FLUTTER UI

SUCCINCTLY

BY **ED FREITAS**

SUCCINCTLY EBOOK SERIES

Syncfusion®

Flutter UI Succinctly

By
Ed Freitas

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

ISBN: 978-1-64200-219-5

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.
Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, CEO
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Ed Freitas is a consultant on software development applied to customer success, mostly related to financial process automation, accounts payable processing, and data extraction.

He likes technology and enjoys playing soccer, running, traveling, life hacking, learning, and spending time with his family.

You can reach him at https://edfreitas.me.

Acknowledgments

Many thanks to all the people who contributed to this book, including the amazing Syncfusion team that helped this book become a reality, especially Jacqueline Bieringer, Tres Watkins, and Graham High.

The manuscript manager and technical editor thoroughly reviewed the book's organization, code quality, and overall accuracy—Graham High from Syncfusion, and James McCaffrey from Microsoft Research. Thank you all.

This book is dedicated to *Puntico* and *Chelin*—may both your journeys be blessed.

Introduction

With the rapid rise of cross-platform mobile frameworks such as Ionic, React Native, and Xamarin, Google decided to step into the game and develop their own framework, with support for both Android and iOS using the same codebase. This is how Flutter came to be.

Flutter is an open-source mobile application development SDK primarily developed and sponsored by Google, used for developing applications for Android and iOS—as well as being the primary method of creating applications for Google Fuchsia.

Flutter is written in C, C++,and Dart, and uses the Skia Graphics Engine. It offers a rich set of fully customizable widgets to build native interfaces that include the beautiful Material Design library and Cupertino (iOS-flavored) widgets, rich motion APIs, smooth natural scrolling, platform awareness, and hot reload, which helps to quickly build user interfaces (UIs) without losing state on emulators, simulators, and any hardware for iOS and Android.

All these features have helped Flutter take off very quickly, and developers are flocking to the framework. It is also one of the trending GitHub projects, which has helped it gain even more popularity.

Flutter has several great features, but one of the best is how easy it is to create high-quality UIs with it. Throughout this book, we will explore the fundamentals of how user interfaces can be created with Flutter, focusing on the existing widgets and tools that Flutter provides out of the box.

I'm thrilled to embark on this adventure with you, and hopefully, by the end of it, you'll have a solid understanding of how to use existing Flutter tools and widgets to make great-looking interfaces with the minimum amount of effort. Without further ado, let's get going.

Chapter 1 Setup

Overview

We won't build a Flutter application from start to finish in this book, at least not in the typical sense of building a complete app. Instead, we will create basic applications, which we will use to explore various examples of how to work with layouts, place widgets, and create user interfaces.

If you would like to explore how to create a Flutter application from start to finish, the *Succinctly* series has you covered with *Flutter Succinctly*, which is a good resource to get up and running quickly with Flutter.

Knowing Flutter is not a prerequisite for following along with the concepts that will be covered throughout this book. This book is more of a complement to the previous book, specifically focusing on user interface concepts, which weren't fully covered in *Flutter Succinctly*.

Installation

Getting Flutter installed is incredibly easy, given that the installation steps are well documented within the official Flutter documentation site.

I'll be using Windows 10, so I'll be describing setup steps and information related to this operating system, but there are also easy-to-follow setup guidelines for both macOS and Linux.

On Windows, some essential system requirements need to be in place, which include having PowerShell 5.0 or later and Git for Windows 2.X or later installed.

Flutter relies on a full installation of Android Studio, as it requires access to all Android platform dependencies. You'll also need to set up an Android device emulator—this is described step by step in the official documentation.

With the prerequisites in place for Windows, all we need to do is download the installation bundle of the Flutter SDK. At the time of writing, it is Flutter's 1.22.6 stable version for Windows.

Once you've downloaded the zip file, extract it to a folder within your hard drive, such as **C:\Flutter**. To make your life easier, I suggest that you not extract the Flutter files to **C:\Program Files** or **C:\Program Files (x86)**, which require elevated or admin permissions.

Once the files are in the desired folder, locate the file **flutter_console.bat**. This is how it looks on my machine.

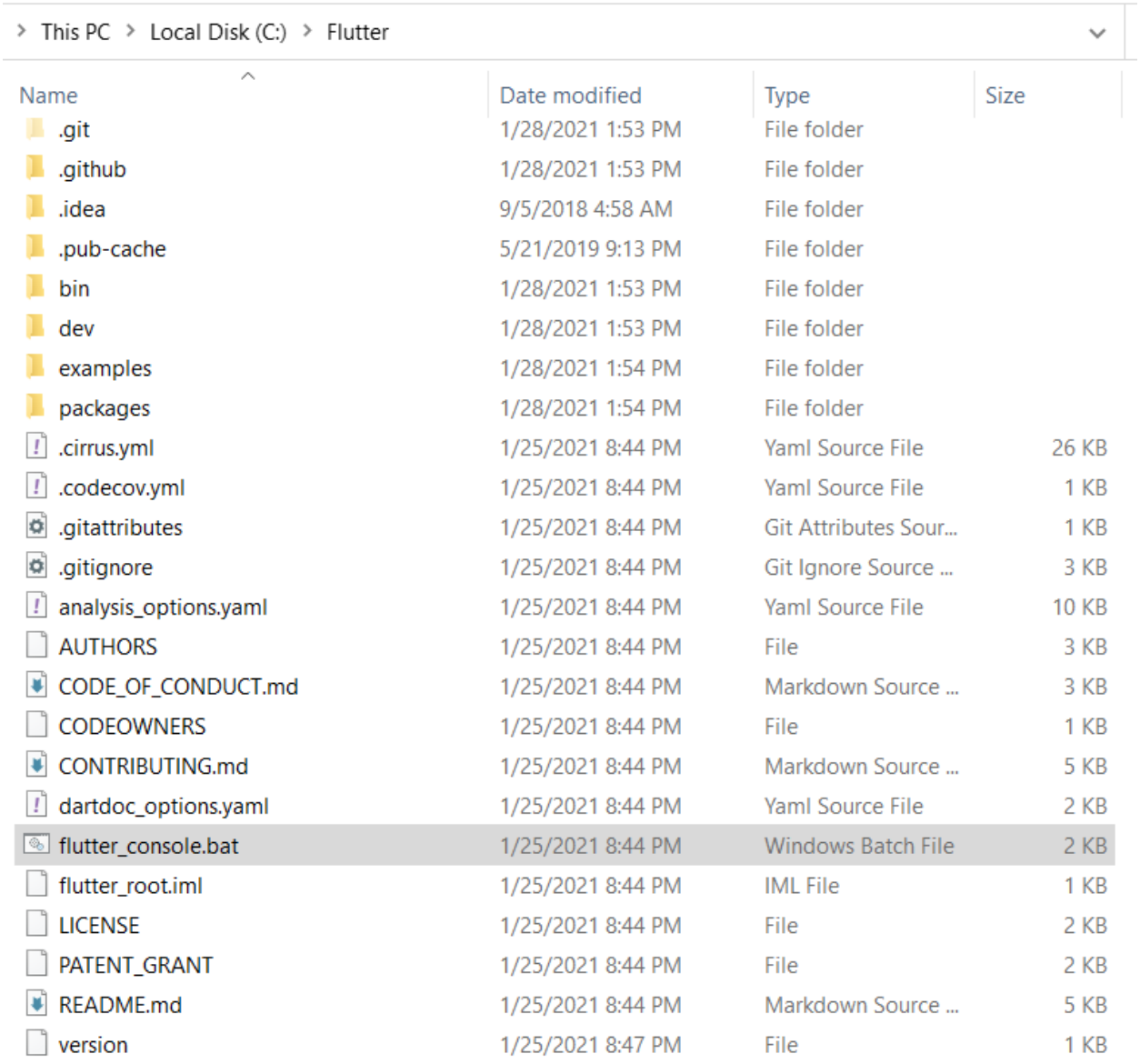

Figure 1-a: The Flutter SDK Files

In principle, you are now ready to run the Flutter console by executing the **flutter_console.bat** file. It's recommended (although not strictly necessary) to add the **flutter\bin** folder to the **Path environment variable** in Windows.

If you are unsure how to add a folder to the **Windows Path variable**, please refer to this article that explains how to do it step by step, with screenshots. On my machine, this looks as follows.

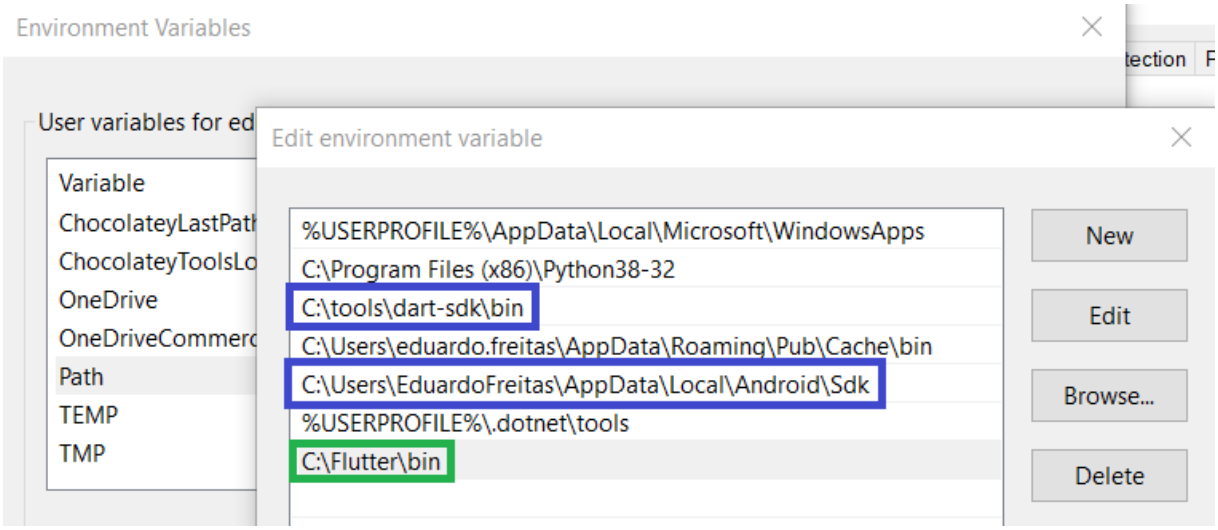

Figure 1-b: Flutter Added to the Path Variable in Windows (Highlighted in Green)

Notice as well that the paths to the Dart and Android SDKs (highlighted in blue in Figure 1-b) should also be added to the environment variables.

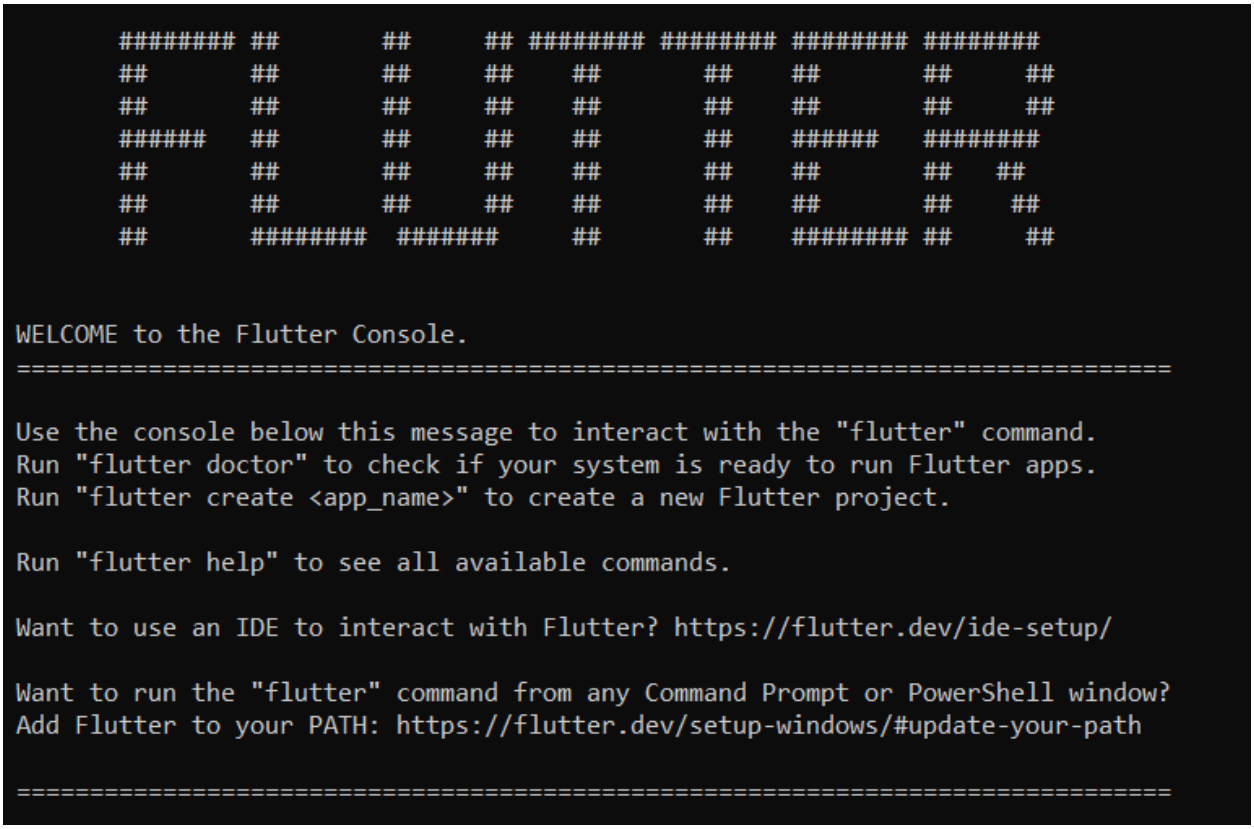

Figure 1-c: The Flutter Console Running

At the prompt, type the following command to check if Flutter is fully operational.

Code Listing 1-a: The "flutter doctor" Command

```
flutter doctor
```

When executing this command, if some updates are available; they will be downloaded and installed accordingly.

```
C:\Users\eduardo.freitas>flutter doctor
Checking Dart SDK version...
Downloading Dart SDK from Flutter engine 2f0af3715217a0c2ada72c717d4ed9178d68f6ed...
Unzipping Dart SDK...
Building flutter tool...
Running pub upgrade...
Downloading package sky_engine...                                  0.8s
Downloading flutter_patched_sdk tools...                           1.5s
Downloading flutter_patched_sdk_product tools...                   1.3s
Downloading windows-x64 tools...                                   2.8s
Downloading windows-x64/font-subset tools...                       0.4s
Downloading android-arm-profile/windows-x64 tools...               0.4s
Downloading android-arm-release/windows-x64 tools...               0.3s
Downloading android-arm64-profile/windows-x64 tools...             0.3s
Downloading android-arm64-release/windows-x64 tools...             0.3s
Downloading android-x64-profile/windows-x64 tools...               0.3s
Downloading android-x64-release/windows-x64 tools...               0.3s
Doctor summary (to see all details, run flutter doctor -v):
```

Figure 1-d: The Flutter Console Running (Continued – Installing Updates)

After you execute this command, you will get a result with any issues found. In my case, because I had previously installed Android Studio and Visual Studio Code, I get the following information.

```
C:\Users\eduardo.freitas>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel stable, 1.22.6, on Microsoft Windows [Version 10.0.19042.746],
    locale en-US)

[√] Android toolchain - develop for Android devices (Android SDK version 29.0.2)
[!] Android Studio (version 4.1.0)
    X Flutter plugin not installed; this adds Flutter specific functionality.
    X Dart plugin not installed; this adds Dart specific functionality.
[√] VS Code, 64-bit edition (version 1.52.1)
[!] Connected device
    ! No devices available

! Doctor found issues in 2 categories.
```

Figure 1-e: The Flutter Console Running (Continued – Results)

Notice that after running this command, I have two issues. In my case, these are irrelevant because I'll be using Visual Studio Code (VS Code) as my Flutter development environment,

instead of Android Studio. It is also highlighted that I don't have a physical device connected, which is fine for now.

If you prefer using Android Studio, make sure you have the Flutter and Dart plugins installed. All the information on how to install both plugins for Android Studio can be found here. When you run the Android Studio installer, please make sure you follow the official documentation so that you end up with a successful Android Studio and SDK setup.

Make sure that you resolve all the conflicts highlighted by the **`flutter doctor`** command before proceeding.

Setting up an editor

Once you have completed all the installation steps, it is necessary to set up Flutter to work with your editor of choice. In my case, I'll be using Visual Studio Code. If you would like to know how to use Android Studio as your editor of choice, then feel free to check *Flutter Succinctly*, which explains how to do this.

Alternatively, the official Flutter documentation describes how to configure Android Studio (IntelliJ) to work with Flutter, so feel free to check our those steps.

For Visual Studio Code, the steps are quite simple. First, go to **View** > **Extensions**.

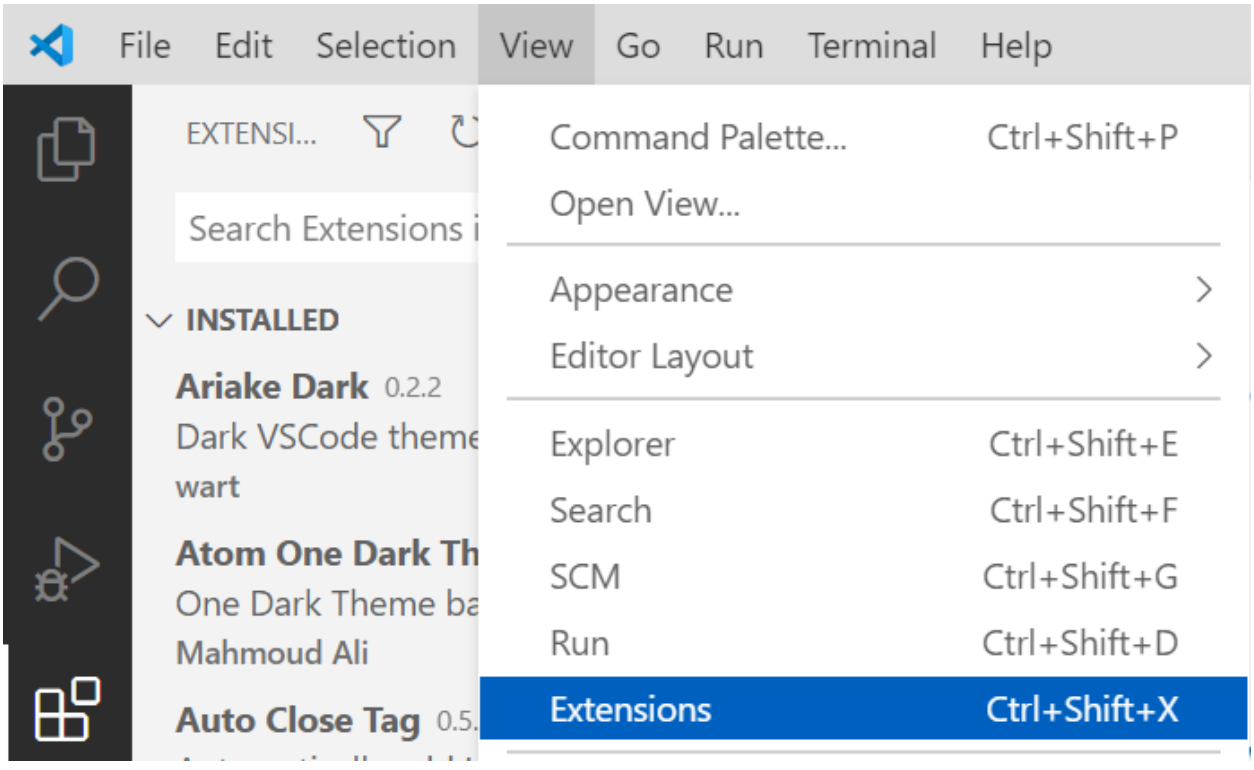

Figure 1-f: The Extensions Option

Type **Flutter** in the search box, select the **Flutter** option, and then click **Install**.

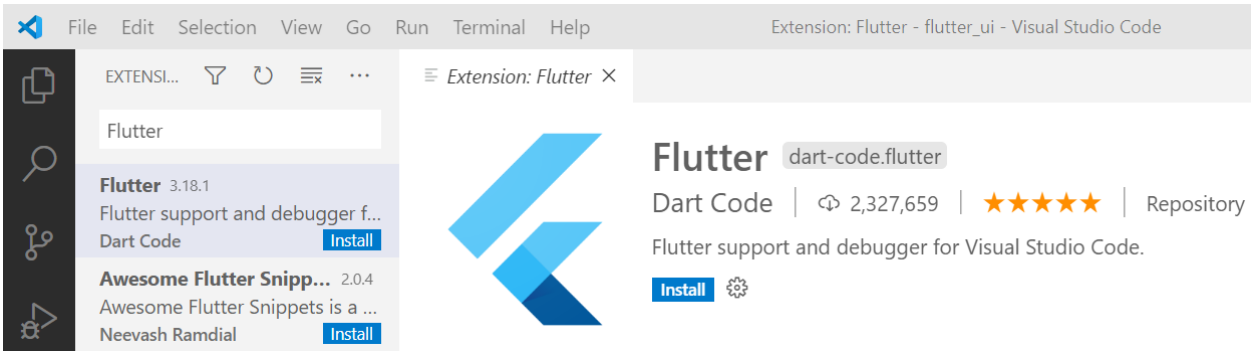

Figure 1-g: The Flutter Extension

Once the Flutter extension has been installed, you might be asked to reload Visual Studio Code.

Next, run the **`flutter doctor`** command to make sure that everything is working as expected. If all is good, you are ready to create a Flutter project.

Creating the app

Once your editor of choice has been correctly set up following the official documentation guidelines and my previous suggestions (in my case using Visual Studio Code), it's time to create a new Flutter project, which we will use throughout the rest of this book.

Creating a new Flutter project with Visual Studio Code is easy. All you need to do is go to **View** > **Command Palette** > **Flutter: New Application Project**.

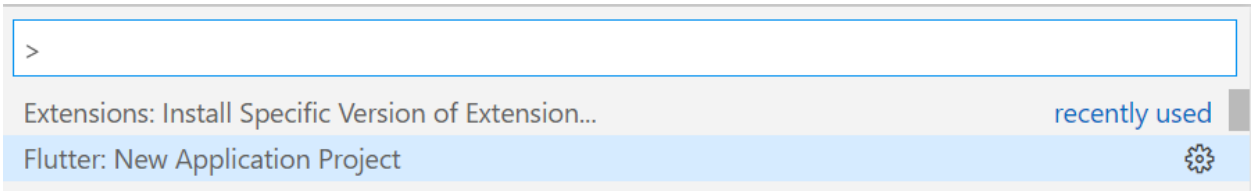

Figure 1-h: The Flutter New Application Project Option

You'll be asked to select the folder where you would like to save your project files, and then to provide a name for your Flutter application. I'll name my application **flutter_ui**.

Once the project has been created, you'll see under the project files within the **Explorer** view of Visual Studio Code, which in my case looks as follows.

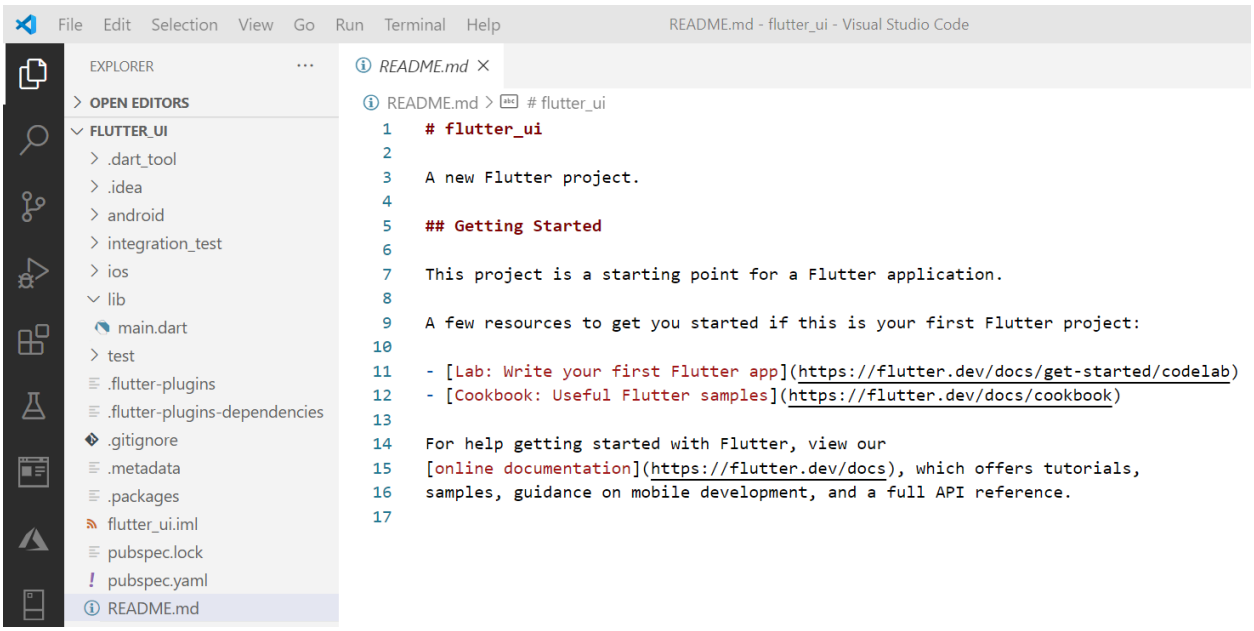

Figure 1-i: The New Flutter Project

The process of creating a new Flutter project with Android Studio is slightly longer than with Visual Studio Code (more steps are required). If you would like to explore this route, *Flutter Succinctly* covers this option in depth.

Before we can run the newly created Flutter project, we need to make sure we have a virtual device created and ready.

Creating a virtual device

Let's quickly go over the steps required to create a virtual device, which can only be done with Android Studio.

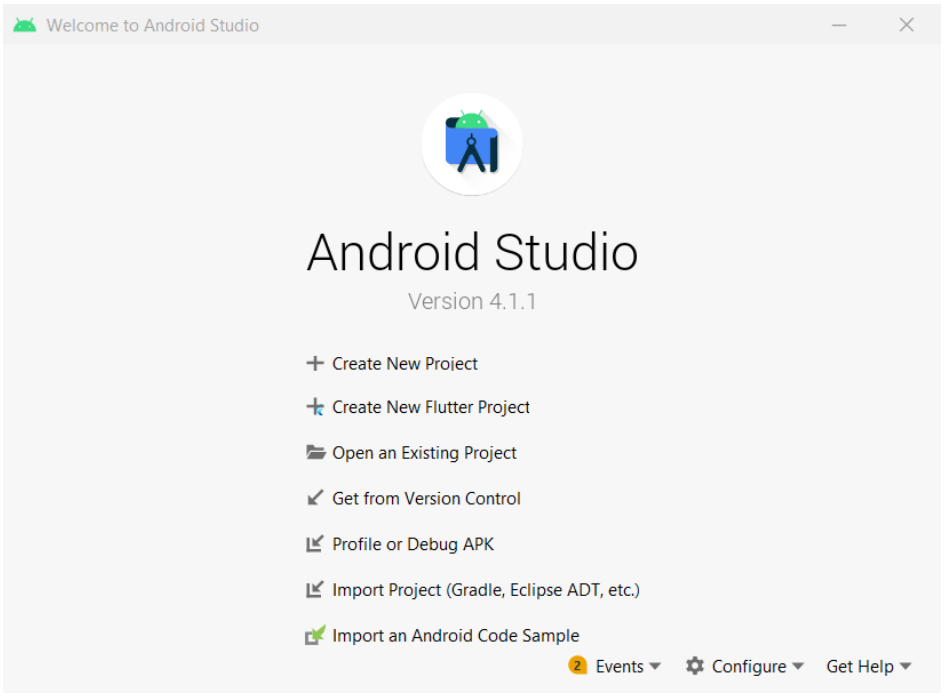

Figure 1-j: The Android Studio Welcome Screen

On the welcome screen of Android Studio, go to **Configure** > **AVD Manager**, which will display the following screen.

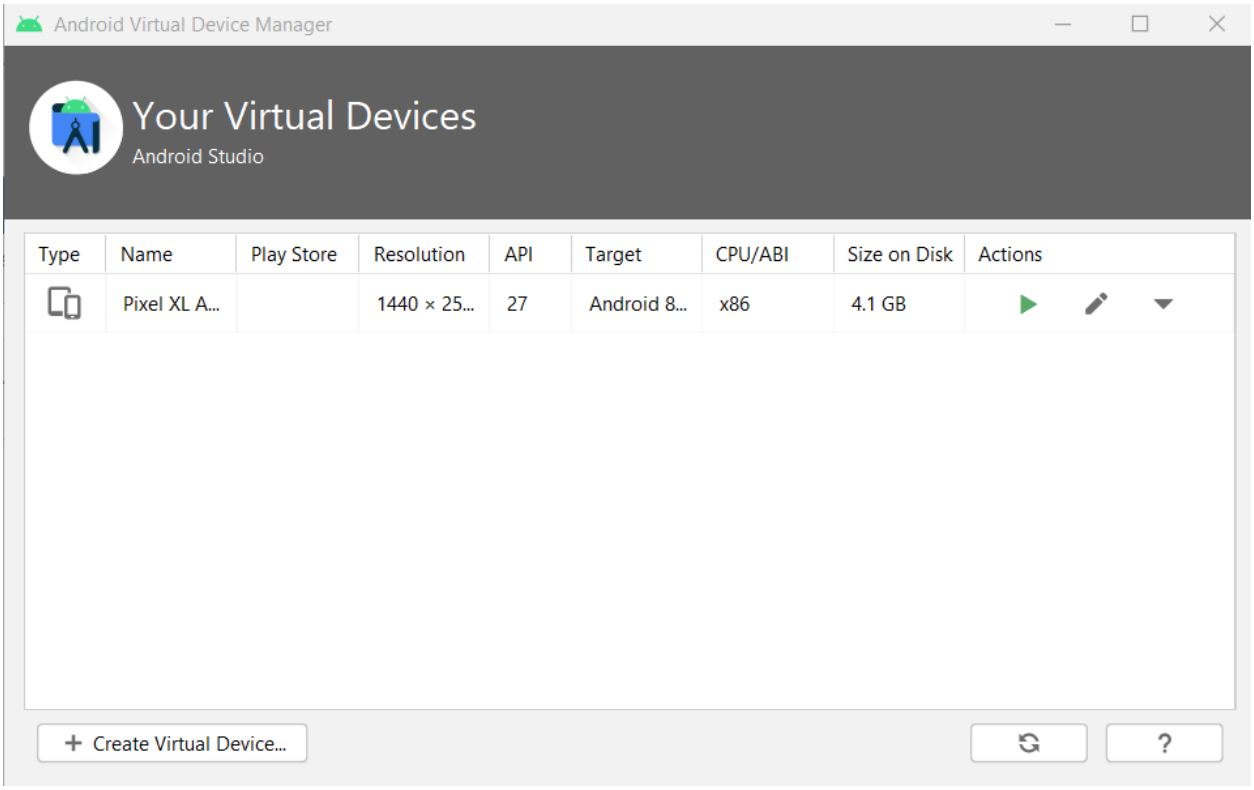

Figure 1-k: The Android Studio Virtual Device Manager (Your Virtual Devices)

You can see that I have a virtual device created. To create a new one, click **Create Virtual Device**, which will display the following screen.

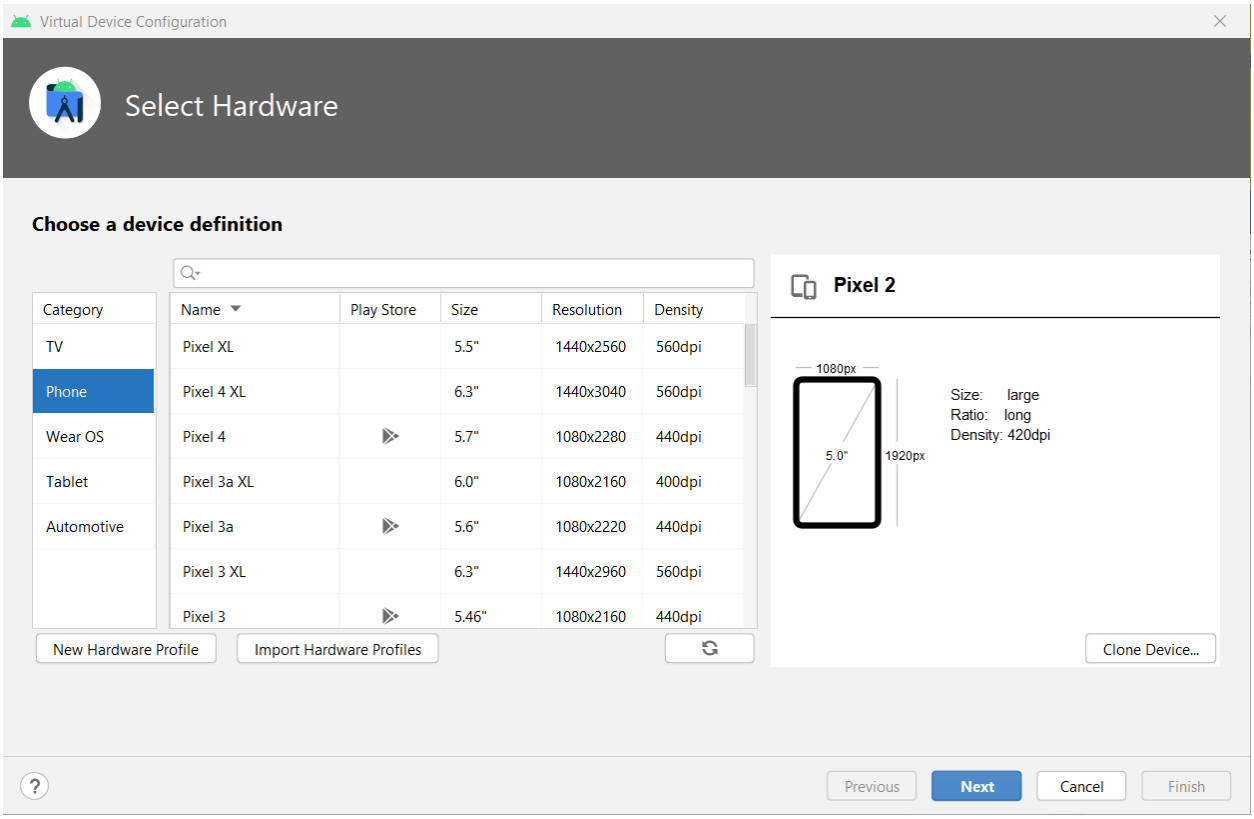

Figure 1-l: The Android Studio Virtual Device Manager (Select Hardware)

At this stage, choose the device that you would like to emulate (such as **Pixel 4 XL**), and then click **Next**. This will display the operating system images available.

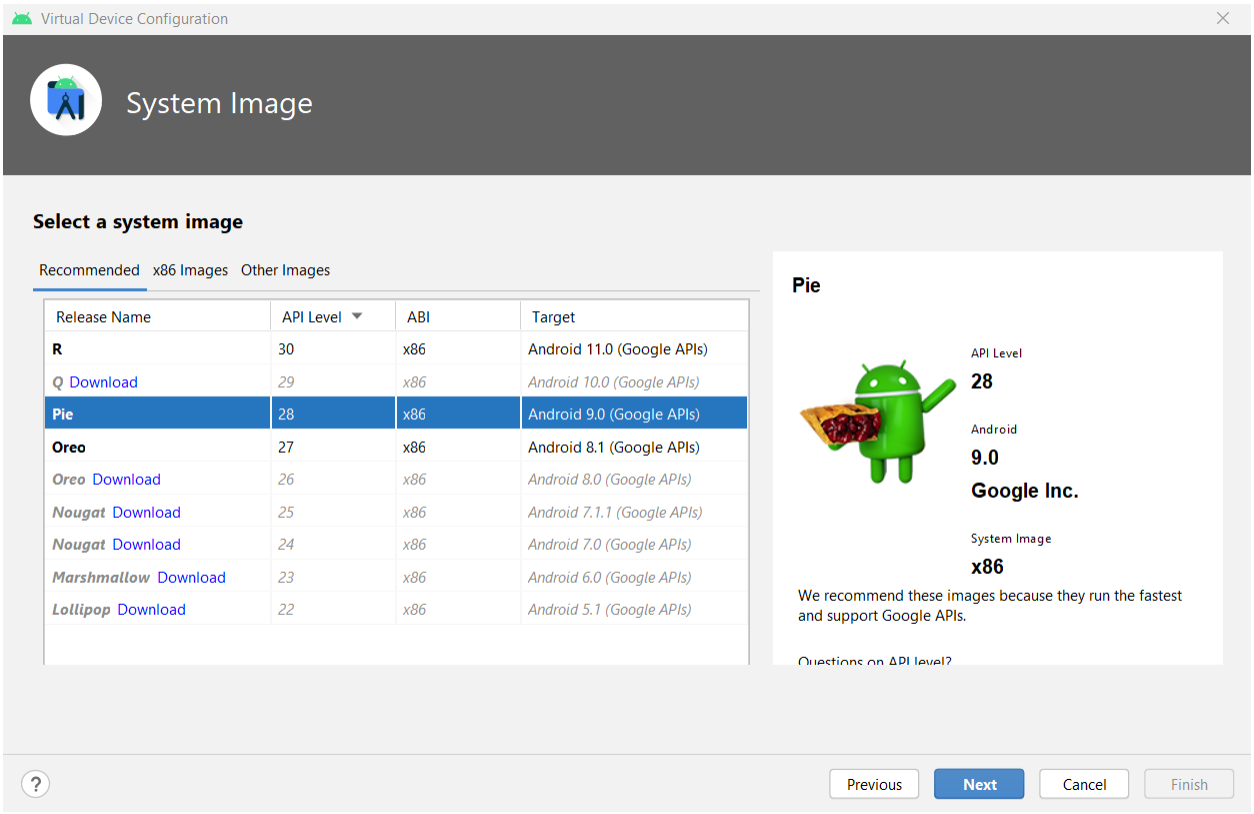

Figure 1-m: The Android Studio Virtual Device Manager (System Image)

It's important to choose an image that plays nicely with your computer's host operating system. In essence, it's not recommended, for emulator performance reasons, to choose an ARM-based image if your computer's host operating system is based on an x86 architecture.

If you've chosen a different image than the one I have highlighted in Figure 1-m, you might have to download the image by using the download link next to the **Release Name**.

Once the image has been selected (and downloaded, if applicable), click **Next** to continue with the last step.

From the list, choose any of the most recent API Level versions, and then click **Next**. This will show a screen to verify the configuration, before creating the image.

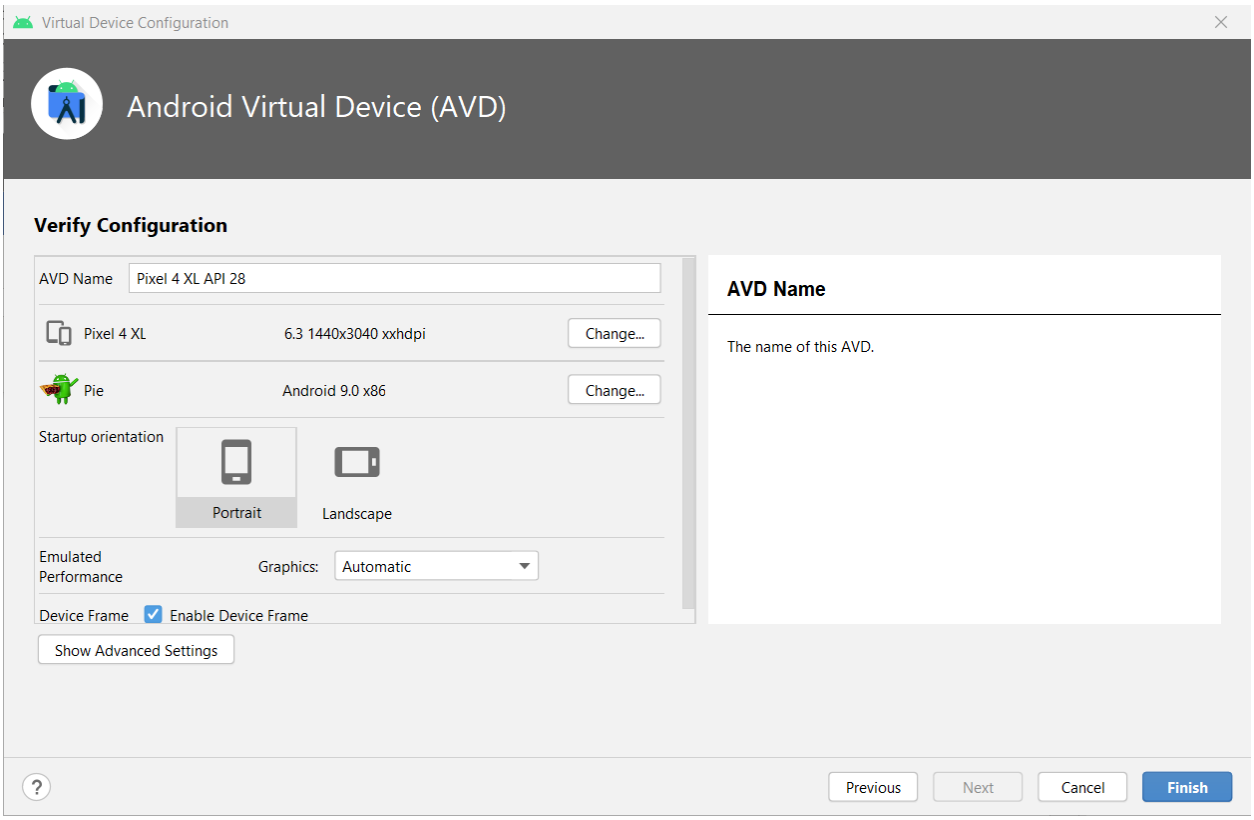

Figure 1-n: The Android Studio Virtual Device Manager (Verify Configuration)

You can use the default configuration settings. To finalize the creation of the virtual device, click **Finish**.

Awesome—you now have created a virtual device. You can create more than one if you wish, as it might help you test your application with multiple devices. The virtual device I have created looks as follows.

Figure 1-o: Virtual Device – Android Emulator

Testing your setup

With our virtual device in place, it's now time to run the application we have created and see what it does. This is the default demo app that comes out of the box with Flutter.

To do that, all we need to do in Visual Studio Code is go to **Run** > **Run Without Debugging**.

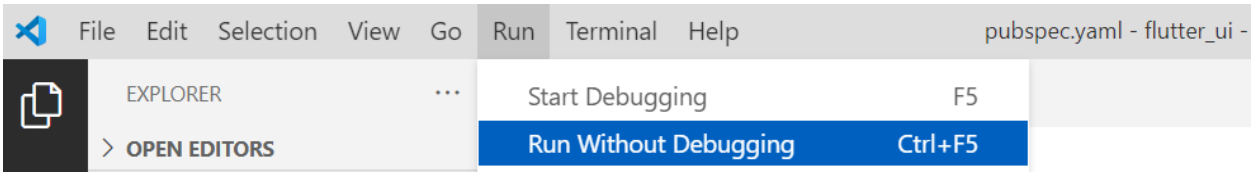

Figure 1-p: Running the App Without Debugging – Visual Studio Code

If you are using Android Studio, select the **Open Android Emulator** option from the **Android SDK built for x86** dropdown menu (which is next to the **Run** button). You'll be able to execute the application when you click **Run**, once the Android emulator is opened.

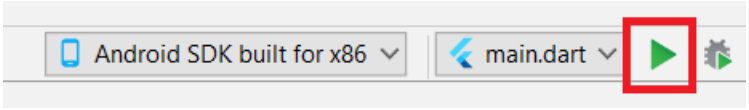

Figure 1-q: The Emulator Dropdown and Run Button – Android Studio

Since I'll be using Visual Studio Code, I'll be focusing entirely on VS Code instead of testing the app with Android Studio.

After clicking the **Run Without Debugging** option in VS Code, the following options are presented. In our case, we need to select **Dart & Flutter**.

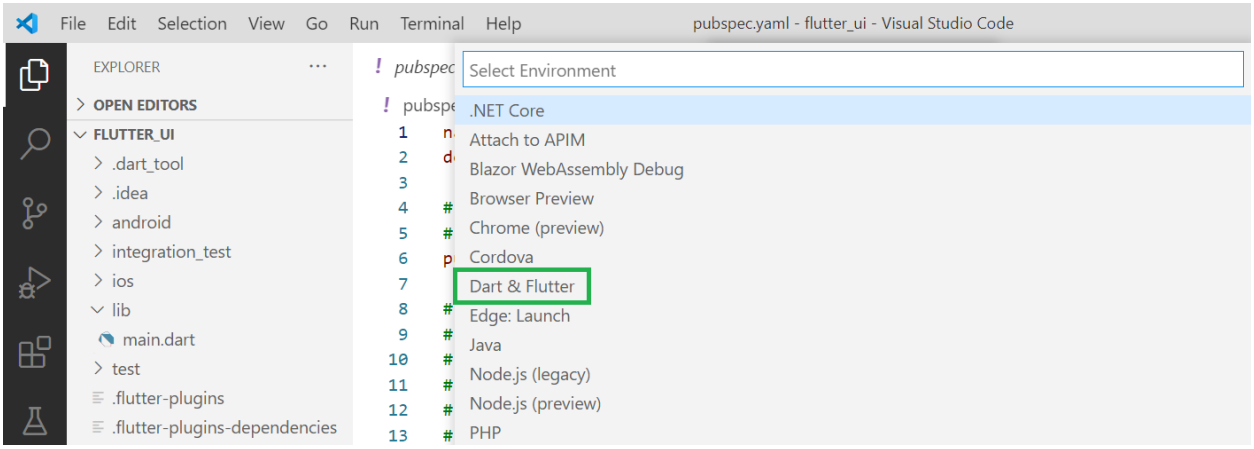

Figure 1-r: Select Environment Options – Visual Studio Code

You will be asked to choose the virtual device to use.

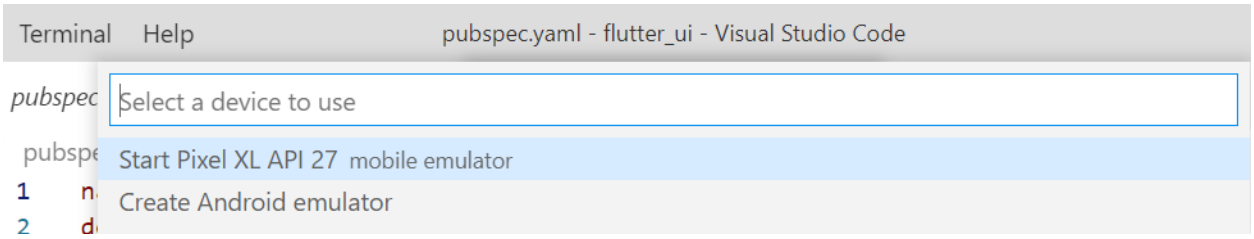

Figure 1-s: Select Device Option – Visual Studio Code

Within that list, the virtual device that was recently created should be shown. If the device is shown on the list, simply click on it to start the emulator.

In my case, my virtual device is a Pixel XL API 27 emulator, so I'll click the **Start Pixel XL API 27 mobile emulator** option. Doing so will open the emulator with the application running, which we can see as follows.

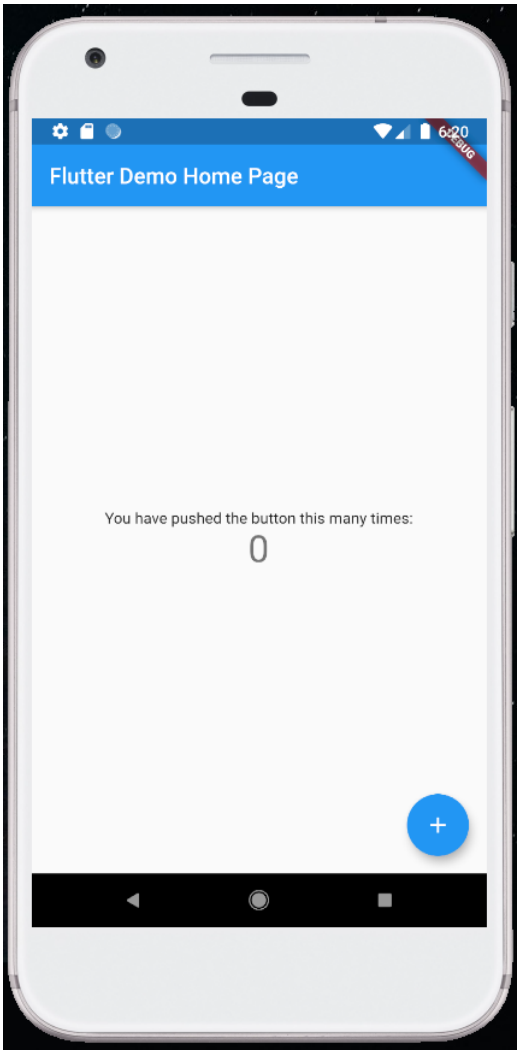

Figure 1-t: Default Flutter App Running

Summary

Throughout this chapter, we've had a look at how to get our Flutter development environment ready. Now that our environment is set up, we are now ready to start exploring how to work with layouts and Flutter UI widgets. This is what we'll do in the next chapter.

Chapter 2 Scaffolds

Overview

Flutter is full of great features, and one of the most important features, in my opinion, is how easy it is to build high-quality user interfaces with it.

The goal of this book is to give you the fundamental knowledge to create engaging user interfaces using layouts, containers, rows and columns, and common widgets.

Scaffolds, as they are known in Flutter, or layouts are at the core of building user interfaces with Flutter, and this is what this chapter is all about.

Our first layout

Building user interfaces is one of those things you can only learn by experimenting, so let's dive right into the action by modifying the default application and building a layout.

The layout that we'll build will be a MaterialApp class, which includes a Scaffold widget with an AppBar, along with a **body** property that includes a FloatingActionButton.

So, with your editor of choice open (I'll be using VS Code), go to the **main.dart** file, delete the existing (default) code, and replace it with the following code.

Code Listing 2-a: Updated main.dart

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Center(
          child: Text(
            'Our first Flutter layout',
            style: TextStyle(fontSize: 24),
          ),
```

```
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
    );
  }
}
```

Once the code has been replaced, save the **main.dart** file.

If you don't have the emulator running, go to **Run** > **Run Without Debugging**, and follow the steps to start the emulator and run the app.

If you already have the emulator running, Flutter's hot reload mechanism will automatically update the application's user interface, which should look as follows.

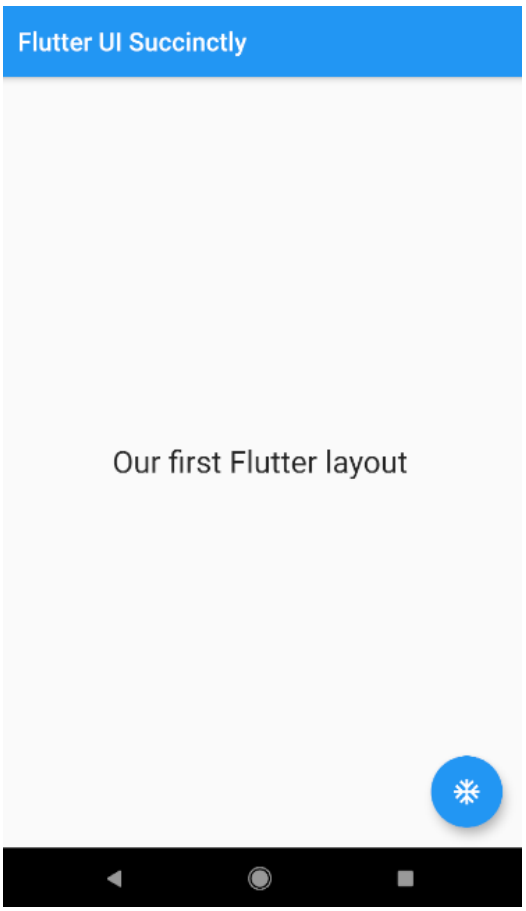

Figure 2-a: Our First Flutter Layout

Before we dive into the details, let's have a look at the following diagram, which describes the relationship between the UI elements and the code.

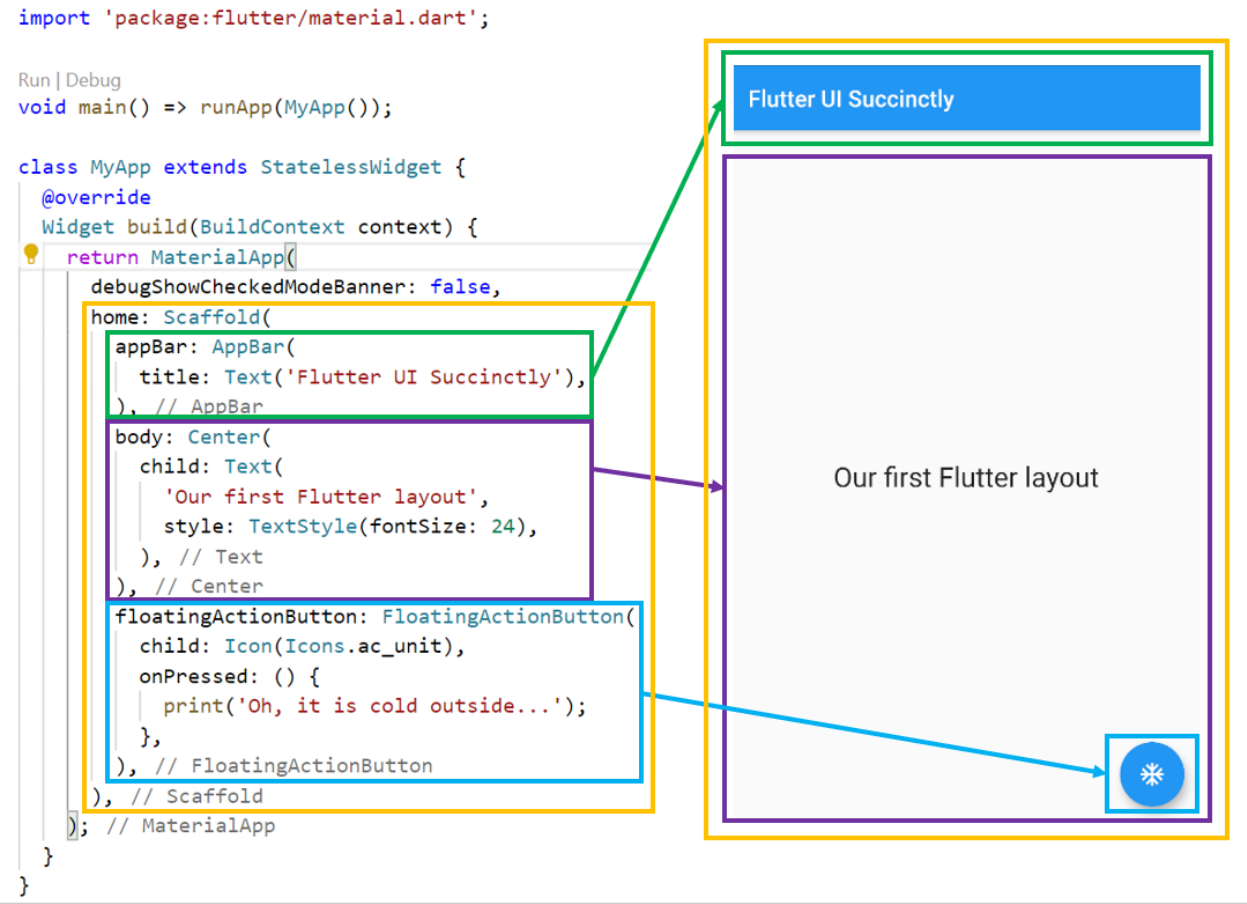

Figure 2-b: UI-to-Code Relationship

As you can see in Figure 2-b, there are three essential parts to this layout, which is the **`Scaffold`** widget (highlighted in yellow).

The first is the **`AppBar`** widget (highlighted in green), the **`body`** property (highlighted in purple), and the **`FloatingActionButton`** widget (highlighted in **blue**).

Let's review in detail what we have done. The first thing we did was import the Material Design Flutter library with this instruction: **`'package:flutter/material.dart'`**.

Next, within the app's **`main`** function, which serves as the application's entry point, we created an instance of the **`MyApp`** class, which we then passed as a parameter to the **`runApp`** method. This method is responsible for executing the Flutter application.

The **`MyApp`** class is a stateless widget that does not require a mutable state and is often used as a starting point for building a user interface.

This is why the **`MyApp`** class inherits (referred to as extends in the Dart programming language) from the **`StatelessWidget`** class.

The **build** method from the **MyApp** class overrides the **build** method inherited from the StatelessWidget class, which is why the **@override** decorator is used.

The **build** method from the **MyApp** class returns an object that renders the user interface. This object is a MaterialApp instance that is used for wrapping several widgets that are required when building Material Design applications.

For the **MaterialApp** widget, there are two properties that we are using. One is the debugShowCheckedModeBanner property, and the other is **home**.

The **debugShowCheckedModeBanner** property, as its name implies, is used to display a debug banner at the top of the application's screen when set to **true**. In this case, its value is set to **false**, so the debug banner is not shown.

The **home** property contains the complete layout that we've created: a **Scaffold** widget, which contains the app's header (**appBar**), the **body**, and the floating blue button (**floatingActionButton**) properties.

The **appBar** property includes only one child property, the **title**, which is assigned the value that is passed to the **Text** widget.

A **Center** widget is assigned to the **body** property of the **Scaffold** widget. As its name implies, the **Center** widget is used to align content to the center of the screen.

The **Center** widget contains a **child** property to which a **Text** widget is assigned. This **Text** widget contains some text which is passed as a string, as well as the **style** property.

The **floatingActionButton** property has a **FloatingActionButton** widget assigned to it. This **FloatingActionButton** widget contains a **child** property and an **onPressed** event, which is triggered when the user taps the floating button.

The **child** property of the **FloatingActionButton** widget is assigned to an **Icon** widget. The **Icon** widget displays the type of icon seen within the floating button, which looks like one of those drawings of a snowflake you see on fridges or air conditioners (**Icons.ac_unit**).

When the **onPressed** event is triggered, a message is printed out to the console, which can be seen in the **Debug Console** area in VS Code.

Figure 2-c: The Debug Console Output – VS Code

We've created our first layout with Flutter. Now, let's have a look at how we can take this further and use colors and themes.

Colors and themes

When you were building the previous layout, you may have noticed that by default, Flutter provided us with a "blue and white" application look and feel (theme), which looks quite good. This theme is based on the Material Design specifications.

However, we can easily customize the appearance of the application while staying within the Material Design specifications. Let's see how we can do that.

The first thing we can do is change the app's brightness settings. This is very simple to do by making a few changes to the **main.dart** code, which are highlighted in the following listing.

Code Listing 2-b: Updated main.dart (Brightness Changes)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Center(
          child: Text(
            'Our first Flutter layout',
            style: TextStyle(fontSize: 24),
          ),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        brightness: Brightness.dark,
```

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

After you have saved those file changes, Flutter should be able to update the application's UI automatically through the hot reload feature. On my machine, it looks as follows.

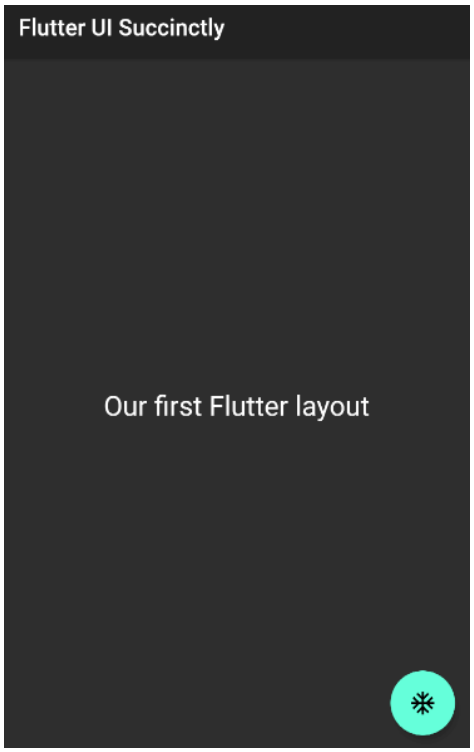

Figure 2-d: The Updated Flutter Layout (Dark Brightness)

Let's see what we have done. With a few lines of code, we were able to change the app's colors. We were able to achieve this by adding a **`theme`** property to the **`MaterialApp`** widget. This **`theme`** property takes a **`Theme`** widget, which contains a **`brightness`** property that has been set to **`Brightness.dark`**.

Although this looks very cool, we might not want to use only one color for most of the app's layout. We can customize the colors of the app's layout, as well as the text font on the app's header. In order to do so, we'll make the following changes to the **main.dart** code.

Code Listing 2-c: Updated main.dart (Color and Text Font Changes)

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

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

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Center(
          child: Text(
            'Our first Flutter layout',
            style: TextStyle(fontSize: 24),
          ),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After you save the changes to **main.dart**, Flutter should be able to update the application's UI automatically. On my machine, it looks as follows.

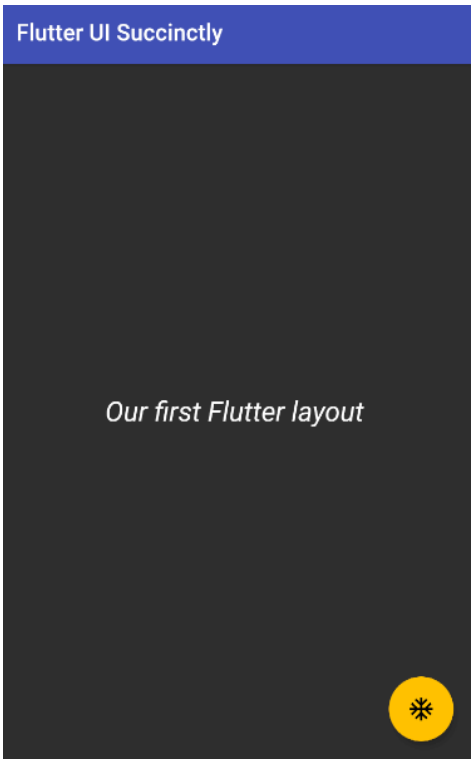

Figure 2-e: The Updated Flutter Layout (Color and Text Font Changes)

Let's have a look at what we have done. To the **`ThemeData`** widget, we've added the **`primaryColor`**, **`accentColor`**, and **`textTheme`** properties.

To know what colors are available to use with Material Design, we can refer to the Material Design Colors website.

The **`primaryColor`** property has been set to **`Colors.indigo`**, which is why the app's header is no longer black. The **`accentColor`** property has been set to **`Colors.amber`**, and the body text has been changed to italic.

To change the body text, a **`TextTheme`** widget instance is assigned to the **`textTheme`** property. This widget contains a **`bodyText2`** property of type **`TextStyle`** that is used to specify the properties of the body text, such as the **`fontSize`** and **`fontStyle`**—in this case, set to **`FontStyle.italic`**.

Summary

Scaffolds, or layouts, are the basic foundations required to build user interfaces with Flutter. Even though the code so far has not been very complex, we've explored the fundamental widgets and properties that are most commonly used to build layouts with Flutter.

With this knowledge, we are ready to explore containers, which are Flutter widgets that allow us to create more complex user interfaces, by embedding widgets within widgets. This is what we'll dive into next.

Chapter 3 Containers

Overview

The most flexible widget in Flutter is the Container class, which allows for painting, sizing, and positioning other widgets within a Flutter application.

The way the **`Container`** widget works is that it wraps the child widget with padding, and then applies constraints to the padded extent by using the width and height as constraints if either is different than **`null`**. The container is then surrounded by an additional space described from the margin.

Containers are used to contain other widgets, with the possibility of applying styling properties to the container itself and its children.

Container sizing

Let's carry on from where we left our code in the previous chapter. We'll start by removing all the code assigned to the **`body`** property of the **`Scaffold`** widget and replacing it with a **`Container`** widget. The changes are highlighted in bold in the listing that follows.

Code Listing 3-a: Updated main.dart (Adding a Container)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          color: Colors.lightBlue,
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
```

```
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After the changes are saved to **main.dart**, Flutter should be able to update the application's UI automatically. On my machine, it looks as follows.

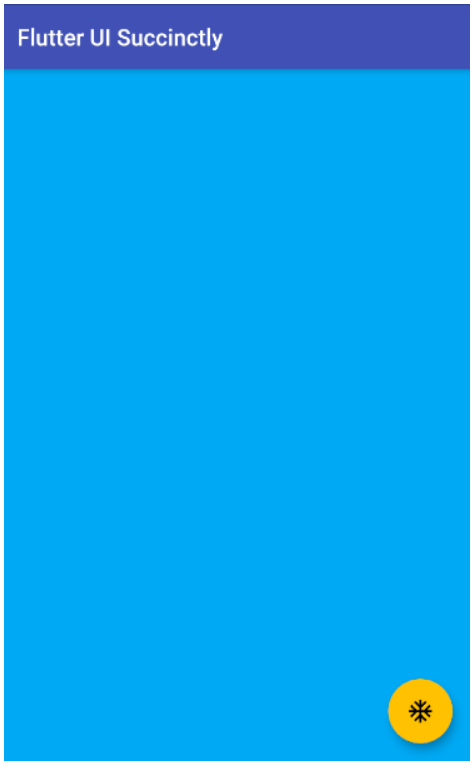

Figure 3-a: The Updated App UI (Adding a Container)

Although to the naked eye it seems the only modification we've made is to change the color of the **body** property, what we have really done is added a **`Container`** widget, which will provide

our application with a lot of flexibility for adding and positioning other widgets within the body of the app.

Before this change, we were limited to only using text within the body of the app. This is a subtle but significant improvement to the app. For now, within the **`Container`** widget, we've set the value of its **`color`** property to **`Colors.lightBlue`**.

Something you've probably noticed is that even if the **`Container`** widget is empty (for now), setting the **`color`** property fills in the entire space of the **`Scaffold`** body.

But what happens when we insert a **`child`** widget into the **`Container`** widget? Let's do precisely that by making the following adjustment to our code (highlighted in bold).

Code Listing 3-b: Updated main.dart (Adding a Child to the Container)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          color: Colors.lightBlue,
          child: ButtonBar()
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
```

```
        brightness: Brightness.dark,
      ),
    );
  }
}
```

What we have done is added a **`ButtonBar`** widget as a **`child`** of the **`Container`** widget. If we save the changes to **main.dart**, we should see the following.

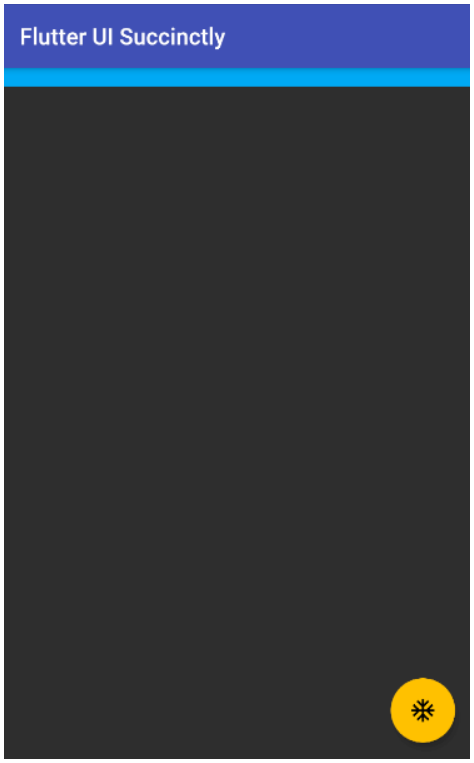

Figure 3-b: The Updated App UI (Adding a Child to the Container)

Notice that after making this change, the **`Container`** widget is as big as the **`ButtonBar`** widget. This means that the size of the **`Container`** widget is dynamic, and may vary depending on the content of its **`child`** property and the widgets that come under it.

So, when using containers, they will size themselves to their **`child`**, unless the **`width`** or **`height`** properties of the **`Container`** widget are set. This is why when we added the **`ButtonBar`** to the **`Container`** widget, it became the same size as the **`ButtonBar`**.

On the other hand, containers without children will follow these two rules:

- If the parent widget provides **unbounded** constraints, the **`Container`** widget without children will always try to be as small as possible.
- If the parent widget provides **bounded** constraints, the **`Container`** widget without children will always try to be as big as possible.

Therefore, we can interpret that four parameters govern how much space a **`Container`** widget may take. These parameters are known as **box** constraints, and can be seen in the following diagram.

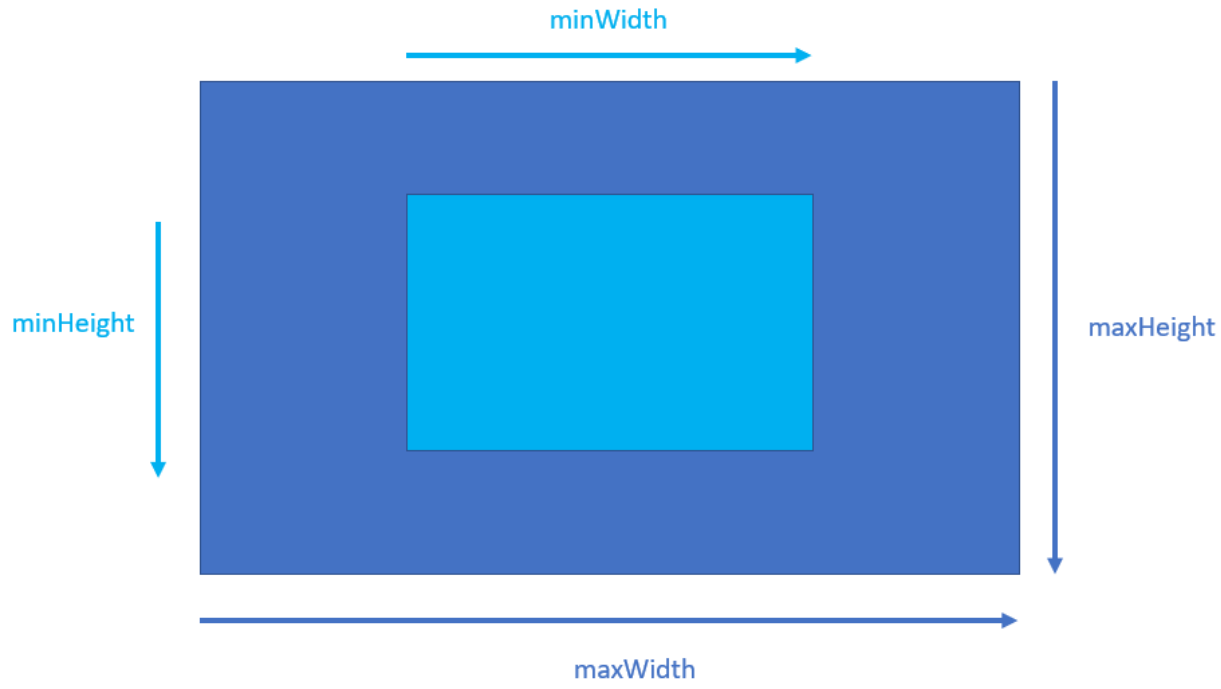

Figure 3-c: Parent and Container Spatial Relationship (Box Constraints)

The area in the lighter blue color represents the **`Container`** widget, and the area in the darker blue color indicates the parent widget.

To see how the **`width`** and **`height`** properties affect the sizing of a **`Container`** widget, let's make some changes to our code. These are highlighted in bold.

Code Listing 3-c: Updated main.dart (Container Width and Height)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
```

```
        body: Container(
            width: 300,
            height: 300,
            color: Colors.lightBlue,
            child: ButtonBar()),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

Save the changes to **main.dart**, and the app's UI should be updated on the emulator. On my machine, it looks as follows.

Figure 3-d: A Container Widget with Specific Width and Height

By specifying fixed values for the **`width`** and **`height`** properties, we were able to overcome the out-of-the-box constraints that govern **`Container`** widget sizing in Flutter.

Container placement

Now that we know how to size a **`Container`** widget, let's explore how we can place it in any part of the screen. But before we do that, we need to get some basic placement concepts clear. When placing any **`Container`** widget, two fundamental definitions must be understood: the **`margin`** and **`padding`** properties.

The **`margin`** property is the space outside the border of a **`Container`** widget, whereas the **`padding`** property is the space between the border of the **`Container`** widget and its content. To understand this better, let's look at the following diagram.

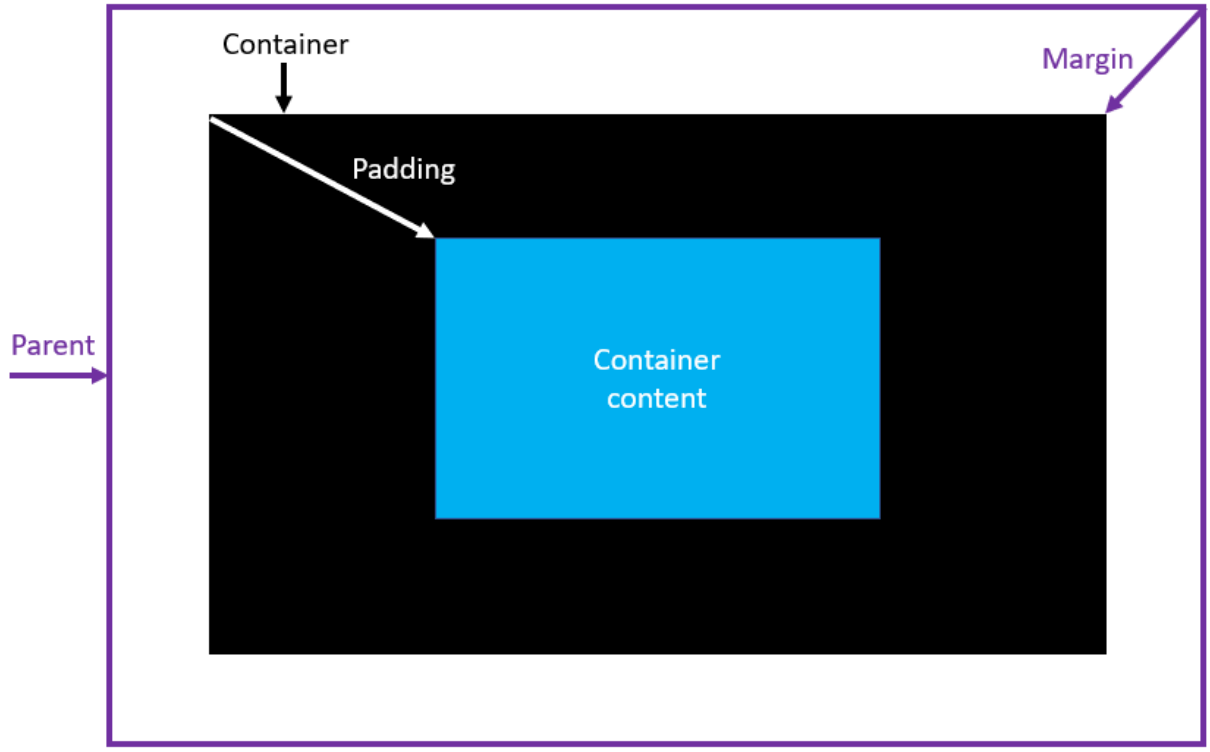

Figure 3-e: Container Margin and Padding

For defining the values of the **`margin`** and **`padding`** properties of a **`Container`** widget, we can use the EdgeInsets class, which is used for setting an offset from each of the four sides of a box.

The **`EdgeInsets`** class contains three constructor methods, which take double values as parameters that can be used to set offsets:

- **`EdgeInsets.all`**: Creates an offset on all four sides of the box.
- **`EdgeInsets.only`**: Allows you to choose on which sides to create an offset.
- **`EdgeInsets.symmetric`**: Allows you to create symmetrical horizontal and vertical offsets.

To understand this better, let's make some adjustments to our code to add some **`margin`** and **`padding`** properties to the **`Container`** widget, and change the **`ButtonBar`** to a **`Text`** widget. The changes are highlighted in bold in the following code.

Code Listing 3-d: Updated main.dart (Container Margin and Padding)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
```

```
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          margin: EdgeInsets.all(100),
          padding: EdgeInsets.all(50),
          width: 300,
          height: 300,
          color: Colors.lightBlue,
          child: Text('Container'),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After the changes are saved to **main.dart**, the app's UI should be updated on the emulator. On my machine, it looks as follows.

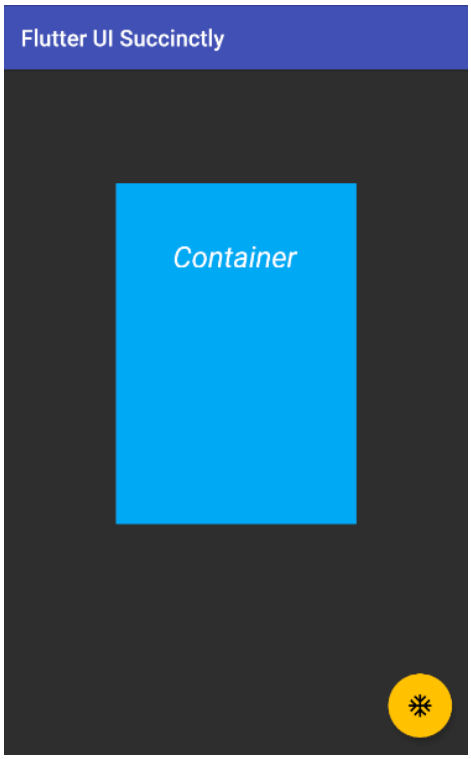

Figure 3-f: A Container Widget with Margin and Padding

By using the **`EdgeInsets.all(100)`** instruction to set the **`margin`** property of the **`Container`** widget, we are indicating that the **`Container`** widget will be placed 100 pixels away, in all offsets in terms of visual edges.

By using the **`EdgeInsets.all(50)`** instruction to set the **`padding`** property of the **`Container`** widget, we are indicating that the content of the **`Container`** widget (in this case the **`Text`** widget assigned to its **`child`** property) will be placed 50 pixels away, in all offsets.

However, you might be asking yourself, why then is the **`Text`** widget within the **`Container`** widget not centered, if **`padding`** of 50 pixels was applied in all four offsets (left, top, right, and bottom)?

The reason is that there is a fixed value assigned to the **`width`** and **`height`** properties. If we want to see the **`Text`** aligned within the **`Container`** widget, based on the **`padding`** assigned, we need to remove the **`width`** and **`height`**. Let's do that now.

Code Listing 3-e: Updated main.dart (Container Margin and Padding, Width and Height Removed)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
```

```
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          margin: EdgeInsets.all(100),
          padding: EdgeInsets.all(50),
          color: Colors.lightBlue,
          child: Text('Container'),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After we remove the **width** and **height** properties and save the changes to **main.dart**, the app's UI should be updated on the emulator. On my machine, it looks as follows.

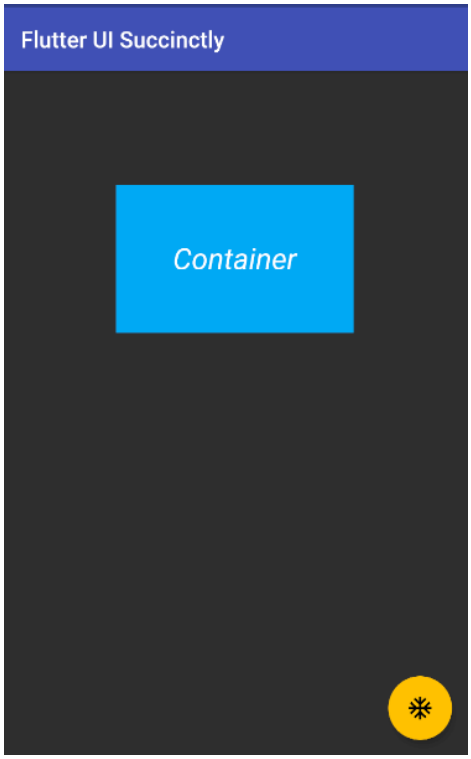

Figure 3-g: A Container Widget with Margin and Padding (Width and Height Removed)

After removing the **`width`** and **`height`** properties from the **`Container`** widget, we can see that the **`padding`** value is visible on all four offsets, which results in the **`Text`** widget being centered within the content of the **`Container`** widget.

After doing this small exercise, there's one key takeaway to keep in mind: sometimes to get a widget's desired look and feel, you'll have to be patient and experiment a bit, even though you might know Flutter very well.

Box decorations

A **`Container`** widget, which has a square or rectangular shape, might not always be best suited for the look and feel of our app's UI. This is when the **`BoxDecoration`** class comes in handy.

Let's give this a try. I've made some changes to the existing code, which are highlighted in bold in the following listing.

Code Listing 3-f: Updated main.dart (Container with a BoxDecoration)

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

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

class MyApp extends StatelessWidget {
```

```
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          margin: EdgeInsets.all(100),
          padding: EdgeInsets.all(50),
          decoration: BoxDecoration(
            color: Colors.lightBlue,
            shape: BoxShape.rectangle,
          ),
          child: Text('Container'),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After these changes are saved to **main.dart**, the app's UI should be updated on the emulator. On my machine, it looks as follows.

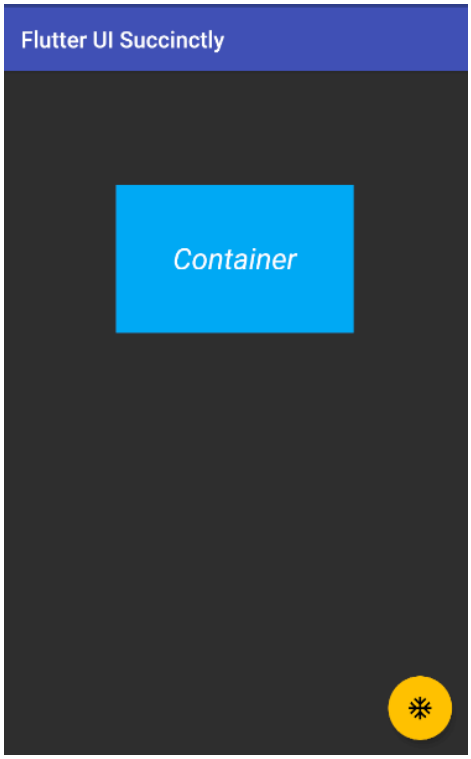

Figure 3-h: A Container Widget with a BoxDecoration (Rectangle)

You might be wondering, what's the point of changing the code to use a **`BoxDecoration`** widget, if we end up with the same look and feel we previously had?

Before answering this question, let's look at the code changes. To the **`Container`** widget, we added a **`decoration`** property, to which we assigned a **`BoxDecoration`** widget.

The **`color`** property was moved within the **`BoxDecoration`** widget, and a **`shape`** property with a value of **`BoxShape.rectangle`** was added.

The reason why no apparent UI changes are visible is that the value of the **`shape`** property was set to **`BoxShape.rectangle`**.

If we want to see some visible changes, let's set the value of the shape property to **`BoxShape.circle`**. This change is highlighted in bold in the following code.

Code Listing 3-g: Updated main.dart (Container with a Box Decoration Using a Circle)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
```

```
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          margin: EdgeInsets.all(100),
          padding: EdgeInsets.all(50),
          decoration: BoxDecoration(
            color: Colors.lightBlue,
            shape: BoxShape.circle,
          ),
          child: Text('Container'),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After these changes are saved to **main.dart**, the app's UI should be updated on the emulator. On my machine, it looks as follows.

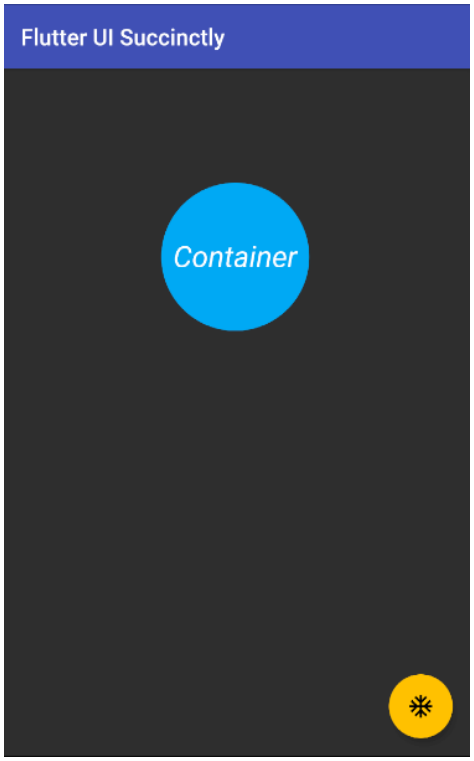

Figure 3-i: A Container Widget with a BoxDecoration (Circle)

Alright, that's certainly different than what we had before. But, in this context, a circle is also not very useful or aligned with the look and feel we want to give our app.

Maybe adding rounded corners would be more practical from a UI perspective. We can do this by adding a **`borderRadius`** property to the **`BoxDecoraton`** widget. The changes to the code are highlighted in bold in the following listing.

Code Listing 3-h: Updated main.dart (Container with a BoxDecoration Using borderRadius)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          margin: EdgeInsets.all(100),
```

```
          padding: EdgeInsets.all(50),
          decoration: BoxDecoration(
            color: Colors.lightBlue,
            shape: BoxShape.rectangle,
            borderRadius: BorderRadius.only(
              topRight: Radius.elliptical(50, 50),
              bottomLeft: Radius.elliptical(25, 25),
            ),
          ),
          child: Text('Container'),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After these changes are saved to **main.dart**, the app's UI should be updated on the emulator. On my machine, it looks as follows.

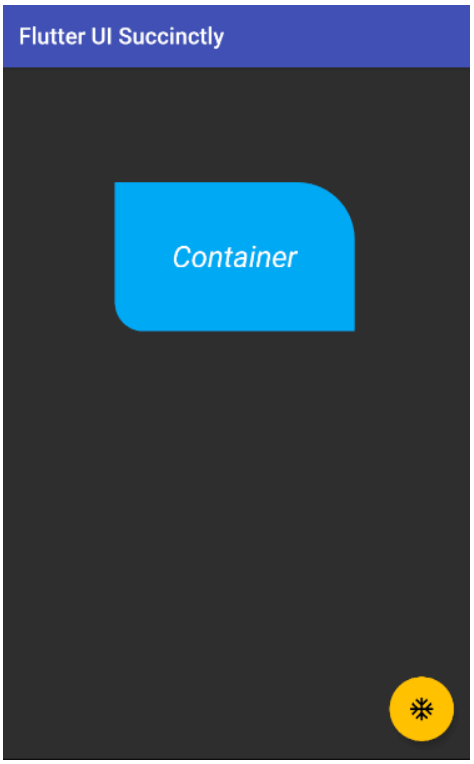

Figure 3-j: A Container Widget with a BoxDecoration (Using borderRadius)

Wow—that looks so much better than using a circle! I'm sure you agree. Let's analyze these changes.

First, we changed the shape of the **`BoxDecoration`** widget back to **`BoxShape.rectangle`**. Second, we have added a **`borderRadius`** property to the **`BoxDecoration`** widget.

The **`borderRadius`** property takes a BorderRadius widget that gets instantiated using the **`only`** constructor method.

This creates a border radius with only the given non-zero values (the **`topRight`** and **`bottomLeft`** corners, in this case). The other corners will be right angles.

Both the **`topRight`** and **`bottomLeft`** corners are assigned the values returned from invoking the Radius.elliptical constructor method, which can create an elliptical radius with the given radii.

The **`topRight`** and **`bottomLeft`** corners look different because the values passed to the **`Radius.elliptical`** method is not the same for both.

You can further experiment with this example, and instead of changing the values for the **`topRight`** and **`bottomLeft`** corners, you could change the **`topLeft`** and **`bottomRight`** values, or do any other combination.

Feel free to use the Radius.circular constructor method as well. The following listing contains an example, where the changes are highlighted in bold.

Code Listing 3-i: Updated main.dart (Container with a BoxDecoration Using borderRadius – Another Example)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          margin: EdgeInsets.all(100),
          padding: EdgeInsets.all(50),
          decoration: BoxDecoration(
            color: Colors.lightBlue,
            shape: BoxShape.rectangle,
            borderRadius: BorderRadius.only(
              topRight: Radius.elliptical(50, 50),
              topLeft: Radius.circular(20),
              bottomRight: Radius.elliptical(25, 25),
            ),
          ),
          child: Text('Container'),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
```

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

After you have made these changes and saved **main.dart**, the app's UI should be updated on the emulator. On my machine, it looks as follows.

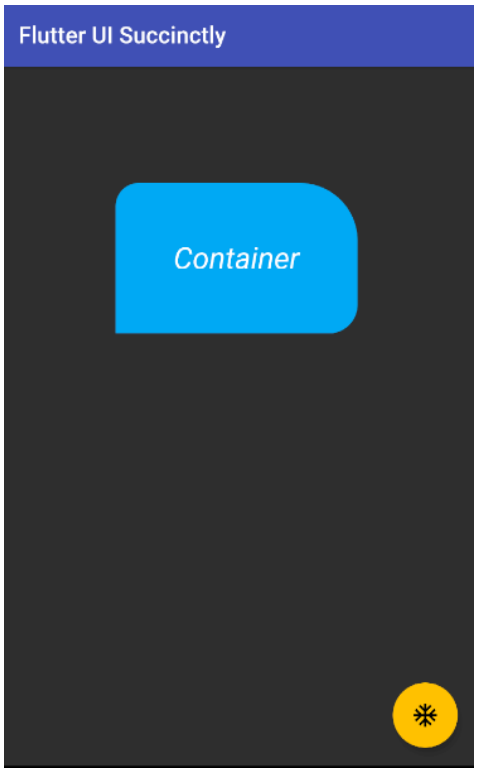

Figure 3-k: A Container Widget with a BoxDecoration (Using borderRadius – Another Example)

As you have seen, there are quite a lot of possibilities to create useful **`Container`** widget shapes when working with the **`BoxDecoration`** class. As long as you have a vivid imagination, there are a ton of interesting combinations you can create.

Images

There are occasions when just having a nice looking **`Container`** widget might not be enough for your app's requirements, and you will have to add an image.

We all know how the old saying goes: *a picture is worth a thousand words*. Images are a great way to add value to your application and provide a user-friendly experience.

Before we can add an image, we need to do a few things. First, we need to find a suitable image. Pixabay is a stock photo and picture website that contains free-to-use images.

I found the following image on the Pixabay website that I'm going to download and use for this example.

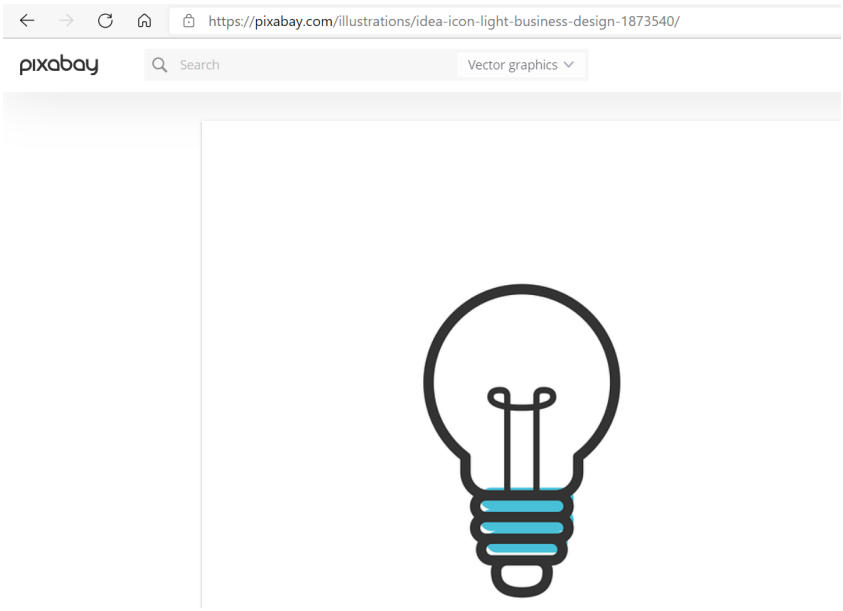

Figure 3-l: Pixabay Stock Image

Once the image has been downloaded, I'm going to switch over to VS Code and create a new folder called **images** by (1) clicking on the new folder icon under the Flutter project directory, and (2) placing the downloaded image within that folder, as you can see in Figure 3-m.

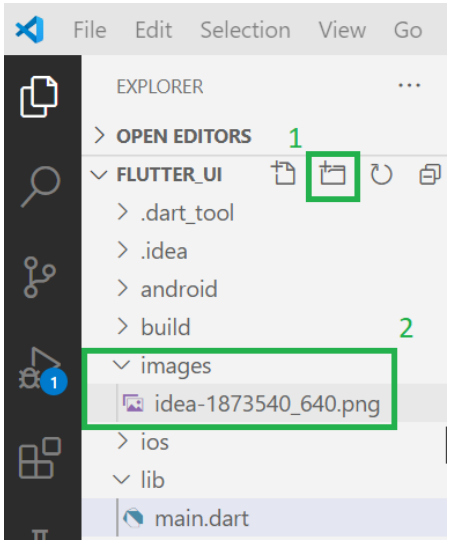

Figure 3-m: Adding an Image Asset to the Flutter Project (VS Code)

Now that we've created the **images** folder and placed the image within it, we need to update our project's **pubspec.yaml** file, add an **assets** section, and add the path to the image.

Here is my updated **pubspec.yaml** file with the changes highlighted in bold.

Code Listing 3-j: Updated pubspec.yaml (Including the Downloaded Image Path)

```
name: flutter_ui
description: A new Flutter project.
publish_to: 'none'
version: 1.0.0+1

environment:
  sdk: ">=2.7.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter
  cupertino_icons: ^1.0.0

dev_dependencies:
  flutter_test:
    sdk: flutter

flutter:
  uses-material-design: true
  assets:
    - images/idea-1873540_640.png
```

As you can see, the **assets** section has been added, along with the file path to the image, within the **images** folder where it has been placed.

With that ready, let's make some code changes to the **main.dart** file to use the image within the **`Container`** widget. The changes are highlighted in bold.

Code Listing 3-k: Updated main.dart (Container with a DecorationImage)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
```

```
          margin: EdgeInsets.all(100),
          padding: EdgeInsets.all(50),
          decoration: BoxDecoration(
            color: Colors.lightBlue,
            shape: BoxShape.rectangle,
            borderRadius: BorderRadius.only(
              topRight: Radius.elliptical(50, 50),
              topLeft: Radius.circular(20),
              bottomRight: Radius.elliptical(25, 25),
            ),
            image: DecorationImage(
              image: AssetImage("images/idea-1873540_640.png"),
              fit: BoxFit.cover,
            ),
          ),
          child: Text('Container'),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After you've made these changes and saved **main.dart**, the app's UI should be updated on the emulator. On my machine, it looks as follows.

Figure 3-n: A Container Widget with a DecorationImage

To achieve this, we added an **`image`** property to the **`BoxDecoration`** widget. This image property takes a **`DecorationImage`** widget.

The **`image`** property of the **`DecorationImage`** widget indicates the file path where the image can be found. This path is passed as a parameter to the constructor of the **`AssetImage`** class, which is used for fetching images that are specified as Flutter project assets.

The **`fit`** property is used to indicate how much space the image will cover within the **`BoxDecoration`** widget. In this case, we are indicating that the image will cover the space available, which is why the **`BoxFit.cover`** value is assigned to it.

Cool—now we know how to add an image. But what happened to the **`Text`** widget? As you can see from the preceding code, the **`Text`** widget is still being used, and it's assigned to the **`child`** property of the **`Container`** widget. So, why isn't it visible?

The reason that the **`Text`** widget is barely visible is that the color of the text is white, so it fades almost entirely into the background image.

Instead of changing the color of the text, let's add opacity to the image. The changes to the code are highlighted in bold in the following listing.

Code Listing 3-l: Updated main.dart (Container with a DecorationImage with Opacity)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          margin: EdgeInsets.all(100),
          padding: EdgeInsets.all(50),
          decoration: BoxDecoration(
            color: Colors.lightBlue,
            shape: BoxShape.rectangle,
            borderRadius: BorderRadius.only(
              topRight: Radius.elliptical(50, 50),
              topLeft: Radius.circular(20),
              bottomRight: Radius.elliptical(25, 25),
            ),
            image: DecorationImage(
              colorFilter: ColorFilter.mode(
                  Colors.black.withOpacity(0.6), BlendMode.dstATop),
              image: AssetImage("images/idea-1873540_640.png"),
              fit: BoxFit.cover,
            ),
          ),
          child: Text('Container'),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
```

```
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After you've made these changes and saved **main.dart**, the app's UI should be updated on the emulator. On my machine, it looks as follows.

Figure 3-o: A Container Widget with a DecorationImage (with Opacity)

Awesome—we can see that text is visible again. Let's have a look at the code changes to understand how this works.

What we have done is added a **`colorFilter`** property to the **`DecorationImage`** widget. To this property, we have assigned the value returned by the ColorFilter.mode constructor method. This method creates a color filter that applies the blend mode given as the second parameter—in this case, **`BlendMode.dstATop`**.

We are creating a new blended color that originates from black, on top of the image, with the alpha channel replaced with the given opacity, which is what the instruction **`Colors.black.withOpacity(0.6)`** does.

We can also do some other interesting things with the image, like repeating it. Let's have a look. The changes are highlighted in bold in the following listing.

Code Listing 3-m: Updated main.dart (Container with Image Repetition)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          width: 450,
          height: 450,
          margin: EdgeInsets.all(100),
          padding: EdgeInsets.all(50),
          decoration: BoxDecoration(
            color: Colors.lightBlue,
            shape: BoxShape.rectangle,
            borderRadius: BorderRadius.only(
              topRight: Radius.elliptical(50, 50),
              topLeft: Radius.circular(20),
              bottomRight: Radius.elliptical(25, 25),
            ),
            image: DecorationImage(
              colorFilter: ColorFilter.mode(
                  Colors.black.withOpacity(0.6), BlendMode.dstATop),
              image: AssetImage("images/idea-1873540_640.png"),
              repeat: ImageRepeat.repeatY,
            ),
          ),
          child: Text('Container'),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
```

```
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After you've made these changes and saved **main.dart**, the app's UI should be updated on the emulator. On my machine, it looks as follows.

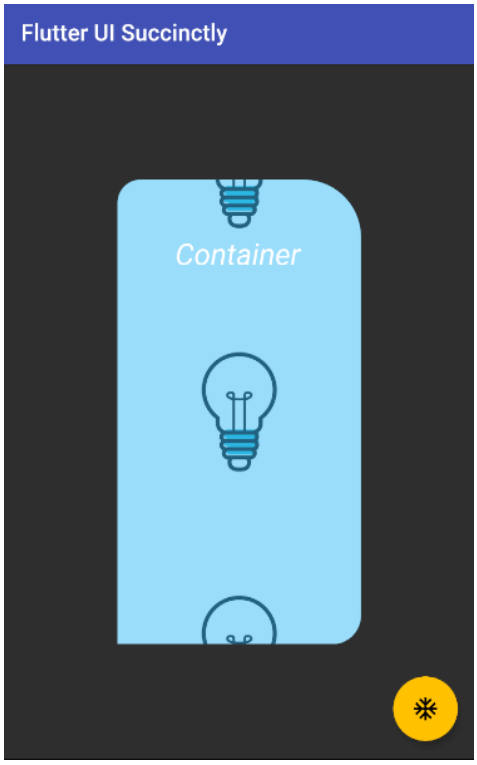

Figure 3-p: A Container Widget with a DecorationImage (with Image Repetition)

The first thing to notice (which might not be that obvious) is that the **`fit`** property was removed from the **`DecorationImage`** widget. This was done so that the image doesn't take all the widget's area.

Next, to the **`Container`** widget, we added fixed values for the **`width`** and **`height`** properties. This was done so that the **`Container`** widget is big enough to be able to repeat the image.

Finally, we added a **repeat** property to the **DecorationImage** widget, setting its value to **ImageRepeat.repeatY**, which indicates that the image will be repeated vertically.

Gradients

Now that we've seen how to use the **BoxDecoration** widget and also how to work with images, we can also use Flutter's Gradient class.

We can easily achieve this by adding a **gradient** property to the **BoxDecoration** widget. Let's have a look. The code changes are highlighted in the following listing.

Code Listing 3-n: Updated main.dart (Container with a LinearGradient)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          decoration: BoxDecoration(
            color: Colors.lightBlue,
            gradient: LinearGradient(
                begin: Alignment.topRight,
                end: Alignment.bottomLeft,
                colors: [Colors.blue, Colors.orange]),
            shape: BoxShape.rectangle,
          ),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
```

```
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After we've made these changes and saved **main.dart**, the app's UI should be updated on the emulator. On my machine, it looks as follows.

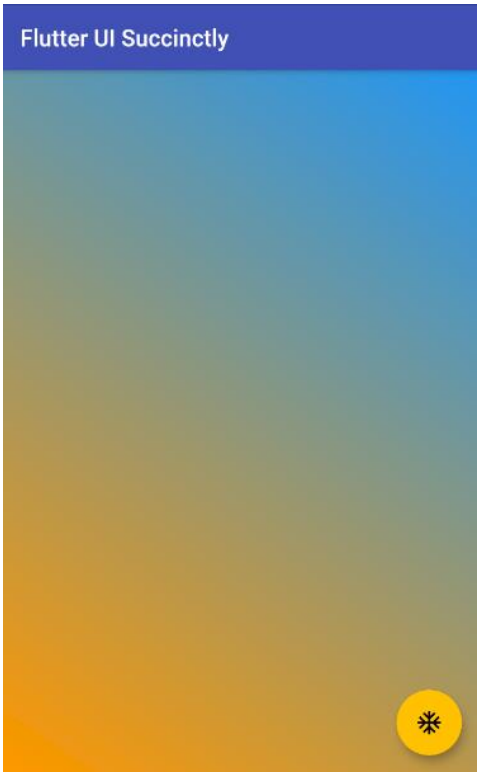

Figure 3-q: A Container Widget with a LinearGradient

Now, look at that—the UI looks awesome! You might have noticed that besides the changes highlighted in bold, the **`margin`**, **`padding`**, and **`child`** properties of the **`Container`** widget were removed. This was done so that the **`Container`** widget could take the available space.

Furthermore, the **`borderRadius`** property was removed from the **`BoxDecoration`** widget, and a **`gradient`** property was added.

With regards to the **gradient** property, this is assigned to a LinearGradient widget, starting with the **blue** color on the **topRight** corner, and ending with the **orange** color on the **bottomLeft** corner. To understand this better, let's look at the following diagram.

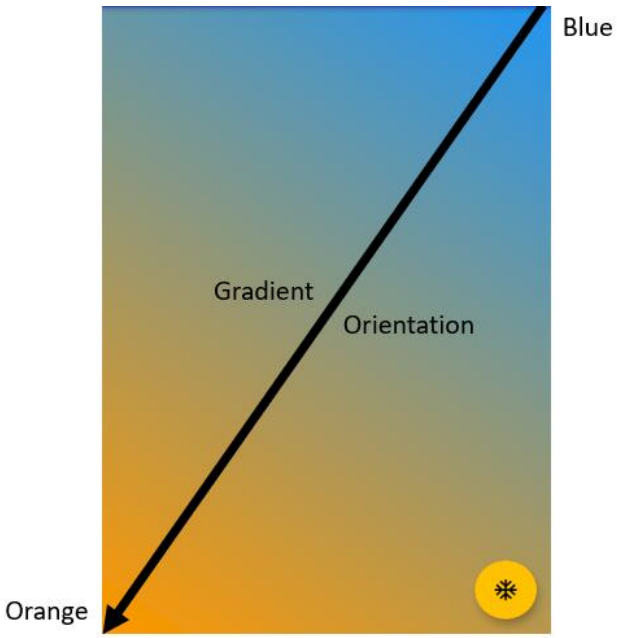

Figure 3-r: LinearGradient Orientation Example

By using the **LinearGradient** widget, we can combine colors from a beginning to an endpoint, resulting in a beautiful display of gradients using the colors specified within the **colors** property.

Experimenting with gradients is a bit like being a painter. You have a canvas, multiple colors, and combinations to experiment with, where you are only limited by your imagination (and the colors available).

Let's experiment a bit more with gradients, and make some changes to the code. These changes are highlighted in bold in the following code.

Code Listing 3-o: Updated main.dart (Container with a LinearGradient – Mirror TileMode)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
```

```
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          decoration: BoxDecoration(
            color: Colors.lightBlue,
            gradient: LinearGradient(
                begin: Alignment(0, -1),
                end: Alignment(0, -0.4),
                tileMode: TileMode.mirror,
                colors: [Colors.blue, Colors.orange]),
            shape: BoxShape.rectangle,
          ),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

Before we save the changes and see them on the emulator, let's try to understand how alignment coordinates work in Flutter by looking at the following diagram.

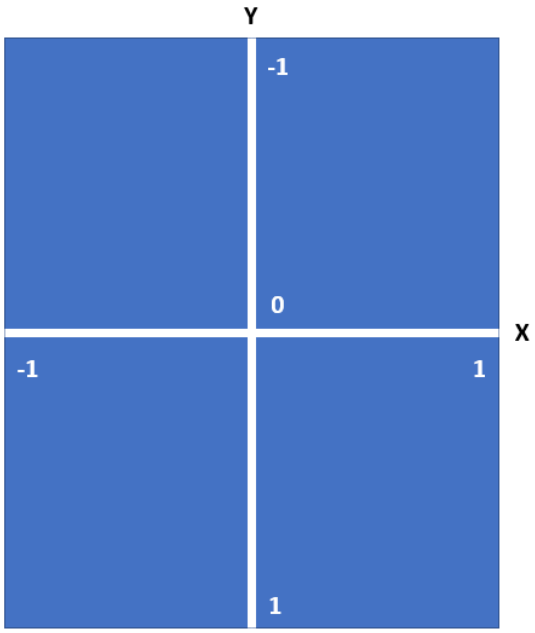

Figure 3-s: Alignment Coordinates in Flutter

By using the alignment coordinates between the values of **-1** and **1**, you can align widgets in the available space, both horizontally and vertically.

So, in this latest example, what we have done is set the value of the **`begin`** property (which uses the **`blue`** color) to start with the coordinates **`X=0`**, **`Y=-1`**. This will make the **`blue`** color appear first within the available space.

We've also set the value of the **`end`** property (which uses the **`orange`** color) to start with the coordinates **`X=0`**, **`Y=-0.4`**. This will make the **`orange`** color appear after the **`blue`** color within the available space, but while taking a bit less space.

After making these changes and saving **main.dart**, we should see the app's UI updated on the emulator. On my machine, it looks as follows.

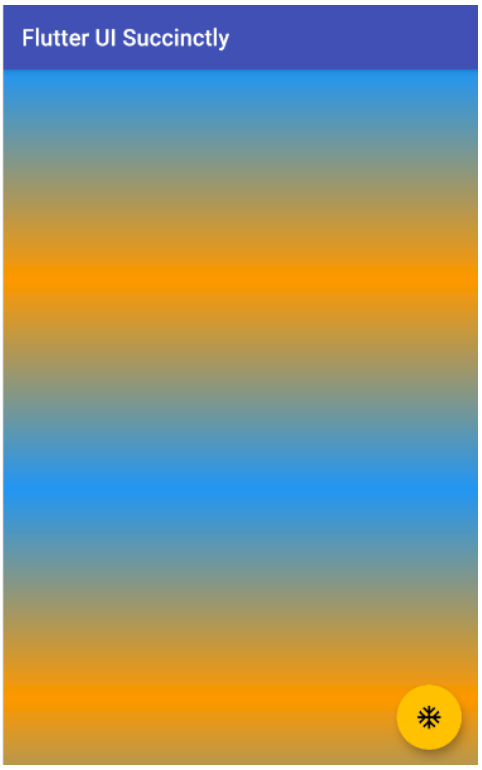

Figure 3-t: A Container Widget with a LinearGradient – Mirror TileMode

I'm sure you agree with me that this effect looks quite cool. This is one of the many reasons why Flutter, in my opinion, is such a great platform for creating amazing UIs.

However, if the **`orange`** color in principle should be taking a bit less space than the **`blue`** color, how is it possible that to the naked eye, it would seem that the **`orange`** color is more prevalent than the **`blue`** color?

The reason that the orange color seems more prevalent is that we are using a **`TileMode.mirror`** effect. This means that the original color reflects itself (including its gradient variations), resulting in two areas with **`blue`**, and two areas with **`orange`**.

Therefore, given the dimensions of the emulator screen, and the order in which the colors are positioned, the **`orange`** color and its gradient variations are slightly favored, rather than the **`blue`** color and its variations. Thus, the **`orange`** seems a bit more prevalent.

Let's experiment a little bit more, by using a RadialGradient class instead of the **`LinearGradient`** we are currently using.

What I would like to achieve is to have a background effect that originates from the center of the **`Container`** widget, using concentric circles that begin with **`blue`**, then use **`deepPurple`** (as the background), and end with **`lightBlue`**.

In other words, something that looks similar to two images I've found on this science photo library website.

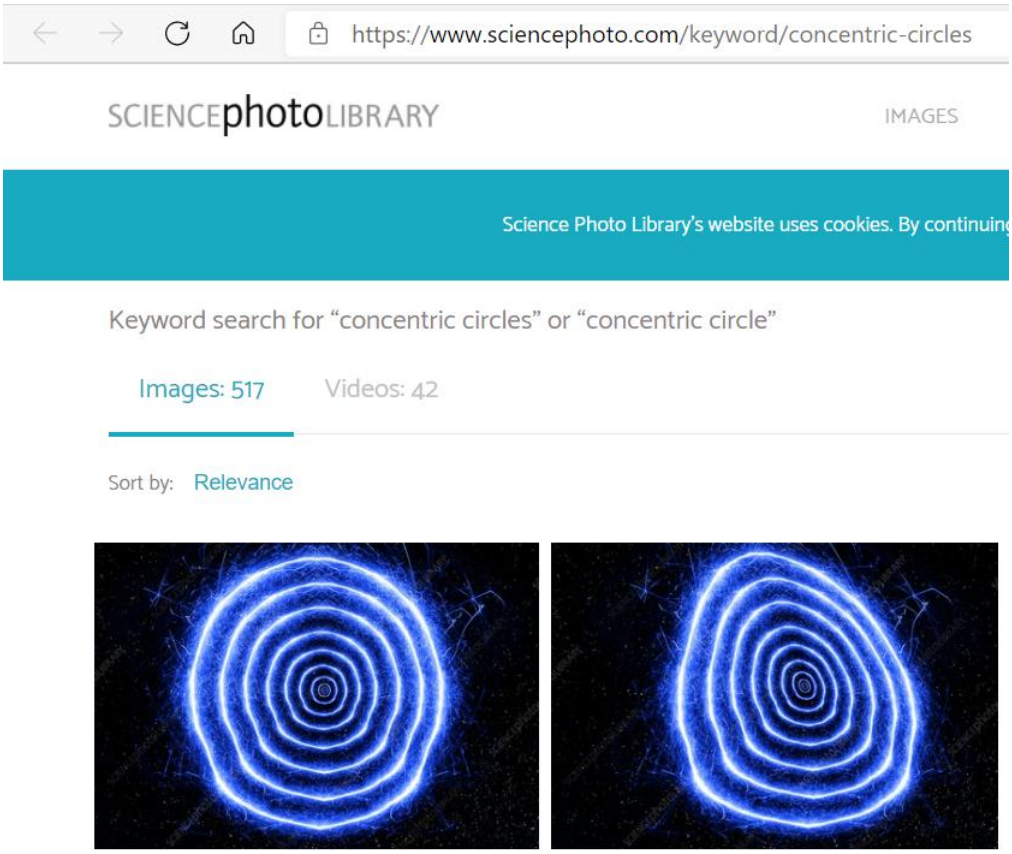

Figure 3-u: The Desired RadialGradient Effect

Creating something similar to this in Flutter is easier than you might think. Let's see how—the changes are highlighted in bold in the following code.

Code Listing 3-p: Updated main.dart (Container with a RadialGradient)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          decoration: BoxDecoration(
```

```
            color: Colors.lightBlue,
            gradient: RadialGradient(
                radius: 0.15,
                center: Alignment(0, 0),
                tileMode: TileMode.mirror,
                colors: [Colors.blue, Colors.deepPurple,
                  Colors.lightBlue]),
            shape: BoxShape.rectangle,
          ),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After making these changes and saving **main.dart**, we should see the app's UI updated on the emulator. On my machine, it looks as follows.

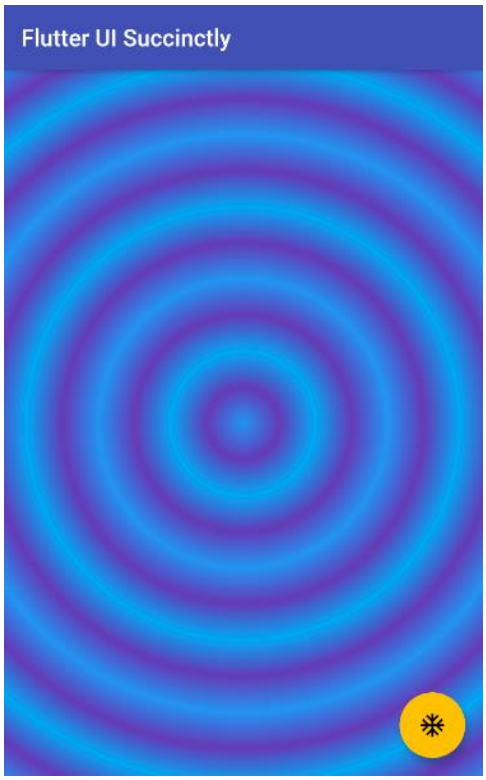

Figure 3-v: A Container Widget with a RadialGradient (Concentric Circles)

That looks fairly similar to the concentric circles from the science photo library website, which is awesome!

By looking at the code changes, we can see that a **`RadialGradient`** widget was used instead of a **`LinearGradient`** widget.

The two key properties that make the effect possible are the **`radius`** property, which indicates how big the circles are, and the **`center`** property, which specifies where the innermost circle (the first drawn) and subsequent ones will be placed on the screen.

The higher the value of the **`radius`** property, the larger the circles, which means that fewer of them would fit on the screen. The lower the value of **`radius`** property, the smaller the circles, which means that more of them would fit on the screen.

By setting the value of the **`center`** property to **`Alignment(0, 0)`**, we are indicating that the most inner circle will be drawn on the alignment coordinates with a value of **`X=0`**, **`Y=0`**, which corresponds to the center of the **`Container`** widget.

Finally, instead of using two colors, we used three colors, which are rendered in the same order as they are described within the **`colors`** property of the **`RadialGradient`** widget.

By implementing these small changes, we were able to come up with an interesting effect.

Summary

Throughout this chapter, we've explored in depth how to create and work with the **`Container`** widget. This is one of the most useful widgets that Flutter has to offer, and it is a fundamental component when it comes to building Flutter UIs.

Although quite a lot can be achieved when working with **`Container`** and related **`child`** widgets, we are still missing a couple of key features required for building a UI with Flutter, such as rows and columns.

When rows and columns are used in combination with **`Scaffold`** and **`Container`** widgets, we suddenly find ourselves with unlimited possibilities of what layouts we can build with Flutter. This is what the next chapter is all about.

Chapter 4 Rows and Columns

Overview

To be able to create and structure a complex UI with Flutter, which involves using multiple widgets, it might be necessary to use Row and Column classes. They are very easy to use and have some unique properties that give them flexibility and power when designing layouts. In this chapter, we will explore how to work with them, which will give us the ability to better understand how to build more complex layouts and UIs.

Definitions

Before we dive into specific Flutter details regarding rows and columns, let's make sure we have a clear understanding of what rows and columns are. So, let's have a look at the following diagram.

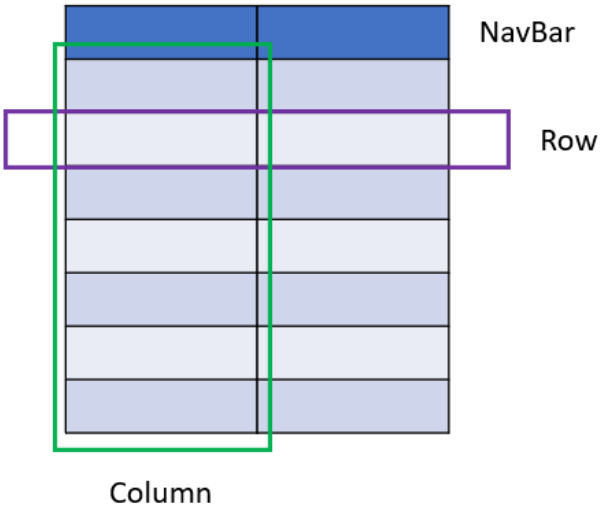

Figure 4-a: App Layout with Rows and Columns

In the preceding diagram, we have our app's layout, which consists of various rows and two columns. One of the rows is highlighted in purple, and the left column is highlighted in green.

Both the rows and columns contain cells, which look like rectangles. Each cell could perfectly contain a Flutter widget. Therefore, a **`Row`** widget is a list of **`child`** widgets placed horizontally, while a **`Column`** widget is a list of **`child`** widgets that are vertically stacked.

Rows and columns have the same properties, which makes it easy to learn how to use them. This means that whatever property you use in a **`Row`** widget, you can also use it in a **`Column`** widget.

Alignment

Before we can start using rows and columns in our Flutter code, we need to understand how the alignment of rows and columns differ one from the other.

Say we have a row (the blue box in Figure 4-b), and we draw a white cross inside of it. The line that goes from top to bottom is known as the **cross axis** of the row. The line that goes from left to right is known as the **main axis** of the row. This can be seen in the following diagram.

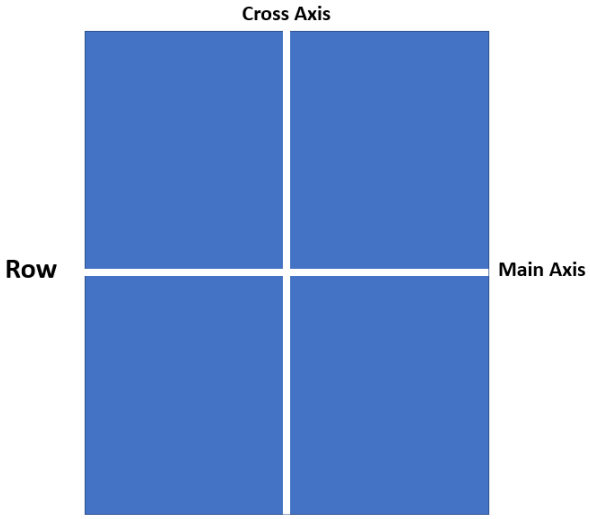

Figure 4-b: Row – Cross and Main Axis

Say we have a column, and we draw that same white cross inside of it. The line that goes from top to bottom is known as the main axis of the column.

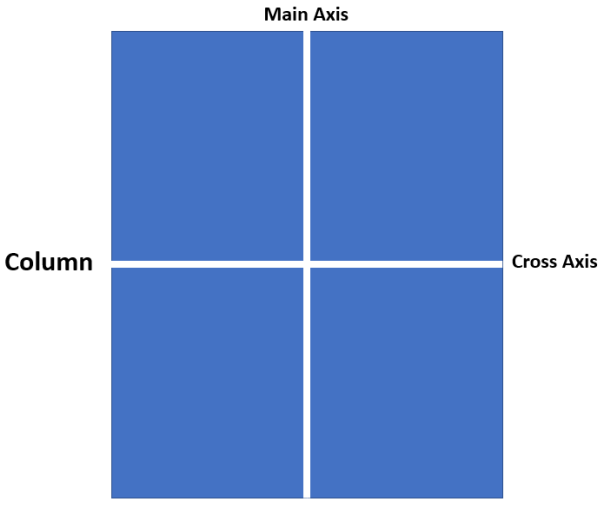

Figure 4-c: Column – Main and Cross Axis

In essence, in any screen, there are always two axes: the vertical axis and the horizontal axis. The vertical axis is called the cross axis for a **`Row`** widget, and the main axis for a **`Column`**

widget. The horizontal axis is called the main axis for a **`Row`** widget and the cross axis for a **`Column`** widget.

This means that the cross axis and the main axis change based on the widget that you are using. Understanding alignment is essential when working with widgets that are placed within rows and columns.

Boxes

To put these concepts into practice, let's take a step back and use the following code as a starting point.

Code Listing 4-a: Updated main.dart (Container with a LinearGradient)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          decoration: BoxDecoration(
            color: Colors.lightBlue,
            gradient: LinearGradient(
                begin: Alignment.topRight,
                end: Alignment.bottomLeft,
                colors: [Colors.blue, Colors.orange]),
            shape: BoxShape.rectangle,
          ),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
```

```
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After making these changes and saving **main.dart**, we should see the app's UI updated on the emulator. On my machine, it looks as follows.

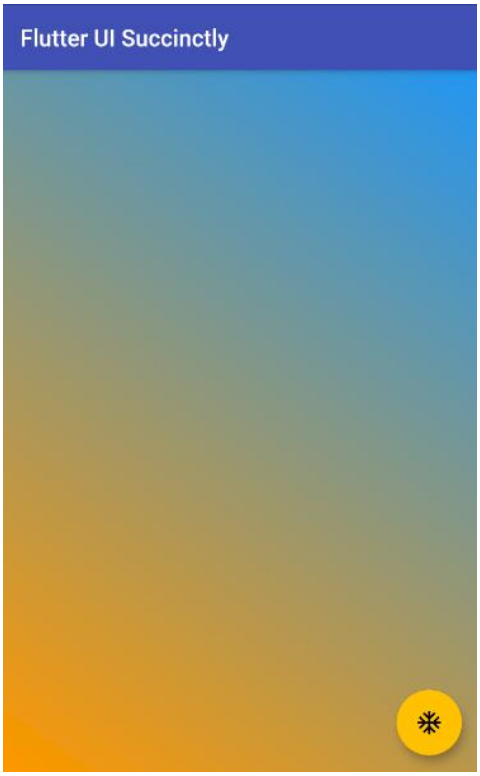

Figure 4-d: A Container Widget with a LinearGradient

We've seen this before, so no surprises here. However, what we want to do next is create a **`Column`** widget (which later we can swap for a **`Row`** widget) that we can add as a **`child`** property to the existing **`Container`** widget.

This **`child`** property of the existing **`Container`** widget will be made up of a list of boxes (each of which will be a **`Container`** widget).

Before we make any changes to the code, let's look at the following diagram to understand what we want to achieve.

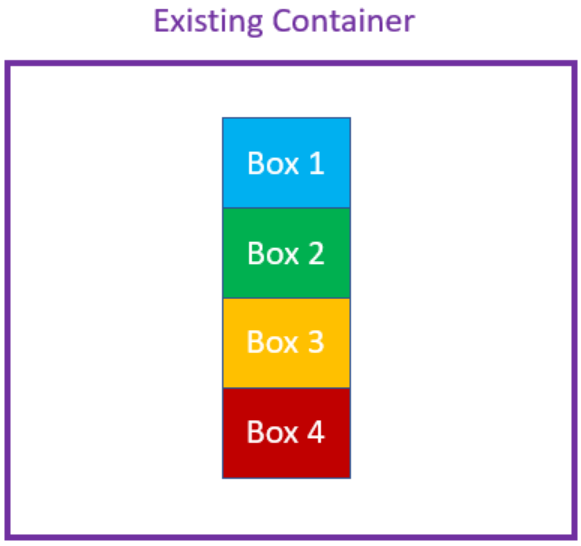

Figure 4-e: A Container Widget with a Column of Boxes (Each Box is a Container)

To achieve this, we need to add a **`Column`** widget to the **`Container`** widget's **`child`** property. The **`Column`** widget contains a **`children`** property, to which we will assign a **`List`** of **`Container`** widgets that we will create with a separate method called **`boxes`**. The code changes are highlighted in bold in the following listing.

Code Listing 4-b: Updated main.dart (Container with a Column of Containers – Dynamically Created)

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

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

class MyApp extends StatelessWidget {
  List<Widget> boxes(int n, double w, double h) {
    List<Widget> bxs = List<Widget>();
    List fill = [Colors.blue, Colors.green, Colors.purple, Colors.pink];
    for (int i = 0; i <= n - 1; i++) {
      Container bx = Container(
        child: Text(i.toString()),
        color: fill[i],
        width: w,
        height: h,
      );
      bxs.add(bx);
    }
    return bxs;
```

```
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          decoration: BoxDecoration(
            color: Colors.lightBlue,
            gradient: LinearGradient(
                begin: Alignment.topRight,
                end: Alignment.bottomLeft,
                colors: [Colors.blue, Colors.orange]),
            shape: BoxShape.rectangle,
          ),
          child: Column(
            children: boxes(4, 40, 40),
          ),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After making these changes and saving **main.dart**, we should see the app's UI updated on the emulator. On my machine, it looks as follows.

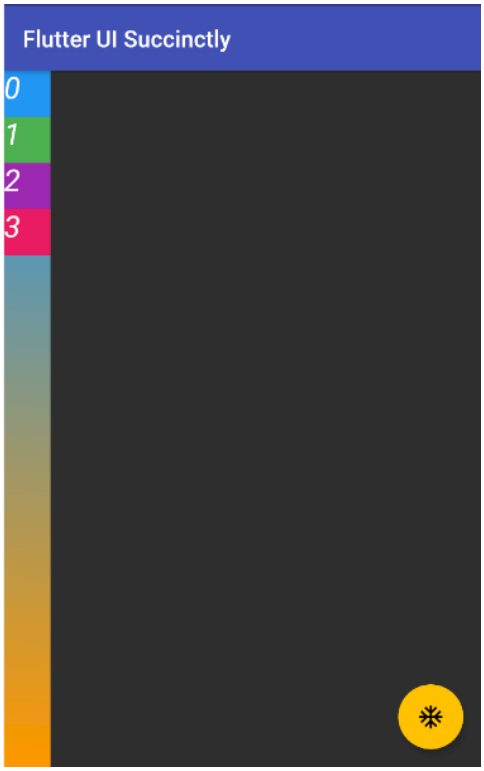

Figure 4-f: A Container with a Column of Containers – Dynamically Created

By looking at the code changes, we can see that the boxes are dynamically created by a method called **boxes**, which returns a list of widgets (**List<Widget>**). Each widget (box) within that list is a **Container**.

This list is assigned to the **children** property of the **Column** widget, which is assigned to the **child** property of the main **Container** widget.

The **boxes** method starts by initializing the **bxs** variable as an empty list of widgets (**List<Widget>**). This will contain each of the **Container** widgets (boxes) that will be dynamically created.

Then a list of colors called **fill** is defined. Each box (**Container**) will have its color—one color for each box.

We loop **n** times, and for each iteration, a **Container** is created, with a defined **width** (**w**) and **height** (**h**), along with its **color** (**fill[i]**).

Each **Container** created is added to the **bxs** list for every iteration, and when the loop has been completed, the **bxs** list is returned, which is assigned to the **children** property of the **Column** widget.

What happens when we change the **Column** widget to a **Row** widget? Let's have a look.

Code Listing 4-c: Updated main.dart (Container with a Row of Containers – Dynamically Created)

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

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

class MyApp extends StatelessWidget {
  List<Widget> boxes(int n, double w, double h) {
    List<Widget> bxs = List<Widget>();
    List fill = [Colors.blue, Colors.green, Colors.purple, Colors.pink];
    for (int i = 0; i <= n - 1; i++) {
      Container bx = Container(
        child: Text(i.toString()),
        color: fill[i],
        width: w,
        height: h,
      );
      bxs.add(bx);
    }
    return bxs;
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          decoration: BoxDecoration(
            color: Colors.lightBlue,
            gradient: LinearGradient(
                begin: Alignment.topRight,
                end: Alignment.bottomLeft,
                colors: [Colors.blue, Colors.orange]),
            shape: BoxShape.rectangle,
          ),
          child: Row(
            children: boxes(4, 40, 40),
          ),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
```

```
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After making these changes and saving **main.dart**, we should see the app's UI updated on the emulator. On my machine, it looks as follows.

Figure 4-g: A Container with a Row of Containers – Dynamically Created

From having a column of boxes, we now have a row of boxes instead. It's the same logic; the **boxes** method hasn't changed. The only difference is that within the main **Container** widget, a **Row** widget is assigned to the **child** property, rather than a **Column** widget.

Alignment adjustment

To position the **Column** or **Row** widget we have created, let's use the axes properties. The changes to the code are highlighted in bold in the following listing.

Code Listing 4-d: Updated main.dart (Row of Containers with Axes – Dynamically Created)

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

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

class MyApp extends StatelessWidget {
  List<Widget> boxes(int n, double w, double h) {
    List<Widget> bxs = List<Widget>();
    List fill = [Colors.blue, Colors.green, Colors.purple, Colors.pink];
    for (int i = 0; i <= n - 1; i++) {
      Container bx = Container(
        color: fill[i],
        width: w,
        height: h,
      );
      bxs.add(bx);
    }
    return bxs;
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          child: Row(
            mainAxisAlignment: MainAxisAlignment.center,
            crossAxisAlignment: CrossAxisAlignment.stretch,
            children: boxes(4, 40, 40),
          ),
```

```
          decoration: BoxDecoration(
            color: Colors.lightBlue,
            gradient: LinearGradient(
                begin: Alignment.topRight,
                end: Alignment.bottomLeft,
                colors: [Colors.blue, Colors.orange]),
            shape: BoxShape.rectangle,
          ),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After making these changes and saving **main.dart**, we should see the app's UI updated on the emulator.

Before you continue reading to have a look at the result, let me give you a heads up. What you will see is counterintuitive to what you expect to see.

The reason for this is that we are using a **`Row`** widget. With that said, you may now have a look.

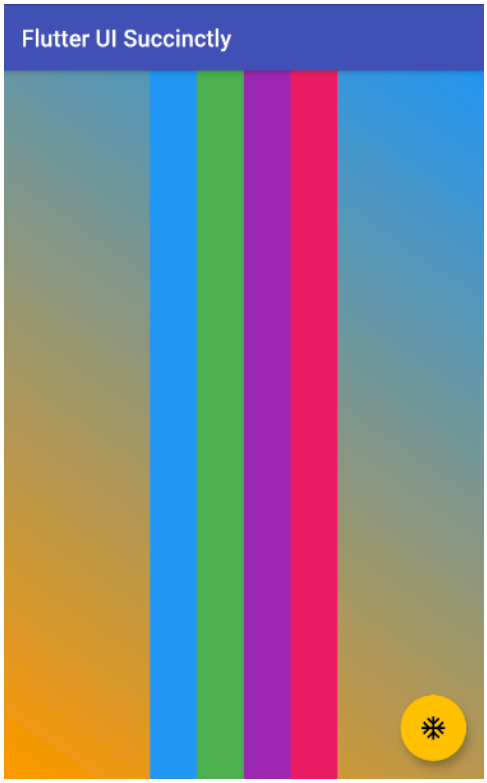

Figure 4-h: Row of Containers with Axes – Dynamically Created

This looks like one of those old TV color bar screens. The really interesting thing about this effect is that we visually ended up with four columns rather than four boxes in a row, even though we are using a **`Row`** widget. Why is that?

Although visually, these look like four columns, they are indeed four child **`Container`** widgets, one after the other, contained within a **`Row`** widget.

The reason they appear as four columns has to do with the **`crossAxisAlignment`** property of the **`Row`** widget being set to **`CrossAxisAlignment.stretch`**. In this way, the boxes have been stretched to the height of the parent **`Container`**.

The key takeaway from this example is that when working with **`Row`** and **`Column`** widgets, there are multiple ways to arrange your widgets on the screen, and sometimes by using a single property, you can end up with a completely different result.

Spacing

Before we wrap up this chapter, there's one more thing I'd like to show you, and that is how to add spacing between the child widgets contained within a **`Row`** or **`Column`** widget.

The following code contains some changes to add spacing between the boxes, highlighted in bold. Let's have a look.

Code Listing 4-e: Updated main.dart (Column of Containers with Axes and Spacing – Dynamically Created)

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

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

class MyApp extends StatelessWidget {
  List<Widget> boxes(int n, double w, double h) {
    List<Widget> bxs = List<Widget>();
    List fill = [Colors.blue, Colors.green, Colors.purple, Colors.pink];
    for (int i = 0; i <= n - 1; i++) {
      Container bx = Container(
        color: fill[i],
        width: w,
        height: h,
      );
      bxs.add(bx);
    }
    return bxs;
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter UI Succinctly'),
        ),
        body: Container(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            crossAxisAlignment: CrossAxisAlignment.stretch,
            children: boxes(4, 40, 40),
          ),
          decoration: BoxDecoration(
            color: Colors.lightBlue,
            gradient: LinearGradient(
                begin: Alignment.topRight,
                end: Alignment.bottomLeft,
                colors: [Colors.blue, Colors.orange]),
            shape: BoxShape.rectangle,
          ),
        ),
```

```
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.ac_unit),
          onPressed: () {
            print('Oh, it is cold outside...');
          },
        ),
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After making these changes and saving **main.dart**, we should see the app's UI updated on the emulator. Let's have a look.

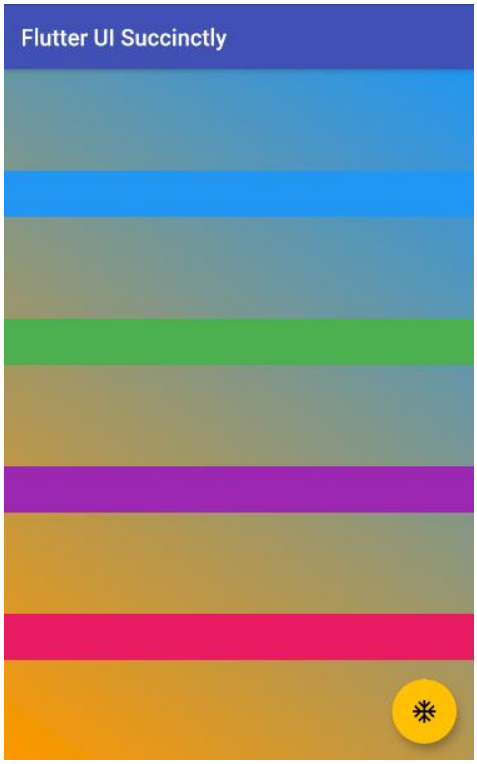

Figure 4-i: Column of Containers with Axes and Spacing – Dynamically Created

From the code changes, two things stand out. The first is that we are no longer using a **`Row`** widget, but instead a **`Column`** widget.

The second is that the value of the **`mainAxisAlignment`** property is now set to **`MainAxisAlignment.spaceEvenly`**, which spaces the columns evenly along the parent **`Container`**.

Although visually these four boxes within a column look like rows, they are four child **`Container`** widgets inside a **`Column`** widget, each box taking the **`width`** of the parent **`Container`** widget.

This is achieved because the **`crossAxisAlignment`** property of the **`Column`** widget is set to **`CrossAxisAlignment.stretch`**, which makes the boxes within the column appear as rows.

Summary

Before we started this chapter, we were only able to work with one **`child`** widget per **`Container`** widget. By adding **`Row`** and **`Column`** widgets, we have seen how we can add more than one **`child`** widget per **`Container`**.

Rows and columns are a great and useful way to enrich a Flutter application's UI, but they are not the only way. Later, we will look at exploring how to use other Flutter widgets, which will allow us to compose more complex UIs.

But before we do that, in the next chapter we will look at navigation widgets, which most Flutter applications rely on.

Chapter 5 Navigation Widgets

Overview

Being able to move around through an app's UI is key for a good user experience and a functional application.

Flutter provides several navigation widgets that can be used to achieve that. This is what this chapter is all about: exploring the navigation possibilities that exist within Flutter.

Succinctly books app

Before we begin exploring the various navigation widgets we have available within Flutter, we are going to refactor the code we've written and add some extra features.

The idea is to showcase the various navigation widgets available in Flutter by creating a Flutter app that can navigate between a reduced list of *Succinctly* books. The changes are highlighted in bold in the following code listing.

Code Listing 5-a: Updated main.dart

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

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

class StaticBooks {
  static const String cdn = "https://cdn.syncfusion.com/";
  static const String path =
  "content/images/downloads/ebook/ebook-cover/";

  static const List<String> covers = [
    "visual-studio-for-mac-succinctly-v1.png",
    "angular-testing-succinctly.png",
    "azure-devops-succinctly.png",
    "asp-net-core-3-1-succinctly.png",
    "angulardart_succinctly.png"
  ];

  static const List<String> titles = [
    "Visual Studio for Mac Succinctly",
    "Angular Testing Succinctly",
    "Azure DevOps Succinctly",
```

```
    "ASP.NET Core 3.1 Succinctly",
    "AngularDart Succinctly"
  ];
}

class Succinctly extends StatelessWidget {
  final String book;
  final String title;

  Succinctly({
    @required this.book,
    @required this.title,
  });

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(title),
      ),
      body: Container(
        decoration: BoxDecoration(
          image: DecorationImage(
            image: NetworkImage(StaticBooks.cdn +
              StaticBooks.path + book),
            fit: BoxFit.scaleDown,
          ),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        child: Icon(Icons.book_online),
        onPressed: () {
          print('Awesome book!');
        },
      ),
    );
  }
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
```

```
      home: Succinctly(
        book: StaticBooks.covers[0],
        title: StaticBooks.titles[0],
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After making these changes and saving **main.dart**, we should see the app's UI updated on the emulator. On my machine, it looks as follows.

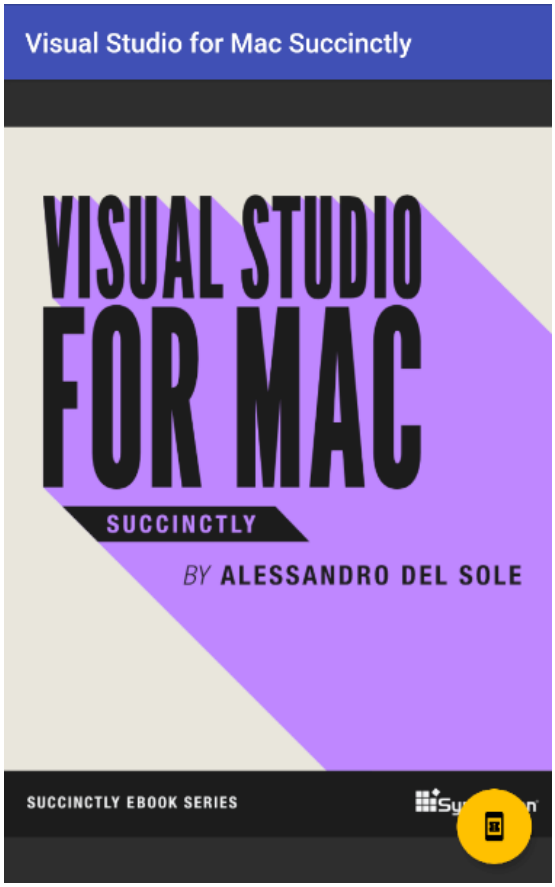

Figure 5-a: Succinctly Books App

That looks cool! With a few code changes, we now have a basic app to view the **covers** of some *Succinctly* books. Let's go over the code changes.

The first thing we did was create the **StaticBooks** class, which contains the hardcoded **covers** and of some **titles** of *Succinctly* books, the base URL (**cdn**) of the content delivery network (CDN), and the location (**path**) where the cover images are located.

Next, we created the **Succinctly** class, which essentially contains the **Scaffold** code that was previously directly assigned to the **home** property of the **MaterialApp** widget.

This means we've taken that code and refactored it into a new class that inherits from **StatelessWidget**. In essence, the **Succinctly** class returns a **Scaffold** widget.

The **Succinctly** class has two properties: **book**, which refers to the book's cover image, and **title**, which refers to the book's name.

The **Succinctly** class has a constructor method with two required parameters: **book** and **title**, which initialize their respective properties.

Finally, the **Succinctly** class has a **build** method that overrides the one inherited from the **StatelessWidget** class, which is responsible for creating the **Scaffold** widget.

With regards to the **appBar** property, the only difference is that we now set the value of the **title** property dynamically.

Regarding the **floatingActionButton** property, the icon was changed to **book_online**, and the text that is printed to the console was also changed to a message more aligned with the topic of this app: books.

The most important changes are related to the **body** property, which is assigned a **Container** widget. The **Container** widget contains a **BoxDecoration** widget, assigned to the **decoration** property, that has a **DecorationImage** widget which uses the NetworkImage widget to fetch an image from the web.

The image that is fetched from the web using the **NetworkImage** widget is the result of the concatenation of the (**StaticBooks.cdn + StaticBooks.path + book**) expression.

PopupMenuButton

Perhaps the easiest way to add navigation features and control to a Flutter application is by using a **PopupMenuButton** widget. The **PopupMenuButton** widget shows a menu when pressed. Let's add a **PopupMenuButton** widget to the app's toolbar as a way to navigate to the different *Succinctly* books. The changes are highlighted in bold.

Code Listing 5-b: Updated main.dart – Using a PopupMenuButton

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

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

class StaticBooks {
  static const String cdn = "https://cdn.syncfusion.com/";
  static const String path =
    "content/images/downloads/ebook/ebook-cover/";

  static const List<String> covers = [
    "visual-studio-for-mac-succinctly-v1.png",
    "angular-testing-succinctly.png",
    "azure-devops-succinctly.png",
    "asp-net-core-3-1-succinctly.png",
    "angulardart_succinctly.png"
  ];

  static const List<String> titles = [
    "Visual Studio for Mac Succinctly",
    "Angular Testing Succinctly",
    "Azure DevOps Succinctly",
    "ASP.NET Core 3.1 Succinctly",
    "AngularDart Succinctly"
  ];
}

class Succinctly extends StatelessWidget {
  final String book;
  final String title;

  Succinctly({
    @required this.book,
    @required this.title,
  });

  PopupMenuItem<String> bookItem(item) {
    return PopupMenuItem<String>(
      child: Text(item),
      value: item,
    );
  }

  List listBooks() {
    return StaticBooks.titles.map((String item) {
      return bookItem(item);
    }).toList();
```

```
  }

  List<Widget> popupMenuButton() {
    return <Widget>[
      PopupMenuButton(
        icon: Icon(Icons.book),
        itemBuilder: (BuildContext context) {
          return listBooks();
        },
      ),
    ];
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(title),
        actions: popupMenuButton(),
      ),
      body: Container(
        decoration: BoxDecoration(
          image: DecorationImage(
            image: NetworkImage(StaticBooks.cdn +
              StaticBooks.path + book),
            fit: BoxFit.scaleDown,
          ),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        child: Icon(Icons.book_online),
        onPressed: () {
          print('Awesome book!');
        },
      ),
    );
  }
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
```

```
      home: Succinctly(
        book: StaticBooks.covers[0],
        title: StaticBooks.titles[0],
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After making these changes and saving **main.dart**, we should see the app's UI updated on the emulator. On my machine, it looks as follows.

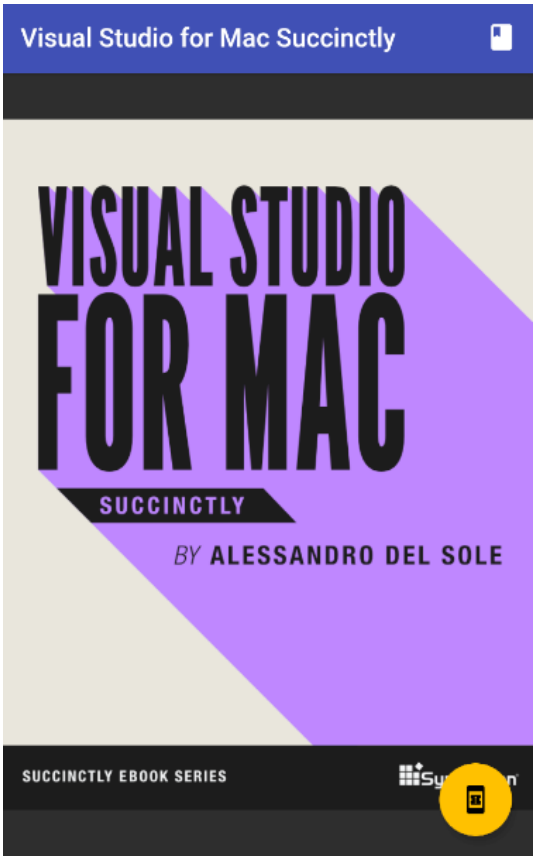

Figure 5-b: Succinctly Books App with a PopupMenuButton (Closed)

Notice that on the top-right corner of the app's toolbar, there is an icon that we can click on. Let's do that.

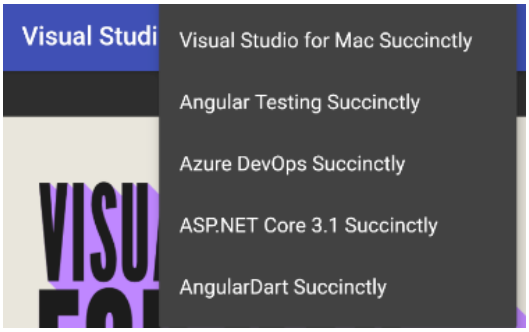

Figure 5-c: Succinctly Books App with a PopupMenuButton (Open)

When clicking the icon, we can see the list of books. The goal is that if you click on any of the book titles, you will be shown its respective cover image, which is something we have yet to implement.

But before we do that, let's have a look at the changes we made. The first thing to notice is that we added the **`actions`** property to the **`AppBar`** widget, which gets assigned the result returned by the **`popupMenuButton`** method (**`List<Widget>`**).

The **`popupMenuButton`** method creates an instance of **`PopupMenuButton`** class, to which its **`itemBuilder`** property is assigned a **`List`** returned by the **`listBooks`** method, which is essentially a list of **`PopupMenuItem<String>`** items.

The way the **`listBooks`** method works is that it creates a list of each of the book titles by invoking the **`map`** method from the **`StaticBooks.titles`** array, and for each title, the **`bookItem`** method is invoked.

All the **`bookItem`** method does is create a **`PopupMenuItem<String>`** instance, which represents a book title on the menu.

Now that we know how this works, we can focus on adding the remaining functionality: that when a menu option is clicked, the correct corresponding cover image is shown, and the book title is updated.

Push and pop

In Flutter, app navigation is based on the Stack class, which contains all the roots that an application has since it began executing. When the app needs to change the page being displayed, the Navigator class is used.

The **`Navigator`** class has two methods that interact with the **`Stack`**: the **`push`** and **`pop`** methods. The **`push`** method inserts a new page at the top of the **`Stack`**, whereas the **`pop`** method removes the page from the top of the **`Stack`** so that the previous page on the **`Stack`** becomes visible again.

When using the **push** method, we need to specify the page we want to load, and to achieve that, we need to use the MaterialPageRoute class. To the **MaterialPageRoute** class, we specify the name of the page we want to pass to the **push** method.

Both the **push** and **pop** methods require the current **context**. So, with the theory covered, let's make the changes necessary to the code. The changes are highlighted in bold in the following listing.

Code Listing 5-c: Updated main.dart – Using a PopupMenuButton with Navigation

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

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

class StaticBooks {
  static const String cdn = "https://cdn.syncfusion.com/";
  static const String path =
  "content/images/downloads/ebook/ebook-cover/";

  static const List<String> covers = [
    "visual-studio-for-mac-succinctly-v1.png",
    "angular-testing-succinctly.png",
    "azure-devops-succinctly.png",
    "asp-net-core-3-1-succinctly.png",
    "angulardart_succinctly.png"
  ];

  static const List<String> titles = [
    "Visual Studio for Mac Succinctly",
    "Angular Testing Succinctly",
    "Azure DevOps Succinctly",
    "ASP.NET Core 3.1 Succinctly",
    "AngularDart Succinctly"
  ];
}

class Succinctly extends StatelessWidget {
  final String book;
  final String title;

  Succinctly({
    @required this.book,
    @required this.title,
  });
```

```
PopupMenuItem<String> bookItem(item) {
  return PopupMenuItem<String>(
    child: Text(item),
    value: item,
  );
}

List listBooks() {
  return StaticBooks.titles.map((String item) {
    return bookItem(item);
  }).toList();
}

List<Widget> popupMenuButton(BuildContext context) {
  return <Widget>[
    PopupMenuButton(
      icon: Icon(Icons.book),
      itemBuilder: (context) {
        return listBooks();
      },
      onSelected: (value) => showBook(context, value),
    ),
  ];
}

void showBook(BuildContext context, String mItem) {
  String title;
  String cover;

  if (mItem == StaticBooks.titles[0]) {
    cover = StaticBooks.covers[0];
    title = StaticBooks.titles[0];
  } else if (mItem == StaticBooks.titles[1]) {
    cover = StaticBooks.covers[1];
    title = StaticBooks.titles[1];
  } else if (mItem == StaticBooks.titles[2]) {
    cover = StaticBooks.covers[2];
    title = StaticBooks.titles[2];
  } else if (mItem == StaticBooks.titles[3]) {
    cover = StaticBooks.covers[3];
    title = StaticBooks.titles[3];
  } else if (mItem == StaticBooks.titles[4]) {
    cover = StaticBooks.covers[4];
    title = StaticBooks.titles[4];
```

```
    }

    Navigator.push(
        context,
        MaterialPageRoute(
            builder: (context) => Succinctly(
                  book: cover,
                  title: title,
                )));
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(title),
        actions: popupMenuButton(context),
      ),
      body: Container(
        decoration: BoxDecoration(
          image: DecorationImage(
            image: NetworkImage(StaticBooks.cdn +
              StaticBooks.path + book),
            fit: BoxFit.scaleDown,
          ),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        child: Icon(Icons.book_online),
        onPressed: () {
          print('Awesome book!');
        },
      ),
    );
  }
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Succinctly(
        book: StaticBooks.covers[0],
```

```
        title: StaticBooks.titles[0],
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After making these changes and saving **main.dart**, we should see the app's UI updated on the emulator. On my machine, it looks as follows.

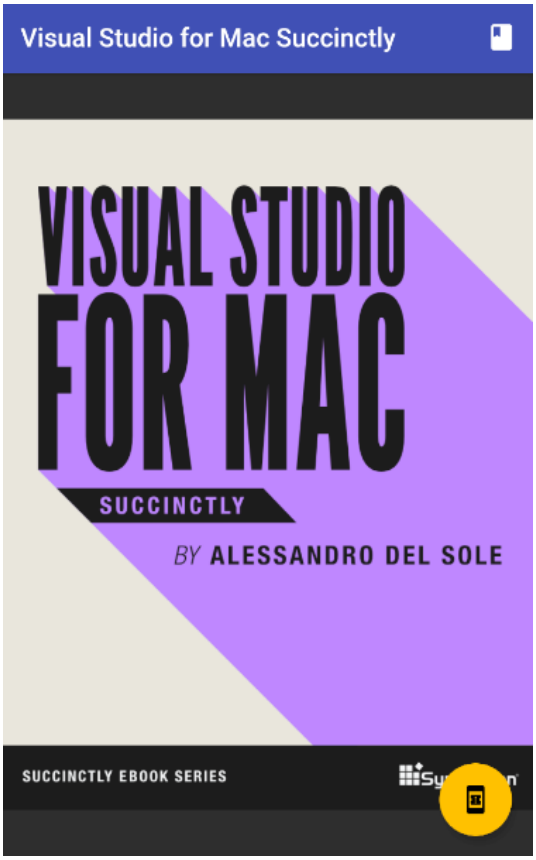

Figure 5-d: Succinctly Books App with a PopupMenuButton with Navigation (Closed)

If we click any of the menu options, we should be able to navigate to the respective book. I'll test it out by clicking the second menu option, **Angular Testing Succinctly**.

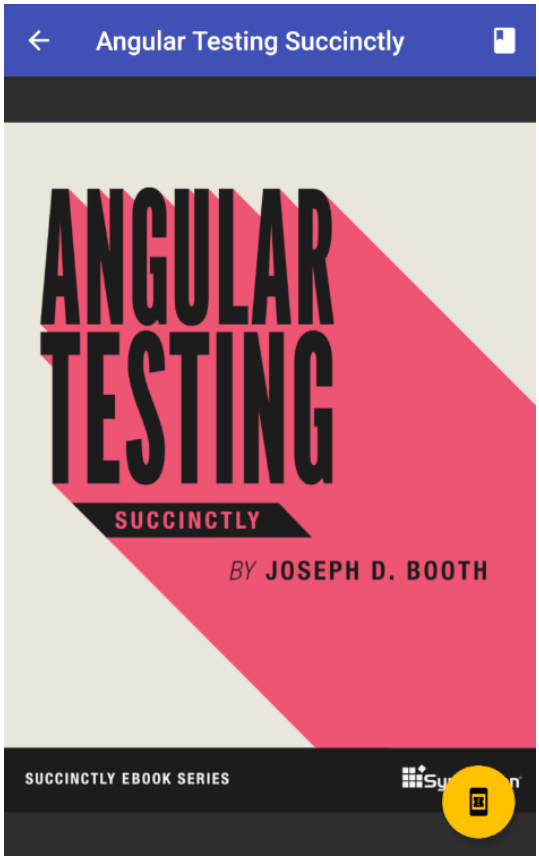

Figure 5-e: Succinctly Books App with Navigation Working

We can see that on the app's title, there's an icon to go back to the previous page. Awesome—we've managed to implement navigation successfully.

To make this happen, the first thing we had to do was pass the **`context`** parameter to the **`popupMenuButton`** method, which is something we hadn't done before. The reason for doing this is that the **`context`** is required when invoking the **`Navigator`** class; therefore, the **`context`** is passed to the **`showBook`** method through the **`onSelected`** event.

The **`onSelected`** event was added to the **`PopupMenuButton`** widget. When the **`onSelected`** event gets triggered, the **`showBook`** method is executed, which is responsible for displaying the book selected through the menu.

The logic behind the **`showBook`** method is very simple. The parameter **`mItem`** indicates the name of the book selected using the menu. The value of the **`mItem`** parameter is compared against the values defined within the **`StaticBooks.titles`** list, and depending on the match, the **`cover`** and **`title`** values are assigned.

Finally, the **`Navigator.push`** method is invoked, to which the **`context`**, **`cover`**, and **`title`** are passed as parameters, resulting in a new page being added to the **`Stack`**, with the corresponding information from the book selected through the menu.

Bottom navigation bar

The BottomNavigationBar is a Flutter widget that is shown at the bottom of the app's screen, regularly used with a Scaffold, and it should be used for a small number of items, usually between three and five.

The **BottomNavigationBar** widget provides a quick and easy solution to include navigation in your apps. Let's get straight into the details and see how we can implement our book app navigation with a **BottomNavigationBar** widget. The changes are highlighted in bold in the following code.

Code Listing 5-d: Updated main.dart – Using a BottomNavigationBar

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

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

class StaticBooks {
  static const String cdn = "https://cdn.syncfusion.com/";
  static const String path =
  "content/images/downloads/ebook/ebook-cover/";

  static const List<String> covers = [
    "visual-studio-for-mac-succinctly-v1.png",
    "angular-testing-succinctly.png",
    "azure-devops-succinctly.png",
    "asp-net-core-3-1-succinctly.png",
    "angulardart_succinctly.png"
  ];

  static const List<String> titles = [
    "Visual Studio for Mac Succinctly",
    "Angular Testing Succinctly",
    "Azure DevOps Succinctly",
    "ASP.NET Core 3.1 Succinctly",
    "AngularDart Succinctly"
  ];
}

class Succinctly extends StatelessWidget {
  final String book;
  final String title;

  Succinctly({
    @required this.book,
```

```
  @required this.title,
});

void showBook(BuildContext context, String mItem) {
  String title;
  String cover;

  if (mItem == StaticBooks.titles[0]) {
    cover = StaticBooks.covers[0];
    title = StaticBooks.titles[0];
  } else if (mItem == StaticBooks.titles[1]) {
    cover = StaticBooks.covers[1];
    title = StaticBooks.titles[1];
  } else if (mItem == StaticBooks.titles[2]) {
    cover = StaticBooks.covers[2];
    title = StaticBooks.titles[2];
  } else if (mItem == StaticBooks.titles[3]) {
    cover = StaticBooks.covers[3];
    title = StaticBooks.titles[3];
  } else if (mItem == StaticBooks.titles[4]) {
    cover = StaticBooks.covers[4];
    title = StaticBooks.titles[4];
  }
  print(cover);

  Navigator.push(
      context,
      MaterialPageRoute(
          builder: (context) => Succinctly(
                book: cover,
                title: title,
              )));
}

static int _index = 0;

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text(title),
    ),
    bottomNavigationBar: BottomNavigationBar(
      selectedItemColor: Colors.orange,
```

```
      currentIndex: _index,
      onTap: (value) {
        String _title = StaticBooks.titles[value];
        _index = value;
        showBook(context, _title);
      },
      items: [
        BottomNavigationBarItem(
          label: 'Visual Studio Mac',
          icon: Icon(Icons.book),
        ),
        BottomNavigationBarItem(
          label: 'Ang. Testing',
          icon: Icon(Icons.book_online),
        ),
        BottomNavigationBarItem(
          label: 'Azure DevOps',
          icon: Icon(Icons.book_online_outlined),
        ),
        BottomNavigationBarItem(
          label: 'ASP.NET Core',
          icon: Icon(Icons.book_online_rounded),
        ),
        BottomNavigationBarItem(
          label: 'AngularDart',
          icon: Icon(Icons.book_online_sharp),
        ),
      ],
    ),
    body: Container(
      decoration: BoxDecoration(
        image: DecorationImage(
          image: NetworkImage(StaticBooks.cdn +
            StaticBooks.path + book),
          fit: BoxFit.scaleDown,
        ),
      ),
    ),
    floatingActionButton: FloatingActionButton(
      child: Icon(Icons.book_online),
      onPressed: () {
        print('Awesome book!');
      },
    ),
```

```
    );
  }
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Succinctly(
        book: StaticBooks.covers[0],
        title: StaticBooks.titles[0],
      ),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After we have made these changes and saved **main.dart**, the app's UI should be updated on the emulator.

I'll click on the fifth icon of the **`BottomNavigationBar`** from left to right, which corresponds to the *AngularDart Succinctly* book. On my machine, it looks as follows.

Figure 5-f: Succinctly Books App – BottomNavigationBar

So, one of the changes made (which is probably not noticeable) was that the **`action`** property from the **`AppBar`** widget was removed, along with all the logic responsible for creating the menu.

The **`bottomNavigationBar`** property was added to the **`build`** method, which gets assigned a **`BottomNavigationBar`** widget, and a **`static`** variable called **`_index`** was added to the **`Succinctly`** class. The **`static _index`** variable is used to keep track of which book is selected using the **`BottomNavigationBar`** widget.

With regards to the **`BottomNavigationBar`** widget, we are using several properties. The **`selectedItemColor`** property indicates the color that is shown when an item is selected—in this case, **`Colors.orange`**, as can be seen in Figure 5-f.

Then we have the **`currentIndex`** property, which gets assigned the value of the **`static _index`** variable. This property is used internally to keep track of which book is selected.

We have the **`onTap`** event, which gets triggered when a book is selected, by tapping on any of the items of the **`BottomNavigationBar`** widget.

When the **`onTap`** event is executed, the title of the book selected is retrieved by the **`StaticBooks.titles[value]`** instruction, and this value is assigned to the **`_title`** variable.

The value of the **`_title`** variable is passed to the **`showBook`** method, which then displays the cover of the selected book.

The items that are shown within the **BottomNavigationBar** widget are specified within the **items** property, which is an array of **BottomNavigationBarItem** widgets.

Each **BottomNavigationBarItem** is a button within the **BottomNavigationBar** widget, and two properties for each are used: an **icon** and a **label**.

As you might have noticed, the **BottomNavigationBar** widget is best suited when you have a fixed number of tabs (items) that you want to display, which we do in this example. When you have a dynamic number of items to display, using the **BottomNavigationBar** widget might not be the best option.

Tab bar

Another way to add navigation to a Flutter application is to use the TabBar class, which is made up of three main parts.

The first part is the TabController, the second is the **TabBar** itself (which contains the tabs), and the third part is the TabBarView with its children.

A **TabBar** displays a horizontal row of tabs, and when you click on one of the tabs, the **TabBarView** class displays different widgets with an animation.

Let's go ahead and implement the app's navigation using a **TabBar** widget. The changes to the code are highlighted in bold in the following listing.

Code Listing 5-e: Updated main.dart – Using a TabBar

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

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

class StaticBooks {
  static const String cdn = "https://cdn.syncfusion.com/";
  static const String path =
  "content/images/downloads/ebook/ebook-cover/";

  static const List<String> covers = [
    "visual-studio-for-mac-succinctly-v1.png",
    "angular-testing-succinctly.png",
    "azure-devops-succinctly.png",
    "asp-net-core-3-1-succinctly.png",
    "angulardart_succinctly.png"
  ];

  static const List<String> titles = [
    "Visual Studio for Mac Succinctly",
```

```
    "Angular Testing Succinctly",
    "Azure DevOps Succinctly",
    "ASP.NET Core 3.1 Succinctly",
    "AngularDart Succinctly"
  ];
}

class Succinctly extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return DefaultTabController(
      length: 5,
      child: Scaffold(
        appBar: AppBar(
          title: Text('Succinctly Books'),
          bottom: TabBar(
            tabs: <Widget>[
              Tab(icon: Icon(Icons.book), text: 'VSM'),
              Tab(icon: Icon(Icons.book_online), text: 'AT'),
              Tab(icon: Icon(Icons.book_online_outlined), text: 'AZ'),
              Tab(icon: Icon(Icons.book_online_rounded), text: 'ASP'),
              Tab(icon: Icon(Icons.book_online_sharp), text: 'AD'),
            ],
          ),
        ),
        body: TabBarView(
          children: <Widget>[
            Container(
              decoration: BoxDecoration(
                image: DecorationImage(
                  image: NetworkImage(StaticBooks.cdn +
                      StaticBooks.path +
                      StaticBooks.covers[0]),
                  fit: BoxFit.scaleDown,
                ),
              ),
            ),
            Container(
              decoration: BoxDecoration(
                image: DecorationImage(
                  image: NetworkImage(StaticBooks.cdn +
                      StaticBooks.path +
                      StaticBooks.covers[1]),
                  fit: BoxFit.scaleDown,
```

```
              ),
            ),
          ),
          Container(
            decoration: BoxDecoration(
              image: DecorationImage(
                image: NetworkImage(StaticBooks.cdn +
                    StaticBooks.path +
                    StaticBooks.covers[2]),
                fit: BoxFit.scaleDown,
              ),
            ),
          ),
          Container(
            decoration: BoxDecoration(
              image: DecorationImage(
                image: NetworkImage(StaticBooks.cdn +
                    StaticBooks.path +
                    StaticBooks.covers[3]),
                fit: BoxFit.scaleDown,
              ),
            ),
          ),
          Container(
            decoration: BoxDecoration(
              image: DecorationImage(
                image: NetworkImage(StaticBooks.cdn +
                    StaticBooks.path +
                    StaticBooks.covers[4]),
                fit: BoxFit.scaleDown,
              ),
            ),
          ),
        ],
      ),
    ),
  );
  }
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
```

```
      debugShowCheckedModeBanner: false,
      home: Succinctly(),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
          fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After we've made these changes and saved **main.dart**, the app's UI should be updated on the emulator.

I'll click the fifth icon of the **TabBar** from left to right, which corresponds to the *Azure DevOps Succinctly* book. On my machine, it looks as follows.

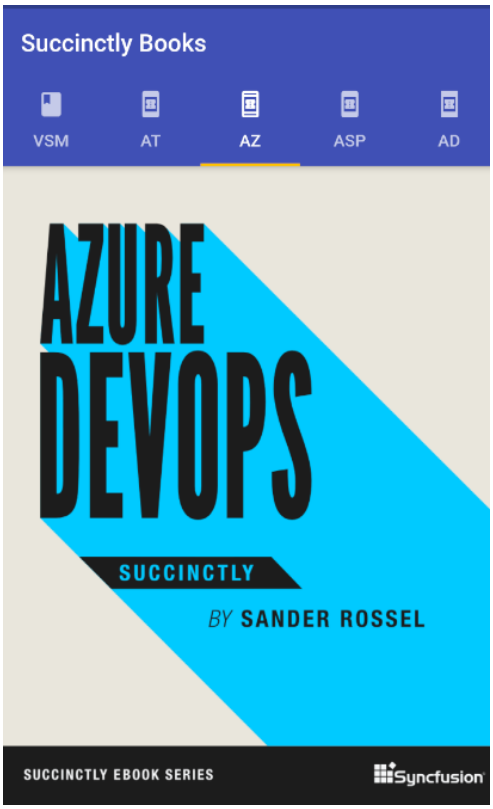

Figure 5-g: Succinctly Books App – TabBar

The first thing to notice is the **Succinctly** class was simplified; only the **build** method remains. Everything else was removed, and the class no longer takes any parameters.

The **build** method of the **Succinctly** class was completely changed. The most noticeable change is that this method now returns a **DefaultTabController** instead of a **Scaffold** widget as before.

The **DefaultTabController** widget contains **length** and **child** properties. The **length** indicates the number of tabs that the widget will contain, and the **child** property is assigned to a **Scaffold** widget.

The **Scaffold** widget has an **AppBar** that includes a **TabBar** widget, which contains an array of **Tab** widgets. Each **Tab** widget contains an **icon** and a **text** property, each corresponding to a tab item from the **TabBar** widget.

The **body** property of the **Scaffold** widget is assigned to a **TabBarView** widget, which contains a **children** property that includes an array of **Container** widgets.

Each **Container** widget corresponds to a *Succinctly* book. Within every **Container** widget, there is a **BoxDecoration** widget, which contains a **DecorationImage** widget.

Each **DecorationImage** contains a **NetworkImage** widget that retrieves the corresponding book image from the web.

As you might have noticed, the **TabBar** widget is also best suited when you have a fixed number of tabs (items) that you want to display, and mostly when the content is static (it doesn't vary), which is the case for this example.

So, when you have a dynamic number of items to display, using the **TabBar** widget might not be the best option.

Summary

Throughout this chapter, we've explored how to add various types of navigation mechanisms to a Flutter application.

Although we touched base on most of the common Flutter app navigation methods, I'd like to leave you with a challenge: how could you implement navigation using the Flutter Drawer class?

If you do have the time to try this out, I'd love to hear from you about your results. In the next and final chapter, we will cover the Stack, ListView, and GridView widgets to wrap things up.

Chapter 6 Stack, ListView, and GridView

Overview

We've covered quite a bit of ground so far, and have now reached the final chapter of this book, which is going to be short but to the point, given what we know so far about creating UIs with Flutter.

User interfaces are one of those topics where there are so many things to talk about that we would need a full library of books to cover it, and even then, we might still miss out on some topics since this is an ever-evolving field.

Take, for example, animations. There are so many different ways to create animations in Flutter that delving into this topic would require a least one book covering the fundamentals. This is something I'll keep in mind for the future.

Nevertheless, most Flutter applications rely on the Stack, ListView, and GridView widgets, and that's exactly what we're going to explore now. Let's get started.

Stack

Before can dive into the details of how the **`ListView`** and **`GridView`** widgets work, we need to understand what the **`Stack`** class is.

According to the official Flutter documentation, a **`Stack`** is a widget that can position its children relative to the edges of its box. A key aspect to know about a **`Stack`** widget is that its size is determined by the size of its largest child component.

What we want to do next is create a screen that uses a **`Stack`**, as this will help us to fully understand how **`ListView`** and **`GridView`** widgets work.

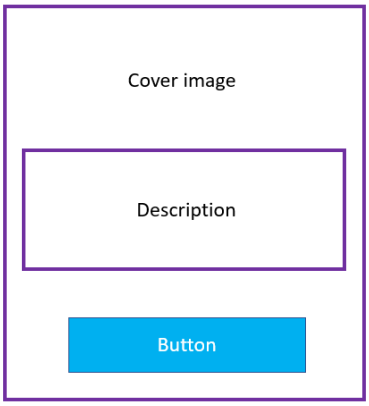

Figure 6-a: The Stack Screen to Build

What we want to build is a screen that has the book's cover image in the background, then an area on top of the image with a description—a Card, and at the bottom of the screen, a RaisedButton. The following code does that, and the changes are highlighted in bold.

Codee Listing 6-a: Updated main.dart – Using a Stack

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

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

class StaticBooks {
  static const String cdn = "https://cdn.syncfusion.com/";
  static const String path =
  "content/images/downloads/ebook/ebook-cover/";

  static const List<String> covers = [
    "visual-studio-for-mac-succinctly-v1.png",
  ];
}

class Succinctly extends StatelessWidget {
  List<Widget> stackScreen(double sizeX, double sizeY) {
    List<Widget> layout = List<Widget>();

    Container cover = Container(
      decoration: BoxDecoration(
        image: DecorationImage(
          image: NetworkImage(
              StaticBooks.cdn +
              StaticBooks.path + StaticBooks.covers[0]),
          fit: BoxFit.scaleDown,
        ),
      ),
    );

    layout.add(cover);

    final card = Positioned(
      top: sizeY / 1.45,
      left: sizeX / 4.2,
      child: Card(
        elevation: 15,
        color: Colors.blue,
        shape: RoundedRectangleBorder(
          borderRadius: BorderRadius.circular(5),
```

```
      ),
      child: Column(
        children: [
          Padding(
            padding: EdgeInsets.all(10),
            child: Text('Succinctly Series'),
          ),
        ],
      ),
    ),
  );

  layout.add(card);

  Positioned button = Positioned(
    width: sizeX - sizeY / 10,
    bottom: sizeY / 40,
    left: sizeX / 12,
    child: RaisedButton( // Or use ElevatedButton
      color: Colors.lightBlue,
      elevation: 8,
      shape: RoundedRectangleBorder(
        borderRadius: BorderRadius.circular(10)
      ),
      child: Text('Browse collection'),
      onPressed: () {
        // Do something later...
      },
    ),
  );

  layout.add(button);

  return layout;
}

@override
Widget build(BuildContext context) {
  final sizeX = MediaQuery.of(context).size.width;
  final sizeY = MediaQuery.of(context).size.height;
  return Container(
    child: Stack(
      children: stackScreen(sizeX, sizeY),
    )
```

```
    );
  }
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Succinctly(),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        textTheme: TextTheme(
          bodyText2: TextStyle(
            fontSize: 26, fontStyle: FontStyle.italic),
        ),
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After we've made these changes and saved **main.dart**, the app's UI should be updated on the emulator. On my machine, this looks as follows.

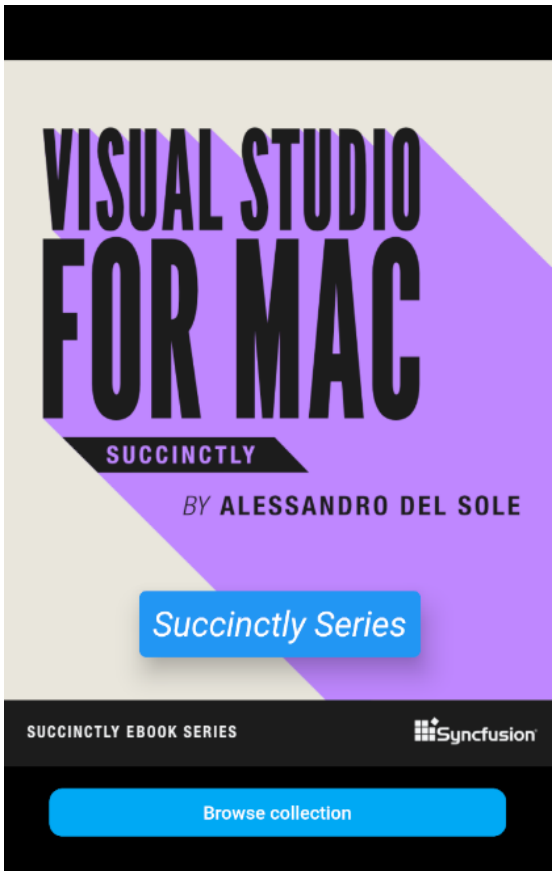

Figure 6-b: Succinctly Books App – Stack

To better understand the code changes, let's have a look at the following diagram, which serves as a visual point of reference between the part of the code contained within the **`stackScreen`** method and the UI.

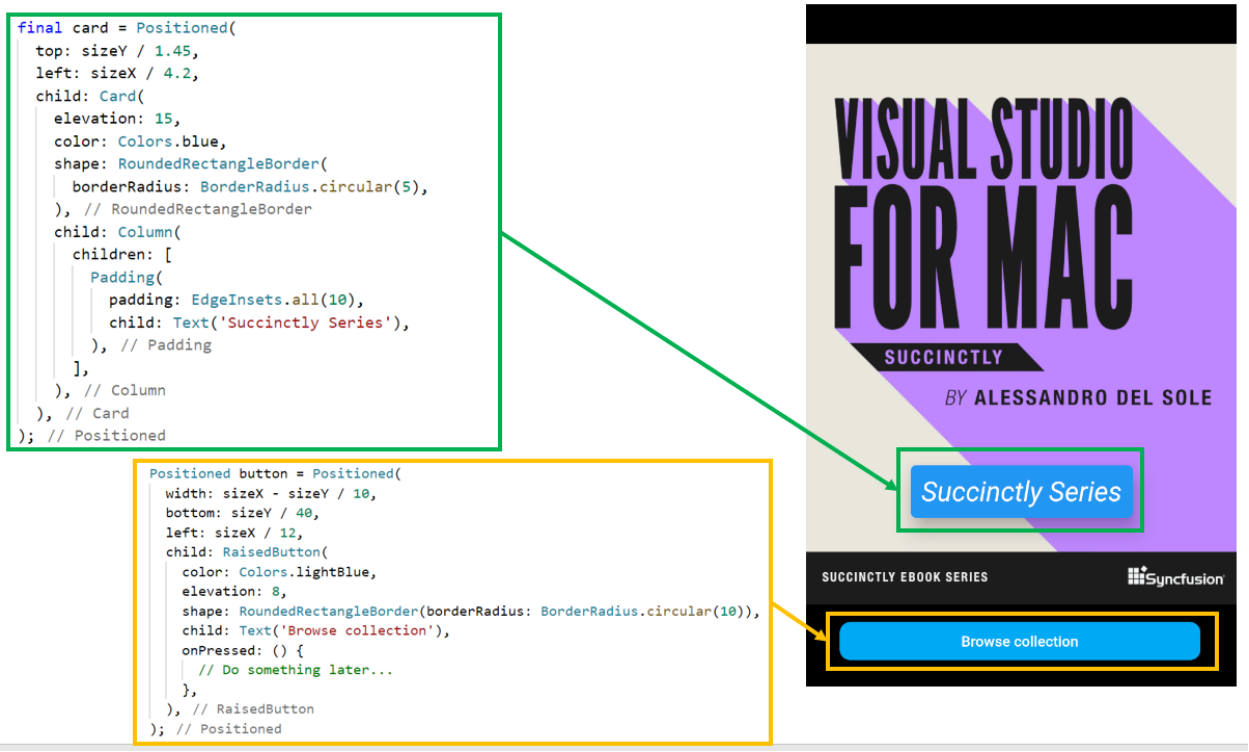

Figure 6-c: Code to UI Relationship (Succinctly Books App – Stack)

As you can see in Figure 6-c, the two main parts of the **`stackScreen`** method have to do with the **`Card`** widget (highlighted in green), and the **`RaisedButton`** (highlighted in yellow). So, with this visual reference in mind, let's explore the code changes, which are quite a few.

The first thing you might have noticed is that the **`titles`** list was removed from the **`StaticBooks`** class, and the **`covers`** list was reduced to one item—this is because we are only showing one book cover.

Next, within the **`Succinctly`** class, we created a **`stackScreen`** method that is responsible for creating a **`layout`** that renders an image (**`cover`**), a **`Card`**, and a **`RaisedButton`**.

This **`layout`** is returned by the **`stackScreen`** method, and it is assigned to the **`children`** property of a **`Stack`** widget, which is assigned to the **`child`** property of a **`Container`** widget.

In essence, we have built a **`layout`** wrapped around a **`Stack`** widget, wrapped in a **`Container`**.

That **`Container`** widget, returned by the **`stackScreen`** method, is then returned by the **`build`** method and assigned to the **`home`** property of the **`MaterialApp`** widget.

Before exploring the **`stackScreen`** method in detail, let's have a look at the other changes made to the **`build`** method.

We can see that within the **`build`** method, we get the width (**`sizeX`**) and height (**`sizeY`**) of the screen by retrieving these values from the **`context`** object using the MediaQuery class.

Both the **`sizeX`** and **`sizeY`** values are then passed to the **`stackScreen`** method and used to position the **`Card`** and the **`RaisedButton`**.

Within the **stackScreen** method, the first thing we did was declare the **layout** variable as a list of widgets (**List<Widget>**). This is because the **Stack** widget will contain the image (**cover**), the **Card**, and the **RaisedButton** widgets.

Next, we created a **Container** instance, which is assigned to the **cover** variable, using the same code we used before when creating the book cover.

The first object that gets added to the **layout** used by the **Stack** widget is the **cover** (image), which happens when the following instruction executes: **layout.add(cover)**.

After that, the **Card** widget is created, and wrapped around a Positioned widget. This is because we want to be able to place the card on a specific location of the screen.

To do that, the **sizeX** and **sizeY** parameters are used to calculate the values of the **top** and **left** properties of the **Positioned** widget.

The **Card** widget is assigned to the **child** property of the **Positioned** widget, and the **elevation**, **color**, and **shape** properties of the **Card** are defined.

The **child** property of the **Card** widget contains a **Column**, and within its **children** property a **Padding** widget that contains a **Text** widget, with some **padding** defined. This is a great example of how a Flutter widget can be composed of many other smaller widgets.

Finally, the **card** object is added to the **layout**, and this is achieved when the following instruction executes: **layout.add(card)**.

The last piece of this puzzle is the button, which is visible at the bottom of the **layout**. Just like with the **Card** widget, the **RaisedButton** widget is wrapped around a **Positioned** widget, so it can be easily placed in a specific part of the screen.

The **width**, **bottom**, and **left** properties of the **Positioned** widget are calculated using the **sizeX** and **sizeY** parameters.

The **RaisedButton** widget is assigned to the **child** property of the **Positioned** widget. For this button, the **color**, **elevation**, and **shape** properties are defined.

The text shown in the button is a **Text** widget assigned to the **child** property of the **RaisedButton** widget.

There's also an **onPressed** event that can be used in the future, which is triggered when the button is pressed, as the name of the event implies.

Finally, the button is added to the **layout** when the following instruction executes: **layout.add(button)**.

There we go—that's how this example using a **Stack** widget was composed.

ListView

One of the most common features you'll find in mobile apps is lists of items, such as products, documents, contacts, or email addresses.

Let's create a list of documents that have an expiration date, such as passports and driver licenses, and this is where we will use the ListView widget. Let's dive right into the code—the changes are highlighted in bold in the following listing.

Code Listing 6-b: Updated main.dart – Using a ListView

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

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

class Doc {
  String name;
  String description;
  DateTime expires;

  Doc(this.name, this.description, this.expires);
}

class Succinctly extends StatelessWidget {
  List<Doc> createDocs() {
    List<Doc> docs = List<Doc>();

    docs.add(Doc('Driver License', 'Florida driver license',
        DateTime.now().add(new Duration(days: 1825))));

    docs.add(Doc('Passport Ed', 'Ed\'s passport',
        DateTime.now().add(new Duration(days: 825))));

    docs.add(Doc('Passport John', 'John\'s passport',
        DateTime.now().add(new Duration(days: 2801))));

    docs.add(Doc('ID card', 'John\'s national ID card',
        DateTime.now().add(new Duration(days: 801))));

    return docs;
  }

  List<ListTile> showList() {
    List<ListTile> items = List<ListTile>();
    List<Doc> docs = createDocs();
```

```
    docs.forEach((doc) {
      items.add(ListTile(
        title: Text(doc.name),
        subtitle: Text(doc.description),
        leading: CircleAvatar(
          child: Icon(Icons.book),
          backgroundColor: Colors.lightBlueAccent,
        ),
        trailing: Icon(Icons.keyboard_arrow_down),
        onTap: () => true,
      ));
    });

    return items;
  }

  @override
  Widget build(BuildContext context) {
    final sizeX = MediaQuery.of(context).size.width;
    final sizeY = MediaQuery.of(context).size.height;
    return Scaffold(
      appBar: AppBar(
        title: Text('Documents'),
      ),
      floatingActionButton: FloatingActionButton(
        child: Icon(Icons.book_online),
        onPressed: () {
          print('New doc!');
        },
      ),
      body: Container(
        width: sizeX,
        height: sizeY,
        child: ListView(
          children: showList(),
        ),
      )
    );
  }
}

class MyApp extends StatelessWidget {
  @override
```

```
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Succinctly(),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        brightness: Brightness.dark,
      ),
    );
  }
}
```

After we've made these changes and saved **main.dart**, the app's UI should be updated on the emulator. On my machine, this looks as follows.

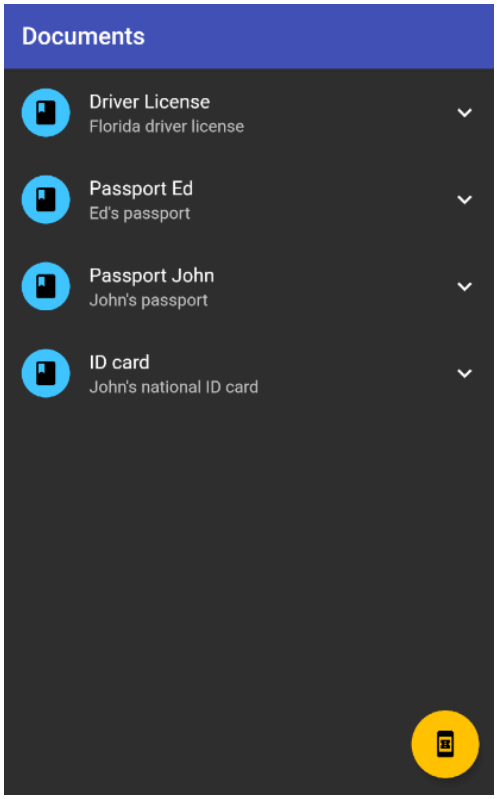

Figure 6-d: Docs App – ListView

Let's review the code changes. The first thing to notice is that the **`Succinctly`** class has been revamped. The **`build`** method returns a **`Scaffold`** widget that includes the usual **`AppBar`** and **`FloatingActionButton`** widgets that we've used before.

The **Scaffold** widget contains a **Container** that includes a **ListView** that is assigned to its **child** property. The **children** property of the **ListView** widget contains an array of **ListTile** widgets, which is what the **showList** method returns.

The **showList** method starts by invoking the **createDocs** method, which returns a list of **Doc** objects (**List<Doc>**).

For each **Doc** object returned by the **createDocs** method, a **ListTile** item is created and added to the **items** list (**List<ListTile>**).

Each **ListTile** item contains a **title**, **subtitle**, **leading**, and **trailing** property, as well as an **onTap** event. To understand this better, let's look at the following diagram.

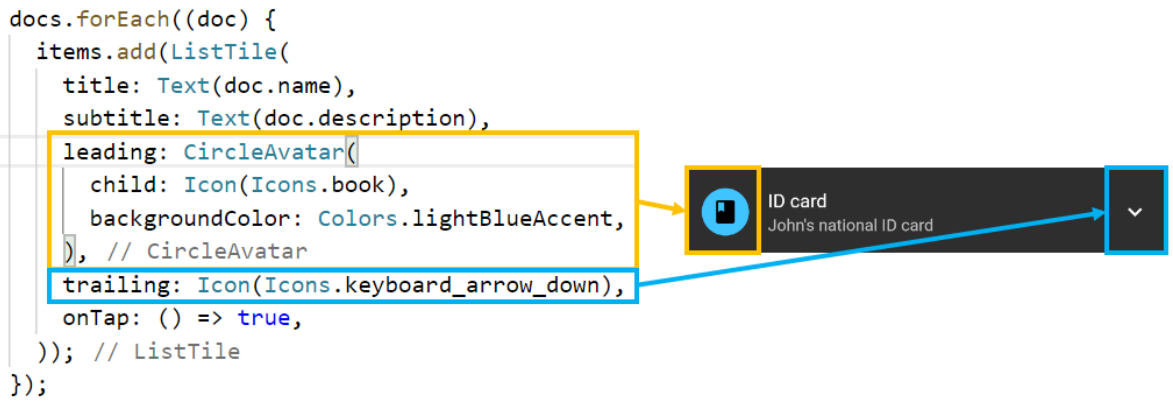

Figure 6-e: ListTile Item – Docs App

From the preceding diagram, we can see that the **leading** property is assigned to a **CircleAvatar** widget, which contains an **Icon** widget and has its **backgroundColor** property set to **Colors.lightBlueAccent**. This is highlighted in yellow.

We can also see that the **trailing** property is assigned to an **Icon** widget, which is the down arrow we can see highlighted in blue.

The **createDocs** method starts by initializing the **docs** variable (**List<Doc>**), and a **Doc** object is created for every item seen in the **ListView** widget. Four items are added to the **docs** list and returned by the method.

The **Doc** class contains a **name**, **description**, and **expires** property. These are the details displayed by each **ListTile**.

That's how easy it is to create a **ListView** widget in Flutter.

GridView

Now that we know how to create a **ListView**, let's cover the final topic of this book, which is how to create a Flutter **GridView**.

A **GridView**, which looks like a set of tiles, is a great way to display images, and can be used as an image gallery. Image galleries are widely used in e-commerce, photo, social media, real estate, and car rental apps, just to name a few.

A great use of a **GridView** widget would be to display some of the *Succinctly* books we saw before. So, let's implement that—the changes are highlighted in bold in the following code.

Code Listing 6-c: Updated main.dart – Using a GridView

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

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

class StaticBooks {
  static const String cdn = "https://cdn.syncfusion.com/";
  static const String path =
  "content/images/downloads/ebook/ebook-cover/";

  static const List<String> covers = [
    "visual-studio-for-mac-succinctly-v1.png",
    "angular-testing-succinctly.png",
    "azure-devops-succinctly.png",
    "asp-net-core-3-1-succinctly.png",
    "angulardart_succinctly.png"
  ];
}

class Succinctly extends StatelessWidget {
  List<Widget> createGrid() {
    List<Widget> imgs = List<Widget>();

    Widget cImage;

    for (int i = 0; i <= StaticBooks.covers.length - 1; i++) {
      cImage = Container(
          child: Image.network(
              StaticBooks.cdn +
              StaticBooks.path + StaticBooks.covers[i])
      );
      imgs.add(cImage);
    }

    return imgs;
  }

  @override
```

```
  Widget build(BuildContext context) {
    final sizeX = MediaQuery.of(context).size.width;
    final sizeY = MediaQuery.of(context).size.height;
    return Scaffold(
        appBar: AppBar(
          title: Text('Succinctly Books'),
        ),
        body: Container(
            width: sizeX,
            height: sizeY,
            child: GridView.count(
              children: createGrid(),
              padding: EdgeInsets.all(10),
              crossAxisSpacing: 4.5,
              mainAxisSpacing: 5.5,
              crossAxisCount: 2,
              scrollDirection: Axis.vertical,
            )));
  }
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Succinctly(),
      theme: ThemeData(
        primaryColor: Colors.indigo,
        accentColor: Colors.amber,
        brightness: Brightness.dark,
      ),
    );
  }
}
```

Before having a look at the updated UI, let's dive into the code changes. The first thing to notice is that we've brought back the **StaticBooks** class, but only with the **covers** array, and without the **titles**.

Within the **Succinctly** class, we have a **createGrid** method that loops through the **covers** array, and for each, a **Container** widget is created.

Within the **Container** widget, the **Image.network** method is used to fetch each of the cover images of the books, which is assigned to the **child** property.

Each image is added to a list of images (**imgs**), which is returned by the **createGrid** method.

The **build** method returns a **Scaffold** widget that contains the usual **AppBar** widget and the **body**, which is what renders the grid.

Before we continue exploring the code changes, let's look at the app's UI, which should be updated on the emulator after we save the **main.dart** file. On my machine, this looks as follows.

Figure 6-f: Books App – GridView

So there's the app using a **GridView** widget—it looks great! The **body** property of the **Scaffold** widget is assigned to a **Container**, which wraps a **GridView** widget.

The **GridView** widget contains several properties, all of which are important to display the grid correctly, such as the crossAxisSpacing, mainAxisSpacing, crossAxisCount, and scrollDirection properties.

The **crossAxisSpacing** and **mainAxisSpacing** properties define the spacing between the images on the **GridView** widget, whereas the **crossAxisCount** property indicates the number of images per row. The **scrollDirection** property indicates that the images are placed vertically.

Finally, the **children** property is assigned the result returned by the **createGrid** method, which retrieves the book cover images.

Final thoughts

I find that creating user interfaces is a rewarding experience, even though most of the work I do is on the back end. I know that everyone might not agree with me, and that's fine. For some developers, UIs are not their thing.

For me, there's something magical about creating a UI and seeing it come to life, catching the eye—a pixel here, a pixel there—and seeing them turn into a shape, widget, or form.

Flutter is a fabulous framework for building rich UIs; it's designed from the ground up with that purpose in mind.

Throughout this book, my goal was to present Flutter's UI capabilities by describing its core features and key widgets so anyone reading could gain enough knowledge to build a UI without getting too deep into the realm of what designers do.

For those who are designers, Flutter is an excellent choice to bring a design to life with relatively few lines of code.

GitHub's Dart repositories are packed with amazing open-source Flutter projects, many focusing on UI design. I invite you to explore them further.

Other than that, another interesting aspect of UIs which we didn't cover in this book is animations. A full book on that topic probably wouldn't cover everything that can be done with them.

So, going forward, if the UI is something that ignites a spark within you, rest assured that we've just scratched the surface of what is possible with Flutter.

I'd personally love to write a book on creating advanced UIs and animations, and perhaps it is something I will bring up in due time with Syncfusion—which, in my opinion, would also add value to their Flutter widgets offering and the customers using them.

I hope this book has given you some solid grounding on how to start creating UIs with Flutter, and that it has inspired you to continue your journey to keep learning about this wonderful framework.

Once again, thank you for taking the time to read a *Succinctly* book. I hope you continue to be inspired by what you can achieve with Flutter. Until next time, take care, and all the best.