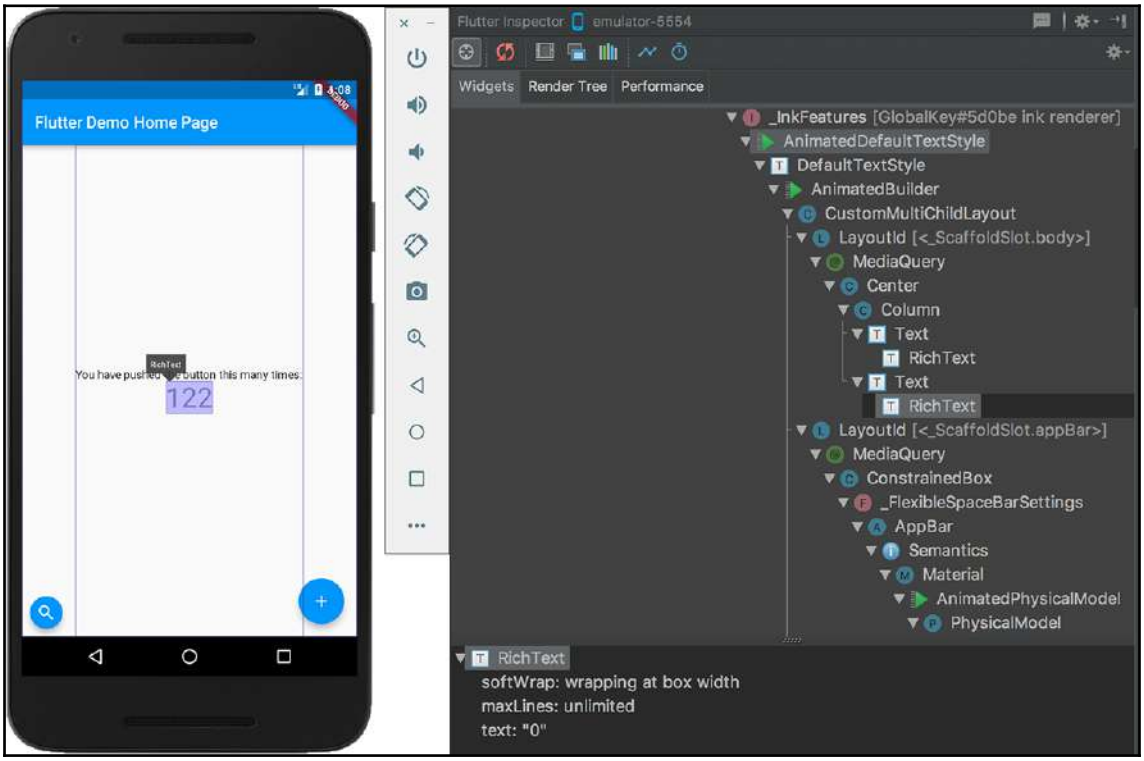
If you are a web developer, you could easily miss the **inspect** option in many browsers. Flutter brings it back to you in the form of a Flutter widget inspector. It is yet another feature that will help you visually debug your application. Let's take a look at a screenshot that displays it:



This is the Flutter widget inspector that we can find in our Android studio. There are many options that this feature presents to us; some of them are shortcuts to the features we mentioned in the visual debugging section. To trigger the inspector, perform the following steps:

1. Click on the **Select widget** option.

2. Then, click on a widget on your device. The widget you click on will be selected and highlighted on the widget tree, as follows:



Once it is triggered and you can see the widget tree, you can take a look at the widget composition and understand whether there is anything wrong in the layout.

We took a look at debugging and also visually debugging our application. Debugging is a good way to find out whether there are any errors in your application. Another good way to find any anomalies or issues in the working of your application is by testing your application. We'll take a look at testing your Flutter application in the next section.

Testing a Flutter application

As your app gets bigger and bigger, a good set of tests may help you save time, as tests can find new bugs that could appear with normal modifications. Even performing **Test Driven Development** (**TDD**) is a good idea, as it can help you define a structure of your project and write less but more efficient code.

In Flutter, there are mainly three kinds of automated testing:

- **Unit testing**
- **Widget testing**
- **Integration testing**

Let's take a look at them in detail.

Unit testing

As the name suggests, a unit test is a type of testing that is used to test a single unit of code. This small unit could be a function, a method, or a class. Generally, in unit testing, we won't need to write on a disk, render to a screen, or receive external input. Unit tests must be as small as possible, so remove any possible external dependencies.

These tests are low maintenance and low in cost, and are very quick in terms of the time they take to execute. The only drawback of unit testing is that you can never completely reply on it, as it does not test the system as a whole. For this reason, there are other kinds of testing that should be used. Let's take a look at how to perform this type of testing:

1. Import `pubspec.yaml` into your testing framework, as follows:

```
dev_dependencies:
     flutter_test:
       sdk: flutter
```

2. Write the test code in `test/unit_test.dart`:

```
import 'package:test/test.dart';
void main() {
   test('the answer to the question', () {
     var answer = 42;
     expect(answer, 42);
   });
 }
```

3. Run the test by running `flutter test test/unit_test.dart` in the project folder. Alternatively, you can run `flutter test` to run all the tests.

Unit tests are run in a local Dart VM with a headless version of the Flutter engine.This makes the process faster because it doesn't need to boot a real Flutter engine or compile a real application.

Widget testing

Widget testing is also known as **component testing**. As its name suggests, it is used for testing a single widget, and the goal of this test is to verify whether the widget works and looks as expected.

In addition to this, you can use the `WidgetTester` utility for multiple things while testing, such as sending input to a widget, finding a component in the widget tree, verifying values, and so on.

Let's take a look at how a widget test looks in code:

```
import 'package:flutter/material.dart';
 import 'package:flutter_test/flutter_test.dart';
void main() {
   testWidgets('my first widget test', (WidgetTester tester) async {
     // You can use keys to locate the widget you need to test
     var sliderKey = UniqueKey();
     var value = 0.0;
    // Tells the tester to build a UI based on the widget tree passed to it
     await tester.pumpWidget(
     StatefulBuilder(
       builder: (BuildContext context, StateSetter setState) {
         return MaterialApp(
           home: Material(
             child: Center(
               child: Slider(
                 key: sliderKey,
                 value: value,
                 onChanged: (double newValue) {
                   setState(() {
                     value = newValue;
                       });
                   },
                 ),
               ),
             ),
           );
```

```
            },
          ),
        );
      expect(value, equals(0.0));
     // Taps on the widget found by key
      await tester.tap(find.byKey(sliderKey));
     // Verifies that the widget updated the value correctly
      expect(value, equals(0.5));
    });
  }
```

While testing, if you need to see the UI, you can always use the `debugDumpApp()` function or run the test using `flutter tun test/widget_test.dart`. In this way, you will also be able to interact with the widgets during testing.

Integration testing

Now, let's take a look at integration testing. This type of testing is used for testing the whole application or a big part of the application. Integration testing can be used to verify that the app does everything as expected or to test the performance of the code. Integration tests are run on a real device or an emulator, but they can't be run with a headless version of Dart VM like as it can in widget testing.

Now, let's get started with writing and running the tests:

1. Add the `flutter_driver` package to `pubspec`:

```
dev_dependencies:
     flutter_driver:
       sdk: flutter
```

2. Enable the Flutter driver extension and add a call to the `enableFlutterDriverExtension()` function in `main.dart`.
3. Run the integration test by using the `flutter drive` command:

```
flutter drive --target=my_app/test_driver/my_test.dart
```

Summary

In this chapter, we have installed the tools to use Flutter; we then became familiar with IDE for our use and looked at Hot Reload, one of the best features in Flutter. We then learned about two essential concepts that are required in every application development workflow, that is, debugging and testing.

These concepts are going to help us get started with Flutter and start building our applications with it.

In the next chapter we'll dive into the widget world and learn the different kinds of widgets that the widget catalog holds for us.

3
Widgets, Widgets Everywhere

In Flutter, the concept of widgets is very important. As stated in `Chapter 1`, *Introducing Flutter*, everything in Flutter is a widget. You might have a lot of questions about widgets, such as "What are the basic types of widgets?", "How do I create one?", "What are some good example of widgets?", and so on.

In this chapter, we will explore these questions together. We will first take a look at the widgets catalog and understand the fundamental widgets that will help you build apps with Flutter. We will also learn how to create custom widgets and then take a look at the concept of routing and navigating in a Flutter app. All of these topics will be covered in the following sections:

- Widgets Catalog
- Creating widgets
- Routing and navigation

Widgets Catalog

The Flutter team built this very good website called the **Widgets Catalog** (`https://flutter.io/widgets/`) where you can explore the variety of components that already exist in Flutter, divided by category. You will be using a lot of these widgets in your applications, so the more you know about them, the more efficiently you can use them in your application.

However, there are a few fundamental widgets listed that will help you get familiar with the types of widgets you will find in the Widgets Catalog. The following is a list of those widgets:

- `Container`
- `Image`
- `Text`
- `Icon`
- `RaisedButton`
- `Scaffold`
- `Appbar`
- `PlaceHolder`
- `Row`
- `Column`
- `ListView`

Let's explore these widgets one by one in detail.

Container

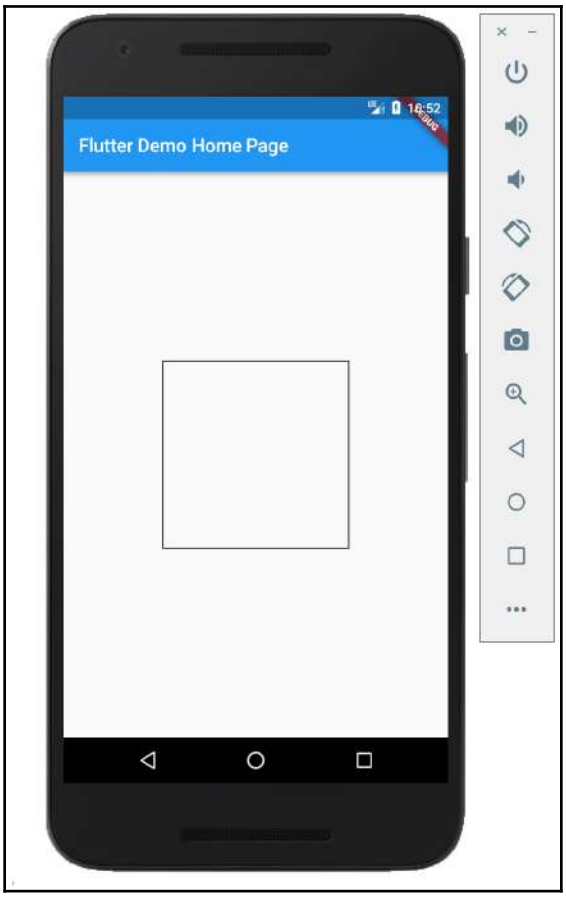
This is one of the complex widgets in the catalog. It is used to contain a child widget within your parent widget, which it does by applying some styling properties on it.

A container makes it possible to apply a variety of features, for example, background color, aligning the child within the container, setting some constraints to the size of the child, and applying some decoration or transformation property to the child (for example, you can rotate a widget). When we look at the amount of things you can do with this widget, it can be considered a complex widget. But in most cases, we will need only a couple of its features.

Now, let's take a look at the code to display the widget. The code will look as follows:

```
Center(
 child: Container(
   decoration: BoxDecoration(border: Border.all()),
    height: 200.0,
    width: 200.0,
  ),
),
```

The following output will be displayed:



Sometimes, you will need to show a widget based on a conditional expression—for example, in this case:

```
function getIcon(bool condition) {
    if (condition == true) return Icon(Icons.edit);
    else return Container();
 }
```

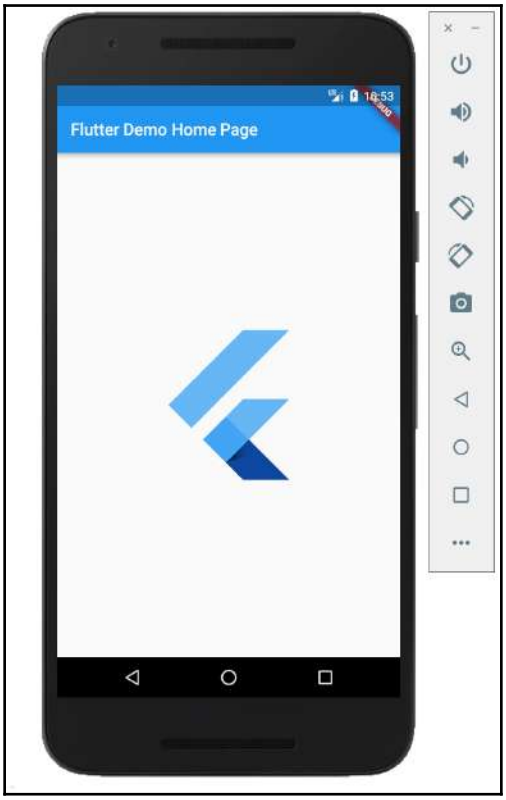
The preceding code shows the conditional expression for a container. It works like most of the conditional expressions, where if the condition is true, you will get your regular widget. But if the condition is false, you will get something called a **null** widget.

Image

Displaying images on your application is one feature that your app must have. There are hardly any apps today that lack the functionality to display an image. And, to do this, the image widget comes into the picture. We can use the following code to use an image widget:

```
Center(
  child: Container(
    height: 200.0,
    width: 200.0,
    child:
Image.network("https://flutter.io/images/flutter-mark-square-100.png"),
  ),
),
```

The following output will be displayed when you use the preceding code:



You can explore the several constructors it has to offer, but I suggest that you try and use them depending on the source you want to use. For example, if you have an `imageProvider`, you will use the default constructor, but if you have the image in an `AssetBundle`, you should use the `Image.asset` constructor.

This is an image-displaying widget, and images come in a few different formats. Here's the list of image formats supported by the image widget:

- `JPEG`
- `PNG`
- `GIF`
- `Animated GIF`
- `WebP`
- `Animated WebP`
- `BMP`
- `WBMP`

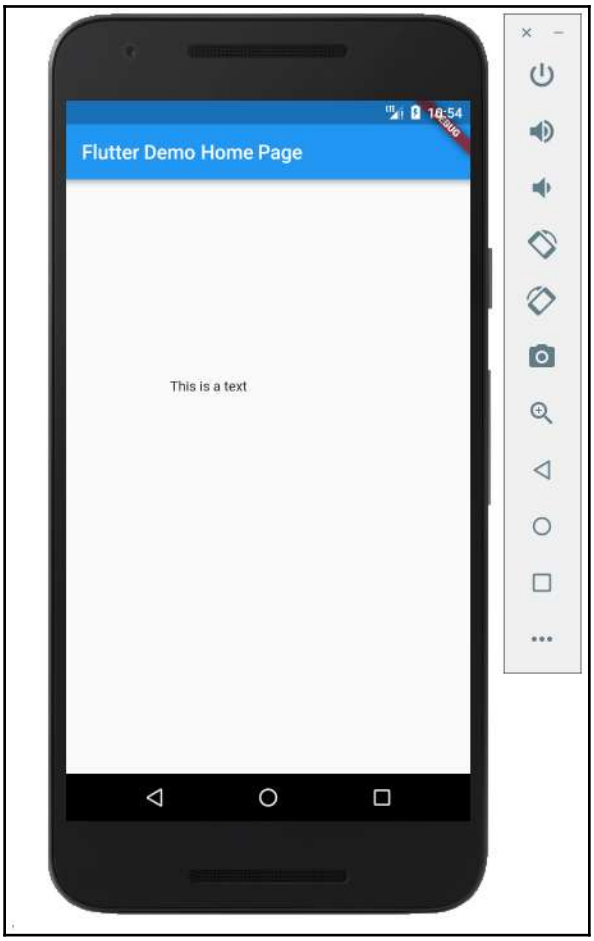
Text

This widget is as self-explanatory as the last one. It is used for displaying text on the screen with a single style. We can also display the text on a single line or multiple lines; this depends on the layout constraints. The style argument when using this widget is optional. If the style argument is not provided, the widget will use the style from enclosing `DefaultTextStyle`, and if the provided style's `TextStyle.inherit` property is true, the given style will be merged with the default one.

The following code can be used for using a text widget:

```
Center(
  child: Container(
    height: 200.0,
    width: 200.0,
    child: Text("This is a text"),
  ),
),
```

The following screenshot will display how the widget is displayed on the screen:



There may be instances when you want to do more with this text widget. For example, to apply more than one style (to display some bold words in a line) to text, you can use the `TextSpan.rich` constructor, or to add interactivity to the text, you can use use a `GestureDetector`.

I would suggest using `FlatButton`, instead of a text widget for interactivity.

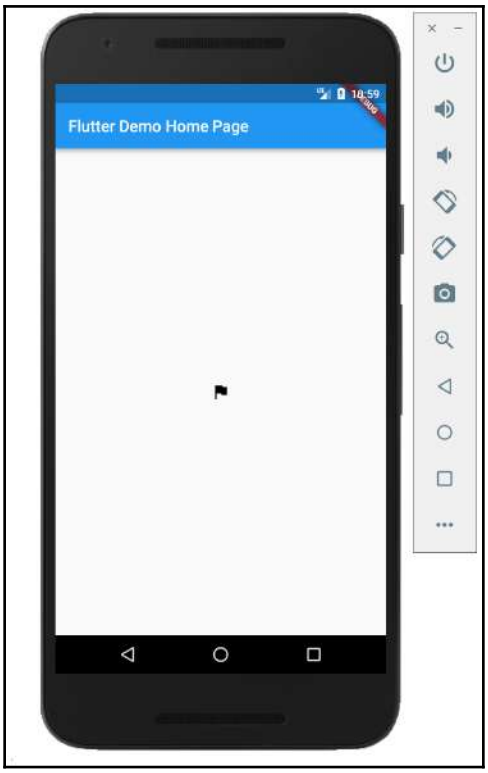
Icon

The icon widget is used to draw an icon using the font described in `IconData`, such as a material's predefined `IconData` in the `Icon` class.

The following code can be used to use the `Icon` widget:

```
Center(
  child: Container(
    height: 200.0,
    width: 200.0,
    child: Icon(Icons.flag),
  ),
),
```

The following is a screenshot displaying how the `Icon` widget looks on the screen:



Just like the text widget, we can add interactivity with the `Icon` widget too. To do that, we can use `GestureDetector`.

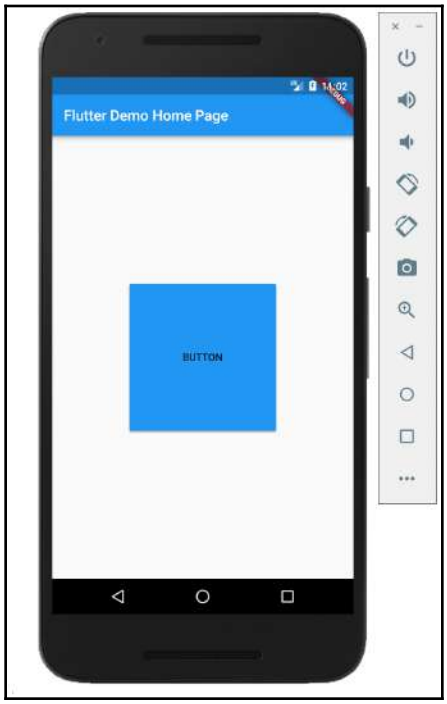
RaisedButton

This widget is used to display a simple elevated button. The button is elevated because the button is based on a material widget whose elevation increases when the button is pressed. If the `onPressed` callback is null, then the button will be disabled, and it will resemble a flat button in `disabledColor`.

The following code can be used to use the `RaisedButton` widget:

```
Center(
  child: Container(
    height: 200.0,
    width: 200.0,
    child: RaisedButton(
      onPressed: () => print("on pressed"),
      child: Text("BUTTON"),
      color: Colors.blue,
    ),
  ),
),
```

The following screenshot will be displayed when you use the preceding code:



The suggestion is to use `RaisedButton` to add dimension in otherwise mostly flat layouts. I would recommend not using such a button in a dialog or a card.

Scaffold

Scaffold is a basic layout structure based on material design. In practice, if you use material design, every screen of your app will have a `Scaffold` as its base. The `Scaffold` widget is used for showing drawers, snackbars, bottomsheets, floating-action buttons, and so on, by offering APIs. To display a snackbar or a bottomsheet, you must use `Scaffoldstate` for the current context. We can use it via `Scaffold.of` and use the `ScaffoldState.showSnackbar` function.

The following code can be used to display a `snackbar` using `Scaffold`:

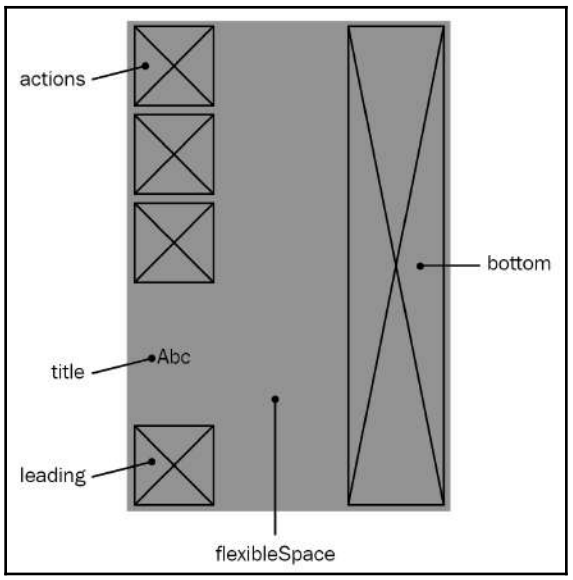
```
Center(
      child: Container(
        height: 200.0,
        width: 200.0,
        child: RaisedButton(
          onPressed: () {
            Scaffold.of(context).showSnackBar(SnackBar(
                  content: Text("HELLO!"),
                ));
          },
          child: Text("BUTTON"),
          color: Colors.blue,
        ),
      ),
    ),
```

The following is the output that will be displayed using the preceding code:

AppBar

`AppBar` is basically used as a property of `Scaffold`, and the majority of `Scaffold`s have app bars. The app bar consists of a toolbar and potentially other widgets. For example, it can host `TabBar`, `FlexibleSpaceBar`, or some actions optionally followed by `PopupMenuButton` for less common operations.

The property that's used for `AppBar` is `Scaffold.appBar`. It looks as follows:



The preceding diagram displays where each widget will be placed by the `appBar` component.

If the leading widget is omitted and `Scaffold` has a drawer, then `appBar` will place a button to open the drawer. If the nearest navigator has any previous routes, a `BackButton` will be inserted.

PlaceHolder

`PlaceHolder` is another widget that explains itself through its name. The `PlaceHolder` widget is used for holding a place for a widget. It draws a box that represents where other widgets will be added later.

The following code can be used for a `PlaceHolder` widget:

```
Center(
     child: Container(
       height: 200.0,
       width: 200.0,
       child: Placeholder(),
     ),
   ),
```

The preceding code will display the following output:

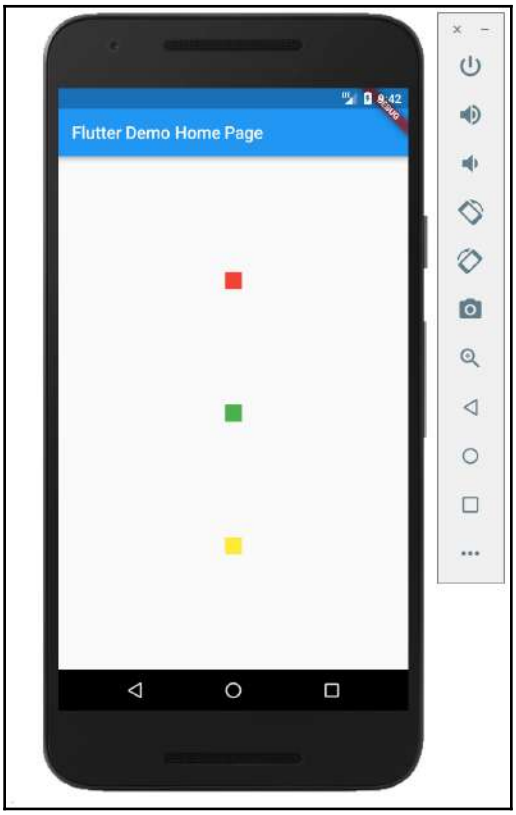
Column

`Column` is essential for composing layout in Flutter apps. It displays its children in a vertical array. The following code can be used for the `Column` widget:

```
Center(
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.center,
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
        children: <Widget>[
          Container(
            height: 20.0,
            width: 20.0,
            color: Colors.red,
          ),
          Container(
            height: 20.0,
```

```
            width: 20.0,
            color: Colors.green,
          ),
          Container(
            height: 20.0,
            width: 20.0,
            color: Colors.yellow,
          ),
        ],
      ),
    ),
```

The following output will be displayed:



The `Column` widget, however, does not support scrolling; for that, we can use `ListView`.

Note that it will be considered as an error by the system if you have more children in a column that will fit into the available room. That's because the column doesn't have the ability to recycle the layout.

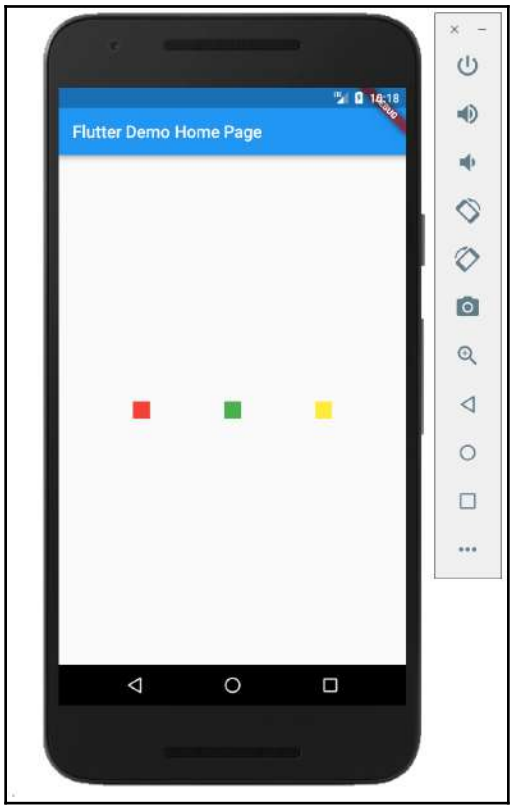
Row

The `Row` widget is similar to the `Column` widget, but still different. We can say that it is the horizontal version of `column`. It draws the children in a horizontal array.

The following code can be used for a `Row` widget:

```
enter(
      child: Row(
        crossAxisAlignment: CrossAxisAlignment.center,
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
        children: <Widget>[
Container(
            height: 20.0,
            width: 20.0,
            color: Colors.red,
          ),
          Container(
            height: 20.0,
            width: 20.0,
            color: Colors.green,
          ),
          Container(
            height: 20.0,
            width: 20.0,
            color: Colors.yellow,
          ),
        ],
      ),
    ),
```

The following output will be displayed:



The story in regard to scrolling remains the same as for the `Column` widget. It is recommended to use `ListView` if you want to scroll the children.

ListView

`ListView` behaves similar to a column or a row; the only difference is that its children can be scrolled.

There are three constructors for the `ListView` widget:

- The default takes a list of widgets in its children property. This is a good choice for small lists because to build it, the list will process every child.
- `ListView.builder` takes an indexed builder to build the children on demand. This is the choice to pick if you have a large number of children, because every time the list processes only the visible children.
- `ListView.custom` takes `SliverChildDelegate`, which provides the ability to customize more aspects of `ListView`.

A note about Row, Column, and ListView

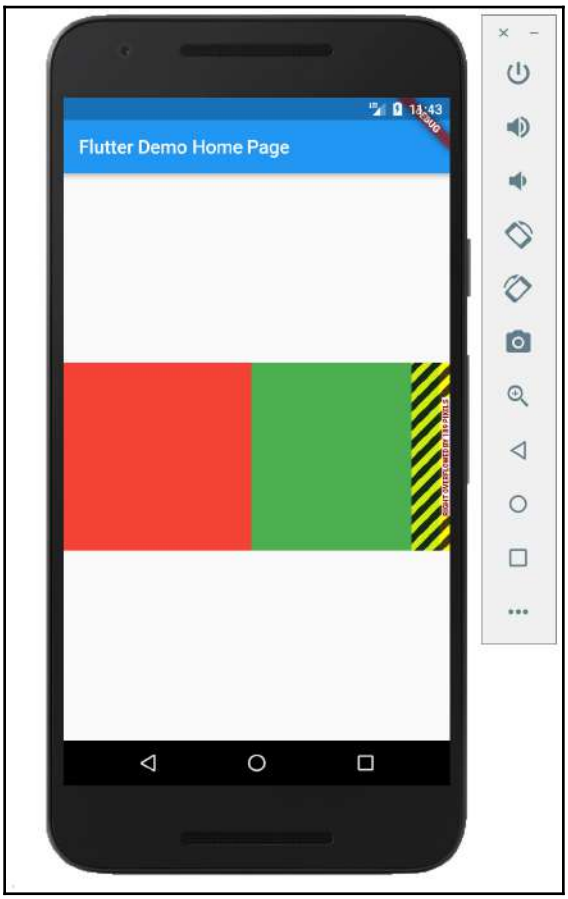
Sometimes, it can happen that you get a runtime exception at the time of building a row or a column that's been placed in another row/column or in any scenario that does not provide a maximum height constraint.

The problem is that the inner widget should fill all the remaining space, but the outer widget has no specific size and should fill the available space too. So, they can't understand where to stop, and then an exception is thrown.

To solve such a problem, you must understand why the inner column/row is receiving unbounded constraints. Consider the following:

- If the column/row is placed in another column/row, you can try to wrap the inner widget in an expanded widget, indicating that it should take the remaining space of the outer widget and not all the space it desires
- If the widget is placed in a `Listview` and is wrapped in an expanded or flexible, then that key is to remove that wrapping widget and to set the size of the inner widget manually

Another problem you may have to encounter the yellow-and-black-striped banner as shown in following screenshot:



This banner indicates that a row or column overflows its size. The solution is to use `ListView` and let the content scroll, or just to reduce the size of the children.

Creating widgets

We saw a number of widgets in the previous section, but there might be a possibility that you don't find the right ready-to-use widget that you want or that you want to combine more widgets in order to create a reusable group. Therefore, you have to create a custom widget.

There are two types of widget in Flutter that you can use to create your own custom widgets:

- Stateless widgets
- Stateful widgets

Let's take a look at them in a bit more detail.

Stateless widgets

Stateless widgets remain the same even if the user interacts with them. This kind of widget has no state, so they can't change according to an internal state. They can only react to higher widget changes.

To build a stateless widget, we will extend the `StatelessWidget` abstract class, as follows:

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

Stateful widgets

Stateful widgets are dynamic components that have an internal state to manage. A stateful widget can react to state changes and change accordingly. The state is stored in a `State` object. To create a `StatefulWidget`, you have to extend the `StatefulWidget` abstract class, as shown in the following code:

```
class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;

  @override
  _MyHomePageState createState() => new _MyHomePageState();
}
```

The state will be a class extending the `State<T extends StatefulWidget>` abstract class. Let's take a look at example where the widget changes the background color according to its state. The code for this is as follows:

```
class _MyHomePageState extends State<MyHomePage> {
  bool value = false;

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      backgroundColor: value ? Colors.black : Colors.white,
      appBar: new AppBar(
        title: new Text(widget.title),
      ),
      body: Center(
        child: Switch(
            value: value,
            onChanged: (v) {
              setState(() {
                value = v;
              });
            }),
      ),
    );
  }
}
```

To trigger the framework to rebuild the widget and apply the changes, you have to call the `setState()` function, or it won't see any changes.

Routing and navigation

We've just looked at how to use widgets, but you won't be using just one widget. In a typical application, it's normal to find more than one screen. When an application has more than one screen, it is essential for the users to have a clear route to move through those pages, and to do so routing and navigating through the pages becomes very important for your application.

To do this, if you are from an Android background, you would use more activities or fragments, and in iOS, you would create a new `viewControllers`.

In the Flutter world, new screens are widgets! To navigate to a new route, we can use the `Navigator.push()` function, passing as an argument the current context and a new `MaterialPageRoute`:

```
Within the `FirstScreen` Widget
 onPressed: () {
   Navigator.push(
     context,
     MaterialPageRoute(builder: (context) => SecondScreen()),
   );
 }
```

The `SecondScreen` will be a normal widget that builds the screen. For example:

```
class SecondScreen extends StatelessWidget {
   @override
   Widget build(BuildContext context) {
     return Scaffold(
       appBar: AppBar(
         title: Text("Second Screen"),
       ),
     );
   }
 }
```

To navigate back, we will use another function of the navigator: `Navigator.pop()`. This function will remove the current route from the stack of routes that are managed by the navigator. We can also use this function to return a value to the users when moving through the screen. Let's take a look at this in detail in the next section.

Returning a value when navigating

Returning a value to the readers when moving from one screen to another screen can improve the user experience of your application. For example, just a simple `welcome` on the screen when opening an app, will increase the user experience. For this purpose, in Flutter, we have `Navigator.pop()`.

`Navigator.pop()` takes the current context as an argument, but it has an optional dynamic argument. This means that you can return any value when popping a screen.

Taking a look at the return value of `Navigator.push()`, you can see that it returns a `Future<dynamic>`. So, when pushing a new screen, you can wait for the popped return value. For example:

```
function getConfirmation(BuildContext context) async {
  return await Navigator.push(context, MaterialPageRoute(
    builder: (context) => ConfirmationScreen(),
  ) ?? false;
}]
```

The `ConfirmationScreen` will be as shown:

```
class ConfirmationScreen extends StatelessWidget {
  Widget build(BuildContext context) => Scaffold(
    body: ButtonBar(
      children: <Widget>[
        RaisedButton(
          child: Text("OK"),
          onPressed: () => Navigator.pop(context, true),
        ),
        RaisedButton(
          child: Text("CANCEL"),
          onPressed: () => Navigator.pop(context, false),
        ),
      ],
    ),
  );
}
```

Summary

In this chapter, we went through the widget catalog; this catalog consists of a number basic widgets that we can start using instantly in our applications without building our own widgets. It is good to understand these basic widgets, as you will be using them in your application a lot. But there will be times when you will need a customized widget, to help you with that we went through stateless and stateful widgets, that will help you customize your widgets. And, finally, we learned how to navigate and route through those widgets.

What's next? There are more and more widgets you can use to build your apps. In the next chapter, we'll see some of them that can be used to build beautiful layouts.

4
Exploiting the Widgets Variety

In this chapter, we will first take a look at constraints in Flutter and understand how it will help in your application development. We will then have a quick introduction to animations and its categories, and take a look at common patterns in it. Then, we will move on to using `ListView` and scrolling widgets, and, finally, have a quick section about silvers. All these topics will be covered in the following sections:

- Constraints in Flutter
- Introducing animations in Flutter
- Using `ListView` and scrolling widgets
- Introducing silvers

Constraints in Flutter

Every widget in Flutter is rendered by a `RenderBox` object that takes the constraints given by the parent and sizes itself within those constraints.

The difference between constraint and size is that the former gives a minimum and maximum of height and width, while the latter consists of a specific height and width.

There are three kinds of RenderBoxes, distinguished by their behavior as follows:

- Those that try to be as big as possible (`ListView`, Center and so on)
- Those that try to be the same size as their children
- Those that try to be a particular size (image, text and so on)

As in every rule, we need exceptions.

Some widgets vary their behavior depending on their constructor arguments. For example, the Container widget tends to be as big as possible, but, if you give it a width (or height), it tries to be that particular size.

A particular constraint is the unbounded (or infinite) one. In this case, either the maximum width or height is set to `double.INFINITY`.

A box that tries to be as big as possible won't work with unbounded constraint, and the framework will throw an exception.
This can happen within flex boxes (row/column) and scrollable regions (`ListView` and other `ScrollView` subclasses).

A constraint can be tight. This means that it leaves no room for the `RenderBox` object to choose a size. An example is the App widget, which forces the view to be as big as the screen.

Flex boxes (row and columns) behave differently based on whether they are in bounded or unbounded constraints:

- In bounded constraints, they try to be as big as possible in that direction
- In unbounded constraints, they try to fit their children in that direction

Introducing animations in Flutter

Animations are one of the important features of a widget. Sometimes, developers think that animations are not very important, but designers know that a good set of animations can attract many users. They also contribute to the look and feel of the application, giving it more personality.

Flutter has a great animation support, making it easy to build nice effects and movements. Many widgets come with standard motion effects designed in their design specification, but you can always customize them according to your own need.

Let's take a look at the animation categories, where we will see the two categories the animations in Flutter are divided into, and then take a look at the common patterns of animations.

Animation categories

In general, animations are defined in two categories:

- **Tween animations**: Short for **in-betweening**. In this case, we define the beginning and ending point, the timeline, and a curve of time and speed. The framework will do the rest of the work, calculating the transition and executing it.
- **Physics-based animations**: These types of animations are made with the aim to represent the real-world behavior.

Common patterns

As a user, you may have noticed that some types of animations are constantly used in most apps. These types of animations are the common patterns in animation.

In Flutter, you can find three common patterns:

- **Animated list/grid**: A simple list or grid animating when adding/removing an element.
- **Shared element transition**: This is used when navigating between two pages that have common elements. For example, an image that shows a thumbnail in one route and a normal picture in another.
- **Staggered animation**: A sequence of animations that compose a bigger one. They can be sequential or overlapping.

Using ListView and scrolling widgets

Flutter supports several scrolling widgets, such as `Gridview`, `ListView`, `and PageView`. Lists are the most commonly used scrolling widgets, and are a scrollable, linear list of widgets. It enables the display of its children one after another in the scroll direction.

ListView

`ListView` is a linear list of scrollable items, and is one of the most commonly used widgets. If you have worked on `ListView`s in Android or iOS, this will be straightforward. As in every case, `ListView` produces child-list items one after another. There are several ways to build `ListView`s, so let's take a look at the approaches one-by-one.

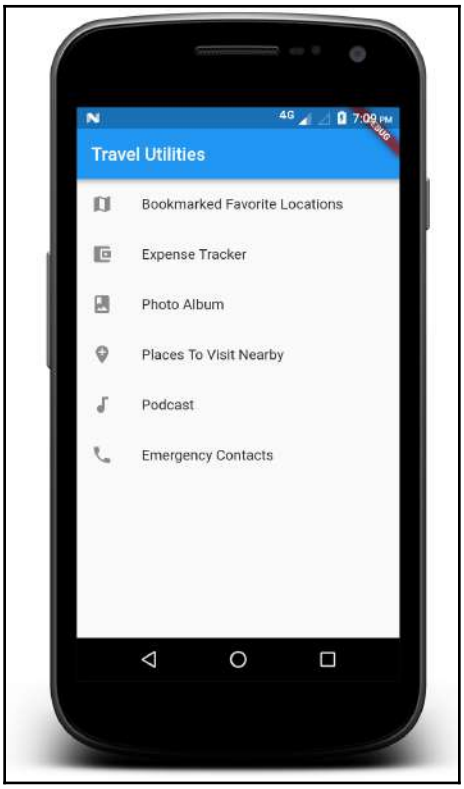
Using List<Widget>

The easiest and most standalone way of building `ListView` is by using an explicit `List<Widget>` of children. This method is ideal for lists with a fixed number of children. Take a look at the following code:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'Travel Utilities';
    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: Text(title),


        ),
        body: ListView(
          children: <Widget>[
            ListTile(
              leading: Icon(Icons.map),
              title: Text('Bookmarked Favorite Locations'),
            ),
            ListTile(
              leading: Icon(Icons.account_balance_wallet),
              title: Text('Expense Tracker'),
            ),
            ListTile(
              leading: Icon(Icons.photo_album),
              title: Text('Photo Album'),
            ),
            ListTile(
              leading: Icon(Icons.add_location),
              title: Text('Places To Visit Nearby'),
            ),
            ListTile(
              leading: Icon(Icons.audiotrack),
              title: Text('Podcast'),
            ),
            ListTile(
              leading: Icon(Icons.phone),
              title: Text('Emergency Contacts'),
            ),
```

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

The following image shows how the preview will look after you run the preceding code:



Using ListView.Builder

The `ListView.builder` constructor calls for `IndexedWidgetBuilder`, which helps developers to build children lists items on demand. This is ideally used for a large or infinite number of visible children, unlike the `ListView` constructor. The other difference is that, while in the case of `ListView`, all the list items have to be defined first, in this case, the `ListView.builder` constructor will create runtimes for the list items as they are scrolled onto the screen.

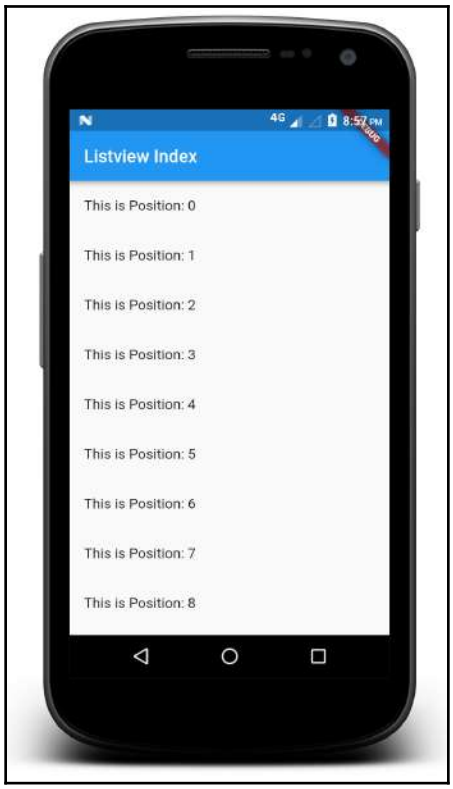
Defining `ListView.builder` is simple and straightforward, as you can see in the following code block:

```
ListView.builder(
  itemCount: 100,
  itemBuilder: (context, index) {
    return ListTile(
      title: Text("Index $index"),
    ); //ListTile
  },
)//ListView.builder
```

Using the preceding code, you will see a `ListView` constructor that shows the index of each item with a text glued to it. The complete code is as follows:

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


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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'ListView Index';

    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: Text(title),
        ),
        body:

        ListView.builder(
          itemCount: 100,
          itemBuilder: (context, index) {
            return ListTile(
              title: Text("This is Position: $index"),
            ); //ListTile
          },
        ) //ListView.builder
    ),//Scaffold
    );//MaterialApp
  }
}
```

After you run the code, the app will appear as follows:



Now, we could add a data source to work with. The data source can be messages, search results, or the sources on the internet that you wish to fetch the data from. We will use the `List<E>.generate` constructor to generate values using the following definition:

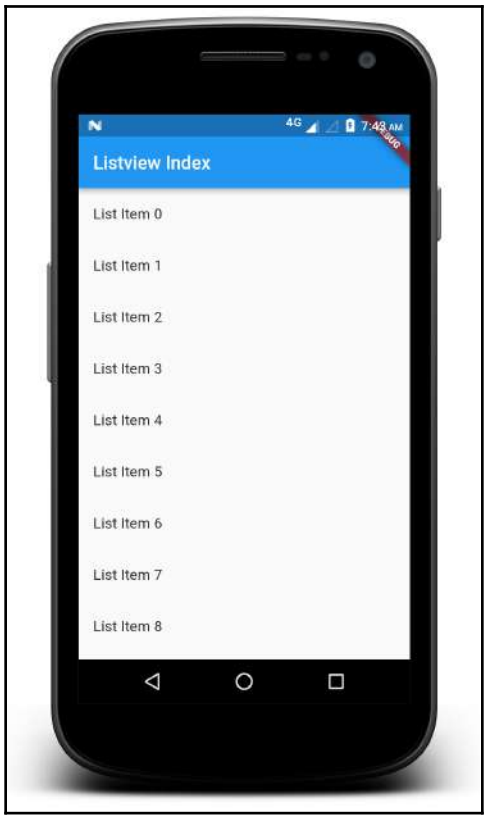
```
List<L>.generate(int length,L generator(int index), {bool growable: true})
```

This creates a list of values with length positions and fills it with values created by calling generator for each index in the range 0 till length–1 in increasing order. The created list is fixed unless the value of `growable` value is set to `true`.

Here is the complete code using the data source to generate `ListView`:

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


void main() {
  runApp(MyApp(
      items: List<String>.generate(100, (i) => "List Item $i"),
  ));
}

class MyApp extends StatelessWidget {
  final List<String> items;

  MyApp({Key key, @required this.items}) : super(key: key);

  Widget build(BuildContext context) {
    final title = 'ListView Index';

    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: Text(title),

        ),
        body:

        ListView.builder(
          itemCount: 100,
          itemBuilder: (context, index) {
            return ListTile(
              title: Text('${items[index]}'),

        );//ListTile
 },
 )//ListView.builder
 ),//Scaffold
 );//MaterialApp
 }
}
```

The output of the preceding code is as follows:



ListView separated by calling ListView.separated

In the previous cases of code executions, we saw that even though the `ListTiles` were listed, there was no separation among them. To build a divider between the `ListTiles`, it also provides a helper constructor for creating a `ListView`. The constructor is `ListView.separated`. This divider is called by the **divider class** to build a one device pixel-thick horizontal line, having padding on either side. Dividers can be used in lists, drawers, or separate content, vertically or horizontally based on the value of the `Axis` enum, as specified in the following `ListView.separated` constructor:

```
ListView.separated({
Key key,
Axis scrollDirection: Axis.vertical,
bool reverse: false,
```

```
ScrollController controller,
bool primary,
ScrollPhysics physics,
bool shrinkWrap: false,
EdgeInsetsGeometry padding,
@required IndexedWidgetBuilder itemBuilder,
@required IndexedWidgetBuilder separatorBuilder,
@required int itemCount,
bool addAutomaticKeepAlives: true,
bool addRepaintBoundaries: true,
bool addSemanticIndexes: true,
double cacheExtent
})
```

The constructor can be called in the following way:

```
ListView.separated(
itemCount: 25,
separatorBuilder: (BuildContext context, int index) => Divider(),
itemBuilder: (BuildContext context, int index) {
return ListTile(
title: Text('item $index'),
);
}, )
```

This builds a fixed-length scrollable linear array of list items that are separated by list items `separators`. The `itemBuilder` callback will be called with indices greater than or equal to `0`, and less than `itemCount`. The separator is built after the first item and before the last item in the list. The `separatorBuilder` callback will be called with indices greater than or equal to `0`, and less than `itemCount 1`.

Here is the sample of the `ListView` constructor using `ListView.separated`:

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


void main() {
  runApp(MyApp(
      items: List<String>.generate(100, (i) => "List Item $i"),
  ));
}

class MyApp extends StatelessWidget {
  final List<String> items;

  MyApp({Key key, @required this.items}) : super(key: key);
```

```
  Widget build(BuildContext context) {
    final title = 'ListView Index';

    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: Text(title),

        ),
        body:

        ListView.separated(
          itemCount: 25,
          separatorBuilder: (BuildContext context, int index) => Divider(),
          itemBuilder: (BuildContext context, int index) {
            return ListTile(
              title: Text('${items[index]}'),
          );//ListTile
          },
        )//ListView.builder
    ),//Scaffold
    );//MaterialApp
  }
}
```

Once you run the preceding code, you will see the `ListView` with separators.

Using ListView.custom constructor

By making use of `SilverChildDelegate`, this method provides the ability to customize several aspects of the child model, defining the way in which they are built. The main parameter required for this is `SliverChildDelegate`, which builds the items. The types of `SliverChildDelegates` are as follows:

- `SliverChildListDelegate`
- `SliverChildBuilderDelegate`

`SliverChildListDelegate` accepts a direct list of children, while, on the other hand, `SliverChildBuiderDelegate` accepts `IndexedWidgetBuilder`. Take a look at the `ListView.custom` constructor:

```
const ListView.custom({
Key key,
Axis scrollDirection: Axis.vertical,
bool reverse: false,
ScrollController controller,
bool primary,
ScrollPhysics physics,
bool shrinkWrap: false,
EdgeInsetsGeometry padding,
double itemExtent,
@required SliverChildDelegate childrenDelegate,
double cacheExtent,
int semanticChildCount
}
)
```

Horizontal lists

Once you have received a hands-on with vertical lists, horizontal lists are simple. In this case, we call the `ListView` constructor, passing through a horizontal `scrollDirection`. This simply overrides the default vertical direction. In this case, we use a `Container` widget, which is an easy-to-use widget that combines common painting, positioning, and sizing widgets. Take a look at the following code:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'Horizontal List Example';

    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: Text(title),
        ),
        body: Container(
          margin: EdgeInsets.symmetric(vertical: 100.0),
```

```
            height: 300.0,
            child: ListView(
              scrollDirection: Axis.horizontal,
              children: <Widget>[
                Container(
                  width: 120.0,
                  color: Colors.orange,

                ),
                Container(
                  width: 120.0,
                  color: Colors.white,
                ),
                Container(
                  width: 120.0,
                  color: Colors.green,
                ),
                Container(
                  width: 120.0,
                  color: Colors.pink,
                ),
                Container(
                  width: 120.0,
                  color: Colors.lime,

              ), // Container
            ], // <Widget>[]
          ), //ListView
          ), //Container
    ), // Scaffold
    ); //MaterialApp
    }
     }
```

The `ListView scrollDirection: Axis.horizontal` property ensures that the list is horizontally scrollable. Once you run the code successfully, you will see the following result:

Grid lists

Just like in the case of horizontal lists, event grid lists are easy to build. It uses a `GridView.count` constructor that allows us to specify how many rows and columns we want on the screen. In the following example, we build `100` widgets that print the value of the position:

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

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'Grid List Example';

    return MaterialApp(
      title: title,
```

```
        home: Scaffold(
          appBar: AppBar(
            title: Text(title),
          ),
          body: GridView.count(
            // Create a grid with 3 columns.
            crossAxisCount: 3,
    // Generate 100 Widgets that display their positions in the List
            children: List.generate(100, (index) {
              return Center(
                child: Text(
                  'Position $index',
                  style: Theme.of(context).textTheme.headline,
              ), // Text
              ); //Center
            }), //List.Generate
          ), //GridView.count
        ), //Scaffold
      );   // MaterialApp
  }
  }
```

The preceding code will produce the following output:

Introducing silvers

We took a look at the `ListView`, now let's quickly take a look at what silvers are, taking a quick example from the `ListView`. Being fancy in your layout can be visually pleasing if executed well. That's exactly how silver can help you. A silver is a portion of a scrollable area using which you can bring in custom scrolling into your view. Let's take a simple example in the form of `ListView`. If an app bar remains static it can sometimes obstruct the view, so, in this case, silver can be used to hide the app bar while you scroll.

One thing that has to be noted is that all of the sliver components go inside `CustomScrollView`. As a developer, you can then combine your lists of silvers to build your custom scrollable area.

Summary

At the start of the chapter, we began discussing constraints in Flutter. We then discussed animations and forms of animation in Flutter. In the next section, we executed some examples of different types of lists. In the final section, we looked at how we can custom animate the portion.

In the next chapter, we will widen our Flutter horizon and look at network and accessibility with Flutter.

5
Widening our Flutter Horizons

In this chapter, we will first discuss networking in Flutter by building a simple application that fetches data from the server using JSON. Every app is required to have accessibility features to cater to mass users, and we will cover this in the accessibility options. In the final section, we will talk about localization for having your app grow globally, supporting multiple languages.

In this chapter, we will cover the following topics:

- Networking in Flutter
- Accessibility in Flutter
- Internationalizing Flutter apps

Networking in Flutter

Networking is the backbone of any app, and knowing how to make network calls is crucial. Working with networking calls in Flutter is simple and follows a streamline standard method. Flutter libraries and methods make it easier for developers to build apps with networking. This chapter will focus on making networking requests.

Using packages

Like many platforms, Flutter supports the use of shared packages that are contributed by the developers to the Flutter and Dart ecosystem. This facilitates development by making developers build apps quickly rather than worry about developing the code from scratch. Some of the most commonly used packages include, but are not limited to: making network requests (HTTP); using device APIs, such as device information (`device_info`); finding information and controlling the camera, including the support for previews of the camera feed and the captured image (camera); finding and using the location of the device using GPS coordinates (geolocator); and using third-party platform SDKs (such as Firebase). You can find the complete list of packages supported by Flutter at `https://pub.dartlang.org/packages`.

Adding existing package dependency to an app

Once you have decided on the set of packages that you want to include, follow these steps to include the dependency. For the purpose of this example, we have chosen HTTP package to an app. This package contains a set of high-level functions and classes, which can help developers consume HTTP resources while working on the app, and it is platform independent. It supports both the command line and the browser:

1. Create the dependency: Open the `pubspec.yaml` file located inside your app folder, and add `http:` under dependencies.
   All packages have a version number, specified in their `pubspec.yaml` file. The current version of the package is displayed next to the package name. When you mention `Plugin_Name_1:`, it is interpreted as `Plugin_Name_1: any`. This indicates that any version of the package may be used. It is advisable to use a specific version to ensure that the app doesn't break when it is updated.

2. Install the package where the dependency has been added. You can install it by running the `flutter packages get` command. If you are using Android Studio/IntelliJ, you can also click the **Package Get** option in the action ribbon at the top of `pubspecs.yaml`. If you are using VS code, click **Get Packages** located on the right-hand side of the action ribbon at the top of `pubspec.yaml`
3. Include the corresponding `import` statement in your Dart code. In this case, it is `import package:http/http.dart`. In case you have missed anything, you can always cross-check using the **Installation** tab option on the package page on `Pub`.
4. At this point, it is better if you stop and restart the app to avoid errors such as `MissingPluginException` when using the package.

Upgrading existing package

When you run `flutter packages get` (this will be `Packages Get` in IntelliJ) for the first time after adding a package in the `pubspec.yaml` file, Flutter will save the version found in the `pubspec.lock lockfile` file. To upgrade the package, you can run the Flutter packages upgrade (`Upgrade dependencies` in IntelliJ). Using this command, Flutter will retrieve the highest available version of the package. In case you have specified range constraint in `pubspec.yaml`, it will fetch the update as per the specification of the constraint.

Building a REST service

One of the most prominent tasks for developers is to build REST services for the project that help you gather data in JSON format, which you can reflect on the front-end of the application. Imagine working on an application, and you want to mock up a REST web service to get the demo data for you. You could certainly build your backend server using Node.js, MongoDB, or other platforms, but one of the easiest ways is to use a JSON server. A JSON server is a simple project that stimulates REST API with CRUD operation. This project hardly consumes time for the setup, and you can swiftly process the data to ensure that everything works as expected. It is ideal for developers who are learning to build REST APIs to understand how the data is processed with a backend for prototyping and mocking.

Setting up JSON Server

The setup of this project can be found at `https://github.com/typicode/json-server`. Note that this project builds a full fake REST API for demo purposes only. Before we begin the setup, ensure that the following components are ready on your system:

1. **Node.js**: JSON-Server is built on top of Node.js. If you already have it in place, please ensure to keep it updated. To find out the version of Node, run the `node -v` command.
2. **NPM package**: NPM stands for Node Package Manager, and comes in handy to easily install, update, configure, and uninstall Node JS platform modules/packages. Ensure NPM is installed on the system. If not, refer to `https://www.npmjs.com/get-npm`. At this point, it would be ideal to quote that NPM is a separate project from the Node, and gets updated frequently. To update NPM, use the `sudo npm install npm@latest -g` command.

3. `cURL`: This open source command line enables the transfer of data with URL syntax. If you have `cURL` installed, use the `curl -V` command (Note that `V` is upper case). In case you need to install `cURL`, run the `brew install curl` command.

The JSON server is available as an NPM package, so we could simply run the following command to install it:

```
$ npm install -g json-server
```

The `-g` option enables the package to be installed globally on your system. Once you have installed it successfully, run the command to cross check:

```
$ json-server -v
```

Building a resource file

A resource is anything that has data associated with it. For example, if you are working on a movie review website, movies, reviewers, users, and so on, would be resources. You could have API endpoints based on these resources. The API endpoints help you to retrieve or update the data on the server. In this case, we will use the resources as a JSON file. This JSON file will act as a config and database file for the mock server you set up using `json-server`.

`Json-server` works in a JSON file, and creating a JSON file is simple. Create a new file as `Books.json`, populate the following content, and save it. Note that the name of the array we specified is `Movie`, so `json-server` will create the REST APIs based on this name:

```
{
  "Movie": [
    {
      "id": 1,
      "Movie Name": "Avengers: Infinity War",
      "Year": "2018",
      "Category": "Science Fiction"
    },
    {
      "id": 2,
      "first_name": "Black Panther",
      "Year": "2018",
      "Category": "Science Fiction"
    },
    {
      "id": 3,
```

```
        "first_name": "Mission: Impossible – Fallout",
        "Year": "2018",
        "Category": "Action"
      },
      {
        "id": 4,
        "first_name": "Annihilation",
        "Year": "2018",
        "Category": "Fantasy"
      }
    ]
}
```

Run the json-server

The final step is to run `json-server` to ensure we have the mock server running locally. Run the following command:

```
$ json-server --watch db.json
```

Congratulations! You have now successfully set up `json-server`. Open the `http://localhost:3000/` URL to check whether you can see the following. Under the resources tag, you will be able to see the Movies JSON file we created.

Do not close the terminal, as that will kill `json-server`. In case your port `3000` is utilized, you can set options for the new port number in a `json-server.json` configuration file.

Fetching data from the server

Fetching data from the server is a commonly used feature. It is performed using the HTTP package that we covered earlier in this chapter. The steps to follow are as follows:

1. Include the HTTP package and make the network request
2. Convert the response into a custom Dart object
3. Fetch the data and display it using the Flutter

Since we have already learned about adding HTTP packages in the *Using packages* section, we will now proceed with making a network request. In our next step, we will fetch the sample JSON content using JSON-Server and the `http.get()` method.

We use the `Future` function, which is a core Dart class for working with async tasks and, together with `http`, it returns the data from a successful `http` call:

```
Future<http.Response> fetchPost() {
  return http.get('http://localhost:3000/Movie/1');
}
```

We now create a `Post` class that will contain the data from our network requests. To ensure that we create a proper `Post` from JSON, we will include a `factory` constructor. In our example, we have four categories of data for each array to fetch, namely, `id`, `movieName`, `year`, and the `category`:

```
class Post {
  final int id;
  final String movieName;
  final int year;
  final String category;

  Post({this.id, this.movieName, this.year, this.category});

  factory Post.fromJson(Map<String, dynamic> json) {
    return Post(
      id: json['id'],
      movieName: json['movieName'],
      year: json['year'],
      category: json['category'],
    );
  }
}
```

Next, we will need to update the `fetchPost` function to return a `Future<Post>`, for which we will follow three steps:

1. First, convert the response body into a JSON Map using the `dart:convert` package. This package contains encoders and decoders for converting between different data representations. To use this, you will first have to add the dependency in your package's `pubspec.yaml` file:

   ```
   dependencies: convert: ^2.1.1
   ```

   Now, use the `package:convert/convert.dart` import in your dart code.

2. If we get an `OK` response from the server with a status code of `200`, it means the data is fetched, and you can convert the JSON Map into a `Post` using the `fromJSON` factory.

3. If the response is unexpected, you can flag an error.

Here is the piece of code that checks the previously-mentioned cases:

```
Future<Post> fetchPost() async {
  final response =
  await http.get('http://localhost:3000/Movies/1');

  if (response.statusCode == 200) {
    // If the call to the server was successful, parse the JSON
    return Post.fromJson(json.decode(response.body));
  } else {
    // If that call was not successful, flag an error.
    throw Exception('Failed to load post');
  }
}
```

In order to fetch the data and display it, we use the `FutureBuilder` widget that is built into Flutter, and helps in working easily with async data sources. To make this happen, we will need two parameters:

- The name of the future we want to work with. In our example, we call it the `fetchPost()` function.
- A builder function that informs Flutter what to render, based on the state of the Flutter—loading, success, or error:

```
FutureBuilder<Post>(
            future: post,
            builder: (context, snapshot) {
              if (snapshot.hasData) {
                return Text(snapshot.data.movieName);
              } else if (snapshot.hasError) {
                return Text("${snapshot.error}");
              }

              // By default, show a loading spinner
              return CircularProgressIndicator();
            },
);
```

Building the code by putting a call to an API in your `build()` method is convenient, but it's not recommended. It will make Flutter call the `build()` method every time when it wants to change anything in the view, making your app slow due to it making unnecessary flooded API calls. A better way is to bit the API when the page is initially loaded, and use `StatelessWidget` for the same.

Using this method, you will make the parent widget responsible for calling the fetch method, storing its result and then passing it to your widget:

```
class MyApp extends StatelessWidget {
  final Future<Post> post;

  MyApp({Key key, this.post}) : super(key: key);
```

This is the complete code that will fetch the JSON content using `json-server` by reading `Movies.json`:

```
import 'dart:async';
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

Future<Post> fetchPost() async {
  final response =
  await http.get('http://localhost:3000/Movies/1');

  if (response.statusCode == 200) {
    // If the call to the server was successful, parse the JSON
    return Post.fromJson(json.decode(response.body));
  } else {
    // If that call was not successful, throw an error.
    throw Exception('Failed to load post');
  }
}

class Post {
  final int id;
  final String movieName;
  final int year;
  final String category;

  Post({this.id, this.movieName, this.year, this.category});

  factory Post.fromJson(Map<String, dynamic> json) {
    return Post(
      id: json['id'],
      movieName: json['movieName'],
      year: json['year'],
      category: json['category'],
    );
  }
}
```

```
void main() => runApp(MyApp(post: fetchPost()));

class MyApp extends StatelessWidget {
  final Future<Post> post;

  MyApp({Key key, this.post}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'JSON Fetch Data Example',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: Scaffold(
        appBar: AppBar(
          title: Text('JSON Fetch Data Example'),
        ),
        body: Center(
          child: FutureBuilder<Post>(
            future: post,
            builder: (context, snapshot) {
              if (snapshot.hasData) {
                return Text(snapshot.data.movieName);
              } else if (snapshot.hasError) {
                return Text("${snapshot.error}");
              }

              // By default, show a loading spinner
              return CircularProgressIndicator();
            },
          ),
        ),
      ),
    );
  }
}
```

At the start of the chapter, we discussed what packages are and how to use them. Once the packages were set up, we discussed how to build a JSON Server to fetch data for our tests. In the final section, we glanced through an example of how to fetch the JSON data to the app, displaying the content of the JSON file we created and ran using the JSON server.

Accessibility in Flutter

Making your app accessible to many users could be a great initiative. That also includes people with disabilities, such as blindness, hearing, voice, or motor impairment. As per the reports on disability by the World Health Organization, there are over 100 million users across the globe who face physical challenges in their daily routine. Technology can be revolutionary in helping people, and that's when building apps catering to their specific needs can aid them well.

Not all the users use the app in a specifically defined manner, so, a focus on accessibility will not only help users to download and use the app, but will also propagate to a new level of users.

Google provides an app to check for accessibility support that is available as accessibility scanner on Google Play at `https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor` This app enables you to find the accessibility provide that a developer can do within the app. For iOS, XCode provides Accessibility Inspector.

Flutter supports three components for accessibility support:

Large font

With age, not many can see the content the way they used to in their youth. Some face issues in reading the text clearly, especially when developers consider using the default size without taking into considering factors such as screen size and orientation. One of the quickest ways to do this is to ensure that the text scales in their accessibility options consider the device specifications of the consumers.

Flutter has a feature that handles text size calculations automatically. For example, the `Text` widget has a `textScaleFactor` property that allows the scaling of the text. Font size is multiplied by the `textScaleFactor` value to determine the new font size that is rendered on the screen in logical pixels. For example, if the `textScaleFactor` is `1.5`, the text will be 50% larger than the specified font size, as follows:

```
Text(
  'Hello India, how are you?',
  textAlign: TextAlign.center,
  overflow: TextOverflow.ellipsis,
  style: TextStyle(fontWeight: FontWeight.normal),
  textScaleFactor: 1.5,
)
```

One point that needs to be kept in mind is that, if you manually set the scale value, the user's accessibility settings will be overridden. You have to ensure that the scale-up value doesn't show up the text to a large extent leading to uninstall of the app. If you do not specify the value, it will check `textScaleFactor` for the nearest `MediaQuery` ancestor (`MediaQueryData.textScaleFactor`) or `1.0`, if no such ancestor exists. Ensure that you test the text displays properly at all accessibility settings.

Screen readers

For those who are visually impaired, this accessiblity option can come in handy. It enables users to receive spoken feedback about the content of the screen. You could turn on VoiceOver in iOS, or TalkBack in an Android application on your device to navigate around your app. For example, when using TalkBack, users perform actions using gestures, and each action is complimented with an audible output that allows users to know that their gesture trigger was successful. There are three types of gestures in TalkBack: basic gestures, back-and-forth gestures, and angle gestures. Note that the users should use single gestures, even finger pressure, and a steady speed to have a seamless experience.

Screen contrast

Specifying background and foreground colors with sufficient color contrast enables better readability for the users. This ratio ranges from 1 to 21, where 21 means the highest.

The W3C recommends the following:

- At least 4.5:1 for smaller text (below 18 point regular, or 14 point bold)
- At least 3.0:1 for larger text (18 point and above regular, or 14 point and above bold)

Accessibility is an important feature and should not be neglected. It ensures that the app is open to a larger audience, enabling the chances for better application usage. It is equally important to test the accessibility options before rolling out to the masses.

Internationalizing Flutter apps

As the name suggests, if your app will be by the international audience, you will have to think of providing **locale** support for the specific language of the target. That means you'll need to write the app in a way that your app renders the values like text and layouts depending on each language or locale that the app supports. Flutter has made it simple by providing support by classes and widgets. Flutter supports the global localization classes for about 24 languages.

Before you start the internationalization, dependencies must be added in `pubspec.yaml`:

```
dependencies:
  flutter:
    sdk: flutter
  flutter_localizations:
    sdk: flutter
```

The next step is to import the `flutter_localizations` library and specify `localizationsDelegates` and `supportedLocales` for `MaterialApp`. Also, import `package:flutter_localizations/flutter_localizations.dart`:

```
Widget build(BuildContext context) {
  return MaterialApp(
    onGenerateTitle: (BuildContext context) =>
DemoLocalizations.of(context).title,
    localizationsDelegates: [

// ... app-specific localization delegate[s] here

      const DemoLocalizationsDelegate(),
      GlobalMaterialLocalizations.delegate,
      GlobalWidgetsLocalizations.delegate,
    ],
    supportedLocales: [
      const Locale('en', ''), //Supporting English
      const Locale('hi', ''), // Supporting Hindi
      const Locale('es', ''), // Supporting Spanish
    ],
```

The `localizationsDelegates` list contains the elements that are factories that produce collections of localized values. `GlobalMaterialLocalizations.delegate` provides strings that are localized and other values for the material components library. The default text direction for the widget library is defined by `GlobalWidgetsLocalizations.delegate`.

There are three methods to keep an eye on:

- `.load`: This method must return an object that contains a collection of related resources
- `.isSupported`: If the support for the locale is found, it returns `True`. Otherwise it will return `False`
- `shouldReload`: If this method returns `True`, then all the app widgets will be rebuilt after a load of resources

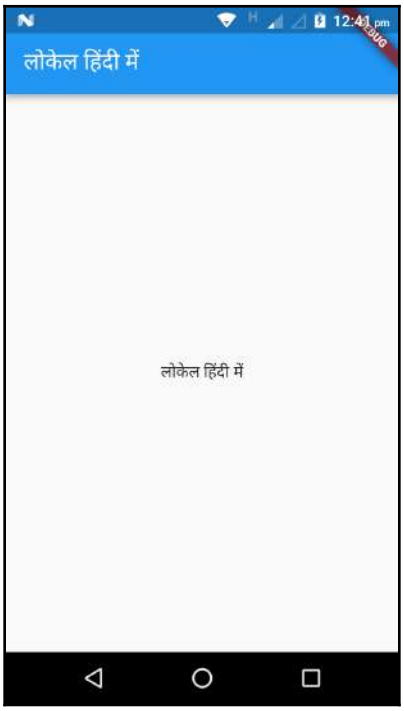
The complete code for your reference is as follows:

```
import 'dart:async';

import 'package:flutter/material.dart';
import 'package:flutter/foundation.dart' show SynchronousFuture;
import 'package:flutter_localizations/flutter_localizations.dart';

class DemoLocalizations {
  DemoLocalizations(this.locale);

  final Locale locale;

  static DemoLocalizations of(BuildContext context) {
    return Localizations.of<DemoLocalizations>(context, DemoLocalizations);
  }

  static Map<String, Map<String, String>> _localizedValues = {
    'en': {
      'title': 'Locale in English',
    },
    'es': {
      'title': 'Local en españa',
    },
    'hi': {
      'title': 'लोकेल हिंदी में',
    },
  };

  String get title {
    return _localizedValues[locale.languageCode]['title'];
  }
}

class DemoLocalizationsDelegate extends
LocalizationsDelegate<DemoLocalizations> {
  const DemoLocalizationsDelegate();
```

```
  @override
  bool isSupported(Locale locale) => ['en', 'es',
'hi'].contains(locale.languageCode);

  @override
  Future<DemoLocalizations> load(Locale locale) {
    // Returning a SynchronousFuture here because an async "load" operation
    // isn't needed to produce an instance of DemoLocalizations.
    return SynchronousFuture<DemoLocalizations>(DemoLocalizations(locale));
  }

  @override
  bool shouldReload(DemoLocalizationsDelegate old) => false;
}

class DemoApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(DemoLocalizations.of(context).title),
      ),
      body: Center(
        child: Text(DemoLocalizations.of(context).title),
      ),
    );
  }
}

class Demo extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      onGenerateTitle: (BuildContext context) =>
DemoLocalizations.of(context).title,
      localizationsDelegates: [
        const DemoLocalizationsDelegate(),
        GlobalMaterialLocalizations.delegate,
        GlobalWidgetsLocalizations.delegate,
      ],
      supportedLocales: [
        const Locale('en', ''),
        const Locale('es', ''),
        const Locale('hi', ''),
      ],

      home: DemoApp(),
    );
```

```
    }
  }

  void main() {
    runApp(Demo());
  }
```

After you run the code successfully, you will see the output as follows:



Summary

We first discussed how networking plays an important role in the apps, along with sample code for setting up and running a local server for fetching JSON code. This section was followed by understanding why accessibility is important, and what improvements developers can provide to support accessibility in the app. The next section showed how to make app support internationalization.

In the next chapter, we will discuss how to use platform powers to build apps.

6
Using a Platform to Power Flutter Apps

With the use of Flutter, you can build for both Android as well as iOS. It uses the Dart programming language to do so. However, Dart does not compile to Android's Dalvik bytecode or Objective C bindings on iOS. This indicates that Dart code, by default, does not have direct access to platform-specific APIs.

Here are a few sets of examples where deeper integration with the host environment might be needed:

- Applications using camera capabilities and geo-tagging features
- Reading device information, such as an OS version and device specifications
- Reading folders and files from the device
- Pushing notifications to the app
- Sharing information with other applications
- Location tracking
- Using sensors
- Using persisted preferences

The list continues as per the support provided by the environment. Using Flutter, enabling the calling of platform-specific APIs, which are available in Java/Kotlin code on Android, Objective C, or Swift on iOS, is not a difficult task.

In this chapter, we will learn how to include packages, followed by learning how to make platform-specific calls. We will learn about how to publish our own plugins.

We will cover the following topics in this chapter:

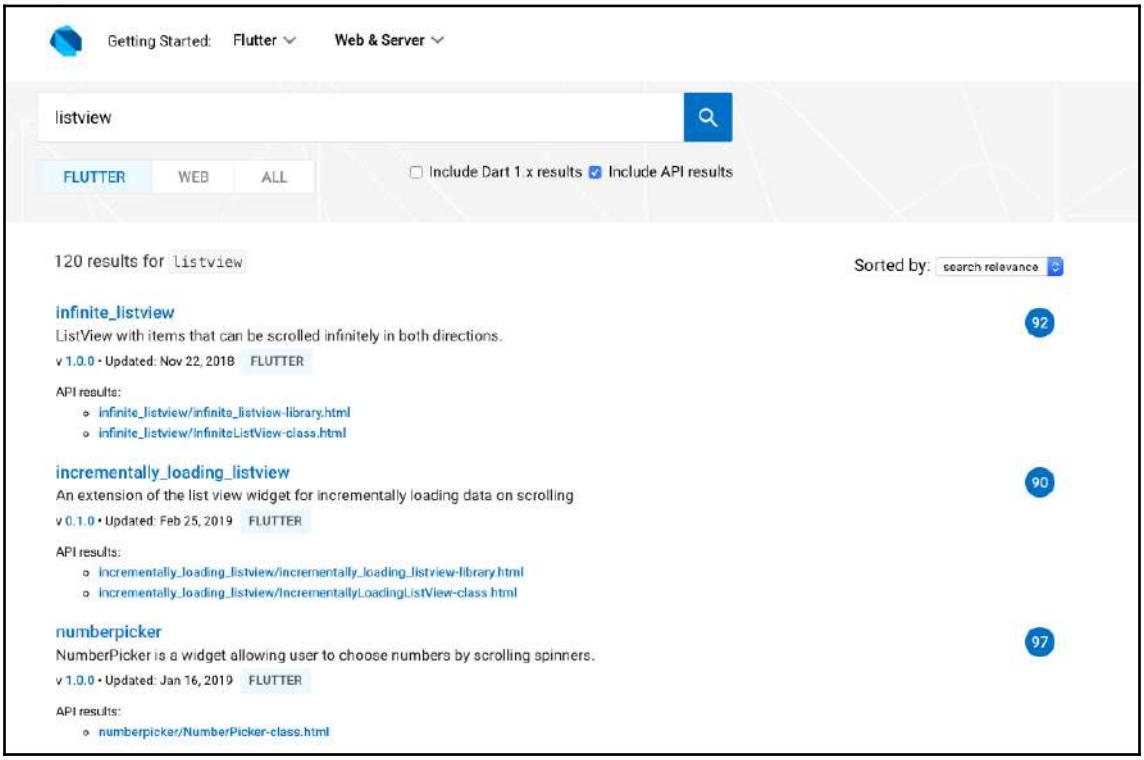
- Using Flutter packages
- Using platform channels
- Building and publishing your own plugin

Using Flutter packages

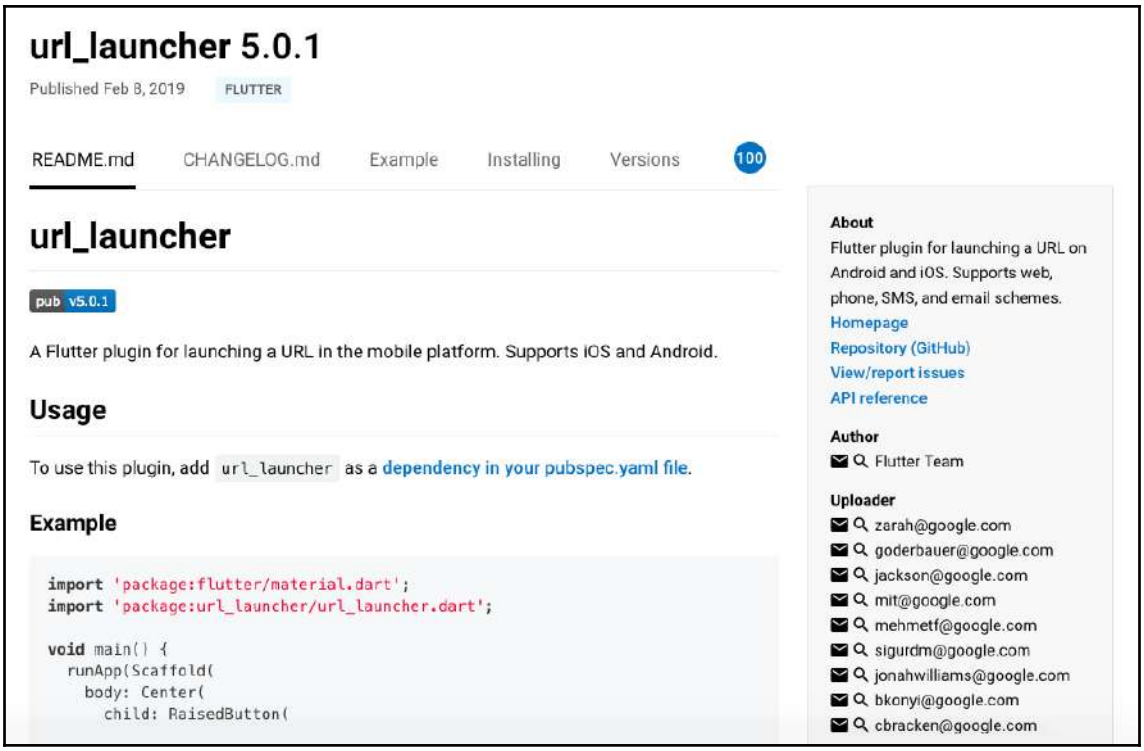
Flutter's package website lists many packages that help us develop applications faster by avoiding the need for developing some features from scratch. These packages are either contributed to by the Flutter team or by developers across the globe who contribute to the Flutter and Dart ecosystems. You can either use the existing packages by visiting the publishing site (`https://pub.dartlang.org/https://pub.dartlang.org/`), or you can develop and publish your own packages. We will learn about building our own packages in the *Building your own packages* section of this chapter.

Searching for the package

Visit the publishing site to search for the packages you want to use in your app. The home page displays some of the popular packages that are used by developers. You can search by keyword to display the results. You may want to look at the weighted score before you opt to choose a plugin. Searching for either `email:@dartlang.org` or `email:flutter-dev@googlegroups.com` will give the results of the official Flutter packages:



Clicking on any of the result items will open the extended preview for the package. In this case, we are using the **url_launcher** package, which is an official Dart package that was developed by the Flutter team. This plugin is used for launching a URL on a mobile platform such as Android and iOS-supporting web, phone, SMS, and email schemes:



When you click on any of the packages, you will see the name of the plugin, along with its latest version and its published date. Following that, you will find five tab options, namely the following:

- `README.md`: Detailed information about the plugin
- `CHANGELOG.md`: Information on the Changelog versions of this plugin with details of each plugin version
- `Example`: Demonstrates how to use the plugin
- `Installing`: Shows the steps to follow to set up the plugin
- `Versions`: Shows the different versions of the plugin, along with the following options: `Version` name (such as 5.0.1), `Uploaded` date (such as Feb 8, 2019), `Documentation`, and `Archive`

The Flutter plugin for launching a URL on Android and iOS supports web, phone, SMS, and email schemes.

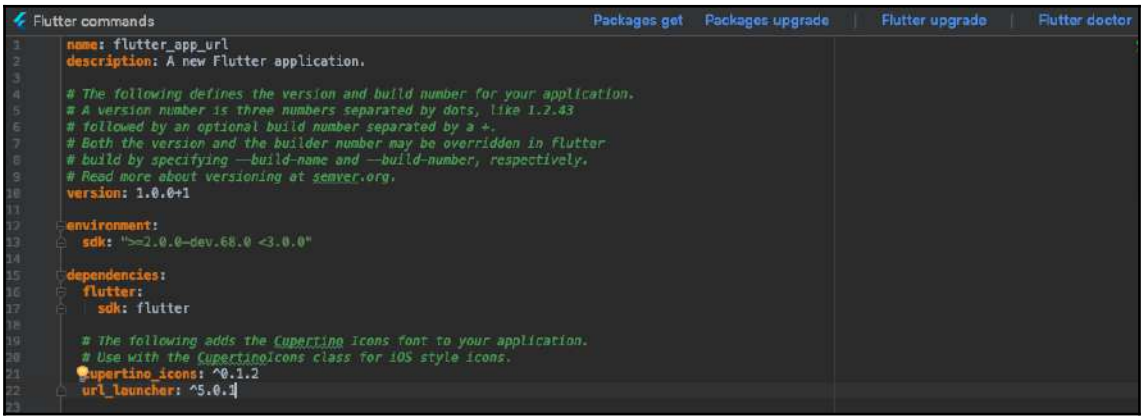
Adding a package dependency to an app

Once you have decided on the package you want to use, you have to make the program depend on it. In this case, we are using the `url_launcher 5.0.1` package:

1. Open the `pubspec.yaml` file located inside your `app` folder.
2. Check out the **Installing** tab on the package's page, and where you will have the option of adding dependencies. In this example, we are adding `url_launcher: ^5.0.1` under dependencies.
3. Next, from the Terminal, run the following command, which will install packages from the command line:

```
flutter packages get
```

   If you are using an editor such as Android Studio/IntelliJ, click **Packages get** on the top of `pubspec.yaml`. In the case of VS code, click **Get Packages** at the top of `Pubspec.yaml`:



4. The next step is to add the correspondent `import` snippet to your Dart code:

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

If you want to upgrade to a new version of the package, use the **Packages upgrade** option. It could be used in the case where there are new features available in that package you wish to include in the project. In the case of IntelliJ, the option to choose is **Upgrade dependencies**.

Ways to specify package dependencies

There are two ways to specify package dependencies. As we discussed earlier, the package is added to `pubspec.yaml` using the shorthand form `plugin1:`. This means `plugin1:any_version`. As a developer, you can always look at `CHANGELOG.md` or the recent version of the package's name to add the correct value. To ensure an app does not break, it is essential to specify a version, which can be done in one of two ways:

1. Specify the version using range constraints, as follows, in which you specify the minimum and maximum versions:

```
dependencies:
url_launcher: '>=0.4.1 <5.0.1'
```

2. Range constraints using `Carat Syntax`:

```
dependencies:
url_launcher: '^5.0.1'
```

Adding the code to the file

Most of the packages give insights on how to include the package components into the code. The package's **Example** tab has more information about the code's execution. In our example, we are using one gradient button to open the URL specified in the code:

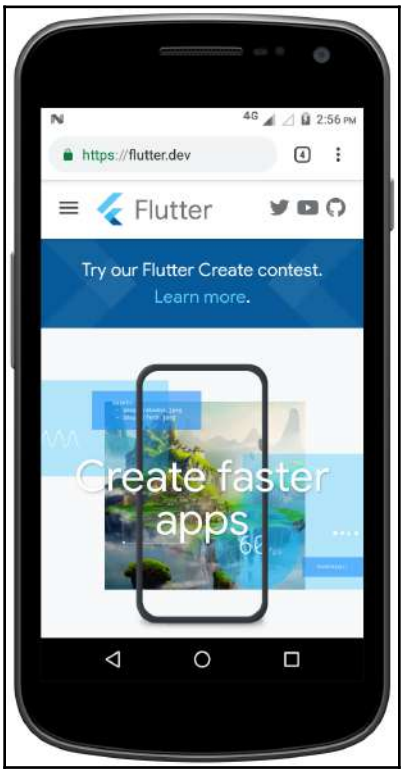
```
import 'package:flutter/material.dart';
import 'package:url_launcher/url_launcher.dart';

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

class MyApp extends StatelessWidget {
  @override

  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Packages',
      theme: ThemeData(
        primarySwatch: Colors.deepOrange,
```

```
      ),
      home: MyStatelessWidget(),
    );
  }
}

class MyStatelessWidget extends StatelessWidget {
  MyStatelessWidget({Key key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Packages in Flutter'),
      ),
      body: Center(
        child: Column(
          mainAxisSize: MainAxisSize.min,
          children: <Widget>[

            RaisedButton(
              onPressed: _initiateURL,
              textColor: Colors.black,
              padding: const EdgeInsets.all(0.0),
              child: Container(
                decoration: const BoxDecoration(
                  gradient: LinearGradient(
                    colors: <Color>[
                      Colors.green, Colors.yellow, Colors.yellowAccent],

                  ),
                ),

                padding: const EdgeInsets.all(10.0),
                child: Text('Open Flutter Website'),
              ),
            ),
          ],
        ),
      ),
    );

  }

  _initiateURL() async {
    const url = 'https://flutter.dev';
    if (await canLaunch(url)) {
      await launch(url);
```

```
      } else {
        throw 'Sorry, We could not launch the URL $url';
      }
    }

}
```

The previous code allows users to simply click on the button and open the Flutter Dev official website. As you may have noticed, we use the `RaisedButton` widget and specify the child property, which allows us to provide gradient to the button.
The `_initiateURL()` method has a launch function that accepts the URL defined and acts to parse the specified URL. It also delegates the handling to the underlying system:

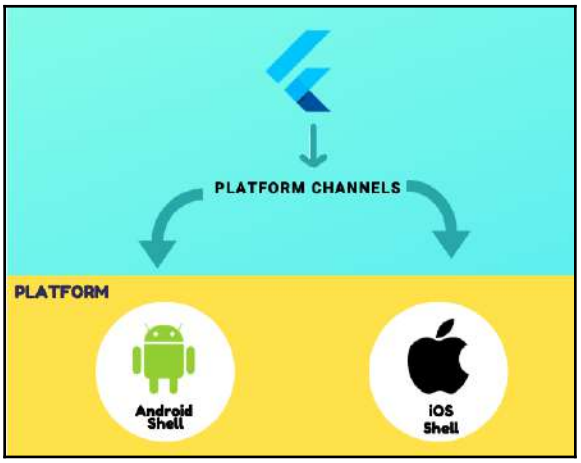
Once you have successfully executed the code and clicked on the button, **Open Flutter Website** will open the URL specified in the parameter URL. You will see the result on the screen on the phone's default browser, as follows:



Using platform channels

As mentioned at the start of the chapter, Flutter is a cross-platform framework, and in order to get access to native APIs, you have to get access to native platform functions. In the case of Flutter, it does this by creating a platform channel to the native platform. Using these platform channels, the developers can call the native functions such as device information, files and folder access, sensor access, camera, shared preferences, and much more.

As shown in the following screenshot, platform channels can be simply imagined to be a communicating mechanism between your Dart code in Flutter and the platform-specific code of your host app. This ensures that the host services are invoked by Flutter's Dart code:



At this point, it is worth noting that you have to set a platform channel for each platform. In the case of Android, they are called `MethodChannels` and in iOS, they are known as `FlutterMethodChannels`.

At this point, it is important to know the platform data type support and codecs. The standard platform channels adopt a standard message codec that supports efficient binary serialization. They are JSON-lookalike. When you send values to and from them, the serialization and deserialization happen automatically.

The Flutter team at Google has listed (source: `https://flutter.dev/docs/development/platform-integration/platform-channels`) the following table, which shows how Dart values are received on the host platform side, and vice versa:

Dart	Android	iOS
`null`	`null`	`nil` (NSNull when nested)
`bool`	`java.lang.Boolean`	`NSNumber numberWithBool:`
`int`	`java.lang.Integer`	`NSNumber numberWithInt:`
`int, if 32 bits, not enough`	`java.lang.Long`	`NSNumber numberWithLong:`

double	java.lang.Double	NSNumber numberWithDouble:
String	java.lang.String	NSString
Uint8List	byte[]	FlutterStandardTypedData typedDataWithBytes:
Int32List	int[]	FlutterStandardTypedData typedDataWithInt32:
Int64List	long[]	FlutterStandardTypedData typedDataWithInt64:
Float64List	double[]	FlutterStandardTypedData typedDataWithFloat64:
List	java.util.ArrayList	NSArray
Map	java.util.HashMap	NSDictionary

In the following example, we will learn how to use platform channel in Android to calculate the battery percentage of an Android phone using the Android `BatteryManager` API. You can read more about the API here: `https://developer.android.com/reference/android/os/BatteryManager`

Creating a new Flutter project

Please create a new Flutter project. Ensure that the Android SDK path has been set up properly and included in the project settings.

Creating a Flutter platform client

For understanding the battery level, Android code be written and then passed on to the Dart code. The applications' state class holds the current state of the app, and we need to extend it to ensure the current battery state is captured.

For this reason, we will use `MethodChannel` as the channel comprising of a single platform method that will return the battery level of the Android phone. The client and the host side communicate by a unique channel name passed in the channel constructor. In our case, we have named it `call.flutter.io/battery`:

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

...
class _MyHomePageState extends State<MyHomePage> {
```

```
  static const platform = const MethodChannel('call.flutter.io/battery');

  // Get Android battery level.
}
```

The next step is to invoke a method on the method channel, and we will use the returned result (the battery level) to update the value inside `setState`:

```
String _batteryLevel = 'Battery Levels are Unknown';

Future<void> _getPhoneBatteryLevel() async {
  String batteryLevel;
  try {
    final int result = await platform.invokeMethod('getBatteryLevel');
    batteryLevel = 'Battery level at $result % .';
  } on PlatformException catch (e) {
    batteryLevel = "Failed to get battery level: '${e.message}'.";
  }

  setState(() {
    _batteryLevel = _getBatteryLevel() asyncbatteryLevel;
  });
}
```

Finally, replace the build method to have a `Button` to have the action, `onPressed`, of which the result of the battery level would be shown as a `Text` value.

The complete `Main.Dart` code is shown here:

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


import 'dart:async';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Platform Channel API',
      theme: ThemeData(

        primarySwatch: Colors.yellow,
      ),
```

```
      home: MyHomePage(title: 'Flutter Platform Channel API'),
    );
  }
}

class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);


  final String title;

  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {

  static const platform = const MethodChannel('call.flutter.io/battery');
  String _batteryLevel = 'Battery Levels are Unknown';

  Future<void> _getPhoneBatteryLevel() async {
    String batteryLevel;
    try {
     final int result = await platform.invokeMethod('getBatteryLevel');
      batteryLevel = 'Battery level at $result % .';
    } on PlatformException catch (e) {
      batteryLevel = "Failed to get battery level: '${e.message}'.";
    }

    setState(() {
      _batteryLevel = batteryLevel;
    });
  }


  @override
  Widget build(BuildContext context) {
    return Material(
      child: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [

            Text ("Click the button to know your phone battery levels"),
            RaisedButton(
              child: Container(
                decoration: const BoxDecoration(
                  gradient: LinearGradient(
```

```
                    colors: <Color>[Colors.red, Colors.green,
Colors.brown],
                  ),
                ),
                padding: const EdgeInsets.all(10.0),
                child: Text('Get Phone Battery Level'),
              ),
              onPressed: _getPhoneBatteryLevel,
              textColor: Colors.white,
              padding: const EdgeInsets.all(0.0),

            ),
            Text(_batteryLevel),

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

Next, we will make changes to the Android code.

Making changes to MainActivity.Java

Navigate to the Android folder inside your project, and locate the `MainActivity.java` file in the Java folder in the project view. Next, inside the `onCreate` method in this file, we create a `MethodChannel` and a set a `MethodCallHandler` inside. Note that the channel name used should be the same on the Flutter client side, as in our case: `call.flutter.io/battery`.

Include the following imports:

```
package androcid.flutterapp1;

import android.os.Bundle;
import io.flutter.app.FlutterActivity;
import io.flutter.plugins.GeneratedPluginRegistrant;
import io.flutter.plugin.common.MethodCall;
import io.flutter.plugin.common.MethodChannel;
import io.flutter.plugin.common.MethodChannel.MethodCallHandler;
import io.flutter.plugin.common.MethodChannel.Result;

import android.content.ContextWrapper;
import android.content.Intent;
import android.content.IntentFilter;
```

```
import android.os.BatteryManager;
import android.os.Build.VERSION;
import android.os.Build.VERSION_CODES;
```

Next, create a `String` to hold the channel name:

```
private static final String CHANNEL = "call.flutter.io/battery";
```

And, finally, add the `MethodChannel` method:

```
new MethodChannel(getFlutterView(), CHANNEL).setMethodCallHandler(
                new MethodCallHandler() {
                    @Override
                public void onMethodCall(MethodCall call, Result result) {
                        // TODO
                    }
                 });
```

In the next step, we have to write the Android `onCreate` method:

```
private int getBatteryLevel() {
    int phoneBatteryLevel = -1;
    if (VERSION.SDK_INT >= VERSION_CODES.LOLLIPOP) {
      BatteryManager batteryManager = (BatteryManager)
getSystemService(BATTERY_SERVICE);
      phoneBatteryLevel =
batteryManager.getIntProperty(BatteryManager.BATTERY_PROPERTY_CAPACITY);
    } else {
      Intent intent = new ContextWrapper(getApplicationContext()).
              registerReceiver(null, new
IntentFilter(Intent.ACTION_BATTERY_CHANGED));
      phoneBatteryLevel = (intent.getIntExtra(BatteryManager.EXTRA_LEVEL,
-1) * 100) /
              intent.getIntExtra(BatteryManager.EXTRA_SCALE, -1);
    }

    return phoneBatteryLevel;
  }
}
```

To use `BatteryManager`, the minimum API is 21, and hence we use `VERSION_CODES.LOLLIPOP`.

Our final step is to complete the `onMethodCall` method added earlier. Using a single platform method, `getbatteryLevel`, you can simply call the Android code written in the previous step, and revert back the response, both in success as well as error cases using the response argument, as follows:

```
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  GeneratedPluginRegistrant.registerWith(this);
  new MethodChannel(getFlutterView(), CHANNEL).setMethodCallHandler(
          new MethodCallHandler() {
            @Override
            public void onMethodCall(MethodCall call, Result result) {

                if (call.method.equals("getBatteryLevel")) {
                  int batteryLevel = getBatteryLevel();

                  if (batteryLevel != -1) {
                    result.success(batteryLevel);
                  } else {
              result.error("Currently unavailable", "Battery level not
              available currently.", null);
                  }
                } else {
                  result.notImplemented();
                }

            }
          });
}
```

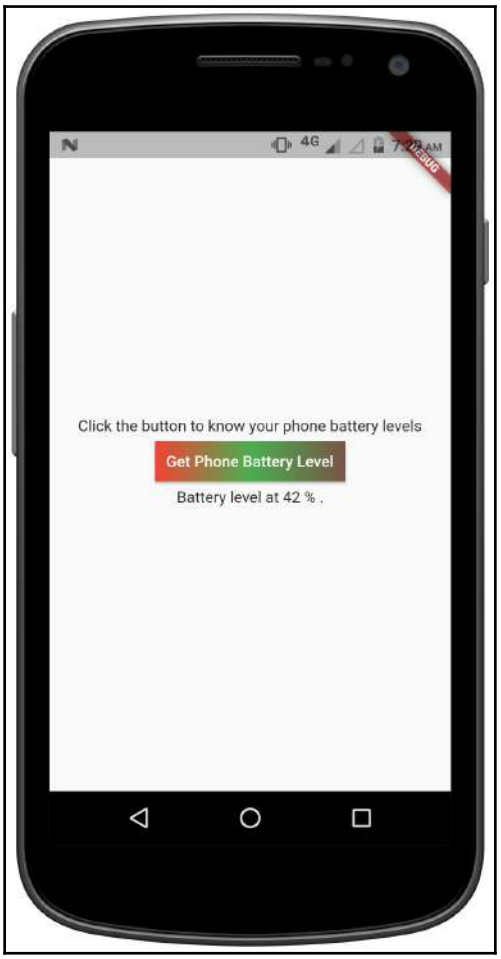
The following is the complete Android code:

```
package androcid.flutterapp1;

import android.os.Bundle;
import io.flutter.app.FlutterActivity;
import io.flutter.plugins.GeneratedPluginRegistrant;
import io.flutter.plugin.common.MethodCall;
import io.flutter.plugin.common.MethodChannel;
import io.flutter.plugin.common.MethodChannel.MethodCallHandler;
import io.flutter.plugin.common.MethodChannel.Result;
import android.content.ContextWrapper;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.BatteryManager;
import android.os.Build.VERSION;
import android.os.Build.VERSION_CODES;
```

```
public class MainActivity extends FlutterActivity {
  private static final String CHANNEL = "call.flutter.io/battery";
  @Override

  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    GeneratedPluginRegistrant.registerWith(this);
    new MethodChannel(getFlutterView(), CHANNEL).setMethodCallHandler(
            new MethodCallHandler() {
              @Override
              public void onMethodCall(MethodCall call, Result result) {

                  if (call.method.equals("getBatteryLevel")) {
                    int batteryLevel = getBatteryLevel();

                    if (batteryLevel != -1) {
                      result.success(batteryLevel);
                    } else {
                result.error("Currently unavailable", "Battery level not
available currently.", null);
                    }
                  } else {
                    result.notImplemented();
                  }

              }
            });
  }

  private int getBatteryLevel() {
    int phoneBatteryLevel = -1;
    if (VERSION.SDK_INT >= VERSION_CODES.LOLLIPOP) {
      BatteryManager batteryManager = (BatteryManager)
getSystemService(BATTERY_SERVICE);
      phoneBatteryLevel =
batteryManager.getIntProperty(BatteryManager.BATTERY_PROPERTY_CAPACITY);
    } else {
      Intent intent = new ContextWrapper(getApplicationContext()).
              registerReceiver(null, new
IntentFilter(Intent.ACTION_BATTERY_CHANGED));
      phoneBatteryLevel = (intent.getIntExtra(BatteryManager.EXTRA_LEVEL,
-1) * 100) /
              intent.getIntExtra(BatteryManager.EXTRA_SCALE, -1);
    }

    return phoneBatteryLevel;
  }
}
```

After running the code successfully, you will see the following output:

When clicking on the button, the **Battery Level** of the phone will be displayed:



Building and publishing your own plugin

Plugins play an important role in the Flutter ecosystem. Google developers and open source contributors have contributed to several published plugins. We have seen in the `Chapter 5`, *Widening our Flutter Horizons*, show to include the plugins in the code. When building a plugin, here are some of the tips to keep in mind:

1. **Check for existing plugin availability**: See whether there are existing plugins available that function the same/similar way, before you code from fresh.
2. **Think Dart**: Since Flutter code is written in Dart, it's ideal if the major logic is crafted in Dart, which not only makes it easy to navigate but also processing the code is easier, across the platform.
3. **Avoid a building platform-specific plugin that is only supports**: The developer might be tempted to start building a plugin including every feature they might desire, but thinking solo-platform might not be such a good idea. Not only will it confuse the users about its use cases on a specific platform, it will also make the app behave in an unexpected way.
4. **Avoid building platform-specific API methods**: As a developer, you might be tempted to build a platform-specific method, but this can go into overkill. Try including the platform-specific logic to the plugin itself.
5. **Build for features**: Ensure that the plugin you are planning to build has a specific use case providing features, rather than just calling existing APIs. There is nothing wrong with using an existing native library, but the problem arises when these APIs don't work as expected across different platforms. Focusing on features more than on API could help you build better plugins.
6. **Detail out the features and installation properly**: When your plugin is published, support the community by giving out more details about the plugin's features and how they seamlessly include this plugin into their code. The visibility of the plugin matters, as if the community loves the plugin, they will rate the plugin higher, thus increasing the visibility. Use examples so that there are no problems regarding including your plugin.

To publish your own plugin, use the following command:

```
flutter packages pub publish --dry-run
```

If there are no errors in the execution, you can execute the following command to publish the plugin and your plugin will be live in a few minutes:

```
flutter packages pub publish
```

Summary

At the start of the chapter, we learned how to include packages in the Flutter code, followed by how to include platform-specific channels to support Flutter code. We also used the `BatteryManager` API to understand the battery state of the Android phone. In the last section, we covered some of the best tips to consider you build your our own plugin, followed by how to publish your own plugin.

In the next chapter, we will take a look at how to use **Firebase** with Flutter.

7
Firebase - Flutter's Best Friend

Building apps using **Firebase** is one of the fastest-growing technology trends in the world. Using Firebase, developers can build apps at a rapid pace without managing the infrastructure, including authentication, storing and syncing data, securely hosting web assets, and cloud storage. Firebase has a base plan that is free, allowing 1 GB the storage and 100 simultaneous connections. If you wish to upgrade, you can check out the plans here: `https://firebase.google.com/pricing/`.

In this chapter, we will cover the following topics:

- Connecting with Firebase
- Creating the Cloud Firestore database
- Firebase Cloud Messaging
- Firebase Remote Config

Connecting with Firebase

Let's first take a look at how to connect with Firebase. We will first need to ensure whether the connections to the Firebase are made properly; to do so, follow these steps:

1. Create a new Flutter project in your IDE or editor
2. Open the file `pubspec.yaml` file
3. Add the following dependency:

```
dependencies:
  flutter:
    sdk: flutter
  cloud_firestore: ^0.9.5+2  //Add this line
```

For details of the latest version of the Cloud Firestore plugin for Flutter, visit the Pub site: `https://pub.dartlang.org/packages/cloud_firestore`.

4. Next, to make your connection, in your IDE or using command line run the following:

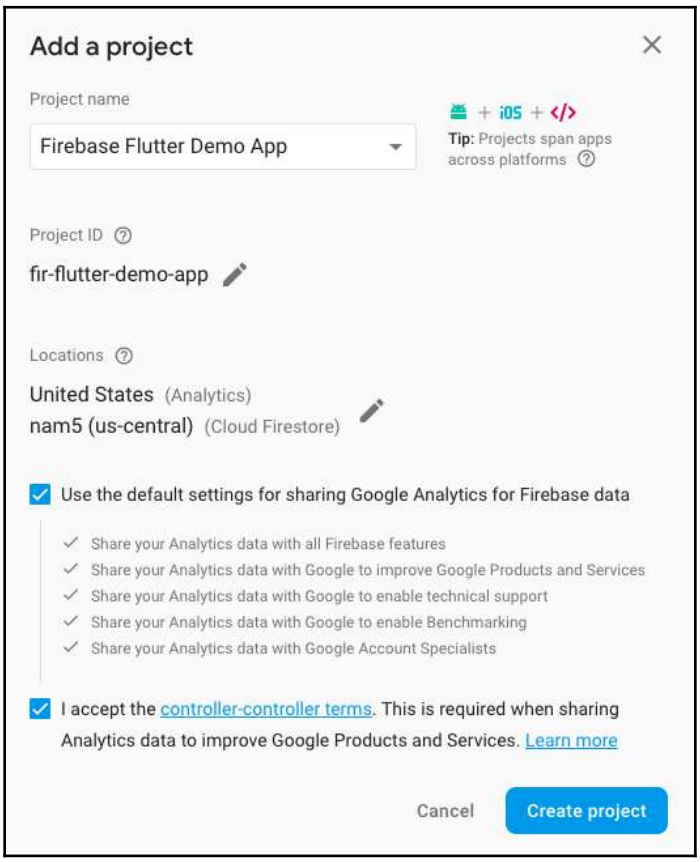
```
flutter packages get
```

Creating a Firebase project

Once the connection has been made, the next step will be to create a Firebase project. So, let's get started. Follow the given steps to create your project:

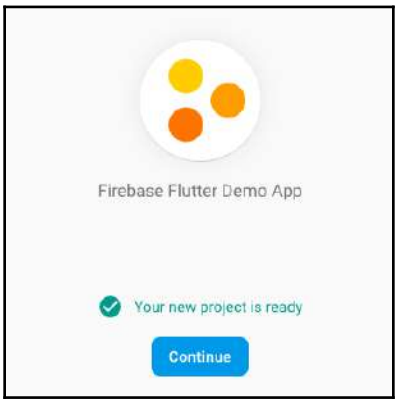
1. Open the website `https://firebase.google.com` and log in or sign up. You can use your Google credentials to log in here.
2. Next, click on Add Project.

3. Once you click this option, you will see the following screen:

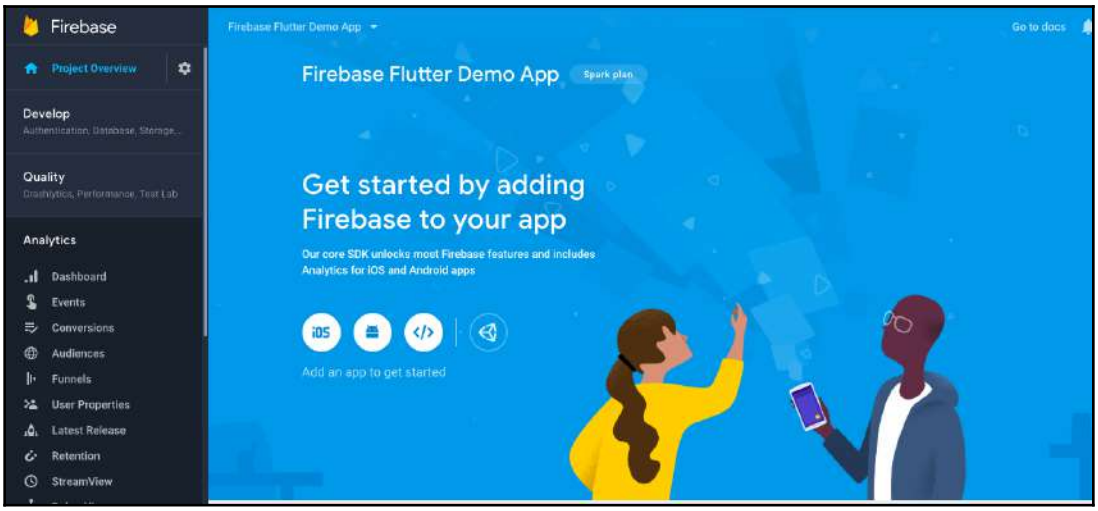


4. Add a **Project name** (for example: `Firebase Flutter Demo App` in our case). The **Project ID** gets auto-generated, or you could type a unique project ID of your own. They are globally unique identifiers.
5. Select the country in **Locations** and then proceed to accept the terms and conditions before you click **Create project.**

6. Click the **Create project** option, and wait a few seconds before the Firebase console shows the message displayed in the following screenshot:



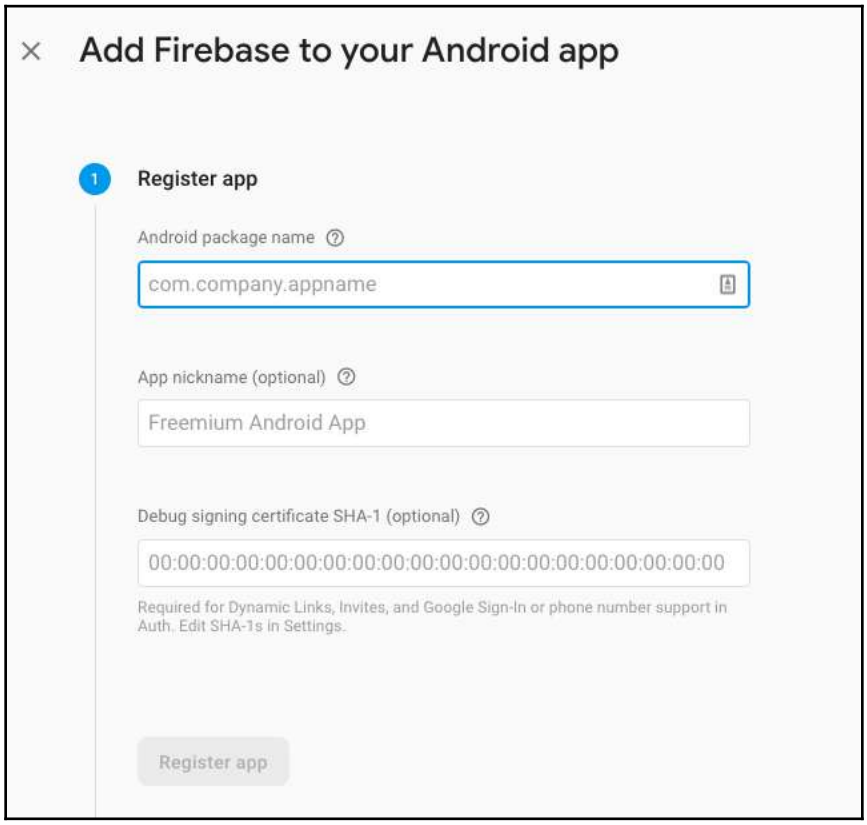
7. If your screen shows the **Your new project is ready** text, as shown in the preceding screenshot, you can click the **Continue** button.
8. Once that is done, you will be shown the project settings dashboard of the app as follows:



9. Choose platform-specific Firebase configurations, based on which app platform you will be building an app for, and click on the respective icon. In our case, since we are building an Android app, we will click on the Android icon to proceed.

Registering an app using a package name

This step is needed to register your app's platform-specific ID with Firebase. This will generate configuration files that we will add to our project folders. Note that in the top-level directory of your Flutter app, iOS and Android are two of the subdirectories that hold the respective platform-specific configuration files:



In the top-level directory of your Flutter app, you can see subdirectories; called Android and iOS. Here,you'll find platform-specific configuration files for iOS and Android.

The most important field here is the **Android package name.** This is generally the `applicationId` in your app-level `build.gradle` file. Another way to find the package name is to follow these steps:

1. In the Flutter app directory, check the `android/app/src/main/AndroidManifest.xml` file.
2. Under the Manifest tag, find the string value of the package, which will be the value of the package name.
3. In the Firebase dialog, paste the copied package name from step 2 into the **Android package name** field.

If you are developing the Flutter app for both iOS and Android, you need to register both the iOS and Android versions within the same Firebase project. But if you are developing it just for one platform, you can just click one of them.

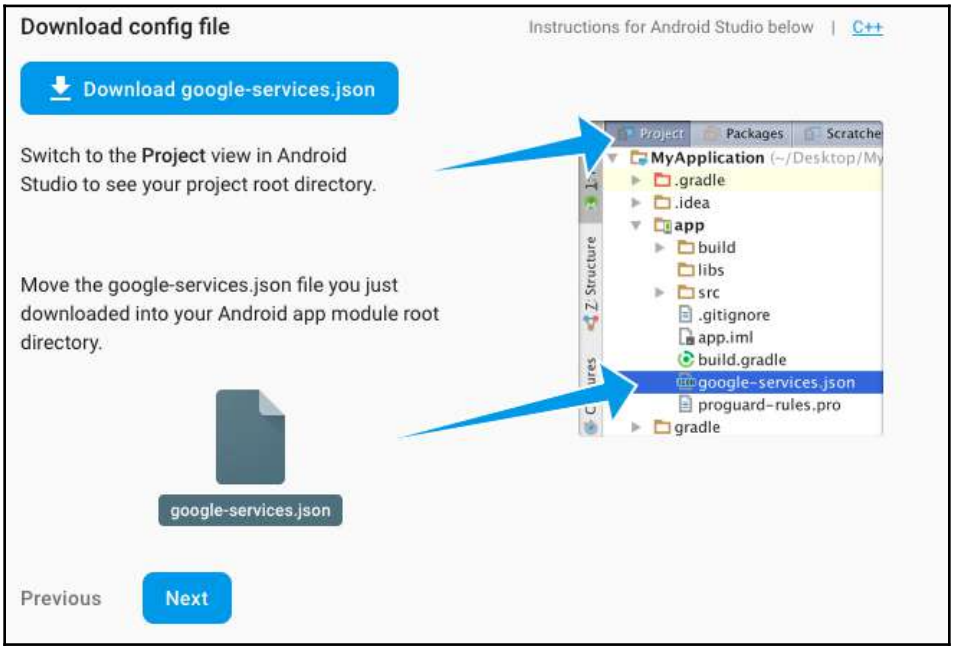
Next, you can add the **App nickname**, which is an optional field. There is another optional field **Debug signing certificate SHA-1,** which has to be used in the same cases if the app uses features such as Google Sign in for Authentication, Dynamic Links, and Invites. In this case, you have to find the debug certificate fingerprint value that you can grab and paste in the field. Refer to the link here, `https://developers.google.com/android/guides/client-auth`, for understanding how to build client auth. Since, in this example, we are not using any of these features, we will leave it blank. Click on **Register app**.

Downloading and setting up the config file

The next part will be to download and set up the config file. Follow the given steps to to download and set up the config file:

1. After clicking **Register app**, the console in this step will generate the `google-services.json` file. Download this file to your computer.
2. Once the file has been downloaded, go to your Flutter app directory, and move the `google-services.json` file that you downloaded previously into the `android/app` directory.

3. After the file has been moved, in the Firebase console, click **Next** as shown in the following screenshot:



Adding Firebase SDK

Now that we have downloaded and set up the config file, the penultimate step is to add Firebase SDK to your project. The Google services plugin for Gradle ensures that the JSON file you downloaded is read. In order to enable Google APIs or Firebase services in your application, you have to add a `google-services` dependency. Two minor modifications are needed to `build.gradle` files to use the plugin. Take a look at the following:

1. Project-level `build.gradle` (`<project>/build.gradle`):

```
buildscript {
dependencies
{
// Add this line
classpath 'com.google.gms:google-services:4.2.0'
}
}
```

2. App-level `build.gradle` (`<project>/<app-module>/build.gradle`):

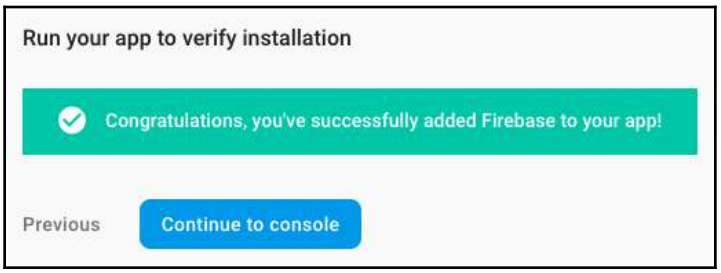
```
dependencies {
  // Add this line
  implementation 'com.google.firebase:firebase-core:16.0.1' }
...
// Add to the bottom of the file
apply plugin: 'com.google.gms.google-services'
```

3. Click the **Sync Now** option to complete this process.

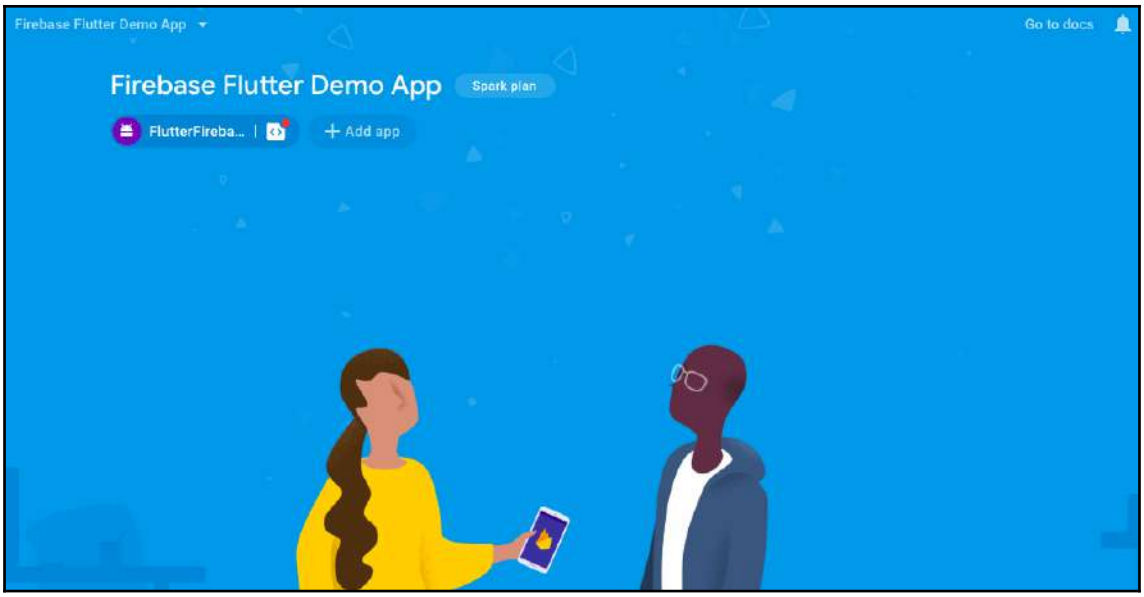
Verifying the configuration

Once the previous steps are complete, we have to verify whether your Flutter app is connected to Firebase. To ensure this, follow these steps:

1. Build the project and run it on the device connected to your computer.
2. Once the app gets run on the phone, the Firebase console automatically detects the configurations and displays a success message as follows:



3. After you click **Continue to console**, you will be taken to the console showing the project name and the other settings:



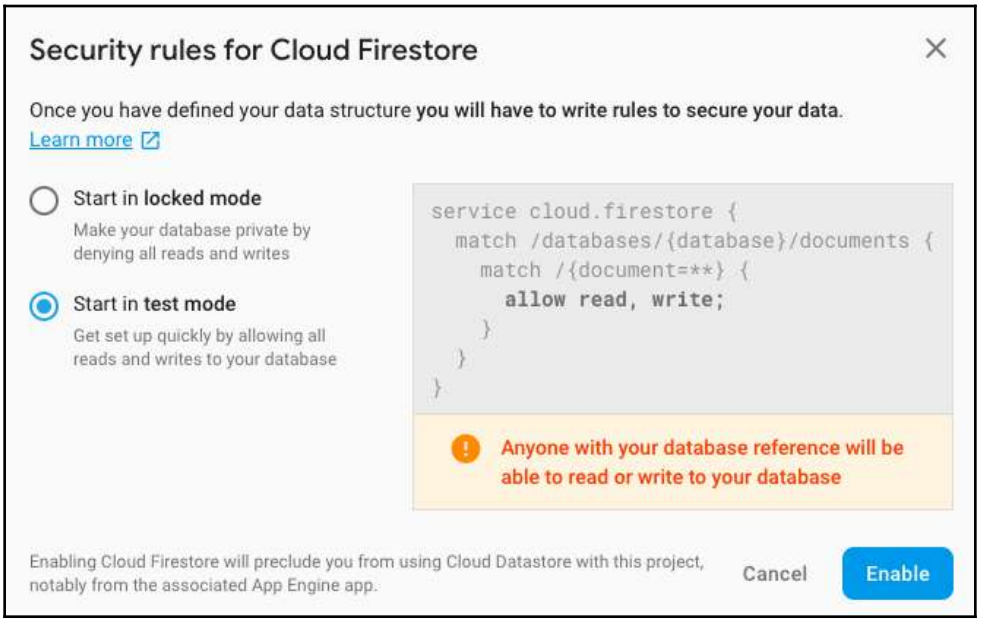
In the next section, we will see how to connect to the Cloud Database.

Creating a Cloud Firestore Database

Once the Firebase-Flutter set up is complete, you are all set to build the app. We will now set up a Cloud Firestore database and initialize some values. Follow these steps:

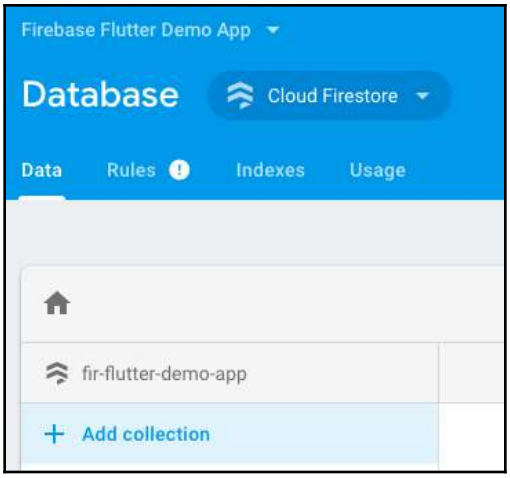
1. Under the **Develop** option, click on **Database.**
2. In the panel shown, click **Create database:**

3. After clicking, you will see a pop-up panel: **Security rules for Cloud Firestore**. Select **Start in test mode** and enable it:

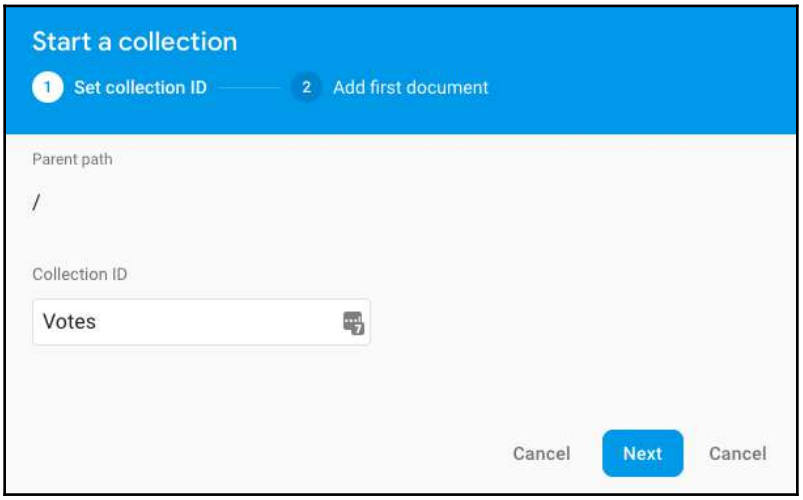


4. We select test mode because we want anyone with the database reference to be able to read or write to the database. When you build the production version of the app, ensure you set up security rules. You can read about these rules here: `https://firebase.google.com/docs/reference/rules/rules`. After clicking **Enable**, the Cloud Firestore will be provisioned with security rules and will be ready for use.

5. From the following panel, click **Add collection**:

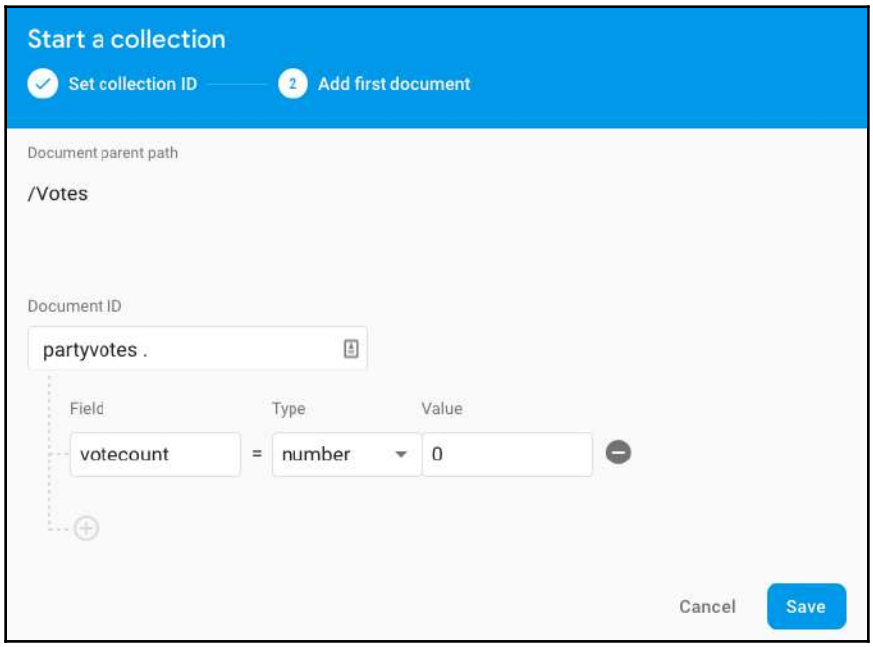


6. We presume that we would have just one collection in Firestore, called **Votes**. A **collection** is a set of documents that comprise the data:



7. Click **Next**.
8. A collection must contain at least one document, which is Cloud Firestore's unit of storage. You can either use an auto-generated ID or have a custom ID. In our case, we use **partyvotes**.

9. For the existing **Field**, enter the value of the name (in our case, it's VoteCount), select the data **Type**, then enter the **Value** of **partyvotes**. Since its an Integer, we select the number and set its initial value to be **0**:



10. Click **Save**.
11. After adding several documents to your collection, your database should look something like this:

Firestore is a NoSQL database, which means that we would not be working with rows and columns. Now we will build the layout of the app. Using the Firestore details, we will construct the list layout, which will generate the list items runtime based on the values in the Firestore and read/update the fresh values when tapped on the list item into the Firestore database. The following is the `main.dart` file:

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

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

// Creating Object to temporary make the list items. We will replace it
when we connect it to Firestore

final party = [
 {"partyname": "BJP", "rating": 1},
 {"partyname": "Congress", "rating": 3},
 {"partyname": "AAP", "rating": 5},
 {"partyname": "Janata Dal Party", "rating": 9},
 {"partyname": "NOTA", "rating": 11},
];

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'State Party Elections - Worker Profile',
     home: MyHomePage(),
   );
 }
}

class MyHomePage extends StatefulWidget {
 @override
 _MyHomePageState createState() {
   return _MyHomePageState();
 }
}

class _MyHomePageState extends State<MyHomePage> {
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(title: Text('Party Votes')),
     body: _buildBody(context),
   );
 }
```

```
 Widget _buildBody(BuildContext context) {
   // We will add the code here in the next section
   return _buildList(context, party);
 }

 Widget _buildList(BuildContext context, List<Map> snapshot) {
   return ListView(
     padding: const EdgeInsets.only(top: 22.0),
     children: snapshot.map((data) => _buildListItem(context,
data)).toList(),
   );
 }

 Widget _buildListItem(BuildContext context, Map data) {
   final result = Record.fromMap(data);

// Adding the padding to ensure enough space is given
   return Padding(
     key: ValueKey(result.name),
     padding: const EdgeInsets.symmetric(horizontal: 15.0, vertical: 7.0),
     child: Container(
       decoration: BoxDecoration(
         border: Border.all(color: Colors.red),
         borderRadius: BorderRadius.circular(6.0),
       ),
// Showing the list item, with name towards the left and the votes to the
right

       child: ListTile(
         title: Text(record.partyname),
         trailing: Text(record.partyvotes.toString()),
         onTap: () => print(record),
       ),
     ),
   );
 }
}

class Record {
 final String partyname;
 final int partyvotes;
 final DocumentReference reference;

 Record.fromMap(Map<String, dynamic> map, {this.reference})
     : assert(map['partyname'] != null),
       assert(map['partyvotes'] != null),
       name = map['partyname'],
       votes = map['partyvotes'];
```

```
 Record.fromSnapshot(DocumentSnapshot snapshot)
     : this.fromMap(snapshot.data, reference: snapshot.reference);

 @override
 String toString() => "Record<$partyname:$partyvotes>";
}
```

We have the collection ready on Firestore Cloud. In the previous example, we have used the `party` object. It's time we now use the Firestore cloud data from our collection to be shown. We can do that by calling Cloud Firestore using a `Firestore.instance` reference. For example, if you wish to call a specific collection from your Firestore Cloud database, you can use the following command to return a stream of snapshots:

```
Firestore.instance.collection('collection_name').snapshots()
```

Streams are of two types: single subscription or broadcast. Streams are responsible for providing an asynchronous sequence of data. User-generated events and data read from files are the two data sequences. Now, using `StreamBuilder` widget, we will inject the stream of data into the user interface we have created. One of the classic use cases of the `StreamBuilder` is that, whenever there is a change in the Firestore values, the list gets updated automatically.

Look for the `_buildBody` method in the previous code and replace the content with this code:

```
Widget _buildBody(BuildContext context) {
 return StreamBuilder<QuerySnapshot>(
   stream: Firestore.instance.collection('party').snapshots(),
   builder: (context, snapshot) {
     if (!snapshot.hasData) return LinearProgressIndicator();
     return _buildList(context, snapshot.data.documents);
   },
 );
}
```

Adding the preceding snippet will produce some errors. The `_buildListItem` method still thinks it's getting a map. Hence, we will need to make a couple of changes.

Firstly, make the method to accept `DocumentSnapshot` instead of a list of a map:

```
Widget _buildList(BuildContext context, List<DocumentSnapshot> snapshot)
{ ....
}
```

Secondly, use the constructor `Record.fromSnapshot()` to build the record. The method's updated code as follows:

```
Widget _buildListItem(BuildContext context, DocumentSnapshot data) {
 final result = Record.fromSnapshot(data);
```

Next, use the `onTap: ()` method to ensure whenever a list item is clicked, the votes are updated into the Firestore database. Whenever you click **List Item**, Cloud Firestore notifies all listeners with the updated snapshot. The app is actively engaged using `StreamBuilder`, which acts to update with the new data. For a single user, it is fine, but when you have multiple users, there is a chance of **Race Condition** may occur.

The complete code for `main.dart` is as follows:

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

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

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'State Party Elections - Worker Profile',
     home: MyHomePage(),
   );
 }
}

class MyHomePage extends StatefulWidget {
 @override
 _MyHomePageState createState() {
   return _MyHomePageState();
 }
}

class _MyHomePageState extends State<MyHomePage> {
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(title: Text('Party Votes')),
     body: _buildBody(context),
   );
 }

 Widget _buildBody(BuildContext context) {
   return StreamBuilder<QuerySnapshot>(
```

```
      stream: Firestore.instance.collection('party').snapshots(),
      builder: (context, snapshot) {
        if (!snapshot.hasData) return LinearProgressIndicator();

        return _buildList(context, snapshot.data.documents);
      },
    );
  }

  Widget _buildList(BuildContext context, List<DocumentSnapshot> snapshot) {
    return ListView(
      padding: const EdgeInsets.only(top: 22.0),
      children: snapshot.map((data) => _buildListItem(context,
data)).toList(),
    );
  }

  Widget _buildListItem(BuildContext context, DocumentSnapshot data) {
    final result = Record.fromSnapshot(data);

    return Padding(
      key: ValueKey(result.name),
      padding: const EdgeInsets.symmetric(horizontal: 15.0, vertical:
      7.0),
      child: Container(
        decoration: BoxDecoration(
          border: Border.all(color: Colors.red),
          borderRadius: BorderRadius.circular(6.0),
        ),
        child: ListTile(
          title: Text(record.name),
          trailing: Text(record.votes.toString()),
          onTap: () => Firestore.instance.runTransaction((transaction)
          async {
            final freshFBsnapshot = await
             transaction.get(record.reference);
            final updated = Record.fromSnapshot(freshFBsnapshot);

               await transaction
                   .update(record.reference, {'partyvotes':
            updated.votes + 1});
             }),
        ),
      ),
    );
  }
}
```

```
class Record {
 final String partyname;
 final int partyvotes;
 final DocumentReference reference;

 Record.fromMap(Map<String, dynamic> map, {this.reference})
     : assert(map['partyname'] != null),
       assert(map['partyvotes'] != null),
       name = map['partyname'],
       votes = map['partyvotes'];

 Record.fromSnapshot(DocumentSnapshot snapshot)
     : this.fromMap(snapshot.data, reference: snapshot.reference);

 @override
 String toString() => "Record<$partyname:$partyvotes>";
}
```

Once you run the code, try clicking on List Items and you will see the updates values mapped on the Firestore Cloud database. You could also try updating the List Item names (in our case, the Party names) in the Firestore Cloud database, and you will see the List Item option updating.

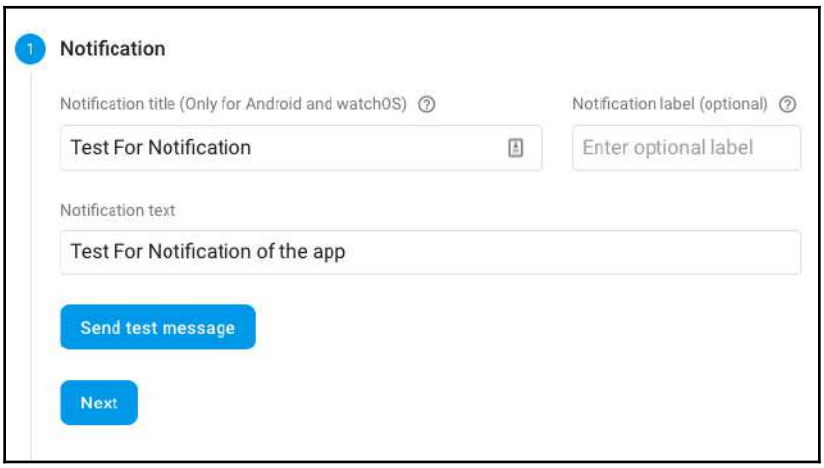
Firebase Cloud Messaging

Firebase Cloud Messaging (**FCM**) is an effective way to drive engagement within the app using the app notification. Using FCM, you can send two kinds of messages to the client device:

1. Notification messages that are directly handled by FCM SDK
2. Data messages

Both these messages have a maximum payload of 4KB. When sending messages from the Firebase console, there is a 1,024 character limit. Firebase has Cloud Messaging as well as In-App messaging, but in this section, we will discuss only Firebase Cloud messaging.

In the Firebase console, click on **Grow | Cloud Messaging** in the left panel. Follow this by clicking **Send your first message.** as shown in the following screenshot:



To test the message on your device, FCM tokens are needed. Use the following Android code to generate these tokens:

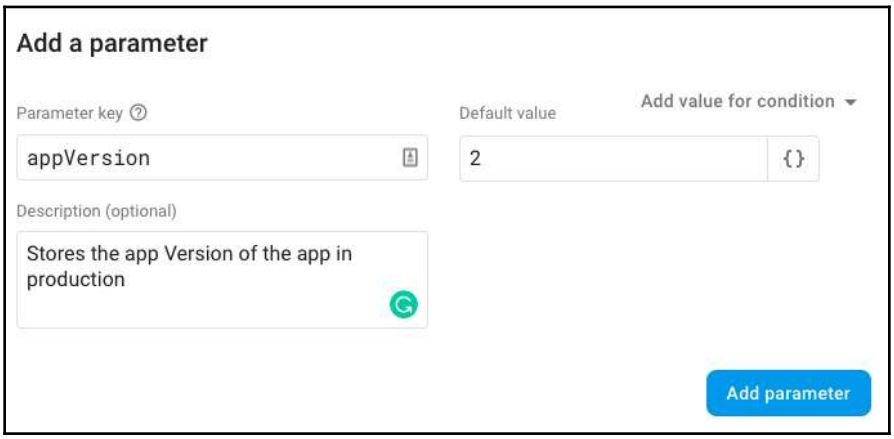
```
FirebaseInstanceId.getInstance().getInstanceId()
        .addOnCompleteListener(new OnCompleteListener<InstanceIdResult>
         () {
            @Override
            public void onComplete(@NonNull Task<InstanceIdResult>
            actionable) {
                if (!actionable.isSuccessful()) {
                    Log.w(TAG, "getInstanceId failed",
                    actionable.getException());
                    return;
                }

                // Get new Instance ID token
                String tokenID = actionable.getResult().getToken();

                // Log and toast
                String message = getString(R.string.msg_token_fmt,
                tokenID);
                Log.d(TAG, message);
                Toast.makeText(MainActivity.this, message,
               Toast.LENGTH_SHORT).show();
            }
        });
```

FCM also allows configuring messages specific to targets such as `GeoLocations`, versions of the app, Languages, and User Audiences. This case is ideal when you wish to send notifications to a specific set of users.

Firebase Remote Config

Using Firebase's Remote Config API, you can make changes to the app without the user actually downloading an app update. One example of this is when you push a new update to the app in production, you can show the pop-up message to the user when they launch the app about the update.

To set up Firebase Remote Config, head over to the **Grow** tab in the Firebase console and click on **Remote Config** as shown in the following screenshot:



Add the **Parameter key** and the **Default value**. There is an optional field for adding the description. Click **Add Parameter** to proceed. Avoid storing any confidential information in the remote config. Firebase also allows the setting of conditions for the parameter. For example, if you want to show a specific welcome message to a user in India and a different message to a user in the USA, remote configuration can come in handy.

Disclaimer: Some of the code files are are licensed under the `Apache 2.0 License` and are available on `https://firebase.google.com/docs/cloud-messaging/android/client`.

Summary

We started this chapter by looking at how Firebase can help us build apps quicker using the Firestore Cloud database—the NoSQL way to help app developers build apps in real-time. We also looked at an example that captured `ListView` with the Firestore Cloud database. This section was followed by looking at how Cloud messaging works. In the last section, we discussed some use cases for using Firebase Remote Config for your apps.

In the next chapter, we will take a look at how to deploy your Flutter applications.

8
Deploying Flutter Apps

Deploying the Flutter app is one of the easiest processes to enable developers to prepare the apps for publishing on app stores. By now, you must have learned Google's mobile app SDK provides several features to craft high-quality native interfaces on iOS and Android, in record time.

In this chapter, we will look at the following topics:

- Deploying on Android
- Deploying on iOS

Deploying on Android

In the course of this book so far, the build that we have built is a debug type. Ideally, this is used for testing the app before following the steps to produce the release version of the app to be uploaded. Flutter also allows creating flavors of the app. If you wish to build a production version of the app you have developed, follow the upcoming steps.

Reviewing the AndroidManifest.xml file

This file holds some of the major global settings of the app that come in handy when building the production version of the app. It can be located at `<app dir>/android/app/src/main`. When you click on the `AndroidManifest.xml` file, you will find the snippet in the `Application` tag shown as follows:

```
…..
android:name="io.flutter.app.FlutterApplication"
android:label="flutter_app_battery"
android:icon="@mipmap/ic_launcher">
….
```

The explanation for the properties visible in the preceding code are as follows:

- `android.name`: This property sets the package name for the app
- `android.label`: This property sets the final name of the app
- `android:icon`: This property sets the launch icon for the app

```
….
<uses-permission android:name="android.permission.INTERNET"/>
…..
```

The `<uses-permission>` tag allows developers to set the permission needed for the developers in the app. For example, if you wish to use the internet, the preceding property has to be used, or if you wish to access the camera, `<uses-permission android:name="android.permission.CAMERA"/>` has to be used. This will manifest element for all camera features. The developers can also demand to ask the users for the permission access in the **runtime** mode.

The build.gradle configurations

The next step is to review the Gradle build file situated in `<app dir>/android/app` and to confirm whether the values entered in the following parameters are correct:

- Set the `VersionCode` and `VersionName` in the following snippet. Please note that the `VersionCode` value should be unique for every uploaded build and that it is an absolute value. The greatest value the Google Play store allows for `versionCode` is `2100000000`. On the other hand, `VersionName` is a string value. There is no purpose of `VersionName` to show up the value on the play store. The string value can be cascaded as a `<major>.<minor>.<point>` string—for example, 1.2.2:

```
…….
…….
def flutterVersionCode =
localProperties.getProperty('flutter.versionCode')
if (flutterVersionCode == null)
{ flutterVersionCode = '1' }
 def flutterVersionName =
localProperties.getProperty('flutter.versionName')
if (flutterVersionName == null)
{ flutterVersionName = '1.0' }
……
……
```

- `applicationId`: This is to enable developers to specify the final, unique application ID.
- `minSdkVersion` and `targetSdkVersion`: These two values specify the minimum API level and the target API level on which the app is designed to run on:

```
defaultConfig
{ // TODO: Specify your own unique Application ID
(https://developer.android.com/studio/build/application-id.html).
applicationId "deviceinformation.flutterappbattery"
minSdkVersion 16
targetSdkVersion 27
versionCode flutterVersionCode.toInteger()
versionName flutterVersionName
testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
}
```

Icons within apps

A trendy, eye-catching icon can be a great trigger for someone to launch the app. By default, the launcher icon is a default icon. By adhering to the Android Launcher Icon guidelines, you can build your own icon that can be used for the app's launch trigger from the mobile screen:

1. Once your icon files are ready, check the `<app dir>/android/app/src/main/res/` directory and place the files in the respective folders using configuration qualifiers. You can read more about this here: `https://developer.android.com/guide/topics/resources/providing-resources#AlternativeResources`.
2. Once you have placed the files in the folders, simply head to `AndroidManifest.xml` and update the application tag's `android:icon` attribute.
3. To ensure the icon has been replaced, `Flutter run` and inspect the app icon in the launcher.

Signing the app

This is one of the key steps before publishing the app on the Google Play store. To publish the app, signing the app using a digital signature is a key part. Follow these steps to sign the app:

1. **Create a keystore**: If you already have a `keystore`, skip to step 2. If you wish to build the new `keystore`, use the `KeyTool` tool to generate one using this command-line code:

```
keytool -genkey -v -keystore ~/appkey.jks -keyalg RSA -keysize 2048
-validity 10000 -alias appkey
```

   `KeyTool` is part of the Java JDK, which is installed as part of Android Studio. Ensure you give an absolute path before running the command line. Also note that the file generated has to be kept private.

2. **Reference keystore from the app**: Next, create a file named `<app dir>/android/key.properties` that contains a reference to your keystore. Keep this file private. Take a look at this code:

```
storePassword=<password used in the previous step>
keyPassword=<password used in the previous step>
keyAlias=appkey
storeFile=<location of the key store file, e.g. /Users/<user
name>/appkey.jks>
```

3. **Configure signing in Gradle**: Head to the `<app dir>/android/app/build.gradle` file and replace **`android {`** with the following code:

```
def keystoreProperties = new Properties()
def keystorePropertiesFile = rootProject.file('key.properties')
if (keystorePropertiesFile.exists()) {
    keystoreProperties.load(new
FileInputStream(keystorePropertiesFile))
}
android {
```

4. Next, remove the following code:

```
buildTypes {
    release {
        // TODO: Add your own signing config for the release build.
        // Signing with the debug keys for now, so `flutter run --
        //release` works.
```

```
            signingConfig signingConfigs.debug
        }
    }
```

Replace it with this code:

```
    signingConfigs {
        release {
            keyAlias keystoreProperties['keyAlias']
            keyPassword keystoreProperties['keyPassword']
            storeFile file(keystoreProperties['storeFile'])
            storePassword keystoreProperties['storePassword']
        }
    }
    buildTypes {
        release {
            signingConfig signingConfigs.release
        }
    }
```

Once these steps are performed, your app's release builds will be signed automatically.

Using ProGuard

By default, Flutter build generation does not obfuscate or minify Android Host. You may want to reduce the size of the APK or save the code from reverse engineering. **ProGuard** is one such way to protect your code:

1. **Configure ProGuard:** Create a new file called `/android/app/proguard-rules.pro` and add these rules:

```
#Flutter Wrapper
-keep class io.flutter.app.** { *; }
-keep class io.flutter.plugin.**  { *; }
-keep class io.flutter.util.**  { *; }
-keep class io.flutter.view.**  { *; }
-keep class io.flutter.**  { *; }
-keep class io.flutter.plugins.**  { *; }
```

   Using the preceding code, you can protect only the engine libraries in Flutter. For protecting the others, add the code as per your development needs.

2. **Enable obfuscation and/or minification**: Open the `/android/app/build.gradle` file and locate the `buildTypes` definition:

```
 …..
 …..
buildTypes {
        release {
            // TODO: Add your own signing config for the release
build.
            // Signing with the debug keys for now, so `flutter run
--release` works.
            signingConfig signingConfigs.debug
        }
…..
…..
```

Inside, release the configuration set `minifyEnabled` and `useProguard` flags to true. Note also point ProGuard to the file you have created in step 1. The refreshed code will look like this:

```
…
…
buildTypes {
        release {
            signingConfig signingConfigs.release
            minifyEnabled true
            useProguard true
            proguardFiles getDefaultProguardFile('proguard-
android.txt'), 'proguard-rules.pro'
        }
…
…
```

Building a release APK

Generating a release is just a two-step process after you have successfully completed the preceding steps. Using the command line, perform the following:

1. `cd <app dir>` (note to replace `<app dir>` with your application's directory path).
2. Run **flutter build apk**. This will create a release APK at `<app dir>/build/app/outputs/apk/release/app-release.apk`.

This build can be published on the Google Play store. Ensure you read the publishing guidelines before the app is uploaded.

Deploying on iOS

As in the case of the Google Play store, Apple follows its own app publishing guidelines as well. Please be sure to read all the information regarding the same, before building the app. Here is the link you can check to read more details about Apple app publishing: `https://developer.apple.com/app-store/review/`. Once the app is submitted, as in the case of Google as well, Apple will check the app to adhere to its publishing guidelines. Note that Flutter supports iOS 8.0 and later. This is important to know when we set the Xcode for build generation.

As in the case of Google, we use the **Google Play Developer** console. In the case of Apple, we will use **App Store Connect**, which was previously known as iTunes. This console is used to manage your app's life cycle. This console will help you set the app's name, descriptions, and app screenshots, which will be published along with the app, pricing, and manage releases.

Registering Bundle ID

Every app that is published on the Apple store has a unique Bundle ID that is identified with Apple. To register a new Bundle ID for your app, follow these steps:

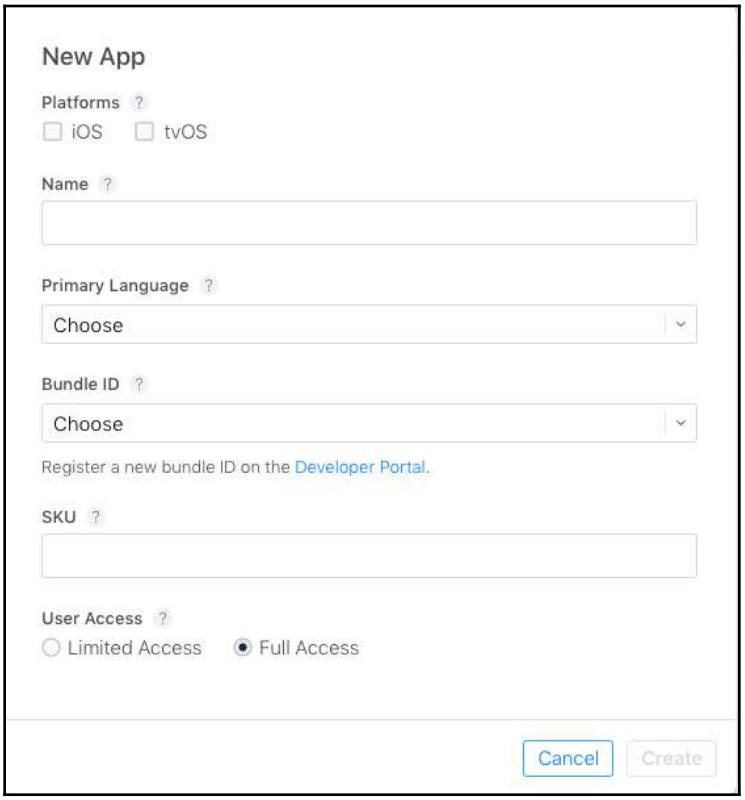
1. Open the App IDs page of your Apple developer account.
2. Click on the + icon to create a new Bundle ID.
3. Type an app name and select **Explicit App ID**, and enter an ID.
4. Select the services your app is going to use and then click **Continue**.
5. The next step confirms the details. Now, click Register to register your Bundle ID.

Generating an application record on App Store connect

To register an app on the **App Store connect**, please follow these steps:

1. Open Apple **App Store connect** in your browser and click **My Apps**.
2. Click on the **+** icon at the top-left corner of the **My Apps page** | **New App**.

3. Within the pop-up screen, fill out your app's details. In the **Platforms** section, ensure that **iOS** is checked. At this point, it is worth mentioning that Flutter does not, as yet, support **tvOS**. So, leave that checkbox unchecked. The name of the app can't be longer than 30 characters. In the **SKU** section, add a unique ID for your app that is not visible in the App Store:



4. Check **Create**.
5. Navigate to the application details for your app that was created using the previous steps, and select **App Information** from the sidebar.
6. Select the **Bundle ID** in the **General Information** section.

Verifying the Xcode settings

Verifying build-publishing settings in Xcode is rather simple compared to that in Android Studio. Firstly, navigate to the target's settings in Xcode and do the following:

1. Open `Runner.xcworkspace` in your app's ios folder, in Xcode.
2. Select the **Runner** project from the Xcode project navigator, which shows up the app's settings. Select the **Runner** target from the main view's sidebar.
3. Select the **General** tab.

The displayed information will need your attention to cross-check the important settings; so in the **Identity** section, look at the following details:

- **Display Name**: This is the name of the app that will be displayed in the App Store and anywhere else where the name is used
- **Bundle Identifier**: This is the app ID you registered on App Store Connect, as discussed in the previous steps

In the *Signing the app* section, please take a look at the following details:

- **Automatically manage signing**: Defines whether Xcode should automatically manage app signing and provisioning. By default, it is set to `True`.
- **The Team**: Select the team associated with your registered Apple Developer account. If you wish to add some more members, click on **Add Account**, followed by updating the settings.

Finally in the **Deployment** section, check the **Deployment Target:** that holds the value for the minimum iOS version your app will support.

Choosing the app icon

As in the case of Android Studio, even in the case of iOS, a placeholder icon is created. If you wish to have your own icon, please read the iOS app icon guidelines before proceeding with the following steps:

1. Select `Assets.xcassets` in the `Runner` folder; this will be present in the Xcode project navigator
2. If your icons are ready, update the placeholder icons with your own app icons that have been generated
3. To check whether the icon is updated, run your app by using **Flutter Run**

Creating the build archive

This is the final step for creating the build archive and then uploading it to the Apple Store. On the command line, follow these steps in your application directory:

1. Run flutter build iOS to create a release build.
2. Perform this only if your Xcode is below version 8.3. To ensure that Xcode refreshes the release mode configuration, restart your Xcode workspace.

In Xcode, please use these steps to configure the app version and build:

1. In Xcode, open `Runner.xcworkspace` in your app's `ios` folder.
2. Select **Product** | **Scheme** | **Runner**.
3. Select **Product** | **Destination** | **Generic iOS Device**.
4. Select **Runner** in the Xcode project navigator followed by **Runner target** in the settings view sidebar.
5. In the **Identity** section, update the version and also update the **Build identifier** to a unique build number. This is used to track the number of the build uploaded. Each build should have a unique build number.

The final step is to create the build archive and upload it to App Store Connect:

1. Select the product and then **Achieve** to product a build archive
2. In the Xcode organizer window in the **sidebar** | **select the iOS app** | **select the build archive** that you just produced
3. Click the **Validate** button
4. After the archive is validated, you can click the **Upload to App store** option

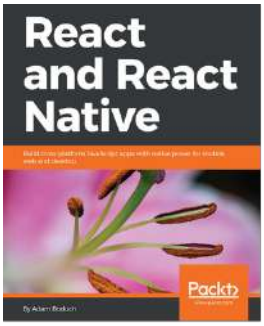
If there are any errors, reproduce the build and try to repeat the process again.

Summary

Once you have worked on your awesome app, deployment and publishing are key aspects. We covered how to publish the android and iOS app on Play Store. It is important to know that app upload just lists the app. You should also look at the App Store as key techniques to have a better visibility for the apps.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



React and React Native
Adam Boduch

ISBN: 9781786465658

- Craft reusable React components
- Control navigation using the React Router to help keep your UI in sync with URLs
- Build isomorphic web applications using Node.js
- Use the Flexbox layout model to create responsive mobile designs
- Leverage the native APIs of Android and iOS to build engaging applications with React Native
- Respond to gestures in a way that's intuitive for the user
- Use Relay to build a unified data architecture for your React UIs

React Native Cookbook
Crysfel Villa

ISBN: 9781786462558

- Build simple and complex UIs using React Native
- Create advanced animations for UI components
- Build universal apps that run on phones and tablets
- Leverage Redux to manage application flow and data
- Expose both custom native UI components and application logic to React Native
- Integrate with existing native applications on iOS and Android
- Deploy your React Native application to the Google Play and Apple App Store
- Add automated testing to your React Native application

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

G

H

I

J

L

X