1

Introducing Flutter and Getting Started

WHAT YOU WILL LEARN IN THIS CHAPTER

- What the Flutter framework is
- What Flutter's benefits are
- How Flutter and Dart work together
- What a Flutter `Widget` is
- What an `Element` is
- What a `RenderObject` is
- What type of Flutter widgets are available
- What the stateless and stateful `Widget` lifecycle is
- How the widget tree and element tree work together
- How to install the Flutter SDK
- How to install Xcode on macOS and Android Studio on macOS, Windows, and Linux
- How to configure an editor
- How to install the Flutter and Dart plugins

In this chapter, you'll learn how the Flutter framework works behind the scenes. Flutter uses widgets to create the user interface (UI), and Dart is the language used to develop the applications. Once you understand how Flutter handles and implements widgets, it will help you in architecting your apps.

You'll learn how to install the Flutter SDK on macOS, Windows, and Linux. You'll configure Android Studio to install the Flutter plugin to run, debug, and use hot reload. You'll install the Dart plugin for code analysis, code validation, and code completion.

INTRODUCING FLUTTER

Flutter is Google's portable UI framework for building modern, native, and reactive applications for iOS and Android. Google is also working on Flutter desktop embedding and Flutter for the Web (Hummingbird) and embedded devices (Raspberry Pi, home, automotive, and more). Flutter is an open-source project hosted on GitHub with contributions from Google and the community. Flutter uses Dart, a modern object-oriented language that compiles to native ARM code and production-ready JavaScript code. Flutter uses the Skia 2D rendering engine that works with different types of hardware and software platforms and is also used by Google Chrome, Chrome OS, Android, Mozilla Firefox, Firefox OS, and others. Skia is sponsored and managed by Google and is available for anyone to use under the BSD Free Software License. Skia uses a CPU-based path render and also supports the OpenGL ES2-accelerated backend.

Dart is the language that you'll use to develop your Flutter applications, and you'll learn more about it in Chapter 3, "Learning Dart Basics." Dart is ahead-of-time (AOT) compiled to native code, making your Flutter application fast. Dart is also just-in-time (JIT) compiled, making it fast to display your code changes such as via Flutter's stateful hot reload feature.

Flutter uses Dart to create your user interface, removing the need to use separate languages like Markup or visual designers. Flutter is declarative; in other words, Flutter builds the UI to reflect the state of the app. When the state (data) changes, the UI is redrawn, and Flutter constructs a new instance of the widget. In the "Understanding the Widget Tree and the Element Tree" section of this chapter, you'll learn how widgets are configured and mounted (rendered) creating the widget tree and element tree, but under the hood, the render tree (a third tree) uses the `RenderObject`, which computes and implements the basic layout and paint protocols. (You won't need to interact directly with the render tree or the `RenderObject`, and I won't discuss them further in this book.)

Flutter is fast, and the rendering runs at 60 frames per second (fps) and 120fps for capable devices. The higher the fps, the smoother the animations and transitions.

Applications made in Flutter are built from a single codebase, are compiled to native ARM code, use the graphics processing unit (GPU), and can access specific iOS and Android APIs (like GPS location, image library) by communicating via platform channels. You'll learn more about platform channels in Chapter 12, "Writing Platform-Native Code."

Flutter provides the developer with tools to create beautiful and professional-looking applications and with the ability to customize any aspect of the application. You'll be able to add smooth animations, gesture detection, and splash feedback behavior to the UI. Flutter applications result in native performance for both iOS and Android platforms. During development, Flutter uses hot reload to refresh the running application in milliseconds when you change the source code to add new features or modify existing ones. Using hot reload is a great way to see the changes you make to your code on the simulator or device while keeping the application's state, the data values, on the screen.

Defining Widgets and Elements

The Flutter UI is implemented by using widgets from a modern reactive framework. Flutter uses its own rendering engine to draw widgets. In Chapter 5, "Understanding the Widget Tree," you'll get an introduction to widgets, and in Chapter 6, "Using Common Widgets," you'll learn how to implement widgets.

You might be asking, what is a widget? Widgets can be compared to LEGO blocks; by adding blocks together, you create an object, and by adding different kinds of blocks, you can alter the look and behavior of the object. Widgets are the building blocks of a Flutter app, and each widget is an immutable declaration of the user interface. In other words, widgets are configurations (instructions) for different parts of the UI. Placing the widgets together creates the widget tree. For example, say an architect draws a blueprint of a house; all of the objects like walls, windows, and doors in the house are the widgets, and all of them work together to create the house or, in this case, the application.

Since widgets are the configuration pieces of the UI and together they create the widget tree, how does Flutter use these configurations? Flutter uses the widget as the configuration to build each element, which means the element is the widget that is mounted (rendered) on the screen. The elements that are mounted on the screen create the element tree. You'll learn more about the widget tree and element tree in the next section, "Understanding the Widget Tree and the Element Tree." You'll also learn to manipulate the widget tree in detail in Chapter 5.

Here's a brief look at the wide array of widgets at your disposal:

- Widgets with structuring elements such as a list, grid, text, and button
- Widgets with input elements such as a form, form fields, and keyboard listeners
- Widgets with styling elements such as font type, size, weight, color, border, and shadow
- Widgets to lay out the UI such as row, column, stack, centering, and padding
- Widgets with interactive elements that respond to touch, gestures, dragging, and dismissible
- Widgets with animation and motion elements such as hero animation, animated container, animated crossfade, fade transition, rotation, scale, size, slide, and opacity
- Widgets with elements like assets, images, and icons
- Widgets that can be nested together to create the UI needed
- Custom widgets you can create yourself

UNDERSTANDING WIDGET LIFECYCLE EVENTS

In programming, you have different lifecycle events that usually happen in a linear mode, one after another as each stage is completed. In this section, you'll learn the widget lifecycle events and their purpose.

To build the UI, you use two main types of widgets, `StatelessWidget` and `StatefulWidget`. A stateless widget is used when the values (state) do not change, and the stateful widget is used when

values (state) change. In Chapter 2, "Creating a Hello World App," you'll learn in detail when to use a `StatelessWidget` or a `StatefulWidget`. Each stateless or stateful widget has a `build` method with a `BuildContext` that handles the location of the widget in the widget tree. The `BuildContext` objects are actually `Element` objects, an instantiation of the `Widget` at a location in the tree.

The StatelessWidget Lifecycle

A `StatelessWidget` is built based on its own configuration and does not change dynamically. For example, the screen displays an image with a description and will not change. The stateless widget is declared with one class, and you'll learn about classes in Chapter 3. The `build` (the UI portions) method of the stateless widget can be called from three different scenarios. It can be called the first time the widget is created, when the widget's parent changes, and when an `InheritedWidget` has changed. In Chapter 15, "Adding State Management to the Firestore Client App," you'll learn how to implement the `InheritedWidget`.

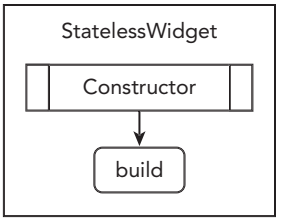

FIGURE 1.1: Stateless-Widget lifecycle

The following sample code shows a `StatelessWidget` base structure, and Figure 1.1 displays the widget's lifecycle.

```
class JournalList extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

The StatefulWidget Lifecycle

A `StatefulWidget` is built based on its own configuration but can change dynamically. For example, the screen displays an icon with a description, but values can change based on the user's interaction, like choosing a different icon or description. This type of widget has a mutable state that can change over time. The stateful widget is declared with two classes, the `StatefulWidget` class and the `State` class. The `StatefulWidget` class is rebuilt when the widget's configuration changes, but the `State` class can persist (remain), enhancing performance. For example, when the state changes, the widget is rebuilt. If the `StatefulWidget` is removed from the tree and then inserted back into the tree sometime in the future, a new `State` object is created. Note that under certain circumstances and restrictions, you can use a `GlobalKey` (unique key across entire app) to reuse (not re-create) the `State` object; however, global keys are expensive, and unless they're needed, you might want to consider not using them. You call the `setState()` method to notify the framework that this object has changes, and the widget's `build` method is called (scheduled). You would set the new state values inside the `setState()` method. In Chapter 2, you'll learn how to call the `setState()` method.

The following example shows a `StatefulWidget` base structure, and Figure 1.2 displays the widget's lifecycle. You have two classes, the `JournalEdit StatefulWidget` class and the `_Journal-EditState` class.

```
class JournalEdit extends StatefulWidget {
  @override
  _ JournalEditState createState() => _ JournalEditState();
```

```
}

class _JournalEditState extends State<JournalEdit> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

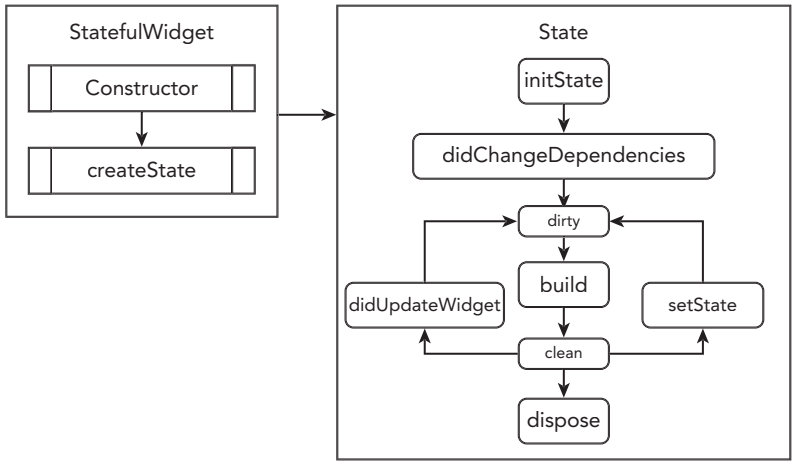

FIGURE 1.2: StatefulWidget lifecycle

You can override different portions of the `StatefulWidget` to customize and manipulate data at different points of the widget lifecycle. Table 1.1 shows some of the stateful widget main overrides, and the majority of the time you'll use the `initState()`, `didChangeDependencies()`, and `dispose()` methods. You'll use the `build()` method all of the time to build your UI.

TABLE 1.1: StatefulWidget lifecycle

METHOD	DESCRIPTION	SAMPLE CODE

`initState()`	Called once when this object is inserted into the tree.	`@override`
		`void initState() {`
		`  super.initState();`
		`  print('initState');`
		`}`

`dispose()`	Called when this object is removed from the tree permanently.	`@override`
		`void dispose() {`
		`  print('dispose');`
		`  super.dispose();`
		`}`

continues

TABLE 1.1 (*continued*)

METHOD	DESCRIPTION	SAMPLE CODE

`didChangeDependencies()`	Called when this `State` object changes.	`@override`
		`void didChangeDependencies() {`
		
		`  super.didChangeDependencies();`
		`  print('didChangeDependencies');`
		`}`

`didUpdateWidget-(Contacts oldWidget)`	Called when the widget configuration changes.	`@override`
		`void didUpdateWidget(Contacts oldWidget) {`
		
		`super.didUpdateWidget(oldWidget);`
		`print('didUpdateWidget: $oldWidget');`
		`}`

`deactivate()`	Called when this object is removed from the tree.	`@override`
		`void deactivate() {`
		
		`  print('deactivate');`
		`  super.deactivate();`
		`}`

`build(BuildContext context)`	Can be called multiple times to build the UI, and the `BuildContext` handles the location of this widget in the widget tree.	`@override`
		`Widget build(BuildContext context) {`
		`  print('build');`
		`  return Container();`
		`}`

`setState()`	Notifies the framework that the state of this object has changed to schedule calling the `build` for this `State` object.	`setState(() {`
		`  name = _newValue;`
		`});`

UNDERSTANDING THE WIDGET TREE AND THE ELEMENT TREE

In the previous section, you learned that widgets contain the instructions to create the UI, and when you compose (nest) widgets together, they create the widget tree. The Flutter framework uses the widgets as the configurations for each element that is mounted (rendered) on the screen. The mounted

elements displayed on the screen create the element tree. You now have two trees, the widget tree that has the widget configurations and the element tree that represents the rendered widgets on the screen (Figure 1.3).

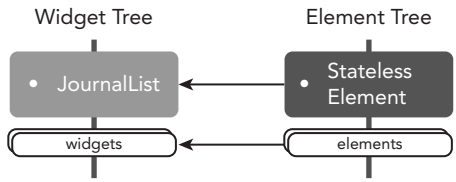

FIGURE 1.3: Widget tree and element tree

When the application starts, the `main()` function calls the `runApp()` method, usually taking a `StatelessWidget` as the argument, and is mounted as the root element for the application. The Flutter framework processes through all of the widgets, and each corresponding element is mounted.

The following is sample code that starts a Flutter application, and the `runApp()` method inflates the `MyApp StatelessWidget`, meaning the main application itself is a widget. As you can see, just about everything in Flutter is a widget.

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

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

Elements have a reference to the widget and are responsible for comparing the widget differences. If a widget is responsible for building child widgets, then elements are created for each child widget. When you see the use of `BuildContext` objects, they are the `Element` objects. To discourage direct manipulation of the `Element` objects, the `BuildContext` interface is used instead. The Flutter framework uses `BuildContext` objects to discourage you from manipulating the `Element` objects. In other words, you'll be using widgets to create your UI layouts, but it's good to know how the Flutter framework is architected and how it works behind the scenes.

As noted earlier, there is a third tree called the *render tree* that is a low-level layout and painting system that inherits from the `RenderObject`. The `RenderObject` computes and implements the basic layout and paint protocols. You won't need to interact directly with the render tree, however, and will be using the widgets.

Stateless Widget and Element Trees

A stateless widget has the configuration to create a stateless element. Each stateless widget has a corresponding stateless element. The Flutter framework calls the `createElement` method (create an instance), and the stateless element is created and mounted to the element tree. In other

words, the Flutter framework makes a request from the widget to create an element and then mounts (adds) the element to the element tree. Each element contains a reference back to the widget. The element calls the widget's `build` method to check for children widgets, and each child widget (like an Icon or Text) creates its own element and is mounted to the element tree. This process results in two trees: the widget tree and the element tree.

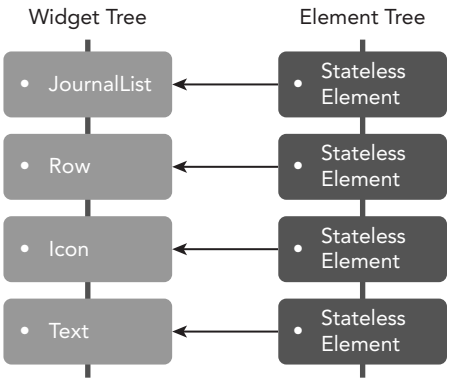

FIGURE 1.4: StatelessWidget widget tree and element tree

Figure 1.4 shows the `JournalList StatelessWidget` that has `Row`, `Icon`, and `Text` widgets representing the widget tree. The Flutter framework asks each widget to create the element, and each element has a reference back to the widget. This process happens for each widget down the widget tree and creates the element tree. The widget contains the instructions to build the element mounted on the screen. Note that you the developer create the widgets, and the Flutter framework handles mounting the elements, creating the element tree.

```
// Simplified sample code
class JournalList extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Row(
      children: <Widget>[
        Icon(),
        Text(),
      ],
    );
  }
}
```

Stateful Widget and Element Trees

A stateful widget has the configuration to create a stateful element. Each stateful widget has a corresponding stateful element. The Flutter framework calls the `createElement` method to create the stateful element, and the stateful element is mounted to the element tree. Since this is a stateful widget, the stateful element requests that the widget create a state object by calling the `StatefulWidget` class's `createState` method.

The stateful element now has a reference to the state object and the widget at the given location in the element tree. The stateful element calls the state object widget's `build` method to check for child widgets, and each child widget creates its own element and is mounted to the element tree. This process results in two trees: the widget tree and the element tree. Note that if a child widget displaying the state (journal note) is a stateless widget like the `Text` widget, then the element created for this widget is a stateless element. The state object maintains a reference to the widget (`StatefulWidget`

class) and also handles the construction for the `Text` widget with the latest value. Figure 1.5 shows the widget tree, the element tree, and the state object. Note that the stateful element has a reference to the stateful widget and the state object.

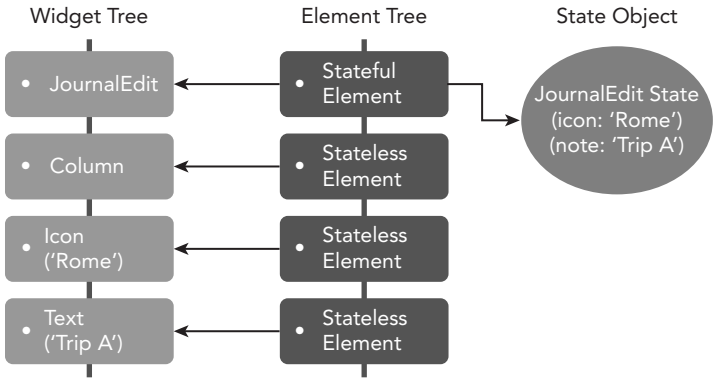

FIGURE 1.5: StatefulWidget widget tree, the element tree, and the state object

To update the UI with new data, you call the `setState()` method that you learned about in the "The StatefulWidget Lifecycle" section. To set new data (properties/variables) values, call the `setState()` method to update the state object, and the state object marks the element as dirty (has changed) and causes the UI to be updated (scheduled). The stateful element calls the state object's `build` method to rebuild the children widgets. A new widget is created with the new state value, and the old widget is removed.

For example, you have a `StatefulWidget JournalEntry` class, and in the `State` object class you call the `setState()` method to change the `Text` widget description from `'Trip A'` to `'Trip B'` by setting the `note` variable value to `'Trip B'`. The state object `note` variable is updated to the `'Trip B'` value, the state object marks the element as dirty, and the `build` method rebuilds the UI children widgets. The new `Text` widget is created with the new `'Trip B'` value, and the old `Text` widget with the `'Trip A'` value is removed (Figure 1.6).

```
// Simplified sample code
class JournalEdit extends StatefulWidget {
  @override
  _JournalEditState createState() => _JournalEditState();
}

class _JournalEditState extends State<JournalEdit> {
  String note = 'Trip A';

  void _onPressed() {
    setState(() {
      note = 'Trip B';
    });
  }
```

```
  @override
  Widget build(BuildContext context) {
    return Column(
      children: <Widget>[
        Icon(),
        Text('$note'),
        FlatButton(
          onPressed: _onPressed,
        ),
      ],
    );
  }
}
```

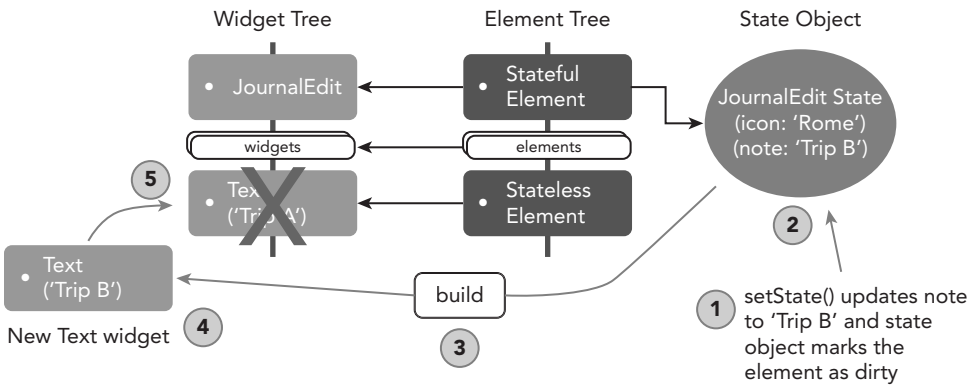

FIGURE 1.6: Updating the state process

Since both the old and new widgets are both `Text` widgets, the existing element updates its reference to the new widget, and the element stays in the element tree. The `Text` widget is a stateless widget, and the corresponding element is a stateless element; although the `Text` widget has been replaced, the state is maintained (persists). State objects have a long life span and remain attached to the element tree as long as the new widget is the same type as the old widget.

Let's continue with the previous example; the old `Text` widget with the `'Trip A'` value is removed and replaced by the new `Text` widget with the `'Trip B'` value. Since both the old and new widgets are of the same type of `Text` widgets, the element stays on the element tree with the updated reference to the new `Text 'Trip B'` widget (Figure 1.7).

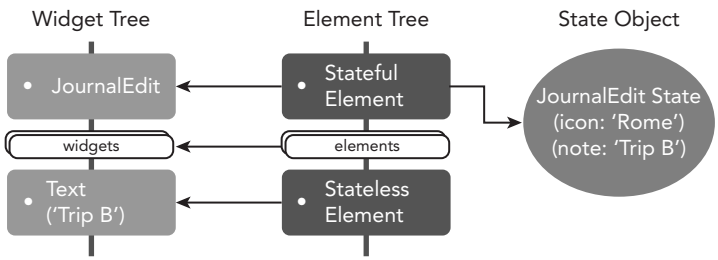

FIGURE 1.7: Updated state for the widget tree and element tree

INSTALLING THE FLUTTER SDK

Installing the Flutter SDK requires downloading the current version, which is 1.5.4 at the time of writing (your version might be higher), of the SDK from the Flutter website. This chapter includes sections for installing on macOS, Windows, and Linux. (Note that targeting and compiling for the iOS platform requires a Mac computer and Xcode, Apple's development environment). Do not get discouraged by the number of steps; you'll do this the first time you install only.

You'll be using the Terminal window to run installation and configuration commands.

Installing on macOS

Before starting the installation, you need to make sure your Mac supports the minimum hardware and software requirements.

System Requirements

- macOS (64-bit)
- 700 MB of disk space (does not include disk space for the integrated development environment and other tools)
- The following command-line tools:
  - `Bash`
  - `mkdir`
  - `rm`
  - `git`
  - `curl`
  - `unzip`
  - `which`

Get the Flutter SDK

The latest installation details are available online on the Flutter website at `https://flutter.dev/docs/get-started/install/macos/`. Execute steps 2 and following from your Mac's Terminal window.

1. Download the following installation file to get the latest release, v1.7.8 (your version might be higher), of the Flutter SDK:

   `https://storage.googleapis.com/flutter_infra/releases/stable/macos/flutter_macos_v1.7.8+hotfix.4-stable.zip`.

2. Extract the file in the desired location by using the Terminal window.

```
cd ~/development
unzip ~/Downloads/flutter_macos_v1.7.8+hotfix.4-stable.zip
```

3. Add the Flutter tool to your path (`pwd` means present working directory; in your case, it will be the `development` folder).

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

4. Update the path permanently.

   a. Get the path that you used in step 3. For example, replace `MacUserName` with your path to the development folder.

   ```
   /Users/MacUserName/development/flutter/bin
   ```

   b. Open or create `$HOME/.bash_profile`. Your file path or filename might be different on your computer.

   c. Edit `.bash_profile` (will open a command-line editor).

   d. Type **`nano .bash_profile`**.

   e. Type **`export PATH=/Users/MacUserName/development/flutter/bin:$PATH`**.

   f. Save by pressing ^X (Control+X), confirm by pressing Y (Yes), and press Enter to accept the filename.

   g. Close the Terminal window.

   h. Reopen the Terminal window and type **`echo $PATH`** to verify that the path has been added. Then type **`flutter`** to make sure the path is working. If the command is not recognized, something went wrong with the `PATH`. Check to make sure you have the correct username for your computer in the path.

Check for Dependencies

Run the following command from the Terminal window to check for dependencies that need to be installed to finish the setup:

```
flutter doctor
```

View the report and check for other software that may be needed.

iOS Setup: Install Xcode

A Mac with Xcode 10.0 or newer is required.

1. Open App Store and install Xcode.
2. Configure the Xcode command-line tools by running **`sudo xcode-select --switch /Applications/Xcode.app/Contents/Developer`** from the Terminal window.
3. Confirm that the Xcode license agreement is signed by **`running sudo xcodebuild -license`** from the terminal.

Android Setup: Install Android Studio

Full installation details are available at `https://flutter.dev/docs/get-started/install/macos#android-setup`. Flutter requires a full installation of Android Studio for the Android platform dependencies. Keep in mind that Flutter apps can be written in different editors such as Visual Code or IntelliJ IDEA.

1. Download and install Android Studio from `https://developer.android.com/studio/`.
2. Start Android Studio and follow the Setup Wizard, which installs all of the Android SDKs required by Flutter. If it asks you to import previous settings, you can click Yes to use the current settings or No to start with the default settings.

Set Up the Android Emulator

You can view details on how to create and manage virtual devices at `https://developer.android.com/studio/run/managing-avds`.

1. Enable VM acceleration on your machine. Directions are available at `https://developer.android.com/studio/run/emulator-acceleration`.
2. If this is your first time installing Android Studio, to access the AVD Manager, you need to create a new project. Launch Android Studio, click Start A New Android Studio Project, and give it any name and accept the defaults. Once the project is created, continue with these steps.

Currently, Android Studio requires an open Android project before you can access the Android submenu.

3. From Android Studio, select Tools ➪ Android ➪ AVD Manager ➪ Create Virtual Device.
4. Select a device of your choice and click Next.
5. Select an x86 or x86_64 image for the Android version to emulate.
6. If available, make sure you select Hardware – GLES 2.0 to enable hardware acceleration. This hardware acceleration will make the Android emulator run faster.
7. Click the Finish button.
8. To check that the emulator image has been installed correctly, select Tools ➪ AVD Manager and then click the Run button (play icon).

Installing on Windows

Before starting the installation, you need to make sure your Windows supports the minimum hardware and software requirements.

System Requirements

- Windows 7 SP1 or later (64-bit)
- 400 MB of disk space (does not include disk space for an integrated development environment and other tools)
- The following command-line tools:
  - PowerShell 5.0 or newer
  - Git for Windows (`Git` from the Windows Command Prompt)

Get the Flutter SDK

The latest installation details are available at `https://flutter.dev/docs/get-started/install/windows/`.

1. Download the following installation file to get the latest release, v1.7.8 at the time of writing (your version might be higher), of the Flutter SDK:

   `https://storage.googleapis.com/flutter_infra/releases/stable/windows/flutter_windows_v1.7.8+hotfix.4-stable.zip`.

2. Extract the file in the desired location.

   Replace `WindowsUserName` with your own Windows username. Do not install Flutter in directories that require elevated privileges like `C:\Program Files\`. I used the following folder location:

   `C:\Users\WindowsUserName\flutter`

3. Look in `C:\Users\WindowsUserName\flutter` and double-click the `flutter_console.bat` file.

4. Update the path permanently (the following is for Windows 10):

   a. Open Control Panel and drill down to Desktop App ➪ User Accounts ➪ User Accounts ➪ Change My Environment Variables.

   b. Under User Variables for `WindowsUserName`, select the `Path` variable and click the Edit button. If the `Path` variable is missing, skip to the next substep, c.

      i. In the Edit environment variable, click the New button.

      ii. Type the path **`C:\Users\WindowsUserName\flutter\bin`**.

      iii. Click the OK button and then close the Environment Variables screen.

   c. Under User Variables for `WindowsUserName`, if the `Path` variable is missing, then click the New button instead and type **`Path`** for the Variable name and **`C:\Users\WindowsUserName\flutter\bin`** for the Variable value.

5. Reboot Windows to apply the changes.

Check for Dependencies

Run the following command from the Windows command prompt to check for dependencies that need to be installed to finish the setup:

```
flutter doctor
```

View the report and check for other software that may need to be installed.

Install Android Studio

Full installation details are available at `https://flutter.dev/docs/get-started/install/windows#android-setup`. Flutter requires a full installation of Android Studio for the Android platform dependencies. Keep in mind that Flutter apps can be written in different editors like Visual Code or IntelliJ IDEA.

1. Download and install Android Studio from `https://developer.android.com/studio/`.
2. Start Android Studio and follow the Setup Wizard, which installs all of the Android SDKs required by Flutter. If it asks you to import previous settings, you can click Yes to use the current settings or No to start with the default settings.

Set Up the Android Emulator

You can view details on how to create and manage virtual devices at `https://developer.android.com/studio/run/managing-avds`.

1. Enable VM acceleration on your machine. Directions are available at `https://developer.android.com/studio/run/emulator-acceleration`.
2. If this is your first time installing Android Studio, to access the AVD Manager, you need to create a new project. Launch Android Studio and click Start A New Android Studio project and give it any name and accept the defaults. Once the project is created, continue with these steps.

   Currently, Android Studio requires an open Android project before you can access the Android submenu.
3. From Android Studio, select Tools ➪ Android ➪ AVD Manager ➪ Create Virtual Device.
4. Select a device of your choice and click Next.
5. Select an x86 or x86_64 image for the Android version to emulate.
6. If available, make sure you select Hardware – GLES 2.0 to enable hardware acceleration. This hardware acceleration will make the Android emulator run faster.
7. Click the Finish button.
8. To check that the emulator image has been installed correctly, select Tools ➪ AVD Manager and then click the Run button (play icon).

Installing on Linux

Before starting the installation, you need to make sure your Linux supports the minimum hardware and software requirements.

System Requirements

- Linux (64-bit)
- 600 MB of disk space (does not include disk space for an integrated development environment and other tools)
- The following command-line tools:
  - `Bash`
  - `curl`
  - `git 2.x`

- `mkdir`
- `rm`
- `unzip`
- `which`
- `xz-utils`

- The libGLU.so.1 shared library provided by mesa packages (e.g., libglu1-mesa on Ubuntu/Debian)

Get the Flutter SDK

The latest installation details are available at `https://flutter.dev/docs/get-started/install/linux/`.

1. Download the following installation file to get the latest release, which is v1.7.8 at the time of writing (your version might be higher), of the Flutter SDK:

   `https://storage.googleapis.com/flutter_infra/releases/stable/linux/flutter_linux_v1.7.8+hotfix.4-stable.tar.xz`.

2. Extract the file in the desired location by using the Terminal window:

   ```
   cd ~/development
   tar xf ~/Downloads/flutter_linux_v1.7.8+hotfix.4-stable.tar.xz
   ```

3. Add the Flutter tool to your path (`pwd` means present working directory; in your case, it will be the `development` folder).

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

4. Update the path permanently.

   a. Get the path that you used in step 3. For example, replace `PathToDev` with your path to the development folder.

      ```
      /PathToDev/development/flutter/bin
      ```

   b. Open or create `$HOME/.bash_profile`. Your file path or filename might be different on your computer.

   c. Edit `.bash_profile` (will open a command-line editor).

   d. Add the following line and make sure you replace the `PathToDev` with your path: **`export PATH="$PATH :/PathToDev/development/flutter/bin"`** .

   e. Run **`source $HOME/.bash_profile`** to refresh the current window.

   f. In the Terminal window, type **`echo $PATH`** to verify that the path is added. Then type **`flutter`** to make sure the path is working. If the command is not recognized, something went wrong with the `PATH`. Check to make sure you have the correct path.

Check for Dependencies

Run the following command from the Terminal window to check for dependencies that need to be installed to finish the setup:

```
flutter doctor
```

View the report and check for other software that may be needed.

Install Android Studio

Full installation details are available at `https://flutter.dev/docs/get-started/install/linux#android-setup`. Flutter requires a full installation of Android Studio for the Android platform dependencies. Keep in mind that Flutter apps can be written in different editors like Visual Code or IntelliJ IDEA.

1. Download and install Android Studio from `https://developer.android.com/studio/`.
2. Start Android Studio and follow the Setup Wizard, which installs all of the Android SDKs required by Flutter. If it asks you to import previous settings, you can click Yes to use current settings or No to start with default settings.

Set Up the Android Emulator

You can view details on how to create and manage virtual devices at `https://developer.android.com/studio/run/managing-avds`.

1. Enable VM acceleration on your machine. Directions are available at `https://developer.android.com/studio/run/emulator-acceleration`.
2. If this is your first time installing Android Studio, to access the AVD Manager, you need to create a new project. Launch Android Studio, click Start A New Android Studio Project, and give it any name and accept the defaults. Once the project is created, continue with these steps.

   Currently, Android Studio requires an open Android project before you can access the Android submenu.
3. From Android Studio, select Tools ➪ Android ➪ AVD Manager ➪ Create Virtual Device.
4. Select a device of your choice and click Next.
5. Select an x86 or x86_64 image for the Android version to emulate.
6. If available, make sure you select Hardware – GLES 2.0 to enable hardware acceleration. This hardware acceleration will make the Android emulator run faster.
7. Click the Finish button.
8. To check that the emulator image has been installed correctly, select Tools ➪ AVD Manager and then click the Run button (play icon).

CONFIGURING THE ANDROID STUDIO EDITOR

The editor you'll be using is Android Studio. Android Studio is the official integrated development environment for Google's Android operating system and is specifically designed for Android development. It's also a great development environment for developing apps with Flutter. Before starting to build an app, the editor needs the Flutter and Dart plugins installed to make it easier to write code. (Other editors that support these plugins are IntelliJ or Visual Studio Code.) The editor plugins give code completion, syntax highlighting, run and debug support, and more. Using a plain-text editor to write your code without any plugins would also work, but without the use of plugin features is not recommended.

Instructions for setting up different code editors are available at `https://flutter.dev/docs/get-started/editor/`. To support Flutter development, install the following plugins:

- Flutter plugin for developer workflows such as running, debugging, and hot reload
- Dart plugin for code analysis such as instant code validation and code completion

Follow these steps to install the Flutter and Dart plugins:

1. Start Android Studio.
2. Click Preferences ➪ Plugins (on macOS) or File ➪ Settings ➪ Plugins (on Windows and Linux).
3. Click Browse Repositories, select the Flutter plug-in, and click the Install button.
4. Click Yes when prompted to install the Dart plugin.
5. Click Restart when prompted.

SUMMARY

In this chapter, you learned how the Flutter framework architecture works behind the scenes. You learned that Flutter is a great portable UI framework to build mobile applications for iOS and Android. Flutter also plans on supporting development for desktop, web, and embedded devices. You learned that Flutter applications are built from a single codebase that uses widgets to create the UI and that you develop with the Dart language. You learned that Flutter uses the Skia 2D rendering engine that works with different types of hardware and software.

You learned that the Dart language is AOT compiled to native code, resulting in fast performance for your applications. You learned that Dart is JIT compiled, making it fast to display code changes with Flutter's stateful hot reload.

You learned that widgets are the building blocks for composing the UI, and each widget is an immutable declaration of the UI. Widgets are the configuration to create elements. Elements are the widgets made concrete, meaning mounted and painted on the screen. You learned that the `RenderObject` implements the basic layout and paint protocols.

You learned about the lifecycle events for the stateless and stateful widgets. You learned that the stateless widget is declared by a single class that extends (inherits) from the `StatelessWidget` class. You learned that the stateful widget is declared with two classes, the `StatefulWidget` class and the `State` class.

You learned that Flutter is declarative and the UI rebuilds itself when the state changes. You learned that widgets are the building blocks of a Flutter app and that widgets are the configurations for the UI.

You learned that nesting (*composition*) widgets results in creating the widget tree. The Flutter framework uses the widget as the configuration to build each element, resulting in creating the element tree. The element is the widget that is mounted (rendered) on the screen. The previous process results in creating the render tree that is a low-level layout and painting system. You'll be using widgets and will not need to interact directly with the render tree. You learned that stateless widgets have the configuration to create stateless elements. You learned that stateful widgets have the configuration to create stateful elements and the stateful element requests from the widget to create a state object.

You learned how to install the Flutter SDK, Xcode for compiling for iOS, and Android Studio for compiling for Android devices. When Android Studio is installed on a Mac, it handles compiling for both iOS (via Xcode) and Android devices. You learned how to install the Flutter and Dart plugins to help developer workflows such as code completion, syntax highlighting, run, debug, hot reload, and code analysis.

In the next chapter, you'll learn how to create your first app and use hot reload to view changes immediately. You'll also learn how to use themes to style the app, when to use stateless or stateful widgets, and how to use external packages to add features such as GPS and charts quickly.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
Flutter	Flutter is a portable UI framework to build modern, native, and reactive mobile applications for iOS and Android from a single codebase. Flutter has also expanded development for desktop, web, and embedded devices.
Skia	Flutter uses the Skia 2D rendering engine that works with different hardware and software platforms.
Dart	Dart is the language used to develop Flutter applications. Dart is ahead-of-time (AOT) compiled to native code for fast performance. Dart is just-in-time (JIT) compiled to quickly display code changes via Flutter's stateful hot reload.
Declarative user interface (UI)	Flutter is declarative and builds the UI to reflect the state of the app. When the state changes, the UI is redrawn. Flutter uses Dart to create the UI.
`Widget`	Widgets are the building blocks of a Flutter app, and each widget is an immutable declaration of the user interface (UI). Widgets are the configuration to create an element.
`Element`	Elements are widgets that are mounted (rendered) on the screen. Elements are created by the widget's configuration.
`RenderObject`	The `RenderObject` is an object in the render tree that computes and implements the basic layout and paint protocols.
Widgets lifecycle events	Each stateless or stateful widget has a build method with a `BuildContext` that handles the location of the widget in the widget tree. The `BuildContext` objects are `Element` objects, meaning an instantiation of the `Widget` at a location in the tree.
`StatelessWidget`	The `StatelessWidget` is built based on its own configuration and does not change dynamically. The stateless widget is declared with one class.
`StatefulWidget`	The `StatefulWidget` is built based on its own configuration but can change dynamically. The stateful widget is declared with two classes, the `StatefulWidget` class and the `State` class.
Widget tree	The widget tree is created when you compose (nest) widgets; this is known as *composition*. Three trees are created: the widget tree, the element tree, and the render tree.
Element tree	The element tree represents each element mounted (rendered) on the screen.

TOPIC	KEY CONCEPTS
Render tree	The render tree is a low-level layout and painting system that inherits from the `RenderObject`. The `RenderObject` implements the basic layout and paint protocols. You'll be using widgets and will not need to interact directly with the render tree.
Flutter SDK and Dart	The mobile software development kit has expanded to desktop, web, and embedded devices.
Xcode and Android Studio	Development tools to build iOS and Android Mobile applications.
Flutter plugin	This plugin helps with developer workflows such as running, debugging, and hot reload.
Dart plugin	This plugin helps with code analysis such as instance code validation and code completion.