4

Introduction to Widgets

It's time to finally start playing with Flutter!

In Flutter, everything you see is a **widget.** Everything your users see and interact with, such as text, a button, an image, a table, or a scrolling list, is a widget. Even the app itself is, in fact, a widget.

Most widgets in Flutter are supposed to perform a single small task. On their own, widgets are classes that perform tasks on the user interface. A `Text` widget displays text. A `Padding` widget adds space between widgets. A `Scaffold` widget provides a structure for a screen.

The real power of widgets comes not from any individual class, but from how you can compose them together to create expressive interfaces. All the widgets on a screen, when combined together, form a **widget tree**.

In Flutter, you do not have any markup language to design the UI, like XML or HTML: you create widgets in Dart.

This chapter will cover the following recipes:

- Creating immutable widgets
- Using a Scaffold
- Using the Container widget
- Printing stylish text on the screen
- Importing fonts and images into your app

Technical requirements

All the code for this project can be downloaded from `https://github.com/PacktPublishing/Flutter-Cookbook/tree/master/chapter_04`.

Creating immutable widgets

There are two kinds of visual widget in Flutter: **stateless** and **stateful**.

A **stateless widget** is simple, lightweight, and performant. Flutter can render hundreds of stateless widgets without breaking a sweat.

Stateless widgets are **immutable**. Once they are created, they cannot be modified. Flutter only has to concern itself with these widgets once. It doesn't have to maintain any complex life cycle states or worry about a block of code modifying them. In fact, the only way to modify a stateless widget is by deleting it and creating a new one.

How to do it...

Start off by creating a brand-new Flutter project called `flutter_layout`, either via your IDE or the command line. Don't worry about the sample code generated. We're going to delete it and start from scratch:

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

```
void main() => runApp(const StaticApp());

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
```

```
        home: const ImmutableWidget(),
      );
    }
  }
```

Note that you have a bunch of red underlines. We need to fix this by importing the `material.dart` library. This can be done manually, but it's more fun to let your IDE do that job. Move your cursor over the word `StatelessWidget`.

3. In VS Code, press *Ctrl* + . or *Command* + . on a Mac. In Android Studio/IntelliJ, press *Alt* + *Enter* or *Option* + *Enter* on a Mac. This will bring up a dialog where you can choose which file to import.
4. Alternatively, you can also click with your mouse on the light bulb that appears on the left of the screen. Choose the file to import from the dialog. Select `material.dart` and most of the errors will go away:

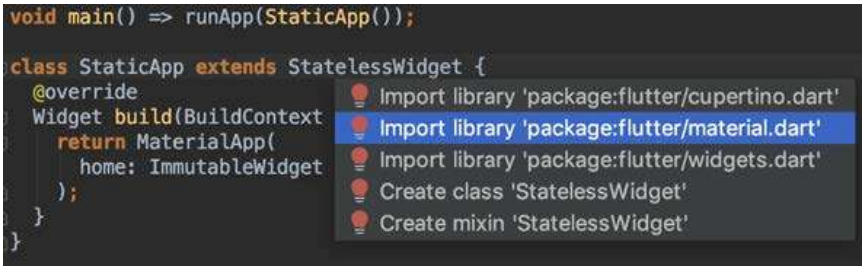

Figure 4.1: The Import helper in Android Studio

5. We'll get to the remaining error in just a second. The `ImmutableWidget` class can't be imported, as it doesn't exist yet.
6. Create a new file, `immutable_widget.dart`, in the project's `lib` folder. You should see a blank file.
7. There is some boilerplate with stateless widgets, but you can create them with a simple code snippet. Just type `stless`. Hit *Enter* on your keyboard and the template will appear:

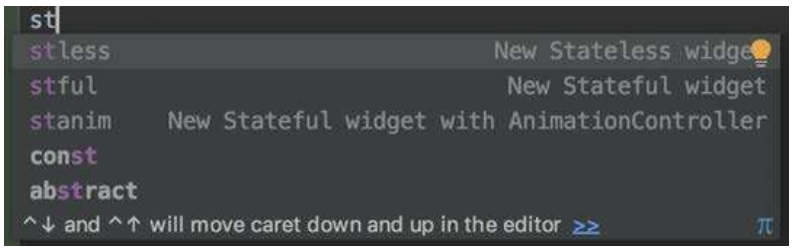

Figure 4.2: stless shortcut

8. Type in the name `ImmutableWidget` and import the material library again just as you did in *Step 2*. Now, type the following code to create a new stateless widget:

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

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

  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.green,
      child: Padding(
        padding: const EdgeInsets.all(40),
        child: Container(
          color: Colors.purple,
          child: Padding(
            padding: const EdgeInsets.all(50.0),
            child: Container(
              color: Colors.blue,
            ),
          ),
        ),
      ),
    );
  }
}
```

9. Now that this widget has been created, we can go back to `main.dart` and import the `immutable_widget.dart` file. Move your cursor over to the constructor for `ImmutableWidget` and then click the light bulb or just type the following code at the top of the `main.dart` file:

```
import './immutable_widget.dart';
```

There are two ways to import files within your project: relative and package imports. Relative imports depend on the location of the current file.

For example, with the instruction:

```
import './immutable_widget.dart';
```

the `immutable_widget.dart` file must be in the same directory as the current file. This is called a **relative import**.

You can also import the same file with the instruction:

```
import 'package:flutter_layout/immutable_widget.dart';
```

In this case, you specify your package name and the name of the file you want to import.

10. And that's it! Hit the run button to run your app in either the iOS Simulator or Android Emulator. You should see three boxes nested inside one another:

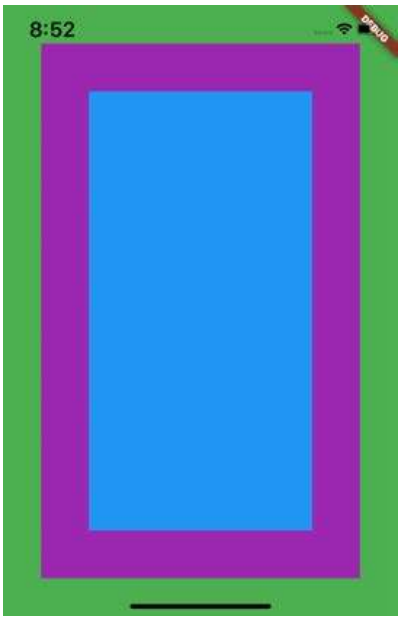

Figure 4.3: The app at the end of this recipe

To see the full-color version of Figure 4.3, and all the other images in this book, download the PDF file here: `https://packt.link/WE615`.

Congratulations! You have now created your first custom Flutter app. That wasn't so hard, was it?

How it works...

Just like every Dart application starts with the `main` function, so does every Flutter app. But in a Flutter app, you also need to call the `runApp` function:

```
void main() => runApp(const StaticApp());
```

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

Our root class, `StaticApp`, is a widget. This class will be used to set up any global data that needs to be accessed by the rest of our app. However, in this case, it will just be used to kick off our widget tree, which consists of a `MaterialApp` (a widget) and the custom `ImmutableWidget` (another widget).

When we say that *Everything you see is a widget*, this implies two things:

- Every visual item in Flutter inherits from the `Widget` class. If you want it on the screen, it's a widget. Boxes are widgets. Padding is a widget. Even screens themselves are widgets.
- The core data structure in a Flutter app is the tree. Every widget can have a child or children, which can have other children, which can have other children... This nesting is referred to as a widget tree.

DevTools is a set of tools to **debug** and **measure performance** on Flutter apps. They include the **Flutter inspector**, which you see in this recipe, but also other tools, including code analysis and diagnostics.

To learn more about DevTools, see `https://flutter.dev/docs/development/tools/devtools/overview`.

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

1. Open the command palette (*Ctrl* + *Shift* + *P*, or *Cmd* + *Shift* + *P* on a Mac).
2. Select the **Flutter: Open DevTools Widget Inspector Page** command.
3. Alternatively, in the bottom panel, hover over the {} icon and click on the **Launch** button near **Dart DevTools**.

In Android Studio/IntelliJ, perform the following steps:

1. Click on the **Flutter Inspector** tab on the right of your screen.
2. Here you can see an image of the Flutter widget inspector:

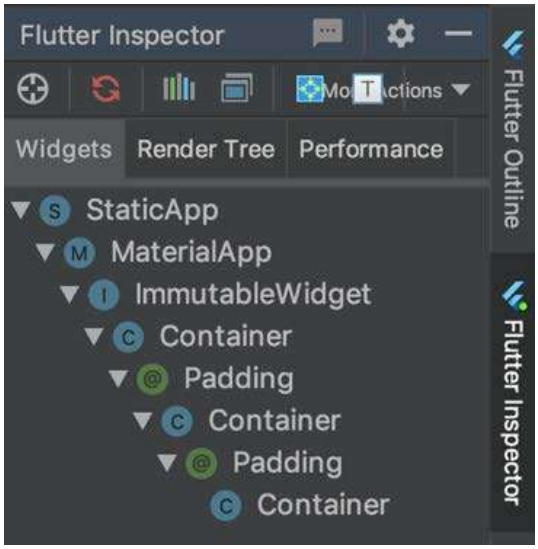

Figure 4.4: The widget tree in Android Studio

This is just a very basic app with only single-child widgets.

The core of every `StatelessWidget` is the `build()` method. Flutter will call this method every time it needs to repaint a given set of widgets. Since we are only using stateless widgets in this recipe, that should never happen, unless you rotate your device/emulator or close the app.

Let's walk through the two build methods in this example:

```
@override
Widget build(BuildContext context) {
   return MaterialApp(
     home: const ImmutableWidget(),
   );
 }
```

This first build method returns `MaterialApp`, which contains our `ImmutableWidget`. Material apps are one of the primary building blocks for apps that follow Google's Material Design specification. This widget implements a lot of features required in an app, such as navigation, theming, and localization. You can also use `CupertinoApp` if you want to follow Apple's design language, or `WidgetsApp` if you want to create your own.

We will typically use material apps at the root of the Flutter tree in this book.

There are some widgets that use different property names to describe their children. `MaterialApp` uses `home`, and `Scaffold` uses `body`. Even though these are named differently, they are still basically the same as the `child` property that you will see on most of the widgets in the Flutter framework.

Let's now have a look at the second `build` method, in the `immutable_widget.dart` file:

```
  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.green,
      child: Padding(
        padding: const EdgeInsets.all(40),
        child: Container(
          color: Colors.purple,
          child: Padding(
            padding: const EdgeInsets.all(50.0),
            child: Container(
              color: Colors.blue,
            ),
          ),
        ),
      ),
    );  }
```

This method returns a `Container` widget (green), which contains a `Padding` widget, which in turn contains another `Container` widget (purple), which contains another `Padding` widget, which contains the last `Container` widget (blue) of this tree.

A `Container` is rendered as a box that has many styling options. The three `Container` widgets are separated by two paddings of slightly different sizes.

In this example, we have chosen to create `Padding` (with an uppercase P) as a widget. `Container` widgets also have a `padding` (lowercase p) property that can specify some padding for the container itself. For example, you can write the following:

```
child: Container(
padding: EdgeInsets.all(24),
color: Colors.blue,)
```

The `Padding` widget will adjust the spacing of its child, which can be any widget of any shape or size.

At the top of the `ImmutableWidget` class, note the line:

```
const ImmutableWidget({super.key});
```

This is a **constructor**. The `const` modifier makes sure that only one global instance of this object is created and no field values can be changed during the life cycle of the object.

`super.key` is a parameter that is passed to the constructor, but because it is enclosed in curly braces, it is an **optional parameter**. In fact, from the `home` property of `MaterialApp` we just call `ImmutableWidget` without passing any parameters. `super` refers to the parent class, i.e., `StatelessWidget`, and `key` is one of its properties. In particular, `key` is used to uniquely identify a widget.

The code in this recipe, while correct, could also be written in a more efficient way. As `Container` has a `padding` property, we could remove the `Padding` widget. And if `color` is the only property of a `Container`, you can also use a `ColoredBox` widget instead of `Container`. So you could achieve the same results with the code shown here:

```
return Container(
      color: Colors.green,
      padding: const EdgeInsets.all(40),
      child: Container(
        color: Colors.purple,
        padding: const EdgeInsets.all(50.0),
        child: const ColoredBox(
          color: Colors.blue,
        ),
      ),
    );
```

Using a Scaffold

Android and iOS user interfaces are based on two different design languages. Android uses **Material Design**, while Apple has created the **Human Interface Guidelines** for iOS, but the Flutter team calls the iOS design pattern **Cupertino**, in honor of Apple's hometown. These two packages, Material Design and Cupertino, provide a set of widgets that mirror the user experience of their respective platforms.

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

In this recipe, you are going to give your app some structure. You will be using the `Scaffold` widget to add an `AppBar` to the top of the screen and a slide-out drawer that you can pull from the left.

Getting ready

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

Create a new file in the project called `basic_screen.dart`. Note that *you can have the app running while you make these code changes*. You could also adjust the size of your IDE so that the iOS Simulator or Android Emulator can fit beside it:

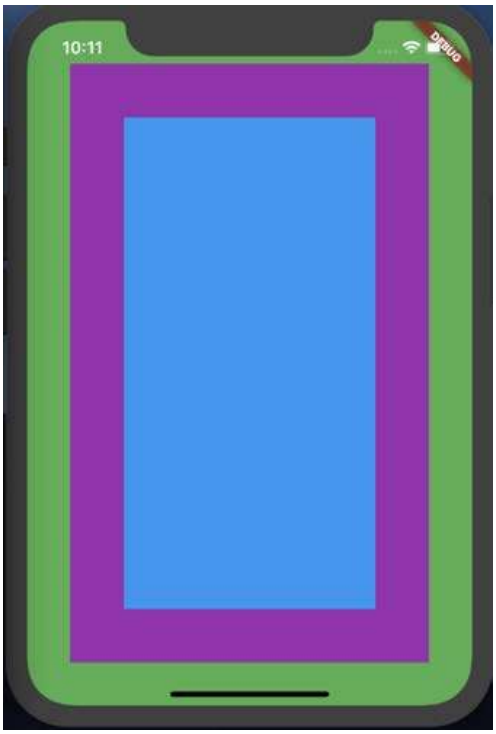

Figure 4.5: The app running on the iOS Simulator

By setting up your workspace in this way, it will be much easier to see code changes automatically injected into your app. (If you are lucky enough to be using two monitors, this does not apply, of course.)

How to do it...

Let's start by setting up a `Scaffold` widget:

1. In `basic_screen.dart`, type `stless` to create a new stateless widget and name that widget `BasicScreen`. Don't forget to import the Material Design library as well:

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

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

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

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

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

void main() => runApp(const StaticApp());

class StaticApp extends StatelessWidget {

  const StaticApp({super.key});

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: BasicScreen(),
    );
  }
}
```

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

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

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
```

```
      body: Center(
        child: AspectRatio(
          aspectRatio: 1.0,
          child: ImmutableWidget(),
        ),
      ),
    );  }
}
```

The screen should now look like this:

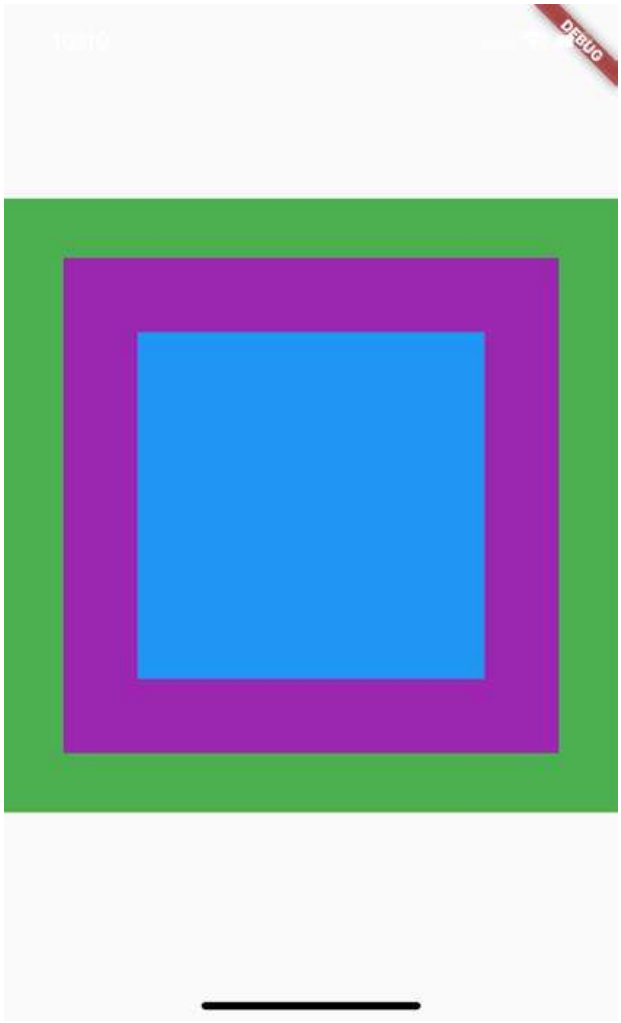

Figure 4.6: The app with a 1.0 aspectRatio

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

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

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

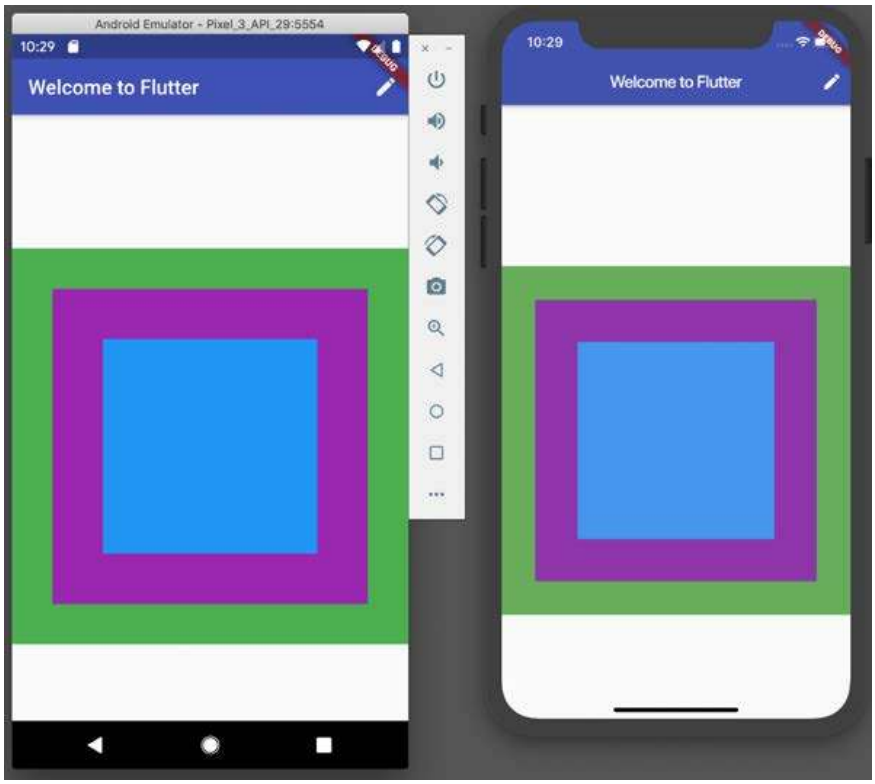

Figure 4.7: The app running on the Android Emulator and iOS Simulator

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

```
body: Center(
  child: AspectRatio(
    aspectRatio: 1.0,
    child: ImmutableWidget(),
  ),
),
drawer: Drawer(
  child: Container(
       color: Colors.lightBlue,
         child: const Center(
         child: Text("I'm a Drawer!"),
    ),
),
),
```

7. The final app should now have a "hamburger" icon (the three lines in the top left, stacked to look like a hamburger) in the `AppBar`. If you press it, the drawer will be shown:

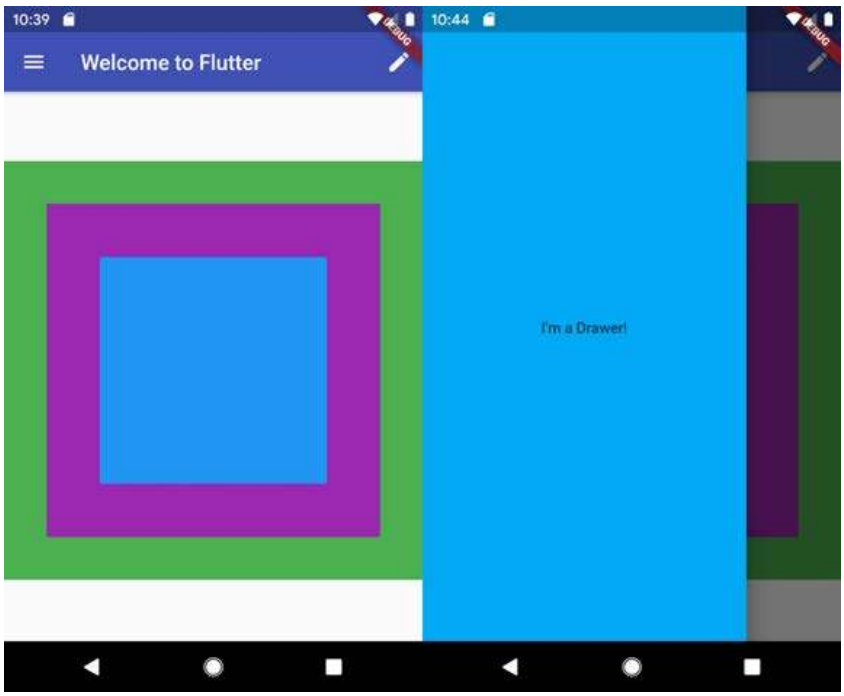

Figure 4.8: A drawer showing after pressing the "hamburger" button

How it works...

`Scaffold` is, as you may guess, a widget. It is usually recommended to use a `Scaffold` widget as the root widget for your screen, as you have in this recipe, even if it is not strictly required. You generally use a `Scaffold` widget when you want to create a screen. Widgets that do not begin with `Scaffold` are intended to be *components* used to compose screens.

Scaffolds are also aware of your device's specifics. `AppBar` will render differently depending on whether you are on iOS or Android. These are known as **platform-aware widgets**. When you add an app bar and you run your app on iOS, `AppBar` formats itself to avoid the iPhone's notch. If you run the app on an iOS device that doesn't have a notch, like the iPhone 8 or an iPad, the extra spacing added for the notch is automatically removed.

There are also other tools for `Scaffold` that we will cover in the next chapters.

Even if you don't plan on using any of the components that `Scaffold` provides, it is recommended to start every screen with a scaffold to bring consistency to your app's layout.

There are two other widgets you used in this recipe: `Center` and `AspectRatio`.

A **Center widget** centers its child both horizontally and vertically.

You can use the **AspectRatio widget** when you want to size a widget following a specific aspect ratio. The `AspectRatio` widget tries the largest width possible in its context, and then sets the height, applying the chosen aspect ratio to the width. For instance, an `AspectRatio` of `1` will set the height to be equal to the width.

Using the Container widget

We've already played around a bit with the `Container` widget in the previous recipes, so now we will build upon what you've seen before and add other features of this versatile widget. In this recipe, you will add some new effects to the existing `ImmutableWidget`.

Getting ready

Before following this recipe, you should have completed the two previous recipes in this chapter, *Creating immutable widgets*, and *Using a Scaffold*.

I suggest you also leave the app running while you are typing your code, so you can see your changes via hot reload every time you save your file.

How to do it...

Let's start by updating the small box in the center and turning it into a circle:

1. In the `ImmutableWidget` class, replace the third `Container` with this method:

```
@override
Widget build(BuildContext context) {
  return Container(
    color: Colors.green,
    child: Padding(
      padding: EdgeInsets.all(40),),
      child: Container(
        color: Colors.purple,
        child: Padding(
          padding: const EdgeInsets.all(50.0),
          child: _buildShinyCircle(),
        ),
      ),
    ),
  );
}
```

2. Write the `_buildShinyCircle()` method. You will be adding `BoxDecoration` to a `Container`, which can include gradients, shapes, shadows, borders, and even images.

After adding `BoxDecoration`, you should make sure to remove the original `color` property on the container; otherwise, you will get an exception. Containers can have a `decoration` or a `color`, but not both.

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

```
Widget _buildShinyCircle() {
   return Container(
     decoration: const BoxDecoration(
       shape: BoxShape.circle,
       gradient: RadialGradient(
         colors: [
           Colors.lightBlueAccent,
```

```
            Colors.blueAccent,
          ],
          center: Alignment(-0.3, -0.5),
        ),
        boxShadow: [
          BoxShadow(blurRadius: 20),
        ],
      ),
    );
  }
}
```

Circles are only one kind of shape that can be defined in a `Container`. You can create rounded rectangles and give these shapes an explicit size instead of letting Flutter figure out how large the shape should be.

4. Add an `import` statement to the top of the screen for Dart's math library and give it an alias of `Math`:

```
import 'dart:math' as math;
```

5. Now, update the second container (the purple one) with the decoration below and wrap it in a `Transform` widget. To make your life easier, you can use your IDE to insert another widget inside the tree. Move your cursor to the declaration of the second `Container` and then, in VS Code, press *Ctrl* + . (*Command* + . on a Mac) and in Android Studio, press *Alt* + *Enter* (*Option* + *Enter* on a Mac) to bring up the following context dialog:

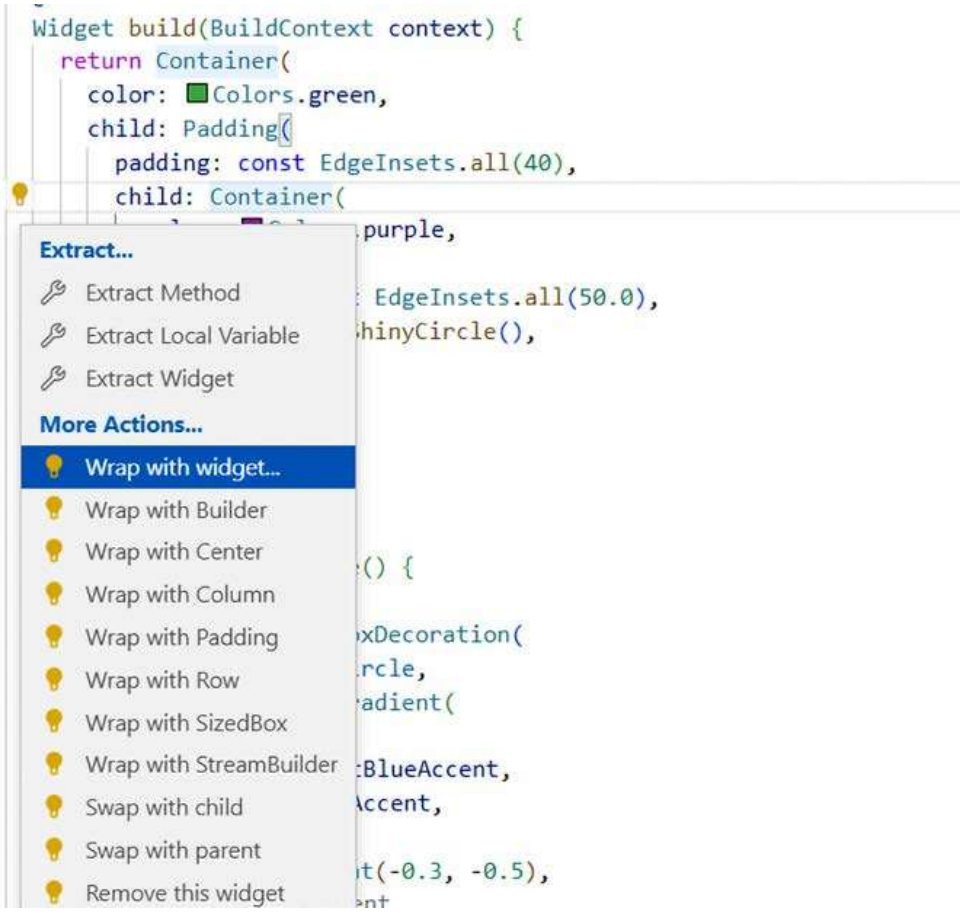

Figure 4.9: A menu showing Flutter code actions

6. Select **Wrap with widget...** or **Wrap with a new widget**, which will insert a placeholder in your code. Replace the placeholder with `Transform.rotate` and add the missing properties, as shown in the updated code here:

```
return Container(
  color: Colors.green,
  child: Center(
```

```
    child: Transform.rotate(
     angle: 180 / Math.pi, // Rotations are supplied in radians
      child: Container(
        width: 250,
        height: 250,
        decoration: BoxDecoration(
          color: Colors.purple,
          boxShadow: [
            BoxShadow(
              color: Colors.deepPurple.withAlpha(120),
              spreadRadius: 4,
              blurRadius: 15,
              offset: Offset.fromDirection(1.0, 30),
            ),
          ],
          borderRadius: const BorderRadius.all(Radius.circular(20)),
        ),
        child: Padding(
          padding: const EdgeInsets.all(50.0),
          child: _buildShinyCircle(),
        ),
      ),
    ),
  ),
);
```

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

```
return Container(
  decoration: const BoxDecoration(color: Colors.green),
      foregroundDecoration: const BoxDecoration(
        backgroundBlendMode: BlendMode.colorBurn,
        gradient: LinearGradient(
          begin: Alignment.topCenter,
          end: Alignment.bottomCenter,
```

```
            colors: [
              Color(0xAA0d6123),
              Color(0x00000000),
              Color(0xAA0d6123),
            ],
          ),
        ),
      ),
  child: […]
```

Your final project should look similar to that in *Figure 4.10*:

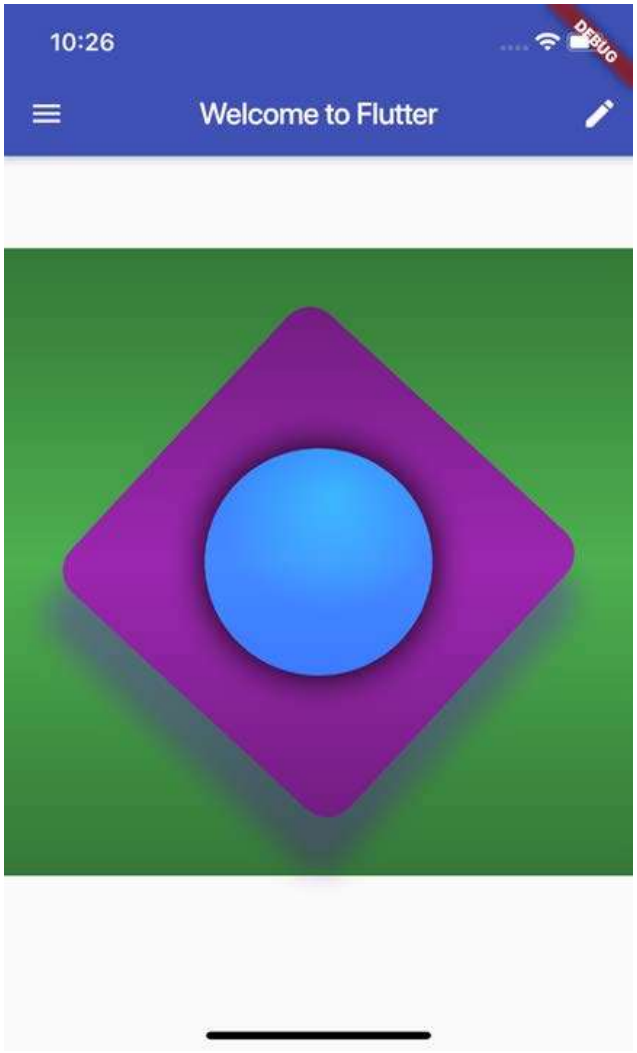

Figure 4.10: The screen after completing this recipe

How it works...

`Container` widgets can add several effects to their children. Like Scaffolds, they enable customizations that can be explored and experimented with.

A `BoxDecoration` can draw:

- Borders
- Shadows
- Colors
- Gradients
- Images
- Shapes (rectangle or circles)

The `Container` itself supports two decorations—the primary background decoration and a foreground decoration, which is painted on top of the container's child.

Containers can also provide their own transforms (like how you rotated the second container), paddings, and margins. Sometimes, you may prefer adding properties such as `padding` inside a container. In other cases, you may use a `Padding` widget and add `Container` as its child, as we did in this recipe. Both achieve the same result.

In this recipe, we *could* also have rotated the box by supplying a `Matrix4` to the `transform` property of the container, but delegating that task to a separate widget follows Flutter's ideology: widgets should only do one very small thing and be composed to create complex designs.

Don't worry about how deep your widget tree gets. Flutter can take it. Widgets are extremely lightweight and are optimized to support hundreds of layers. The widget itself doesn't do any rendering; it just provides a configuration for the renderer. The actual rendering is done in two more parallel trees, the `Element` tree and the `RenderObject` tree. Flutter uses these internal trees to talk to the GPU.

Printing stylish text on the screen

Almost every app has to display some text. Flutter has a powerful and fast text engine that can render all the rich text that you'd expect from a modern mobile framework.

In this recipe, we will be drawing text with Flutter's two primary widgets—`Text` and `RichText`. The `Text` widget is the most common way to quickly print text on the screen, but you will also occasionally need `RichText` when you want to add even more style within a single line.

Getting ready

To follow along with this recipe, you should have completed all of the previous recipes in this chapter.

Create a new file called `text_layout.dart` in your project's `lib` directory.

As always, make sure that your app is running in either a simulator/emulator or an actual device to see the changes in your app in real time with the hot reload feature.

How to do it...

Let's get started with some basic text:

1. In your `text_layout.dart` file, add the shell for a class called `TextLayout`, which will extend `StatelessWidget`. Import all the requisite packages:

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

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

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

2. Open `basic_screen.dart` and perform these updates so that the `TextLayout` widget will be displayed underneath `ImmutableWidget`.

   For the sake of brevity, `AppBar` and the `Drawer` code have been removed and an ellipsis (...) is shown instead:

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

```
class BasicScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(...),
      body: Column(
        children: const [
          AspectRatio(
            aspectRatio: 1.0,
            child: ImmutableWidget(),
          ),
          TextLayout(),
        ],
      ),
 drawer: Drawer(...),
    );
  }
}
```

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

```
  @override
  Widget build(BuildContext context) {
    return Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: [
        const Text(
          'Hello, World!',
          style: TextStyle(fontSize: 16),
        ),
        Text(
          'Text can wrap without issue',
          style: Theme.of(context).textTheme.headline6,
        ),
//make sure the Text below is all in one line:
        const Text(
```

```
            'Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Etiam at mauris massa. Suspendisse potenti. Aenean aliquet eu
nisl vitae tempus.'),
      ],
    );  }
```

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

Figure 4.11: Formatted text

4. All of these `Text` widgets can take a single style object, but what if you want to add multiple styles to different parts of your sentences? That is the job of `RichText`. Add these new widgets just after the last widget in your column:

```
const Divider(),
   RichText(
      text: const TextSpan(
      text: 'Flutter text is ',
      style: TextStyle(fontSize: 22, color: Colors.black),
        children: <TextSpan>[
          TextSpan(
            text: 'really ',
            style: TextStyle(
              fontWeight: FontWeight.bold,
              color: Colors.red,
            ),
            children: [
              TextSpan(
                text: 'powerful.',
                style: TextStyle(
                  decoration: TextDecoration.underline,
                  decorationStyle: TextDecorationStyle.double,
                  fontSize: 40,
                ),
              ),
            ],
          ),
        ],
      ),),
```

This is what the final screen should look like:

Figure 4.12: The screen after completing this recipe

How it works...

Hopefully, most of the code for the `Text` widget should be self-evident. It's just a matter of creating hundreds of these widgets over time, which will eventually create fluency with this API. The `Text` widget has some basic properties that warrant discussion, such as text alignment and setting a maximum number of lines, but the real meat is in the `TextStyle` object. There are several properties in `TextStyles` that are exhaustively covered in the official documentation, but you will most frequently be adjusting the font size, color, weight, and font.

As a bonus, `Text` widgets are accessibility-aware out of the box. There is no extra code that you'll need to write. `Text` widgets respond to the text-to-speech synthesizers and will even scale their font size up and down if the user decides to adjust the system's font size.

The `RichText` widget creates another tree of `TextSpan`, where each child inherits its parent's style, but can override it with its own.

We have three spans in the example and each one adds a bit more to the inherited style:

- Font size 22, colored black
- Font weight bold, colored red
- Font size 40, double underline

The final span will be styled with the sum of all its parent spans in the tree.

There's more...

At the beginning of the recipe, did you notice this line?

```
Theme.of(context).textTheme.headline6,
```

This is a very common Flutter design pattern, and we will call it the **of-context** pattern in this book. This is used to access data from other parts higher up the widget tree.

Every build method in every widget is provided with a `BuildContext` object, which is a very abstract-sounding name. `BuildContext`, or `context` for short, can be thought of as a marker of your widget's location in the tree. This `context` can then be used to travel up and down the widget tree.

In this case, we're handing our context to the static `Theme.of(context)` method, which will then search up the tree to find the closest `Theme` widget. The theme has predetermined colors and text styles that can be added to your widgets so that they will have a consistent look in your app. This code is adding the global `headline6` style to this text widget.

See also

If you want to learn more about how to set themes in your apps and even create your own custom ones, have a look at the official guide at `https://docs.flutter.dev/cookbook/design/themes`.

Importing fonts and images into your app

Text and colors are nice, but pictures are often worth a thousand words. The process of adding custom images and fonts to your app is a little more complex than you might expect. Flutter has to work within the constraints of its host operating systems, and since iOS and Android like to do similar things in different ways, Flutter creates a unified abstraction layer on top of both of their filesystems.

In this recipe, we will be using asset bundles to add a photo at the top of the screen and use a custom font.

Getting ready

You should have completed the previous recipe in this chapter, *Printing stylish text on the screen*, before following along with this one.

You will add an image to the app. You can get some great free stock photography from Unsplash. Download this beach image by Khachik Simonian as well: `https://unsplash.com/photos/nXOB-wh4Oyc`.

How to do it...

Let's update the previous recipe's code with some new fonts:

1. Open the `pubspec.yaml` file in the root folder of your project.
2. In the `pubspec.yaml` file, add the `google_fonts` package in the `dependencies` section, but be careful. YAML is one of the few languages where white space matters, so be sure to put the `google_fonts` dependency precisely with the same indent as `flutter`, as shown here:

```
dependencies:
  flutter:
    sdk: flutter
  google_fonts: ^3.0.1
```

3. You can also add a dependency from the terminal, typing:

```
flutter pub add google_fonts
```

4. After this is done, run `flutter pub get` to rebuild your asset bundle, or alternatively press the **Get Packages** button in your editor (in Android Studio/IntelliJ you'll find it in the action ribbon at the top of `pubspec.yaml`, in VS Code it's an icon at the top right of the screen).
5. We can now add any Google font to the text widgets in `text_layout.dart`. Add the `google_fonts` import at the top of the file:

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

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

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

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

Figure 4.13: Hello World with the leckerliOne font

7. Now you will add a picture to the screen. Stop the app. Then, at the root of your project, create a new folder called `assets`.
8. Rename the file you have downloaded (refer to the *Getting ready* section of this recipe) to something simple, such as `beach.jpg`, and drag the image to the `assets` folder.
9. Update the `pubspec.yaml` file once again. Locate and uncomment the `assets` section of the file to include the image folder in your project:

```
# To add assets to your application, add an assets section, like this:
  assets:
    - assets/
```

10. In `basic_screen.dart`, replace `body` of the `Scaffold` with this code:

```
body: Column(
  crossAxisAlignment: CrossAxisAlignment.start,
  children: [
    Image.asset('assets/beach.jpg'),
    const TextLayout(),
  ],
),
```

11. Wrap the image with a `Semantics` widget, and add a description of the image you are showing:

```
Semantics(
     image: true,
     label: 'A beautiful beach',
     child: Image.asset('assets/beach.jpg')),
```

The final layout should show the image at the top of the screen:

Figure 4.14: An image showing on the screen

How it works...

In this recipe, you have seen two common features in Flutter: choosing fonts for your `Text` widgets and adding images to your screens.

You may use any custom font in Flutter. When working with Google Fonts, adding them to your project is extremely easy. You just need to add the `google_fonts` package dependency in the `pubspec.yaml` file to your app:

```
google_fonts: ^3.0.1
```

There are currently about 1,000 fonts you can use for free in your apps with Google Fonts. Have a look at the official site, `https://fonts.google.com/`, to choose the right one for your apps.

When you want to use the `google_fonts` package in one or more of your screens, you need to import the package at the top of the file. You can do this in the `text_layout.dart` file with the help of the following command:

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

From there, you just need to use the package. You add the `GoogleFonts` widget to the `style` property of your `Text`:

```
  style: GoogleFonts.leckerliOne(fontSize: 40),
```

When adding the image to the `pubspec.yaml` file, you have provided Flutter with instructions on how to build an asset bundle. The bundles then get converted to their platform equivalents (`NSBundle` on iOS and `AssetManager` on Android) where they can be retrieved through the appropriate file API.

For listing assets, we thankfully do not have to explicitly reference every single file in our assets directory:

```
- assets/
```

This shorthand is equivalent to saying that you want to add each file in the `assets` directory to the asset bundle.

You can also write this:

```
- assets/beach.jpg
```

This notation will only add the file that you specify in that exact directory.

If you have any sub-directories, they will also have to be declared in `pubspec`. For example, if you have `images` and `icons` folders in the `assets` folder, you should write the following:

```
- assets/
- assets/images/
- assets/icons/
```

You might want to keep all your project assets in one flat directory because of this and not go too crazy with the file organizations.

The last step in this recipe was wrapping the image within a `Semantics` widget:

```
Semantics(
     image: true,
     label: 'A beautiful beach',
     child: Image.asset('assets/beach.jpg')),
```

You can use the `Semantics` widget to describe the purpose or meaning of a widget to users with accessibility needs. In our example, we provide a label to describe the content of the image. The text description in the label can be read by screen readers and other accessibility tools.

See also

For a comprehensive guide on how to add assets in Flutter, have a look at the assets guide: `https://docs.flutter.dev/development/ui/assets-and-images`.

Summary

In this chapter, you have had an introduction to building user interfaces in a Flutter app.

You have seen how to create immutable widgets and use them in your apps.

The chapter covered the use of the `Scaffold` widget, which provides a basic structure for screens in your apps.

You have also used Containers, powerful widgets that provide several layout-related properties and effects.

You have used the `Text` widget to print text on the screen, and seen how to customize the font, color, size, and other properties of the text.

You've seen how to import fonts and images into a Flutter app: in particular, you added and used Google fonts and displayed an image from the assets directory in your project.

Finally, you have seen how to describe onscreen content using the `Semantics` widget: this can help to improve the accessibility of your app.

In general, you are now familiar with the basics of creating user interfaces. The skills that you have acquired in this chapter will provide the foundation for building engaging and performant UIs in Flutter, in both simple and more complex apps.