Beginning App Development with Flutter

Rap Payne

Apress®

Beginning App Development with Flutter

Create Cross-Platform Mobile Apps

Rap Payne

Beginning App Development with Flutter: Create Cross-Platform Mobile Apps

Rap Payne
Dallas, TX, USA

ISBN-13 (pbk): 978-1-4842-5180-5 ISBN-13 (electronic): 978-1-4842-5181-2

https://doi.org/10.1007/978-1-4842-5181-2

Copyright © 2019 by Rap Payne

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress. com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-5180-5. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to the men and women of the Flutter Community. I've never seen a group more devoted to the success of others. You're an inspiration and example to me.

Particular thanks to these members of the community who've helped me with Flutter

issues. This Texan owes y'all!

Andrew "Red" Brogdon (Columbus, Ohio),

Brian Egan (Montana),

Emily Fortuna (San Francisco),

Frederik Schwieger (Düsseldorf, Germany), Jeroen "Jay" Meijer (Rotterdam, Netherlands), Martin Rybak (New York), Martin Jeret (Estonia), Nash Ramdial (Trinidad), Nilay Yenner (San Francisco), Norbert Kozsir (Karlsruhe, Germany), Pooja Bhaumik (Bengaluru, India), Raouf Rahiche (Casablanca by way of Algeria), Remi Rousselet (Paris), Rohan Tanaja (Berlin), Scott Stoll (Cleveland, Ohio),

But especially Simon Lightfoot (London), who we all call "The Flutter Whisperer" He taught me much of what I know about Flutter.

Praise for Beginning App Development with Flutter

"Rap has written a great starting guide full of information for those who are new to developing multi-platform apps with Flutter."

—Frederik Schwieger (Düsseldorf, Germany), Organizer of the International Flutter Hackathon and creator of flutter school

"A great read! This covers everything a beginner might want to know, and more. It explains not only what Flutter is but why it exists works the way it does. It also provides great tips for common pitfalls along the way. Definitely recommended."

—Jeroen "Jay" Meijer (Rotterdam, Netherlands), Leader of Flutter Community Github

“Rap’s book is a great book to get started with Flutter. It covers every important topic to write your very first app but also contains valuable information for more seasoned developers.”

—Norbert Kozsir (Karlsruhe, Germany)
Flutter Community Editor

“As a non-native English speaker, I’m totally impressed by the simplicity of this book and how much I can read and understand without getting bored.”

—Raouf Rahiche (Algeria) Flutter speaker, developer, and instructor

“As an early adopter and one of the original members of the Flutter Community, Rap is one of the world’s foremost authorities on Flutter. Where documentation is written for Engineers, by Engineers, Rap is a human who (thankfully!) writes in an enjoyable style that can easily be understood by other humans.”

—Scott Stoll (Cleveland, Ohio), Contributor to the Flutter codebase and Co-founder of the Flutter Study Group

Table of Contents

About the Author
xvii About the Technical Reviewer

xi

x Who is this book for?

xxi

Part I: Introduction to Flutter

1 Chapter 1: Hello Flutter

3 What is Flutter?

4 Why Flutter?

5 The other options

5 Native solutions

7 Conclusion

8

Chapter 2: Developing in Flutter

9 The Flutter toolchain

10 The Flutter SDK

10 IDEs

10 IDE DevTools

12 Emulators

13 Keeping the tools up to date

15 The Flutter development process

Part II: Foundational Flutter

29 Chapter 3: Everything Is Widgets

31 UI as code

33 Built-in Flutter widgets

35 Value widgets

Chapter 4: Value Widgets 47
The Text widget 47
The Icon widget 48
The Image widget 49
Embedded images 50
Network images 51
Sizing an image

Input widgets
Text fields
55 Putting the form widgets together
65 Form widget
65

FormField
widget
67 One big Form example
71 Conclusion

76

Chapter 5: Responding to Gestures

77 Meet the button family

78

RaisedButton

80 FlatButton and IconButton

81 FloatingActionButton

81

CupertinoButton

82

Dismissible

83 Custom gestures for your custom widgets

Step 1: Decide on your gestures and behaviors 83
Step 2: Create your custom widget 84
Step 3: Add a GestureDetector widget 85
Step 4: Associate your gesture with its behavior 86
Example 1: Reacting to a long press 87
Example 2: Pinching to add a new item 87
Example 3: Swiping left or right 89
What if there are two or more gestures happening at the same time? 90
Conclusion 92

Chapter 6: Laying Out Your Widgets 92
Laying out the whole scene 93

100 MaterialApp widget

100 The Scaffold widget

101 The AppBar widget

102 SafeArea widget

104 SnackBar widget

105

How Flutter decides on a widget's size

106 The dreaded "unbounded height" error

107 Flutter's layout algorithm

108

Putting widgets next to or below others
110
Your widgets will never fit!
113 What if there's extra space left over?
113
mainAxisAlignment
113
crossAxisAlignment
115 Expanded
widget
117
What if there's not enough space?
121 The ListView widget
121 Container
widget and the box model

140 Navigating forward and back
141 Get result after a scene is closed
143 Drawer navigation
144 The Drawer widget
146 Filling the drawer
148

Tab Navigation
150 TabController
151

TabBarView
151 TabBar and TabBarView
152 TabBar at the bottom
153 The Dialog widget
153 showDialog() and AlertDialog
154 Responses with a Dialog
155 Navigation methods can be combined
157

Chapter 8: Styling Your Widgets
159

Thinking in Flutter Styles
160 A word about

colors
1
61 Styling
Text
163
TextStyle
163 Custom fonts
165 Container
decorations
168
Border
170
BorderRadius
172 BoxShape

173 Stacking widgets
176 Positioned widget
178 Card widget
180 Themes
181 Applying theme properties
183 Conclusion
186

Chapter 9: Managing State

InheritedWidget

200 BLoC


200
ScopedModel


201
Hooks

201 Provider


202 Redux


202 Whoa! That’s a lot of packages!

203
Conclusion

Part III: Above and Beyond 203

Chapter 10: Your Flutter App Can Work with Files 205

Including libraries in your Flutter app 207

Finding a library 208

Adding it to pubspec.yaml 208

Importing the library 210

Using the library 210

Futures, async, and await 211

Why would it wait? 211

await

214 async

215 Including a file with your app

216 Writing a file

218 And reading it!

219 Using JSON

220 Writing your app's memory to JSON

Reading JSON into your app's memory 222
Shared preferences 223
To write preferences 224
To read preferences 224
Conclusion 225

Chapter 11: Making RESTful API Calls with HTTP 227
What is an *API* call? 228
The flavors of API requests 228
Making an HTTP GET or

DELETE request
230
Making an HTTP PUT, POST, or PATCH request
231 HTTP responses to widgets
232
Brute force – The easy way
233 FutureBuilder – The clean way
234 Strongly typed classes
238
Create a business class
238 Write a .fromJSON() method
239 Use .fromJSON() to hydrate the object
240
One big example

Making a strongly typed business class

PeopleList.dart

244 A GET request in Flutter

247 A DELETE request in Flutter

247

PeopleUpsert.dart

248 A POST and PUT request in Flutter

252 Conclusion
254

Chapter 12: Using Firebase with Flutter
255 Introducing Firebase
256 Cloud Firestore
257 Cloud Functions
25 8 Authentication
259 Setting up Firebase itself
259 (1) Creating a Firebase project
260 (2) Creating the database

Appendix A: Dart Language Overview
287 What is Dart?
287 Expected features – Dart Cheatsheet
288
Data types
288 Arrays/lists
289 Conditional expressions
289
Looping

290

Classes

290 Class constructors

2

91 Unexpected things about Dart

291 Type inference

2

92 final and const

292 Variables are initialized to null

293 String interpolation with $

294 Multiline strings

294 Spread operator

Named constructors

About the Author

Rap Payne has focused on mobile development since he started Agile Gadgets, a mobile app development company, in 2003. He is a consultant, trainer, and entrepreneur who has written apps, mentored developers, and taught software development classes for Fortune 500 companies like Boeing, Walmart, Coca-Cola, Wells Fargo, Honda, CVS, GE, Chase, HP, Lockheed, ExxonMobil, Lowe's, Nike, J.C. Penney, USAA, and Walgreens; government agencies like the NSA, the US Air Force, Navy, Army, NASA, Britain's GCHQ, and Canada's postal service; and several provincial governments, to name a few.

As a professional mentor and trainer, Rap has developed a talent for communicating highly complex ideas in easy-to-understand ways. And as a real-world developer, he understands the need to teach these topics using practical and realistic examples and exercises.

About the Technical Reviewer

Massimo Nardone has more than 22 years of experience in Security, Web/Mobile development, Cloud, and IT Architecture. His true IT passions are Security and Android.

He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

He holds a Master of Science in Computing Science from the University of Salerno, Italy.

He has worked as a Project Manager, Software Engineer, Research Engineer, Chief Security Architect, Information Security Manager, PCI/ SCADA Auditor, and Senior Lead IT Security/Cloud/SCADA Architect for many years.

His technical skills include Security, Android, Cloud, Java, MySQL, Drupal, Cobol, Perl, Web/Mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, django CMS, Jekyll, Scratch, and so on.

He works as Chief Information Security Officer (CISO) for Cargotec Oyj. He worked as visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto

University). He holds four international patents (PKI, SIP, SAML, and Proxy areas).

Who is this book for?

If you're a developer with experience in some object-oriented language like Java, C#, C++, or Objective-C and you want to create Android apps, iOS apps, or web apps with Flutter, this book is for you. It is especially important for you if you want to create an app that runs on multiple platforms and if you are new to Flutter.

If you've got some experience already with Flutter, you'll undoubtedly learn something, but we're not expecting that you have any prerequisite knowledge or experience with Flutter. All of our chapters are written with the assumption that everything in Flutter is completely new to you.

If you know anything about iOS development, Android development, or web development, that will certainly help with understanding the topics because there are lots of analogies in them for Flutter. The more you know about those things, the better, especially JavaScript and React. But if you know none of them, don't fret. They're by no means necessary.

Knowledge of the Dart language also will help. We've found that Dart has got its unique features for sure, but it is extremely easy to pick up if you understand object-oriented concepts. Heck, if you know Java or C#, most code snippets are understandable without any explanation of the language. Read a few and you'll be writing your own in no time.

At the same time, there are some unique but very cool Dart features that we consider best practices. We could have "simplified" the code for Java devs by not using these best practices, but in the long run that's not doing you any favors. Instead, we go ahead and use them,

but we do explain those things in “Appendix A: Dart Language Overview.” In there, we give you a cheat sheet with just enough detail to write code, followed

by a more in-depth explanation of the features that will be unexpected by developers of other languages. Pay special attention to the section called “Unexpected things about Dart.”

What is covered?

This book teaches you how to create fully functioning and feature-rich apps that run on iOS, Android, and the Web. We do this in three sections.

Part I: Introduction to Flutter

1. **Hello Flutter** – We’re setting the stage for the book. Giving you a feel for why you’re here. What problems does Flutter solve? Why the boss would choose Flutter vs. some other solution.
2. **Developing in Flutter** – Flutter has a unique set of tools, but it isn’t always straightforward what each tool does and how to use it. This chapter guides you through the process of write-debug-test-run. We get an understanding of the tooling including installation and maintenance.

Part II: Foundational Flutter

3. **Everything Is Widgets** – Widgets are super important to Flutter since they’re the building

blocks of every Flutter app. We show why and provide the motivation and basic tools to create widgets. Topics include composition, UI as code, widget types, keys, and stateless vs. stateful widgets.

4. **Value Widgets** – A deep dive into widgets that hold a value, especially user-input fields. Topics include the pubspec.yaml file; Text, Image, and Icon widgets; and how to create forms in Flutter.

5. **Responding to Gestures** – How to make your program do things in response to user actions like taps, swiping, pinching, and the like. We'll show you the button family and the GestureDetector widget.

6. **Laying Out Your Widgets** – We'll learn how to lay out a view, controlling how widgets are placed side by side and/or above and below, defining the amount of space between widgets, and aligning them vertically and horizontally.

7. **Navigation and Routing** – Navigation is making the app hide one widget and show another in response
to user actions. This makes them feel like they're moving from one scene to another. We'll cover stack navigation, tab navigation, and drawer navigation.

8. **Styling Your Widgets** – Then we'll look at how to control each widget's color, borders, decorations,
shapes, and other presentational characteristics. We handled light styling as we introduced each widget

earlier, but this is where we answer all the questions needed to get a real-world app looking good and staying consistent throughout with themes.

9. **Managing State** – How to get data from one widget to another and how to change that data. We cover how to create StatefulWidgets and design them in the best way. We also provide a high-level overview of tools to handle real-world complex state management.

Part III: Above and Beyond

10. **Your Flutter App Can Work with Files** – Using libraries. Futures, async, await. Bundling files with your app. Reading and writing a file. JSON serialization.
11. **Making RESTful API Calls with Ajax** – How to read from and write to an HTTP API server. This is where we show how to make GET, POST, PUT, DELETE, and PATCH requests.
12. **Using Firebase with Flutter** – We will show you a real-world, robust cloud solution that works like a dream with Flutter. No surprise that it is also a Google offering.

What is not covered and where can I find it?

As importantly, you should know what not to expect in the book. We will not give you a primer on the Dart programming language beyond the aforementioned appendix. We simply didn't think it was the best

use of your time and wanted to dive right into Flutter. If you feel you need a primer later on, go here: https://dart.dev/guides/language/language tour followed by https://dart.dev/tutorials. We chose not to discuss deploying to the app stores. The stores already do a fine job of explaining how to submit an app. That, and the process, changes so frequently that your definitive resource ought to be the stores themselves. You'll find

instructions at https://developer.apple.com/ios/submit/ and here: https://play.google.com/apps/publish. And we aren't going to cover certain advanced topics like device-specific development in iOS and Android or adding Flutter to an existing iOS/Android project. This is a beginner's book and we didn't want to overwhelm you. These and so many other topics can be found on the Web by searching and through some of the other resources we'll point you to in the last chapter of book.

PART I

Introduction to Flutter

CHAPTER 1

Hello Flutter

Picture this in your mind's eye. You are the superintelligent and capable CEO of a new business. Obviously your mission is to maximize sales while minimizing expenses. "Hmmm.", you think. "I can really increase sales if I make our products available on the Web." So you ask your friends how to create a web app and they say ...

"You need to hire a web developer. They should know HTML, CSS, JavaScript, and probably some framework like React, Vue, or Angular." It's expensive but you do it and your gamble pays off. Sales increase markedly. Trying to keep on top of demand, you monitor social media and engage your customers. You hear them say that this web app is great and all but "We'd have been here earlier if you had an app in the App Store." So you talk to your team who, while being experts in the Web, are not iOS developers. They tell you ...

"You need to hire an iOS expert. They should know iOS, Swift or Objective-C, Xcode, macOS, and CocoaPods for development." Your research shows that this person is even more specialized and therefore expensive than your web devs. But again, it seems to be the right thing to do, so you bite the bullet and hire them. But even while this app is being developed, you see that the feedback was not isolated to iOS apps, but instead was looking at all mobile devices. And – oh, snap! – 85% of devices worldwide run Android, not iOS. You bury your head in your hands as you ponder whether or not you can afford to ignore 85% of your potential customers. Your advisors tell you ...

© Rap Payne 2019

R. Payne, *Beginning App Development with Flutter*,
https://doi.org/10.1007/978-1-4842-5181-2_1

"You need to hire an Android expert. They should know the Android

OS, Gradle, Android SDK, XML, Android Studio, and Java or Kotlin." "Really?!? Another developer?", you say. "Yes. And one just as expensive as your iOS developer," they respond.

Isn't there one person who can do all three things? Some way to share the code between all of those environments? Then you could hire just one person. In fact, they could write the code one time and deploy it to the Web, to the App Store, and to the Google Play Store. One codebase to maintain. One place to make improvements and upgrades. One place to squash bugs. Ladies and gentlemen, allow me to introduce you to Flutter!

What is Flutter?

Flutter is a set of tooling that allows us to create beautiful apps that run on iOS, Android, the Web, and desktop.[1]

Flutter is ...

- Free (as in free beer. No cost)
- Open source (that's the other sense of the word "free") • Backed by and originated at Google
- Being enhanced and maintained by a team of developers at Google and hundreds of non-Google contributors around the globe
- Currently being used by thousands of developers in organizations across the world for production apps
- Fast because it compiles to truly native apps that don't use crutches like WebViews and JavaScript bridges

[1]Desktop is coming soon. Flutter will work on Windows, macOS, Chromebooks, and Linux.

- Written one place and compiled to a web app for

billions of browsers, an iOS app for iPhones and iPads, and an Android app for all of the rest of the phones and tablets out there

Why Flutter?

Google's mission with Flutter is ...

To build a better way to develop for mobile

Notice what is not in that mission. There's no mention of Android (which is also owned by Google) nor of iOS nor of the Web. Flutter's goal is to create a better way to develop for all devices. In other words, Flutter should be better to create iOS apps than Swift. It should be better to create Android apps than Kotlin. It should be better to create web apps than HTML/JavaScript. And if you get all of those things simultaneously with one codebase, all the better.

The Flutter team has succeeded spectacularly with this mission. As proof, Eric Seidel offers this example.[2] The Google CRM team used Flutter to build an internal Android app and did it **three times** faster than with their traditional Android toolchain!

But it turns out that Flutter isn't the only game in town for cross platform. You have other options.

The other options

Cross-platform development comes in three general flavors listed in Table 1-1.

[2] http://bit.ly/eric_seidel_flutter_keynote_video at 21:47 in.

Table 1-1. *Cross-platform development categories*

	Some technologies	Cons	Pros
Progressive Web Apps (PWA)	HTML/CSS, React, Angular, Vue	Not a real app. Runs in a web browser. Not available in app stores. Hard to create a desktop shortcut. Cannot access many of the device's resources like accelerometer,	compass, proximity sensor, Bluetooth, NFC, and more Easy to write
Hybrid	PhoneGap, Cordova, Sencha, Ionic	Runs in a WebView so it can be slow. Nearly impossible to share code with the web app	Easier for web devs to learn because it uses HTML and JavaScript as its language and structure
Compile-to-native solutions	React Native, NativeScript, Flutter, Xamarin	Learning a framework may be difficult. Mastering the toolchain definitely is	Real apps that can be found in the stores and run fast

If you have a captive audience, one where users value your app so much that they're willing to accept a poorer user experience, the cheapest solution is to create a PWA. If your app is extremely naive and speed is not expected to be an issue, a hybrid solution might be appropriate. But if speed, smoothness, and sophisticated capability are important, you will need to go with a native solution.

Native solutions

As of today, there are four fairly popular compile-to-native solutions (Table 1-2).

Table 1-2. *Compile-to-native cross-platform frameworks*

	 Xamarin	NativeScript	React Native	Flutter
Year introduced	2011	2014	2015	2018
Backed by	Microsoft	Telerik	Facebook	Google
Presentation language	XAML and/or xamarin.forms	Proprietary but looks like XML	Proprietary but looks like JSX	Dart
Procedural language	C#	JavaScript	JavaScript	Dart

These are all decent options. All are free to develop in and are well tested, having many production applications created. All have been used in large organizations.

But only one has an option to create a web application in addition to the iOS and Android apps that will be deployed to the app stores – Flutter. Flutter is the latest of these frameworks to be released. As such it has a distinct advantage of observing those that had come before. The Flutter team took note of what worked well with other frameworks and what failed. In addition, Flutter added new innovations and ideas – all baked in from the start rather than being bolted on as improvements are made.

But I suspect that if you've bought this book, you don't need much convincing so I'll stop. Suffice it to say that Flutter is amazing! It is easy to write, elegant, well-designed – an absolute pleasure to code in.[3]

Conclusion

Now, if you're the kind of developer I hope you are, you're chomping at the bit to get your hands dirty writing some code! So let's get to it. We'll start by installing and learning the Flutter development toolchain.

[3]But if you do want to read more, here's a deeper discussion of Flutter vs. some other frameworks: http://bit.ly/2HC9Khm

CHAPTER 2

Developing in Flutter

As we saw in the last chapter, Flutter enables us to create apps that run on the Web, on desktop computers, and on mobile devices (which seems to be the main draw). But wait a second, how exactly do we create these apps? What editor should we use? What is

needed in the Flutter project? How do you compile the Dart source code? Do we need any other tools to support the project? How do you get it into a browser or on a device in order to test it out? Good questions, right?

Let's answer those questions and more in this chapter. Let's cover two significant topics:

1. Tools needed – How to install and maintain them
2. The development process – How to create the app, run it, and debug it

> Caution By its nature, cross-platform app development tooling involves an awful lot of moving parts from various organizations, few of whom consult with the others before making changes. And since we're dealing with boundary-pushing and young technology, changes happen frequently. We've tried in this chapter to stick with timeless information but even it is likely to become stale eventually. Please check with the authors of these tools for the latest and greatest information.

© Rap Payne 2019
9 R. Payne, *Beginning App Development with Flutter*,
https://doi.org/10.1007/978-1-4842-5181-2_2

The Flutter toolchain

There is no end to the list of helpful tools that the development community has produced. It is truly overwhelming. We're making no attempt at covering them all. We want to give you just enough for you to be proficient but not so many that you're overburdened. Forgive me if I've skipped your favorite.

The Flutter SDK

The Flutter SDK is the only indispensable tool. It includes the Flutter compiler, project creator, device manager, test runner, and tools that diagnose – and even correct – problems with the Flutter configuration.

Installing the flutter SDK

The installation instructions are found here: https://flutter.dev/docs/get-started/install. Long story short – it will involve downloading the latest zip file of tools and setting your PATH to point to the folder where you unzipped them. The steps vary per operating system, but they're very plain on that web site.

> Tip This step seems very low level and sounds intimidating, but after this step, things get easier and less error-prone. Don't let it discourage you.

IDEs

In theory an IDE isn't really needed. Flutter can be written using any editor and then compiled and run using the flutter SDK that you installed earlier. But in reality almost nobody ever does that. Why would they? The following IDEs have Flutter support built right in!

VS Code from Microsoft

VS Code is from Microsoft. Its official name is "Microsoft Visual Studio Code," but most of us just call it *VS Code*. Whatever you call it, please do not confuse it with Microsoft's other product called "Microsoft Visual Studio." They are not the same thing regardless of the similar names. You can get VS Code here:

https://code.visualstudio.com.

Android Studio/IntelliJ from JetBrains

Android Studio and IntelliJ are essentially the same thing. They are built from the same codebase and have the same features.

You can get Android Studio at https://developer.android.com/studio and IntelliJ IDEA here: www.jetbrains.com/idea/download.

Which IDE should I use?

Both VS Code and Android Studio/IntelliJ are free and open source. Both run cross-platform on Windows, Mac, and Linux. Both are roughly equally popular with Flutter developers,[1] neither having a clear market advantage over the other. You can't go wrong with either one.

But if you must choose one, what we've found is that your background may affect how you like the tools. Developers from the web development world, those who use tech like HTML, CSS, JavaScript, NodeJS, React, Angular, or Vue, strongly prefer VS Code. On the other hand, those developers who came from a Java world, especially Android developers, seem to lean toward Android Studio/IntelliJ.

The good news is that this is a very low-pressure choice. It is trivial to switch editors – even while working on a given project. Start in one and see

[1]A recent poll of Flutter devs by Andrew Brogdon (@redbrogdon) of the Flutter team showed that 53% use VS Code, 30% use Android Studio, and 15% use IntelliJ. See http://bit.ly/flutter_devtools_poll

how you like it. If you don't, you can give the other a test drive for awhile. Go back and forth a couple of times until you have a strong preference. It's really no big deal to switch.

IDE DevTools

While those IDEs are great, they're not built for Flutter exclusively; they're used for developing in other languages and frameworks as well. So to improve the Flutter development flow, we should install the Flutter DevTools. It adds in debugger support, lets you look at logs, connects seamlessly with emulators, and a few more things.

Installing the DevTools is done from within each IDE. Within Android Studio/IntelliJ, go to "Preferences ➤ Plugins" from the main menu (Figure 2-1). In VS Code, go to "View ➤ Extensions" (Figure 2-2). The Flutter devtools are simply called "Flutter" and a search will turn them up. In either platform, hit the green "Install" button.

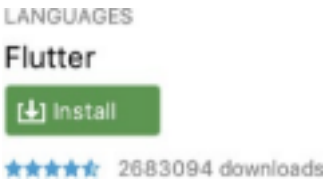


Figure 2-1. *DevTools install in Android Studio*

Figure 2-2. *DevTools install in VS Code*

You may need to restart the IDE after you install.

Emulators

Once you've got the IDE and DevTools installed, you're ready to compile your app. But to run it, you need to get it on a device. An emulator – a virtual device that runs on your laptop/desktop – makes it really easy to run, test, debug, and show your app. You'll probably want to test on both iOS and Android, so you'll need emulators for

each. There are several emulators available, but I'll mention just a couple, Xcode's iOS simulator and AVD's Android emulator.

iOS simulator

If you don't own a Mac, you won't be running an iOS emulator or even compiling for iOS for that matter.[2] But if you do and you have Xcode installed, you're in luck; you have the iOS simulator already. To run it, you open Xcode, then go to Xcode ➤ Open Developer Tool ➤ Simulator (Figure 2-3). The simulator will start up, and from within it, you can select any iOS device including iPhones and iPads.

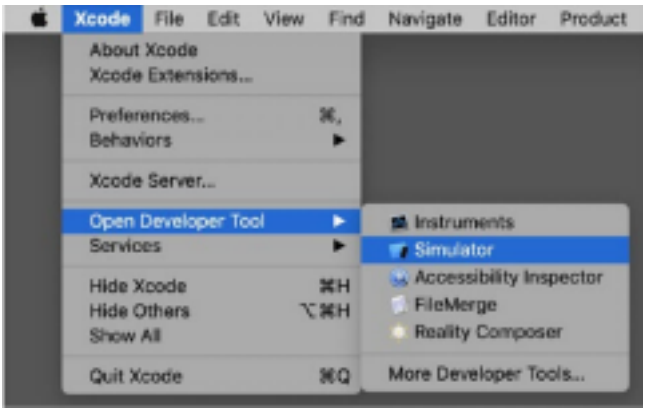


Figure 2-3. *Opening the iOS Simulator from Xcode*

[2]<sarcasm>Thanks, Apple.</sarcasm>

Android emulator

Just like there are tons of Android models, so are there tons of Android emulators, but there are only two popular ways to interact with them: Genymotion and AVD Manager. Genymotion is a for-profit company, so when you visit their web site, they'll do their level best to steer you toward their paid version. That's understandable. We'll focus on AVD Manager because it is totally free and more popular with Flutter devs.

AVD stands for "Android Virtual Device." The AVD Manager is found

in Android Studio under Tools (Figure 2-4).

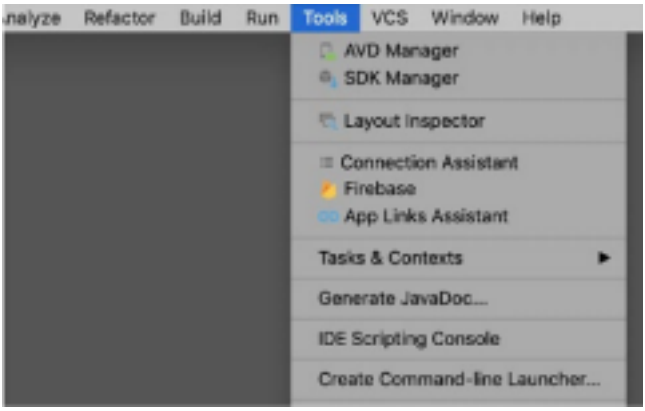


Figure 2-4. *Finding the AVD Manager in Android Studio*

Once opened, you'll see a list of your currently installed emulators. It should start out empty at first. You'll have the ability to install one or more of the hundreds of Android device emulators available by hitting the "+ Create Virtual Device..." button at the bottom (Figure 2-5).

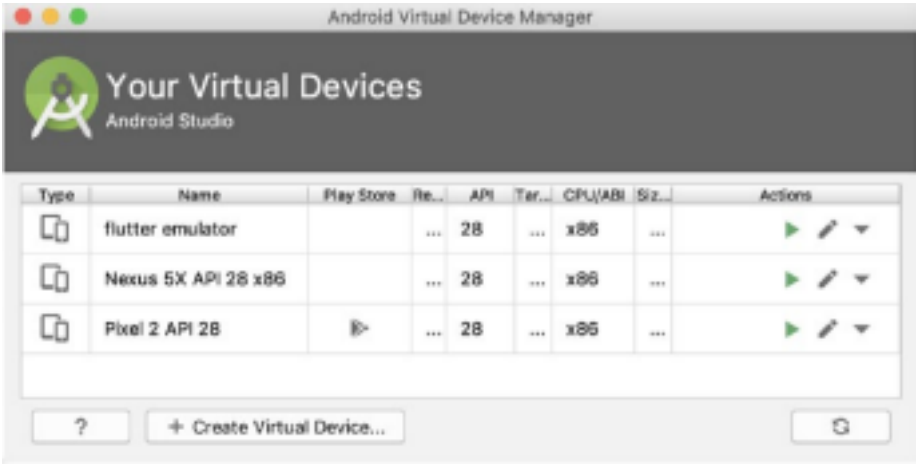


Figure 2-5. *AVD Manager has a list of available devices. Click "+" to add more*

Hit it and you can choose from all kinds of devices or create one of your own. You'll only need to install a device once. After it's installed, that emulated device is usable from any IDE, whether IntelliJ/Android Studio or VS Code. No need for a separate setup on VS Code.

Keeping the tools up to date

Early on, cross-platform development with tools like Xamarin and React Native was terribly difficult because of the sheer number of the tools involved and the interdependencies between them. I'm still in therapy from the pain.

But because Flutter arrived on the scene later it can learn from others' mistakes. The Flutter team, recognizing these pain points, gave us an innovative tool to manage the rest of the toolchain. It will examine your development machine, looking for all the tools you'll need to develop Flutter apps, the versions you have, the versions that are available, the interdependencies between them, and then make a diagnosis of problems. It will even prescribe a solution to those problems. Kind of sounds like a doctor, right? Well, let me introduce you to flutter doctor!

flutter doctor

You'll run flutter doctor from the command line. It checks all the tools in your toolchain and reports back any problems it encounters. Here's one where Xcode needed some help:

```
$ flutter doctor
Doctor summary (to see all details, run flutter doctor -v):

[✓] Flutter (Channel beta, vX.Y.Z, on Mac OS X X.Y.Z, locale
    en-US)
[✓] Android toolchain - develop for Android devices (SDK
    version X.Y.Z)
[!] Xcode - develop for iOS and macOS (Xcode X.Y) ✘ Xcode
requires additional components to be installed in order to run.
 Launch Xcode and install additional required components when
prompted.
[✓] Android Studio (version X.Y)
[✓] VS Code (version X.Y.Z)
[!] Connected device
 ! No devices available

 ! Doctor found issues in 2 categories.
$
```

The "No devices available" error is common, and you can usually ignore that one. It just means that at that moment no emulators were running. Here's an example of what we prefer to see – everything checks out:

```
$ flutter doctor
Doctor summary (to see all details, run flutter doctor -v):

[✓] Flutter (Channel beta, vX.Y.Z, on Mac OS X X.Y.Z, locale
    en-US)
```

[✓] Android toolchain - develop for Android devices (SDK version X.Y.Z)
[✓] Xcode - develop for iOS and macOS (Xcode X.Y)
[✓] Android Studio (version X.Y)
[✓] VS Code (version X.Y.Z)
[✓] Connected device (1 available)

• No issues found!

flutter doctor not only detects and reports problems but it usually prescribes the fix for each. It will even tell you when it is time to upgrade itself via “flutter upgrade.”

flutter upgrade

Yes, the initial installation of the Flutter SDK was a little daunting but the upgrade is a breeze. You’ll literally type two words, “flutter upgrade”:

```
$ flutter upgrade
Upgrading Flutter from /usr/local/bin/flutter...
From https://github.com/flutter/flutter
 2d2a1ff..a72edc2 beta -> origin/beta
 3932ffb..cc3ca9a dev -> origin/dev

 5a3a46a..a085635 master -> origin/master * [new
branch] refactor -> origin/refactor <snip>

* [new tag] v1.10.5 -> v1.10.5
Updating c382b8e..a72edc2
 11 files changed, 413 insertions(+), 302 deletions(-) Building
flutter tool...

Upgrading engine...
                        Downloading ios-deploy... 0.3s
```

```
Flutter X.Y.Z • channel beta • https://github.com/flutter/ flutter.git
Framework • revision a72e06 (23 hours ago) • 20XX-YY-ZZ
15:41:01 -0700
Engine • revision b863200c37
Tools • Dart X.Y.Z

Running flutter doctor...
Doctor summary (to see all details, run flutter doctor -v): [✓] Flutter
(Channel beta, vX.Y.Z, on Mac OS X X.Y.Z, locale  en-US)
[✓] Android toolchain - develop for Android devices (SDK
      version X.Y.Z)
[✓] Xcode - develop for iOS and macOS (Xcode X.Y)
[✓] Android Studio (version X.Y)
[✓] VS Code (version X.Y.Z)
[✓] Connected device (1 available)

• No issues found!
```

Note that flutter doctor is automatically run as the last step, confirming that all is well. Upgrading is a piece of cake.

The Flutter development process

Now that we have all the tools installed and up to date, let's create an app and run it through the debugger.

Scaffolding the app and files

Create a whole new Flutter app by running ...

```
$ flutter create my_app
```


This will create a subfolder under the current folder called my_app. It will be full of ready-to-run Dart code.

Tip The app name is case insensitive, so you should make it all lowercase. Dashes are illegal characters, so you can't use kebab casing. The recommended casing is lowercase_with_underscores.

Anatomy of a Flutter project

It's not critical that you know about all of the files and folders that are in the project you just created. But if you're curious, let's quickly walk through a newly created Flutter project shown in Figure 2-6.

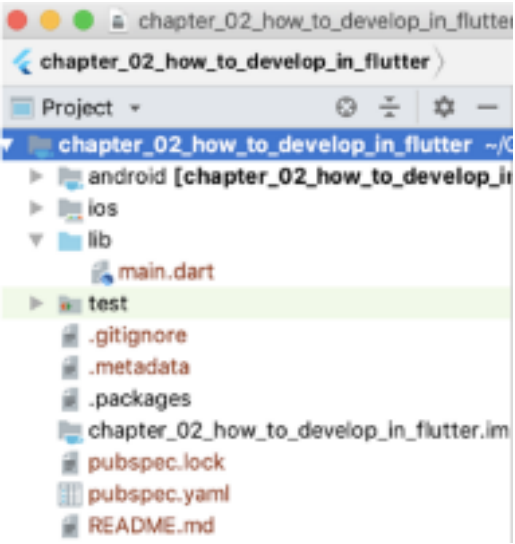


Figure 2-6. *A fresh Flutter project made by flutter create*

You'll have these folders:

- android and ios – These are the platform-specific parts

of the project. For the most part, you won't need to touch these.

- lib – This is the home of all of your Dart source code. You will build your app's hierarchy here. This is where you'll spend nearly all of your time and attention.
- test – If you have unit tests (and you probably should eventually), put them here.

And you'll have these files:

- pubspec.yaml – This is essentially the project file for Dart projects. This is where we set our project name, description, dependencies and more. Be sure to read the comments in here to get a better picture of what is suggested and possible.
- .gitignore and README.md – These will be very familiar to devs who use git and github for their source code repository. Others won't care.
- .metadata and .packages – These are important config files which you'll never open. But Flutter needs them.

Tip There's one more file you should be aware of: analysis-options. yaml. Having this file is not required, but if you do, you'll write better code. This file signals the IDE to perform linting (aka static analysis) on the code as you write it. With analysis-options.yaml, the IDE will warn you when you don't use best practices.

Rather than writing one from scratch, let me suggest that you start with someone else's. Here's one that is very popular with the Flutter

community: https://github.com/flutter/plugins/blob/master/analysis_options.yaml.

It is aggressive. If you want to turn off some of the rules, just delete the lines or comment them out. An explanation of all of the linting rules can be found here: http://dart-lang.github.io/ linter/lints/.

Running your app

You now have a Flutter app created. Let's go run it. There are multiple ways of running your app. The most popular way is to hit the green "Play" button in either Android Studio/IntelliJ or VS Code. You can also do it from the command line using "flutter run":

```
$ flutter run
Running "flutter pub get" in chapter_02_how_to_develop_in_
flutter... 0.5s Launching lib/main.dart on iPhone X in debug mode...
Running Xcode build...
    ├─Assembling Flutter resources... 6.1s └─Compiling, linking and
signing... 5.9s Xcode build done. 13.8s Syncing files to device iPhone
                                    X... 1,852ms

To hot reload changes while running, press "r". To hot restart
  (and rebuild state), press "R".
An Observatory debugger and profiler on iPhone X is available at:
http://127.0.0.1:52550/8m0h8zacV58=/
For a more detailed help message, press "h". To detach, press "d"; to
quit, press "q".
```

But if you hit the green Play/Debug button in your IDE (Figure 2-7), you'll have the option of debugging your app by setting breakpoints and stepping through the code using the developer tools (Figure 2-8).

Figure 2-7. *The Play and Debug buttons are at the top in Android Studio*

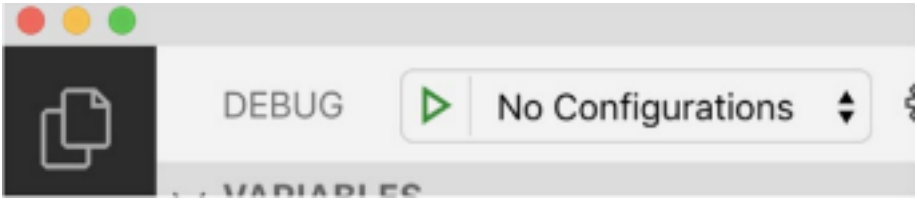


Figure 2-8. *The Play button is in the upper left in VS Code*

Obviously you'll need to run your app in a device of some kind. There are several: the Chrome browser for a web app, emulators, or a physical device that is tethered to your development machine via a cable. When you click the Play/Debug button, you get to choose which device you want to run at that moment. Notice that in the preceding screenshot of Android Studio, there's a dropdown menu with a list of available devices. In VS Code, hit the Play button, and a menu immediately pops up with your choices. With either IDE, you are in control.

Tip You can check what devices are currently available to you by running "flutter devices" from the command line.

```
$ flutter devices
3 connected devices:
```

Vivo XL3 • 55S...KF • android-arm64 • Android 8.0.0 (API 26)
Android SDK • emul...4 • android-x86 • Android 9 (API 28) (emulator)
iPhone X • E6...39A • ios • com. apple...OS-12-1 (simulator)

The preceding sample output tells us that we have three devices. The first and second are Android devices and the third runs iOS. The first device is a tethered physical device. The second and third are emulators.

Note that this command is different from the "flutter emulators" command which tells you all possible emulators you could potentially choose from. The flutter devices command tells you which devices are currently available to run your app.

Running it as a web app

Flutter considers your browser to be a device when you're running as a web app. So all that is needed to run as a web app is to enable the Google Chrome web browser as a device. You can enable it with this one-time command:

```
$ flutter config --enable-web
Setting "enable-web" value to "true".
```

From then on, when you get a list of devices on which to run your app, "Chrome" will appear as one of them. Simply choose to run your app in Chrome and the IDE will load your web app in it.

Running it on a tethered device

There are times when you need to run your app on a physical device. For example, I was developing a project that involved printing labels to a physical printer connected by Bluetooth. Emulators don't pair via Bluetooth. To test the printing, I needed an actual physical device that was already paired to my Bluetooth printer.

To tether a physical device to your development machine, you'll use a USB cable for most Android devices and a Lightning cable for most iPhones.

Tips #1 When connecting an Android device, it will initially think you're trying to charge it or transfer photos. To let it know you're trying to debug, open the Developer Options screen on the device and select "Enable USB debugging".

#2 Many connection issues can be caused by an inferior USB cable. Counterintuitively, not all USB cables are created equal. Switch to a higher-quality cable if you still can't connect after changing settings.

Hot reloading

Once the app is running in your emulator/browser/physical device/ whatever, you'll want to make changes to the source code and rerun. Here's the really cool thing: any time you save a change to the source code, it is recompiled and the new version is loaded instantly. Your app picks up where you left off – in the same spot, with the same state, and same data. We call it "hot reloading," and it makes the development cycle ridiculously fast and frictionless.

Debugging

Both IDEs have essentially the same debugging tools you've become accustomed to in all IDEs. When you start your project running, the debugging tools will appear.

In Android Studio the debug window opens, usually at the bottom of the IDE. It has a tiny toolbar which looks like Figure 2-9.

Figure 2-9. *The debugging toolbar in Android Studio*

The options are "step over," "step into," "force step into," and "step out" from left to right.

In VS Code the toolbar appears floating over your source code (Figure 2-10).

Figure 2-10. *The debugging toolbar in VS Code*

Its options are "play/pause," "step over," "step into," "step out," "hot reload," "restart," and "stop debugging."

Note Flutter is pickier when you're debugging than when running for real in a device. This is a good thing because during debugging it makes obvious certain errors that you should probably fix but aren't necessarily fatal. In the release version, it swallows those same errors and (hopefully) allows our users to continue running our app.

One family of those errors is "runtime assertions." You'll know you're dealing with one of these when the debugger gives you an error like this:

```
═══════ Exception caught by gesture ═══════

The following assertion was thrown while handling a gesture:
setState() callback argument returned a Future. The setState() method on _FooState#236 was called with a closure or method that returned a Future. Maybe it is marked as "async".
etc. etc. etc.
```

Your takeaway is this: when you see one of these, fix the problem. It's the right thing to do. But don't be confused if you don't see that same problem after you've deployed it.

Conclusion

Look, I know that this is a lot of stuff to absorb. The nature of cross platform development makes the tooling hairy. But the worst is behind us. Once you've got the Flutter SDK and an IDE (VS Code/Android Studio/IntelliJ IDEA) installed, that's all you really need. And granted, the DevTools and an emulator or two can really help. All that's left is getting some repetitions in for practice. You're going to be great!

So now that we've seen the Flutter toolchain, let's start creating widgets!

PART II

Foundational Flutter

CHAPTER 3

Everything Is Widgets

Let's pretend that you are an insanely talented Lego nerd and got offered one of the few coveted jobs as a Lego Master Builder. Congrats! Let's also say that your first assignment is to build a six-foot-tall Thor made from 26,000 Legos (Figure 3-1).

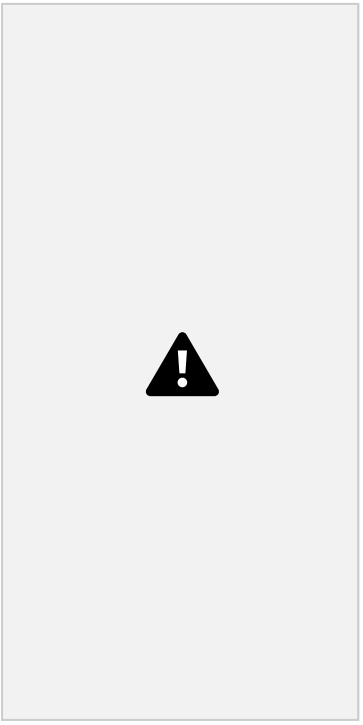


Figure 3-1. *A Lego Thor. The author snapped this picture at a movie theater once*

© Rap Payne 2019
31 R. Payne, *Beginning App Development with Flutter*,
https://doi.org/10.1007/978-1-4842-5181-2_3

How would you go about doing that? Ponder that for a minute. Go ahead, we'll wait.

Would you just start grabbing bricks and putting them together? Probably not. Would you lay out the soles of Thor's feet and build from the bottom up? Again, no. Here's my guess as to your common-sense strategy:

1. You'd get a vision of what you're building. Figure the whole thing out.
2. Realize that the entire project is too complex to build at once.
3. Break the project into sections (legs, left arm, right arm, torso, left sword, right sword, helmet, cape, head).
4. Realize that each of them is still too complex.
5. For each section, you break it into sub-sections.
6. Repeat steps 4 and 5 until you've got simple enough components that each is easy to understand, build, and maintain – for you and for any teammates that you may have.
7. Create each simple component.
8. Combine simple components to form the larger, more complex components.
9. Repeat steps 7 and 8 until you've got your entire project created.

This process has a name: *componentization*, and is exactly the thought process we'll go through with our Flutter projects.

Componentization is not something new. In fact, it was proposed as

far back as 1968.[1] But the technique has recently exploded in popularity thanks to web frameworks like Angular, React, Vue, Polymer, and native web components. Seems like all the cool kids are doing software components these days. The idea of recursively breaking down the complex bits into simpler bits is called *decomposition*. And the act of putting the written pieces back together into larger components is called *composition*.

In the world of Flutter, these components are referred to as *widgets*. Flutter people like to say "everything is widgets," meaning that you and I will be using the Google-provided widgets – the ones that ship with Flutter. We'll compose them together to create our own custom widgets. And our custom widgets will be composed together to create more and more complex custom widgets. This continues until you've got yourself a full-blown app.

> *In the world of Flutter, components are referred to as **widgets**.*

Every app can be thought of in two parts:

1. Behavior – What the software does. All of the business logic goes here: the data reading, writing, and processing.
2. Presentation – How the software looks. The user interface. The buttons, textboxes, labels.

Only Flutter combines these into one language instead of two.

UI as code

Other development frameworks have proven componentization to be the way to go. The Flutter team has openly stated that they were heavily

[1] http://bit.ly/componentHistory

inspired by React[2] which is based on componentization. In fact, all framework makers seem to borrow heavily from one another. But Flutter is unique in the way that the user interface is expressed. Developers use the same Dart language to express an app's graphical user interface as well as the behavior (Table 3-1). We call this "UI as code."

Table 3-1. *Only Flutter uses the same language for presentation and behavior*

Framework	Behavior expressed in ...	UI expressed in
...	Xamarin	C# XAML
React Native	JavaScript	JSX
NativeScript	JavaScript	XML
Flutter	Dart	Dart

So how does this UI get created? Like many other frameworks and languages, a flutter app starts with a *main* function. In Flutter, main will call a function called runApp(). This runApp() receives one widget, the root widget which can be named anything, but it should be a class that extends a Flutter StatelessWidget. It looks like this:

```
// import the Dart package needed for all Flutter apps import
'package:flutter/material.dart';

// Here is main calling runApp
void main() => runApp(RootWidget());

// And here is your root widget
class RootWidget extends StatelessWidget {
 @override
```

[2]Source: https://flutter.dev/docs/resources/faq#does-flutter-come-with a-framework

```
Widget build(BuildContext context) {
return Text("Hello world");
}
}
```

And that's all you need to create a "Hello world" in Flutter. But wait ... what is this Text() thing? It's a built-in Flutter widget. Since these built-in widgets are so important, we need to take a look at them.

Built-in Flutter widgets

Flutter's foundational widgets are the building blocks of everything we create and there are tons of them – about 160 at last count.[3] This is a lot of widgets for you and I to keep track. But if you mentally organize them, it becomes much more manageable.

They fall into these major categories:

- Value widgets
- Layout widgets
- Navigation widgets
- Other widgets

Note These are not Flutter's official list of categories. Their 14 categories are listed here: https://flutter.dev/docs/ development/ui/widgets. We just felt that reorganizing them helps to keep them straight.

[3]You can find a list of them all here: https://flutter.dev/docs/reference/widgets

We'll take a brief look at each of these categories with an example or two, and then we'll do some deep dives in later chapters. Let's start with value widgets.

Value widgets

Certain widgets hold a value, maybe values that came from local storage, a service on the Internet, or from the user themselves. These are used to display values to the user and to get values from the user into the app. The seminal example is the Text widget which displays a little bit of text. Another is the Image widget which displays a .jpg, .png, or another picture. Here are some more value widgets:

Checkbox
CircularProgressIndicator
Date & Time Pickers
DataTable
DropdownButton
FlatButton
FloatingActionButton
FlutterLogo
Form
FormField
Icon
IconButton
Image
LinearProgressIndicator
PopupMenuButton
Radio
RaisedButton
RawImage
RefreshIndicator
RichText
Slider
Switch
Text
TextField
Tooltip

We'll explore value widgets in more detail in the next chapter.

Layout widgets

Layout widgets give us tons of control in making our scene lay out properly – placing widgets side by side or above and beneath, making them scrollable, making them wrap, determining the space around widgets so they don't feel crowded, and so on:

Align
AppBar
FittedBox Flow
Chapter 3 Everything Is Widgets
Padding
PageView
AspectRatio
Baseline
BottomSheet
ButtonBar
Card
Center
Column
ConstrainedBox
Container
CustomMultiChildLayout Divider
Expanded
ExpansionPanel
FractionallySizedBox GridView
IndexedStack
IntrinsicHeight
IntrinsicWidth
LayoutBuilder
LimitedBox
ListBody
ListTile
ListView
MediaQuery
NestedScrollview OverflowBox
Placeholder
Row
Scaffold
Scrollable
Scrollbar
SingleChildScrollView SizedBox
SizedOverflowBox SliverAppBar
SnackBar
Stack
Table
Wrap

This is a huge topic which we've given its own chapter, Chapter 6, "Laying Out Your Widgets."

Navigation widgets

When your app has multiple scenes ("screens," "pages," whatever you want to call them), you'll need some way to move between them. That's where Navigation widgets come in. These will control how your user sees one scene and then moves to the next. Usually this is done when the user taps a button. And sometimes the navigation button is located on a tab bar or in a drawer that slides in from the left side of the screen. Here are some navigation widgets:

AlertDialog
BottomNavigationBar Drawer
MaterialApp
Navigator
SimpleDialog
TabBar
TabBarView

We'll learn how they work in Chapter 7, "Navigation and Routing."

Other widgets

And no, not all widgets fall into these neat categories. Let's lump the rest into a miscellaneous category. Here are some miscellaneous widgets:

GestureDetector	Dismissible	Transitions
	Cupertino Theme	Transforms

Many of these miscellaneous widgets are covered throughout the book where they fit naturally. GestureDetector is crucial enough that it gets its own chapter, Chapter 5, "Responding to Gestures."

How to create your own stateless widgets

So we know that we will be composing these built-in widgets to form our own custom widgets which will then be composed with other built-in widgets to eventually form an app.

Widgets are masterfully designed because each widget is easy to understand and therefore easy to maintain. Widgets are abstract from the outside while being logical and predictable on the inside. They are a dream to work with.

Every widget is a class that can have properties and methods. Every widget can have a constructor with zero or more parameters. And most importantly, every widget has a build method which receives a BuildContext[4] and returns a single Flutter widget. If you're ever wondering how a widget got to look the way it does, locate its build method:

[4]Don't get distracted by the BuildContext. It's used by the framework and we do occasionally refer to it, but we'll save those examples later in the book. For now, just think of it as part of the recipe to write a custom widget.

```
class RootWidget extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
 return Text('Hello world');
 }
}
```

In this hello world example which we repeated from earlier in the chapter, we're displaying a Text widget (Figure 3-2). A single inner widget works but real-world apps will be a whole lot more complex. The root widget could be composed of many other subwidgets:

```
class FancyHelloWidget extends StatelessWidget {
Widget build(BuildContext context) {
 return MaterialApp(
 home: Scaffold(
 appBar: AppBar(
 title: Text("A fancier app"),
 ),
 body: Container(
 alignment: Alignment.center,
 child: Text("Hello world"),
 ),
 floatingActionButton: FloatingActionButton(  child:
Icon(Icons.thumb_up),
 onPressed: () => {},
 ),
 ),
 );
 }
}
```

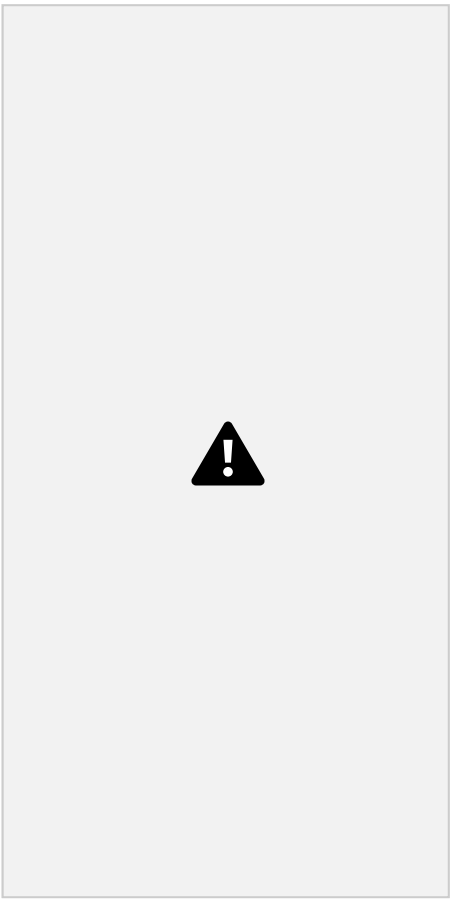


Figure 3-2. *The app created by this simple widget*

So as you can see, the build method is returning a single widget, a MaterialApp, but it contains a Scaffold which contains three subwidgets: an AppBar, a Container, and a FloatingActionButton (Figure 3-3). Each of those in turn contains sub-subwidgets of their own.

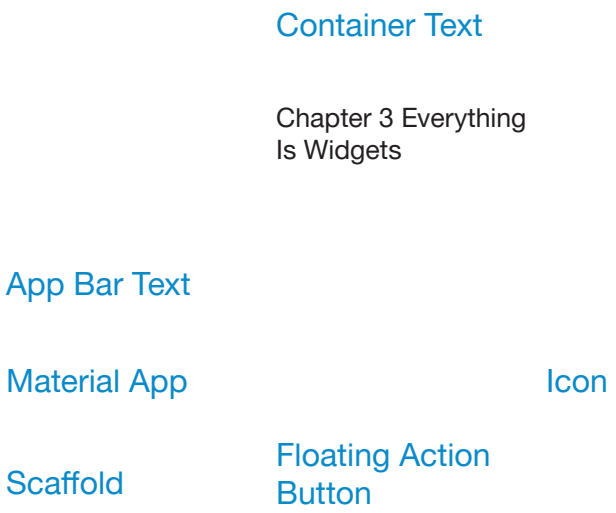


Figure 3-3. *The widget tree from our example app above*

This is how your build method will always work. It will return a single, massive, nested expression. It is widgets inside widgets inside widgets that enable you to create your own elaborate custom widget.

Widgets have keys

You may hear about a virtual DOM when other developers talk about Flutter. This comes from the world of React. (Remember that Flutter borrowed heavily from React's excellent architecture.) Well, strictly speaking, Flutter doesn't have a DOM, but it does maintain something resembling it – the element tree. The element tree is a tiny copy of all the widgets on the screen. Flutter maintains a current element tree and one with batched changes applied.

You see, Flutter might be really slow if it applied every tiny change

to the screen and then tried to re-render it hundreds of times per second. Instead, Flutter applies all of those changes to a copy of the element tree. It then periodically “diffs” the current element tree with the modified one and decides what truly needs to be re-rendered. It only re-renders those parts that need it. This is much, much faster.

But occasionally Flutter gets confused when matching the widgets in the element trees. You’ll know to programmatically assign keys if your data changes and widgets get drawn in the wrong location, the data isn’t updated on the screen, or your scroll position isn’t preserved.

You don’t need to worry about keys most of the time. It is needed so rarely that we’re going to be satisfied if you understand that ...

1. Keys exist and why Flutter may need them.
2. If your widgets aren’t being redrawn as you might expect when data changes, keys may solve problems.
3. You have the opportunity to assign keys to certain widgets.

If that’s not enough to satisfy you for now, the great Emily Fortuna has recorded a super ten-minute video on keys.[5]

Passing a value into your widget

Do you know what this formula means?

y = f(x)

Math majors will recognize this as reading “Y is a function of X.” It concisely communicates that as X (the independent variable) changes, Y (the dependent variable) will change in a predictable way. Flutter lives on this idea, but in Flutter the formula reads like this:

Scene = f(Data)

In other words, as the data in your app changes, the screen will change accordingly. And you, the developer, get to decide how that data is presented as you write a build method in your widgets. It is a foundational concept of Flutter.

[5]You can find Emily's video here: http://bit.ly/FlutterKeys

Now how might that data change? There's two ways:

1. The widget can be re-rendered with new data passed from outside.
2. Data can be maintained within certain widgets.

Let's talk about the first. To pass data into a widget, you'll send it in as a constructor parameter like this:

```
Widget build(BuildContext context) {
 return Person("Sarah"); // Passing "Sarah" into a widget }
```

If a widget represents how to render a Person, it would be a very normal thing to pass in a firstName, like we just did with "Sarah" earlier. If you do that, you'll need to write your widget's constructor to receive that value:

```
class Person extends StatelessWidget {
 final String firstName;
 Person(this.firstName) {}
 Widget build(BuildContext context) {
 return Text('$firstName');
 }
}
```

This is Dart syntax. Note three things. First, you'll list the input parameter in the constructor ("this.firstName" in the preceding example). Second, make sure you put "this." in front of it. The "this." matches it to a class-level property rather than a parameter that is local to the constructor function. And third, mark the corresponding

class property as final. You might want to pass in two or more properties like this:

```
Widget build(BuildContext context) {
 return Person("Sarah","Ali");
}
```


Of course passing in two values means creating two final variables and two constructor parameters to handle them:

```
class Person extends StatelessWidget {
 final String firstName;
 final String lastName;
 Person(this.firstName, this.lastName) {}
 Widget build(BuildContext context) {
 return Text('$firstName $lastName');
 }
}
```

As you can guess, these are matched positionally which can be easy to mess up and not terribly flexible. A better practice is to have named parameters:

```
Widget build(BuildContext context) {
 return Person(firstName:"Sarah", lastName:"Ali"); }
```

This reduces confusion for the other developers who use your widget. Here's how you'd write your widget to receive that value:

```
class Person extends StatelessWidget {
 final String firstName;
 final String lastName;
 Person({this.firstName, this.lastName}) {}
 Widget build(BuildContext context) {
 return Container(child: Text('$firstName $lastName'));  }
}
```

Do you see the difference? It's subtle. There are now curly braces around the constructor parameters. This makes them optional and named.

Tip Note that in all three of the preceding examples, we are using a Person class that might have been defined in the same dart file where you're using it. But a better practice is to create each class in a separate dart file and import it into other dart files where it is used.

```
import 'Person.dart';
```

Stateless and Stateful widgets

So far we've been going out of our way to create stateless widgets. So you probably guessed that there's also a stateful widget. You were right. A stateless widget is one that doesn't maintain its own state. A stateful widget does.

"State" in this context refers to data within the widget that can change during its lifetime. Think about our Person widget from earlier. If it's a widget that just displays the person's information, it should be stateless. But if it is a person **maintenance** widget where we allow the user to change the data by typing into a TextField, then we'd need a StatefulWidget.

There's a whole chapter on stateful widgets later. If you just can't wait to know more about them, you can read Chapter 9, "Managing State," later in this book. Then come back here.

So which one should I create?

The short answer is create a stateless widget. Never use a stateful

widget until you must. Assume all widgets you make will be stateless and start them out that way. Refactor them into stateful widgets when you're sure you really do need state. But recognize that state can be avoided more often than developers think. Avoid it when you can to make widgets simpler and therefore easier to write, to maintain, and to extend. Your team members will thank you for it.

Note There is actually a third type of widget, the InheritedWidget. You set a value in your InheritedWidget and any descendent can reach back up through the tree and ask for that data directly. It is kind of an advanced topic, but Rémi Rousselet would have had my head if I hadn't mentioned it. You can read more about it in Chapter 9, "Managing State," or watch Emily Fortuna's concise overview of InheritedWidget here: http://bit.ly/inheritedWidget.

Conclusion

So now we know that Flutter apps are all about widgets. You'll compose your own custom Stateless or Stateful widgets that have a build method which will render a tree of built-in Flutter widgets. So clearly we need to know about the built-in Flutter widgets which we'll learn beginning in the
next chapter.

CHAPTER 4

Value Widgets

We learned in the last chapter that *everything is a widget.* Everything you create is a widget and everything that Flutter provides us is a widget. Sure, there are exceptions to that, but it never hurts to think of it this way, especially as you're getting started in Flutter. In this chapter we're going to drill down into the most fundamental group of widgets that Flutter provides us – the ones that hold a value. We'll talk about the Text widget, the Icon widget, and the Image widget, all of which display exactly what their names imply. Then we'll dive into the input widgets – ones designed to get input from the user.

The Text widget

If you want to display a string to the screen, the Text widget is what you'll need. Text('Hello world'),

> Tip If your Text is a literal, put the word const in front of it and the widget will be created at compile time instead of runtime. Your apk/ ipa file will be slightly larger but they'll run faster on the device. Well worth it.

You have control over the Text's size, font, weight, color, and more with its style property. But we'll cover that in Chapter 8, "Styling Your Widgets."

© Rap Payne 2019

R. Payne, *Beginning App Development with Flutter*,
https://doi.org/10.1007/978-1-4842-5181-2_4

The Icon widget

Flutter comes with a rich set of built-in icons (Figure 4-1), from cameras to people to cards to vehicles to arrows to batteries to Android/iOS devices. A full list can be found here: https://api.flutter.dev/flutter/material/ Icons-class.html.

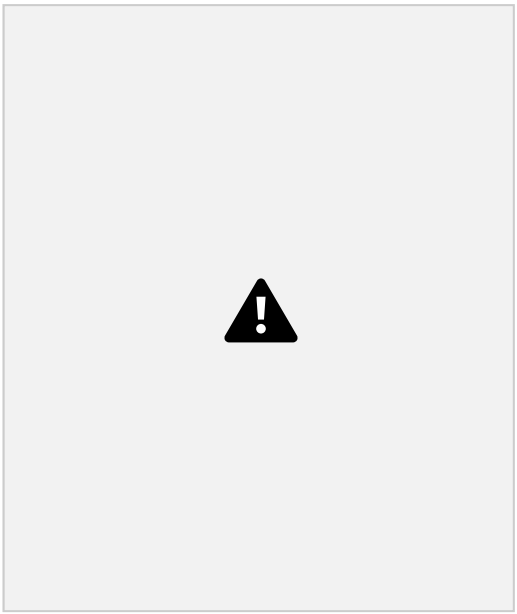


Figure 4-1. *An assortment of Flutter's built-in widgets in random colors*

To place an icon, you use the Icon widget. No surprise there. You use the Icons class to specify which one. This class has hundreds of static values like Icons.phone_android and Icons.phone_iphone and Icons.cake. Each points to a different icon like the ones pictured previously. Here's how you'd put a big red birthday cake (Figure 4-2) on your app:


```
Icon(
 Icons.cake,
 color: Colors.red,
 size: 200,
 )
```

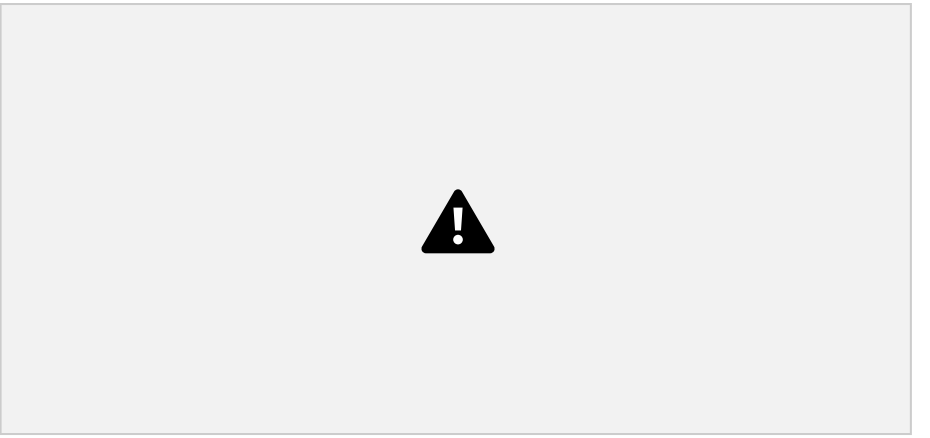


Figure 4-2. *The red cake icon*

The Image widget

Displaying images in Flutter is a bit more complex than Text or Icons. It involves a few things:

1. Getting the image source – This could be an image embedded in the app itself or fetched live from the Internet. If the image will never change through the life of your app like a logo or decorations, it should be an embedded image.
2. Sizing it – Scaling it up or down to the right size and shape.

Embedded images

Embedded images are much faster but will increase your app's install size. To embed the image, put the image file in your project folder, probably in a subfolder called images just to keep things straight. Something like assets/ images will do nicely.

Then edit pubspec.yaml. Add this to it:

```
flutter:
 assets:
 - assets/images/photo1.png
 - assets/images/photo2.jpg
```

Save the file and run "flutter pub get" from the command line to have your project process the file.

Tip The pubspec.yaml file holds all kinds of great information about your project. It holds project metadata like the name, description, repository location, and version number. It lists library dependencies and fonts. It is the go-to location for other developers new to your project. For any of you JavaScript developers, it is the package.json file of your Dart project.

Then you'll put the image in your custom widget by calling the asset() constructor like this:

```
Image.asset('assets/images/photo1.jpg',),
```


Network images

Network images are much more like what web developers might be accustomed to. It is simply fetching an image over the Internet via HTTP. You'll use the network constructor and pass in a URL as a string.

Image.network(imageUrl),

As you'd expect, these are slower than embedded images because there's a delay while the request is being sent to a server over the Internet and the image is being downloaded by your device. The advantage is that these images are live; any image can be loaded dynamically by simply changing the image URL.

Sizing an image

Images are nearly always put in a container. Not that this is a requirement, it's just that I can't imagine a real-world use case where it won't be inside another widget. The container has a say in the size that an image is drawn. It would be an amazing coincidence if the Image's natural size fit its container's size perfectly. Instead, Flutter's layout engine will shrink the image to fit its container, but not grow it. This fit is called *BoxFit. scaleDown,* and it makes sense for the default behavior. But what other options are available and how do we decide which to use? Table 4-1 provides your BoxFit options.

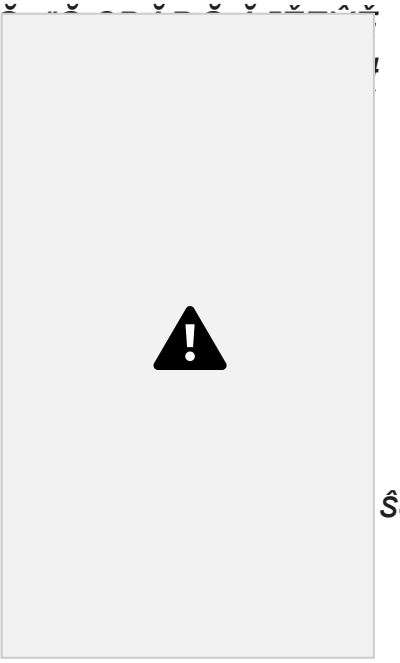
Table 4-1. *BoxFit options*

fill	Stretch it so that both the width and the height fit exactly. Distort the image
cover	Shrink or grow until the space is filled. The top/bottom or sides will be clipped
fitHeight	Make the height fit exactly. Clip the width or add extra space as needed
fitWidth	Make the width fit. Clip the height or add extra space as needed
contain	Shrink until both the height and the width fit. There will be extra space on the top/bottom or sides

Photo courtesy of Eye for Ebony on Unsplash

So those are your options, but how do you choose? Figure 4-3 may help you decide which fit to use in different situations.

Ž LJŽƵ ŶĞĞĚ łŚĞ ŝŵĂĞĞ łŽ Ĩŝůů łŚĞ ĞŶŧŝđĞ

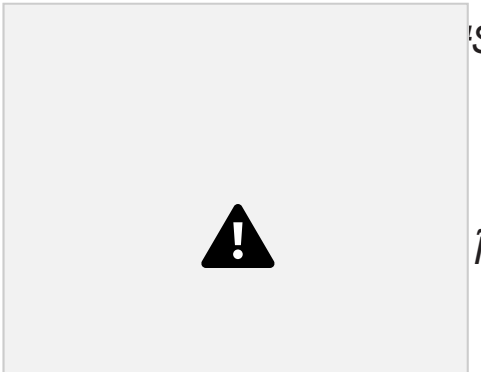


^ĐĂĐĞ ĂđŽƵŶŶĚ ŝł *ŝƐ*

ŽŬĂLJ &ŝůů łŚĞ

ĐŽŶĂŝŶĞđ &ŝůů Ăůů łŚĞ ‖ŝĚłŚ Žđ Ăůů łŚĞ

ŚĞŝĞŚĘ



łŚĞ ŚĞŝĞŚĘ

ĨŝŀĘ,ĞŝĘŚ ĨŝłtŝĚłŚ

Figure 4-3. *How to decide an image's fit*

To specify the fit, you'll set the *fit* property.

```
Image.asset('assets/images/woman.jpg',
 fit: BoxFit.contain,),
```

Input widgets

Many of us came from a web background where from the very beginning there were HTML <form>s with <input>s and <select>s. All of these exist to enable the user to get data into web apps, an activity we can't live without in mobile apps as well. Flutter provides widgets for entering data like we have in the Web, but they don't work the same way. They take much more work to create and use. Sorry about that. But they are also safer and give us much more control.

Part of the complication is that these widgets don't maintain their own state; you have to do it manually.

Another part of the complication is that input widgets are unaware of each other. In other words, they don't play well together until you group them with a Form widget. We eventually need to focus on the Form widget. But before we do, let's study how to create text fields, checkboxes, radio buttons, sliders, and dropdowns.

Caution Input widgets are really tough to work with unless they are used within a StatefulWidget because by nature, they change state. Remember that we mentioned StatefulWidgets briefly in the last chapter and we're going to talk about them in depth in Chapter 9, "Managing State." But until then, please just take our word for it and put them in a stateful widget for now.

Text fields

If all you have is a single textbox, you probably want a TextField widget. Here's a simple example of the TextField widget with a Text label above it:

```
const Text('Search terms'),
TextField(
 onChanged: (String val) => _searchTerm = val,
),
```

That onChanged property is an event handler that fires after every keystroke. It receives a single value – a String. This is the value that the user is typing. In the preceding example, we're setting a local variable

called _searchTerm to whatever the user types.

To provide an initial value with a TextField, you need the unnecessarily complex TextInputController:

```
TextEditingController _controller =
 TextEditingController(text: "Initial value here"); Then tell
```

your TextField about the controller

```
const Text('Search terms'),
TextField(
 controller: _controller,
 onChanged: (String val) => _searchTerm = val,
),
```

You can also use that _controller.text property to retrieve the value that the user is typing into the box.

Did you notice the Text('Search terms')? That is our lame attempt at putting a label above the TextField. There's a much, much better way. Check this out ...

Making your TextField fancy

There's a ton of options to make your TextField more useful – not infinite options, but lots. And they're all available through the InputDecoration widget (Figure 4-4):

```
return TextField(
 controller: _emailController,
 decoration: InputDecoration(
 labelText: 'Email',
 hintText: 'you@email.com',
 icon: Icon(Icons.contact_mail),
 ),
),
```

Figure 4-4. *A TextField with an InputDecoration*

Table 4-2 presents some more InputDecoration options.

Table 4-2. *Input decoration options*

Property	Description
labelText	Appears above the TextField. Tells the user what this TextField is for
hintText	Light ghost text inside the TextField. Disappears as the user begins typing
errorText	Error message that appears below the TextField. Usually in red. It is set automatically by validation (covered later), but you can set it manually if you need to
prefixText	Text in the TextField to the left of the stuff the user types in
suffixText	Same as prefixText but to the far right
icon	Draws an icon to the left of the entire TextField
prefixIcon	Draws one inside the TextField to the left
suffixIcon	Same as prefixIcon but to the far right

(*continued*)

Table 4-2. (*continued*) Property Description

Tip To make it a password box (Figure 4-5), set obscureText property to true. As the user types, each character appears for a second and is replaced by a dot.

```
return TextField(
 obscureText: true,
 decoration: InputDecoration(
 labelText: 'Password',
 ),
);
```

Figure 4-5. *A password box with obscureText*

Want a special soft keyboard? No problem. Just use the keyboardType property. Results are shown in Figures 4-6 through 4-9.

```
return TextField(
 keyboardType: TextInputType.number,
);
```

Figure 4-6. *TextInputType.datetime*

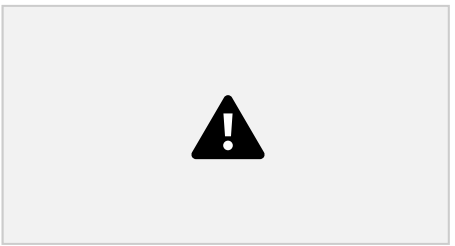


Figure 4-7. *TextInputType.email. Note the @ sign*

Figure 4-8. *TextInputType.number*

Figure 4-9. *TextInputType.phone*

Tip If you want to limit the type of text that is allowed to be entered, you can do so with the TextInput's inputFormatters property. It's actually an array so you can combine one or more of ...

- BlacklistingTextInputFormatter – Forbids certain characters from being entered. They just don't appear when the user types.

- WhitelistingTextInputFormatter – Allows only these characters to be entered. Anything outside this list doesn't appear.

- LengthLimitingTextInputFormatter – Can't type more than X characters.

Those first two will allow you to use regular expressions to specify patterns that you want (white list) or don't want (black list). Here's an example:

```
return TextField(
 inputFormatters: [
 WhitelistingTextInputFormatter(RegExp('[0-9 -]')),
LengthLimitingTextInputFormatter(16)
 ],
```



```
 decoration: InputDecoration(
 labelText: 'Credit Card',
 ),
);
```

In the WhitelistingTextInputFormatter, we're only

allowing numbers 0–9, a space, or a dash. Then the LengthLimitingTextInputFormatter is keeping to a max of 16 characters.

Checkboxes

Flutter checkboxes (Figure 4-10) have a boolean value property and an onChanged method which fires after every change. Like all of the other input widgets, the onChanged method receives the value that the user set. Therefore, in the case of Checkboxes, that value is a bool.

```
Checkbox(
 value: true,
 onChanged: (bool val) => print(val)),
```


Figure 4-10. *A Flutter Checkbox widget*

Tip A Flutter Switch (Figure 4-11) serves the same purpose as a Checkbox – it is on or off. So the Switch widget has the same options and works in the same way. It just looks different.

Figure 4-11. *A Flutter Switch widget*

Radio buttons

Of course the magic in a radio button is that if you select one, the others in the same group are deselected. So obviously we need to group them somehow. In Flutter, Radio widgets are grouped when you set the groupValue property to the same local variable. This variable holds the value of the one Radio that is currently turned on.

Each Radio also has its own value property, the value associated with that particular widget whether it is selected or not. In the onChanged method, you'll set the groupValue variable to the radio's value:

```
SearchType _searchType;
//Other code goes here
Radio<SearchType>(
 groupValue: _searchType,
 value: SearchType.anywhere,
 onChanged: (SearchType val) => _searchType = val), const
Text('Search anywhere'),
Radio<SearchType>(
 groupValue: _searchType,
 value: SearchType.text,
 onChanged: (SearchType val) => _searchType = val), const
Text('Search page text'),
Radio<SearchType>(
 groupValue: _searchType,
 value: SearchType.title,
 onChanged: (SearchType val) => _searchType = val), const
Text('Search page title'),
```