Building Cross-Platform Apps with Flutter and Dart

Build scalable apps for Android, iOS, and web from a single codebase

Deven Joshi

bpb

Building Cross-Platform Apps with Flutter and Dart

Build scalable apps for Android, iOS, and web from a single codebase

Deven Joshi

www.bpbonline.com

Copyright © 2023 BPB Online

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

First published: 2023

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-89423-570

www.bpbonline.com

www.bpbonline.com

Dedicated to

My beloved family & friends,
who keep pushing me on.

About the Author

Deven Joshi is an avid app developer who loves all things mobile. Tinkering with Flutter since its early days, he is heavily involved in the Flutter community and has open-source contributions including apps, articles, videos, and packages. He has worked with several startups and companies across multiple domains helping them build out their Flutter products. He is an active Flutter evangelist speaking about the topic at various local and international conferences. He is currently a Developer Advocate at Stream and specializes in their Flutter SDKs. Based on all his contributions to Flutter, he is recognized by Google as a Google Developer Expert (GDE) in Flutter and Dart.

About the Reviewers

- **Amadi Promise** is a highly skilled Android and Flutter developer with a track record of success in building robust and scalable mobile applications. Passionate about problem-solving through coding, technical writing, and sharing knowledge through public speaking engagements. Committed to helping individuals succeed in the rapidly evolving digital landscape through education and training in digital skills."

- **Santosh Das** is an experienced software developer who provided valuable feedback on this book. With expertise in Flutter technology and 5 years of experience in software development, he ensured that the book met high standards for technical accuracy, clarity, and relevance. He holds a bachelor's degree in computer science and technology from Gujarat Technological University, and this book marks his debut as a technical reviewer. Santosh's commitment to staying up to date with the latest developments in Flutter makes him a valuable resource for the technical community.

Acknowledgement

I want to express my deepest gratitude to my family and friends for their unwavering support and encouragement throughout this book's writing.

I am also grateful to BPB Publications for their guidance and expertise in bringing this book to fruition. It was a long journey of revising this book, with valuable participation and collaboration of reviewers, technical experts, and editors.

I would also like to acknowledge the unwavering support and guidance of everyone in the Flutter community who were pivotal in my growth.

Finally, I would like to thank all the readers who have taken an interest in my book and for their support in making it a reality.

Preface

Flutter, the open-source UI toolkit developed by Google has gained immense popularity among developers due to its ability to create stunning, fast, and fluid user interfaces. By using a single codebase, Flutter enables you to build applications that feel native to each platform, offering a consistent user experience regardless of the device. With its powerful features and extensive Widget library, Flutter provides an efficient and productive environment for building complex applications in record time.

Accompanying Flutter is Dart, a modern, object-oriented programming language designed to optimize the development process. Dart combines the best aspects of familiar programming languages, offering a concise syntax, strong type system, and advanced features such as asynchronous programming and reactive programming patterns. As the primary language for Flutter development, Dart brings efficiency and elegance to your codebase, allowing you to write expressive and maintainable code.

This book curates a comprehensive journey through the world of cross-platform development with Flutter and Dart. Whether you are a beginner or an experienced developer, the aim is to provide you with the knowledge and practical skills necessary to build robust, scalable, and visually appealing applications. From setting up your development environment to exploring advanced topics such as state management, animations, and testing, each chapter is crafted to deliver hands-on insights and real-world examples.

Chapter 1: An Overview of Dart- provides a general introduction to the Dart language. It also provides the first look into the language and talks about its history. The evolution of the language over time is also shown.

Chapter 2: Data Types- introduces the various built-in data types offered by Dart, such as numbers, strings, booleans, lists, and maps. It also explores the characteristics and usage of each data type, highlights their strengths and will guide you on when and how to employ them in your code. The chapter also talks about null-safety and deals with nullable and non-nullable variables.

Chapter 3: Conditionals and Loops- explores conditional statements (if, else if, else) and loops (while, do-while, for, for-each) in Dart. These constructs provide

powerful tools for controlling program flow and executing code based on conditions or for a specific number of iterations.

Chapter 4: Functions and Classes- explores Functions and Classes in Dart. Goes into various kinds of functions with respect to parameters as well as extension functions. The chapter also dives into the nuances of classes and creating a hierarchy.

Chapter 5: Operators- talks about using operators effectively in your Dart code, enabling you to manipulate and evaluate data with precision and efficiency. The chapter goes through the operators available in the Dart language.

Chapter 6: Asynchronous Programming- explores the power of asynchronous programming in Dart. Asynchronous programming allows you to execute concurrent and non-blocking operations, enabling your applications to handle time-consuming tasks without blocking the user interface. This chapter dives into concepts such as futures, async, await, and streams, equipping you with the knowledge and techniques to write efficient and responsive code.

Chapter 7: Why Flutter?- delves into the unique features and advantages of Flutter that makes it stand out among other frameworks. Additionally, we discuss the extensive widget library, the vibrant Flutter community, and the ability to build high-performance applications with a single codebase. By the end, you will have a clear understanding of why Flutter is a powerful framework for creating stunning cross-platform applications.

Chapter 8: Installing Flutter- guides you through the process of installing Flutter, the open-source UI framework for cross-platform app development. This chapter provides step-by-step instructions for setting up Flutter on your preferred operating system. It also covers the installation of necessary dependencies, configuring the Flutter SDK, and setting up the development environment.

Chapter 9: Flutter Project Structure and Package Ecosystem- explores the project structure and package ecosystem in Flutter, providing insights into organizing and managing your Flutter projects effectively. This chapter examines the essential directories and files that make up a Flutter project. It also delves into the Flutter package ecosystem, which offers a wide range of pre-built packages and libraries to enhance your app development process and talks about leveraging packages from the official Flutter package repository, as well as how to manage dependencies using Flutter's package manager, pub.dev.

Chapter 10: Diving into Widgets- explores the different types of Widgets, including stateless and stateful Widgets, and their role in creating interactive user interfaces. This chapter discusses the Widget tree and how Widgets are composed hierarchically to form the UI structure. You will also learn about the Widget lifecycle, handling user interactions, and updating UI elements based on state changes.

Chapter 11: Basic Widgets and Layouts- delves into Flutter's extensive Widget library, showcasing commonly used widgets and demonstrating how to customize and combine them to create visually appealing and functional interfaces.

Chapter 12: Networking in Flutter- covers the various techniques and tools available in Flutter for making HTTP requests, handling responses, and managing network connectivity. This chapter also discusses techniques for handling asynchronous operations during network requests, ensuring your application remains responsive while data is being fetched or uploaded.

Chapter 13: Local Data Persistence- delves into the realm of local data persistence in Flutter, enabling you to store and retrieve data locally on the user's device. This chapter explores various techniques and mechanisms for persisting data, ensuring that your application can retain and access information even when offline or between app sessions. We will also look into the usage of local databases, such as SQLite, in Flutter for managing structured data and performing advanced data manipulation operations.

Chapter 14: Theming, Navigation, and State Management- begins by diving into theming, which allows you to customize the visual appearance and styling of your Flutter application. This chapter delves into navigation techniques in Flutter, enabling you to create intuitive app flows and handle user interactions. It also discusses state management - a crucial aspect of building scalable and maintainable Flutter applications. We explore different state management approaches, including Provider and BLoC, empowering you to manage and update app state efficiently, resulting in responsive and interactive user experiences.

Chapter 15: Advanced Flutter-Animations- explores the realm of advanced Flutter animations, empowering you to bring your applications to life with fluid and captivating motion. This chapter talks about the fundamentals of animations in Flutter and goes into various kinds of animations such as implicit and explicit animations.

Chapter 16: Advanced Flutter – Under the Hood- breaks down Widgets to their fundamental building blocks including Elements and RenderObjects. This chapter talks about the various trees of Flutter and how Widgets work internally.

Chapter 17: Writing Tests in Flutter- discusses the different types of tests in Flutter, including unit tests, widget tests, and golden tests, and provide guidance on when and how to use each type effectively.

Chapter 18: Popular Flutter Packages- explores a selection of popular Flutter packages that extend the capabilities of the framework and enable you to build feature-rich and robust applications. This chapter highlights a variety of packages across different categories, each offering unique functionalities and solutions to common development challenges.

Chapter 19: Deploying Applications- explores the process of deploying Flutter applications, ensuring that your creations reach users on various platforms. This chapter goes into the deployment options available for different target platforms, including iOS, Android, and the web. We will also discuss platform-specific deployment considerations, such as submitting apps to the Apple App Store and Google Play Store, as well as the other respective platforms.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/szxebry

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Building-Cross-Platform-Apps-with-Flutter-and-Dart**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At **www.bpbonline.com**, you can also read a collection of free technical

articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

1. **An Overview of Dart** **1**
Structure 1
Objectives 2
History of Dart 2
About Dart 3
Working of Dart in Flutter 5
Hello, World! 6
The evolution of the Dart language 6
Dart 1.0 .... *7*
Dart 2.0 .... *7*
Dart 2.12 .... *7*
Dart 3.0 .... *8*
What will we learn about Dart? 8
Conclusion 9
Questions 9

2. **Data Types** **11**
Introduction 11
Structure 12
Objectives 12
The Basic Data Types 12
Numbers 12
num 13
Integer *13*
Double *14*
Strings 14
Booleans 15
Lists 16
forEach() *17*

map() 17
Sets 18
Maps 19
Other types 20
Type inference 20
Public/private variables 21
Null safety 21
Late variables 23
Converting nullable to non-nullable 23
Conclusion 24
Questions 25

3. Conditionals and Loops 27
Introduction 27
Structure 27
Objectives 28
Conditionals 28
The if statement 29
The else block 29
The ternary operator 30
The switch statement 31
Loops 33
The while loop 34
The for loop 35
The for…in (for…each) loop 36
The do…while loop 37
Iterable specific loops 38
A little more about loops 39
Conclusion 40
Questions 40

4. Functions and Classes 41
Introduction 41

Structure 41
Objectives 42
Starting with functions 42
Structure of a Dart function 42
Optional parameters 43
Positional parameters 43
Named parameters 44
Passing default values to optional parameters 45
Functions as first-class objects 45
Defining function types in Dart 46
Extension methods 47
Starting with Dart classes 48
Constructors in Dart 48
Named constructors 49
Inheritance, interface, and mixins 50
Basic inheritance 50
Interfaces 51
Mixins 51
Conclusion 52
Questions 52

5. Operators 53
Structure 53
Objectives 54
The ternary operator (A ? B : C) 54
The ?? operator 55
The ?. operator 56
The ??= operator 57
The cascade ("..") notation 58
The fat arrow (=>) operator 59
The ~/ operator 60
The spread ("...") operator 60

The null-aware index ("?[]") operator 61
The null-aware cascade ("?..") operator 61
Conclusion 62
Questions 62

6. Asynchronous Programming 63
Introduction 63
Structure 64
Objectives 64
The need of asynchronous programming 64
Understanding the async/await structure 65
Looking into futures 65
Handling futures 66
Delayed futures 68
FutureBuilder 68
Streams 69
Isolates 70
Conclusion 71
Questions 71

7. Why Flutter? 73
Introduction 73
Structure 73
Objectives 74
Another framework 74
A look back in time 75
Netscape 75
Understanding mobile applications 76
The first mobile apps 76
Hybrid vs. cross-platform 77
The rise of hybrid frameworks 78
Things we can do differently 79

How does React Native work? *81*
Enter Flutter 82
What does Flutter do differently? *82*
Flutter solving React Native's issues *84*
Hot restart, hot reload 86
Docs, support, and community 86
Conclusion 87
Questions 87

8. Installing Flutter 89
Structure 90
Objectives 90
Initial setup 90
Installing for Windows 90
Installing for macOS 92
Installing for Linux 93
IDEs for Flutter development 94
Setting up Android Studio 94
Setting up VS Code 97
Flutter extensions 98
VS Code extensions *99*
Awesome Flutter snippets *99*
Pubspec assist *100*
Rainbow brackets *100*
Android Studio extensions *101*
Flutter Pub Version Checker *101*
Flutter Snippets *102*
Rainbow brackets *103*
Conclusion 104

9. Flutter Project Structure and Package Ecosystem 105
Introduction 105
Structure 105

Objectives....106
Beginning the Flutter journey....106
Breaking down UI and logic files....108
Understanding lib and main.dart....108
Understanding pubspec.yaml....109
What is a YAML file?....109
Setting basic project metadata....110
Versioning and environment....110
Adding dependencies (packages and plugins)....111
Understanding Flutter's package repository: pub.dev....111
Packages outside pub.dev....113
Package vs plugin....113
Versioning dependencies....113
Dependency overrides....114
Developer dependencies....115
Adding assets....115
Specifying publish_to....116
The test folder....116
Importing packages....117
README.md and LICENSE....118
Important files to remember in a Flutter project....118
Android manifest....118
build.gradle....119
Info.plist....120
AppDelegate....121
Conclusion....121

10. Diving into Widgets....123
Introduction....123
Structure....123
Objectives....124
What is a widget?....124

Understanding composition 125
Refreshing UI: setState() 126
Stateful vs. stateless widgets 127
What is state? *127*
An easy rule to remember *129*
StatelessWidgets *129*
StatefulWidgets *130*
Underlying differences and performance differences 131
The layers beneath Flutter 132
Material, Cupertino and more 133
Bonus: WidgetsApp *137*
The Widgets layer 138
Understanding RenderObjects 138
About Elements 140
Conclusion 142
Questions 142

11. Basic Widgets and Layouts 143
Introduction 143
Structure 143
Objectives 144
Approaching Flutter layouts 144
Text 145
TextStyle *145*
Changing font size, font weight and font family *145*
Text color *146*
Other customizations *146*
Max lines and overflow *148*
Buttons 149
TextButton *149*
ElevatedButton *150*
OutlinedButton *150*

Button properties *151*

Color *151*

Button shape *152*

Column and row 153

Rows: columns - but horizontal *156*

Icon 156

Icon customization *157*

Padding 157

Container 159

Adding size and color *159*

Alignment and padding *160*

Decoration *161*

Stack 162

Basic implementation *162*

Positioned *163*

AppBar 164

Title, size, color *165*

Leading and actions *165*

CenterTitle *166*

Scaffold 166

AppBar *167*

FloatingActionButton *167*

Bottom 168

BottomNavigationBar *168*

Body *169*

ListView 171

Scroll direction *172*

Scroll physics *173*

Understanding the basic Counter app 174

Conclusion 177

Questions 177

12. Networking in Flutter..........179

Introduction..........179

Structure..........179

Objectives..........180

Connecting to the internet..........180

The TMDB API..........180

The HTTP packages..........182

Understanding data models..........185

Modifying the fetch function..........188

Ways to create data models..........189

QuickType..........189

Code generation..........191

Problems with building UI from data in API calls..........192

Building UI from network data..........193

Conclusion..........196

Questions..........196

13. Local Data Persistence..........197

Introduction..........197

Structure..........198

Objectives..........198

Getting started..........198

Types of data..........198

Settings / preferences..........199

App feed..........199

A note on multi-platform support..........200

SharedPreferences..........201

Adding SharedPreferences to your app..........201

Using SharedPreferences..........202

Create..........202

Read..........202

Update and delete..........203

sqflite 203

Adding sqflite to your app 204

Creating a database 204

CRUD operations 205

Create 205

Update 206

Delete 206

Hive 206

Adding Hive to your app 207

Basic data storage 207

Storing objects in Hive 208

Conclusion 209

Questions 210

14. Theming, Navigation, and State Management 211

Structure 212

Objectives 212

Adding theming 212

Creating and adding themes 213

Adding dark mode 216

Understanding navigation 217

Navigator methods 218

Push page 218

Pop page 219

Push replacement 219

Push and remove until 220

Pop until 220

Introduction to state management 220

The Problems With setState() 221

InheritedWidget 223

Provider 224

Riverpod 227

bloc/flutter_bloc ... 229
Conclusion ... 232
Questions ... 232

15. Advanced Flutter - Animations ... 233
Introduction ... 233
Structure ... 233
Objectives ... 234
What is an animation? ... 234
About the Flutter animation framework ... 235
Basic building blocks of animations ... 235
Tween ... 235
The question here: What is "lerp"? ... 236
AnimationController ... 237
Animation ... 239
Creating a basic animation from scratch ... 241
AnimatedBuilder ... 248
TweenAnimationBuilder ... 249
Implicit animations ... 252
A few widgets that use implicit animations ... 254
AnimatedOpacity ... 255
AnimatedPositioned ... 256
AnimatedCrossFade ... 257
Conclusion ... 259
Questions ... 259

16. Advanced Flutter - Under the Hood ... 261
Structure ... 261
Objectives ... 262
Understanding Flutter as a UI toolkit ... 262
Concerns about the Flutter approach ... 263
The Trees of Flutter ... 263

Understanding RenderObjects 264
Types Of RenderObjects 264
Understanding Elements 265
RenderObjectWidgets 266
Breaking down Widgets 267
Breaking down Opacity 267
Breaking down Text 270
Conclusion 276

17. Writing Tests in Flutter 279
Structure 280
Objectives 280
The different types of testing 280
Setting up tests 281
Exploring setUp() and tearDown() 281
Exploring test variants 282
Adding timeouts for a test 284
Unit tests 285
Widget tests 287
Creating (pumping) a widget to test 288
Understanding Finders 289
find.byType() 289
find.text() 290
find.byKey() 291
find.descendant() and find.ancestor() 292
Understanding the WidgetTester 294
A bit about pumpWidget() 294
A bit about pump() 294
Going to pumpAndSettle() 297
Interaction with the environment 297
Integration tests 298
Golden tests 300

Conclusion 301
Questions 301

18. Popular Flutter Packages 303
Introduction 303
Structure 304
Objectives 304
dio 304
url_launcher 306
Web link 307
Mail 307
Phone 307
SMS 308
file_picker 308
Picking single file 308
Picking multiple files 309
Pick certain types of files 309
image_picker 309
Picking a single image 309
Picking multiple images 309
Capture image or video 310
geolocator 310
Getting location 310
Getting last known location 310
Listening to location 311
connectivity_plus 311
Check connection status 311
Check WiFi vs cellular 312
Listen to connection status 312
sensors_plus 312
Listening to accelerometer events 312
Listening to user accelerometer events 313

Listening to user accelerometer events 313

Listening to magnetometer events 313

Listening to gyroscope events 313

google_maps_flutter 314

animated_text_kit 315

Rotate 315

Scale 316

Fade 316

Typewriter 317

cached_network_image 317

chewie 318

auto_size_text 320

flame 321

Drawbacks of packages 324

Conclusion 324

Questions 325

19. Deploying Applications 327

Structure 327

Objectives 328

Versioning your application 328

Deploying to the Google Play Store 328

Deploying to the Apple App Store 331

Deploying to Web 334

Uploading to Firebase 335

Deploying to macOS 337

Deploying to Linux 338

Deploying to Windows 340

Conclusion 341

Questions 342

Index 343-351

CHAPTER 1
An Overview of Dart

The plethora of programming languages available in the world is often the reason potential developers are confounded when a framework opts to go for a relatively unknown language in the mainstream development world. However, Dart was chosen for a very specific feature set that gives Flutter an edge over existing application development frameworks, and features that we will explore in detail in due time. As I often see it, Dart is a language that offers features from all over the programming sphere without seeming unfamiliar. If you are familiar with any major programming language, Dart should not seem very new, albeit with a few surprises here and there.

Structure

In this chapter, the following topics will be learned:

- History of Dart
- About Dart
- Working of Dart in Flutter

- Hello, World!
- The evolution of the Dart language
- What will we learn about Dart?

Objectives

After studying this chapter, you should gain a greater understanding of Dart's origin and why it is used in Flutter.

History of Dart

Dart is a comparatively recent language, debuting in 2011 and 1.0 released in 2013. However, people often point out correctly that newer languages like Swift came out after Dart and are widely used, whereas Dart is not. The reason for that pertains more to the original purpose of the language than its feature set. Dart originally served a very different purpose, with Google pitching it as a replacement for JavaScript. Dart can be directly compiled to JavaScript using the **`dart2js`** compiler.

However, Dart failed to catch on to its original purpose and became neglected, even going to the top of lists such as *Worst programming languages to learn this year* on blogging sites. Again, not because of the language and semantics, but how little it was used outside Google. Then, when very few expected the Dart language to take off, Flutter arrived and brought a storm of developers to try it out (fast-forward to now, and Dart and Flutter are some of the fastest-growing requirements on the market). This influx of new developers brought more rapid changes and new features to the language. Dart was not explicitly designed with Flutter in mind, however, it was a good fit when several languages were compared. *Figure 1.1* illustrates the logo of Dart:

***Figure 1.1**: The Dart logo*

I remember giving Dart a try in 2013/14 (my memory is a bit foggy on this one) – when Flutter likely existed in a conceptual stage, if at all. They had examples like creating a pirate badge with Dart on the website, which I followed along just for fun - not expecting I would have anything to do with the language in the future since I was into native Android development, which was something geared towards the web. I did not even faintly expect that a language I did just because of an obscure blog post would be my primary programming language almost a decade later.

blog post would be my primary programming language almost a decade later.

Funnily enough, the next time I heard of Dart was when I happened to run across the alpha version of Flutter almost 2-3 years later. Even then, Flutter was not nearly as developed as it is today. The tooling is the most annoying thing because the code structure of Flutter was quite different from the Java/XML I was used to. But after that, I just found a flow that I had not in other mobile frameworks – and Dart felt

instantly familiar, obviously partly from my earlier exposure, but more because it was designed to be. Almost overnight, I could develop ideas in hours instead of days or weeks. Mobile development was not the same for me – and I clung to Flutter and Dart because I knew they were the next big things.

Moving on from the memory lane to something more objective.

About Dart

Dart is open-source, object-oriented, and statically typed (2.x). Most features of the language should be familiar to most people, with some added features like mixins, and syntactic sugar designed to make common tasks in development easier.

Here are a few reasons Flutter chose Dart in particular:

- **Easier to learn**: Dart does not require an exorbitant time required to learn. Therefore, developers can focus on the Flutter framework instead of spending time adapting their existing knowledge to fit the new language and semantic constraints.

- **Can be Ahead-Of-Time (AOT) or Just-In-Time (JIT) compiled according to need**: AOT compilation eliminates the need for code to be compiled every time it is run, leading to faster start-up times. This is effective when apps are installed on a user's device, leading to much quicker app launches.

  AOT, however, leads to slower development times when the code is changed or updated. This is when JIT compilation makes for a pleasant development experience offering easy and quick code changes. Dart uses JIT in development and AOT in production apps, the best of both worlds. The JIT technique allows for the stateful hot reload Flutter is famous for. (More to come in later chapters).

  Here is a small tidbit for Android lovers: If you ever had a phone with Android Jellybean and, subsequently, Android KitKat, you may have noticed that KitKat apps opened way quicker but required more time for installation compared to Jellybean. This was due to the new **Android Runtime** (**ART**) being implemented. ART leveraged AOT compilation – hence taking extra time but didn't need to do any extra work when opening an app, therefore the much faster loading times. If you are young enough not to know Android KitKat and Jellybean, you sincerely make me feel old.

- **Eliminates the need for a declarative language similar to XML (Android) or JSX (React)**: Declarative layout languages often add a lot of code to the application by defining a separate language for UI and code. For example, in Android, before Kotlin came along, developers needed to get references to views before using them, leading to unnecessary steps that can be attributed

to context-switching. However, Dart allows Flutter to declare UI alongside normal code, making several common tasks like creating lists easier than equivalents on Android. *Figure 1.2* describes the contents of a Flutter app when it is in development:

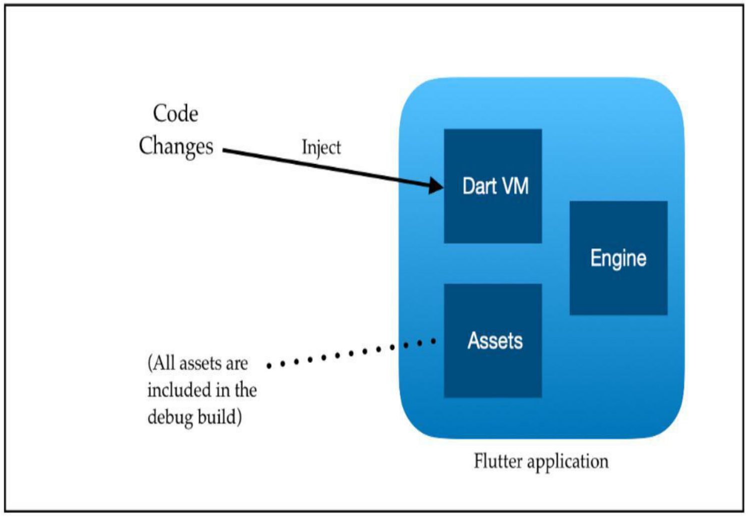


***Figure 1.2**: Running a Flutter app in development.*

- **Null-safe**: Null-safety is a favourite feature of mobile development folks since it drastically reduces errors when creating a mobile application. While we talk about this a bit more later, it means that you explicitly need to tell Dart what data in your app can and cannot be null. Dart will enforce that you do not allow null values or explicitly check for them. Over time, at least in my experience, this saves hundreds of hours on a single project.

  While null safety is now implemented in Dart, this was not the case when it originally chose Dart. Implementing null safety later was a significant effort by the Dart team and the entire community in unison since all Dart and Flutter packages needed to be upgraded. Not having null-safety was one of, if not the largest complaint that Android/iOS developers had coming to Flutter since languages like Kotlin provided immense relief to the `NullPointerException` plagued Java developers. They did not want to give up that luxury – which, if you can completely understand even if you

have developed a single Java Android project to completion.

Alongside Flutter, **AngularDart** also allowed the creation of web apps using Dart. In 2019, Flutter also announced its move to the web with a project named *Hummingbird* — Flutter for the web. Flutter Web is now stable alongside many desktop platforms such as Windows, MacOS, and Linux. After Flutter Web's development, Google's focus on the web seems to have shifted a bit away from AngularDart and now recommends Flutter for developing web projects.

Frameworks like Dart Frog (**`dart_frog`** on **`pub.dev`**) also allow the creation of REST APIs using Dart.

However, Flutter is by far the most popular framework that uses Dart, and the one being more rapidly updated, developed, and has greater community support.

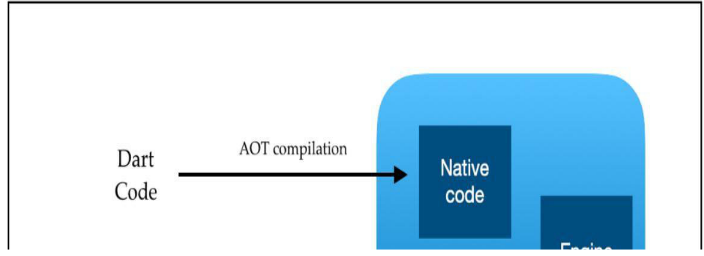
Working of Dart in Flutter

How Dart code runs in a Flutter app is an interesting topic. This is a small overview for the curious, and Flutter development does not need explicit knowledge of this topic, so feel free to skip this section if you want to learn Dart instead.

As discussed in the previous section, Flutter uses Just-In-Time compilation in development and AOT compilation in release mode. When using JIT compilation, Flutter uses a **virtual machine** (VM) to run Dart code. This runs code slower than release applications might and adds some size to the debug **`.apk`** or **`.ipa`** file. This is not the complete explanation for the huge debug build file size you might experience when building a Flutter app. The large debug build size is due to the app building with basically everything in the project, so you do not need to rebuild to use new assets. Alongside this, certain optimizations which remove unused code – known as tree shaking are also not used when running a debug build of the app.

Things change when a release build is built — no Dart VM runs code in the release build. Instead, the code is compiled (ahead of time) to native ARM libraries and added to the project, avoiding the need for a Dart VM.

A Flutter app does not use platform widgets and paints all UI itself. Hence, all taps/ gestures on the screen and rendering are handled by compiled Flutter code. *Figure 1.3* describes the contents of a Flutter app when in production:



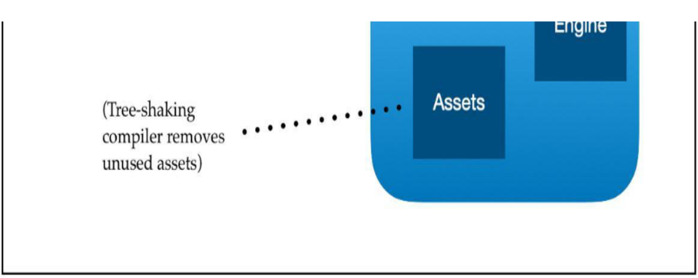


***Figure 1.3**: Running a Flutter app in release mode*

For Android: The Flutter code is compiled AOT into native, ARM, and x86 libraries. The Flutter engine's code is converted into C and C++ code using the **Native Development Kit** (**NDK**) that Android provides. This is bundled into a **`.apk`** file, and a build is created.

Note: There is no virtual machine in the release build, only the debug build.

For iOS: iOS apps are built like the Android files, with the main difference being that, using the LLVM compiler substituting Android's NDK. These are then packaged into a **`.ipa`** file.

Hello, World!

As with any language, we must carry on the tradition of creating a program that prints **`Hello, World!`** as our first-ever Dart program. This should be a pleasant break from the daunting introductory programs in other programming languages.

Code 1.1:

```
void main() {

    print('Hello, World!');

}
```

Note: Writing `void` is optional here as function return types are optional (but you still should).

A few things to note:

- Top-level functions are allowed; hence, no class encloses the main function.
- Console output is done by a simple **`print()`** function.
- The **`public`** keyword does not exist in Dart. More on that in the next chapter.
- Unlike languages like Kotlin and Python, semicolons still exist in Dart-land.

However, semicolons may soon be optional as well.

The evolution of the Dart language

This section looks at the evolution of the Dart language through important releases. This helps us understand the development direction of the language and the fundamental shifts taken in its history.

Dart 1.0

The 1.0 version release of Dart came at a time when Dart was meant to be a JavaScript replacement. There were also plans to include a Dart VM directly in Chrome which would make it easy to run Dart without transforming it. However, these plans were changed later, and Dart was compiled into JavaScript instead.

Version 1.0 was quite different from the Dart we know now. It featured a type system that was not sound. Types were simply annotations in the language allowing snippets like these to run successfully:

Code 1.2:

```
void main() {

    int a = 3; // a is declared as an integer
    a = 'This is a string'; // a is assigned an integer

}
```

Dart 2.0

Dart version 2.0 was one of the most substantial Dart releases ever. It brought a sound-type system and steered the language towards Flutter rather than the web. The release made Dart more relatable to Java/Kotlin/Swift developers, which made the core of Flutter's developer base. However, null safety was seen as a significant missing feature by most developers recently switching to null-safe languages such as Kotlin.

Without null safety, this was valid code in Dart:

Code 1.3:

```
Shape shape; // Notice that is still null

shape.calculateArea(); // No compile-time errors
```

Output: **`Runtime error since shape is null`**

Dart 2.12

Dart 2.12 finally brought sound null-safety to Dart alongside a number of other changes, such as a stable **Foreign Function Interface** (**FFI**), which allows efficient communication with native platforms. After sounding null safety, getting runtime

errors for developers was much more challenging. However, this version also allowed running the app without null-safety or partially with null-safety.

This is what null-safe code looks like in the previous example:

Code 1.4:

```
Shape? shape; // Shape now has to be declared as a nullable type

shape?.calculateArea(); // Method will only be called if Shape object is
not null
```

Dart 3.0

Dart 3.0 brings along several features asked for by Flutter developers worldwide, such as patterns, records, and capability controls for classes. It also makes it compulsory to use null-safe projects. Projects cannot run with a partial null-safety post this version. This results in performance improvements since several runtime checks can be eliminated.

Here is a snippet demonstrating records and patterns in simplified Dart 3.0 code:

Code 1.5:

```
void main() {

    double length = 0;
    double width = 0;

    Shape shape = Shape();

    (length, width) = shape.calculateLengthAndWidth();
```

```
}

// Inside the Shape class
(double, double) calculateLengthAndWidth() {
    return(shapeLength, shapeWidth);
}
```

What will we learn about Dart?

As this book's primary focus is on Flutter, the Dart programming constructs discussed will be angled towards topics that are specially required by Flutter rather than being

an exhaustive Dart reference. Some topics which are not required for Flutter, will be avoided for brevity. However, any topic related to Flutter development will be covered in detail, along with its examples.

Learning Dart specifically for Flutter allows us to leverage the operators, functions, and syntactic sugar it provides to build better applications. For example, learning the **`async`**/**`await`** pattern for asynchronous tasks and **`map()`** / **`reduce()`** / **`filter()`** to handle lists of widgets better.

Now that we are done with a general introduction to why Flutter chose Dart to develop apps, let us get hands-on with actual Dart code.

Conclusion

Dart is the language that all Flutter apps are written in. Hence, it is critical to understand the language well before proceeding to the more complex world of mobile app development using Flutter. Fortunately, there are not many complicated Dart-specific concepts to learn either – leading to an easy transition to the Flutter framework once basic Dart concepts are clear. This chapter focused on background details and how Flutter deals with Dart code during development and production.

In the next chapter, we start with our Dart journey by understanding the data types available in Dart, which are an essential building block to learning efficient Dart programming.

Questions

1. Is Dart Ahead-Of-Time or Just-In-Time compiled?
2. What was Dart originally meant for?

3. Does Dart use a virtual machine (VM)?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 2
Data Types

Introduction

In theory, holding data in a programming language is quite easy - we would have numbers, strings, some ideal data structures, and we would be good to go to create the program we want. Sadly, in the real world, we deal with things like hardware

and memory constraints - the new MacBooks likely do not offer an infinite RAM option. Every language has its own primitives, its own way of defining variables, and almost always some curious choices when it comes to language design. This can also make or break a language since variables form the base of everything we do in a program - if you cannot declare an array or make it five lines long, there will be considerable friction when adopting the language. While being unable to declare an array is an absurd example, things like null safety (discussed in the first chapter and at the end of this one) are not.

In this chapter, we will go through data types and everything you may need to know about variables and data types in the Dart language. Data types in Dart should be familiar to any programmer and variable declaration offers unique flexibility that accommodates both the `var` developers from JavaScript and similar, as well as folks who prefer more verbose type declaration in their code.

Feel free to try all the examples mentioned in the chapter and experiment with all the possible combinations, the best way to know what can and cannot be done.

Structure

In this chapter, we will cover the following points:

- The Basic data types
- Numbers
- Strings
- Booleans
- Lists
- Sets
- Maps
- Other types
- Type inference
- Public/private variables
- Null safety
- Late variables
- Converting nullable to non-nullable

Objectives

After studying this chapter, you should better understand the in-built data types of Dart and the working of variables in Dart. Null-safety as a concept should also be better understood.

The Basic Data Types

In this section, we go into the data types available in Dart. Some of these are used extensively when building a Flutter app, and others may not be. As you will find out, Dart is relatively less verbose than most other languages when declaring variables and creating arrays, maps, and so on - not Pythonic when eliminating verbosity. Still, it can be seen as a no-nonsense approach while eliminating unnecessary code.

Numbers

There are various interpretations of numbers in the Dart language. Any language needs multiple types to hold numbers because different types have different

characteristics and limitations. These types differ in aspects such as range of values, accuracy, performance, and more.

num

Numbers allow you to define numerical values in a program. In Dart, there are a few different types of numbers. The **num** is the superclass of double and integer and is the most basic way to define a number. However, in most cases, we do not use num, using **double** or **int** instead.

The **num** keyword allows you to define values as a number:

Code 2.1:

```
num a = 3;

// OR

num b = 3.14;
```

The **num** variable carries along various functions to help us get the desired value, a few of them are as follows:

Code 2.2:

```
b.toString() // Converts b to String type
```

```
b.toStringAsPrecision(2) // Converts to string with certain precision
(Here: 3.1)

b.ceil() // Rounds to higher value (Result: 4)

b.floor() // Rounds to lower value (Result: 3)
```

Since **int** and **double** are subclasses of **num**, they also inherit most of these functions.

Apart from specialized cases, there is usually no immediate need to use num apart from accommodating cases where a number may or may not have a decimal. We use the next two types for most of the work.

Integer

An integer stores a numerical integer value in a variable. Dart allows 64-bit integer values, meaning the number can range from -2^63 to 2^63 -1.

The variable declaration carries along from other languages, using the **int** keyword:

Code 2.3:

```
int a = 3;
```

We simply call functions available through the keywords to convert strings to doubles or integers.

Code 2.4:

```
int a = int.parse('314');
```

While rare, if you need to store a value even bigger than 64 bits can hold, you can use the **BigInt** class instead of int to hold an arbitrarily large number.

Double

A **double** can store decimal values in a variable and is also a 64-bit value. The **double** keyword is used for declaration.

Code 2.5:

```
double pi = 3.14;
```

An important thing to note (especially for Flutter) is that a double can implicitly take an integer value and convert it to a **double**. For example:

Code 2.6:

```
double a = 1;
```

Primitive types can also have function calls. For example, let us do the same example where we add an explicit conversion to a **double**:

Code 2.7:

```
double a = 1.toDouble();
```

This also extends to other function calls like **toString()**, **abs()**, **ceil()**, and other math functions without using a separate library that provides math functions.

We simply call functions available through the keywords to convert strings to doubles or integers.

Code 2.8:

```
double a = double.parse('3.14');
```

Strings

Strings carry over the same principles as most other languages. They can be declared using single or double quotes:

Code 2.9:

```
String s = "Hello, World!";

// OR

String s = 'Hello, World!';
```

Dart also allows multi-line strings that can continue over multiple lines without using concatenation.

Code 2.10:

```
String s = ''' This
Is a multi-line String ''';
```

Dart also allows string interpolation, which allows the insertion of values into a string without heavy use of the concatenation operator. As an example:

Code 2.11:

```
int answer = 42;

String response = "The answer is: " + answer.toString();
```

This gets complicated when a lot of values are involved. Instead, we can directly use string interpolation, which allows us to use the **$** operator for directly adding values in an understandable format to the string.

Code 2.12:

```
int answer = 42;

String response = "The answer is: $answer";
```

Note that **$** only works for literals, and if an inner field is to be accessed or an operation needs to be done, use **${}**.

Code 2.13:

```
String response = "The answer is: ${answer + 5}";
```

Booleans

Boolean values store true/false values in a variable. You can use the bool keyword to declare them:

Code 2.14:

```
bool demoValue = true;

bool otherValue = false;
```

Boolean values are especially useful when we use them in conditionals or loops, as seen in the next chapter, since their primary role is controlling the flow of the program.

Lists

Lists are collections of items of one or multiple types. They can be growable (we can keep on items forever - or until memory finishes) or fixed length. To declare a normal growable list, we can do:

Code 2.15:

```
List<int> integers = [1, 2, 3, 4, 5];

List<double> doubles = [1.1, 1.2, 1.3];
```

```
List<dynamic> dynamicList = [1, 3.14, "Hello, world!"];
```

To declare a fixed length list, we can do the following:

Code 2.16:

```
var fixedLengthList = List.filled(3, 0);
```

This creates a list with a length of **3**, with all elements being **0**. Lists have common operations, such as:

Code 2.17:

```
// Adding elements to the list
integers.add(6);

// Deleting elements from the list
integers.remove(5);

// Sorting a list
integers.sort() ;


// Clear the list
integers.clear();
```


Lists also supply several methods to iterate or transform the list into another object. Let us discuss a few that are important to us in Flutter to understand.

All the functions ahead will be discussed in detail in *Chapter 4, Functions and Classes.* It also covers functions as first-class objects which are used here.

forEach()

Here is an example of the **forEach()** function:

Code 2.18:

```
integers.forEach((value) {
    print(value);
});
```

The forEach() function allows us to iterate over each element in a collection. In this example, we are going over every element in the integer list and printing it. If you are confused over the slightly odd syntax, please refer to the next chapter to understand anonymous functions. Note that there is also an equivalent to the **for-**

each loop in other languages, which is the for-in loop (discussed later).

map()

The **map()** function allows us to transform the list from one list type to another. To do this, it calls the function we provide to the **map()** function once for every element in the original list. This is compiled into a new **Iterable** (a list-like object which can be iterated over) and supplied back.

As an example, if there's a list of integers that we need to convert to a list of strings, it is possible to do it with the **forEach()** function and a new list to store it like this:

Code 2.19:

```
List<String> strings = [];

integers.forEach((value) {

   strings.add(value.toString());

});
```


However, the **map()** function allows us to do something much simpler:

Code 2.20:

```
List<String> strings = integers.map((value) {

   return value.toString();

}).toList();
```

Note: The end toList() conversion is because map() returns an Iterable, not a list.

There are several more that I recommend you explore, like, **filter()**, **reduce()**, and **fold(),** which help you handle data more easily than conventional **for** loops.

Sets

Sets

Sets can be viewed as a simpler version of lists where a certain value can only be added once. This is extremely useful for getting all the unique objects on the list instead of items being repeated. The competitive coding crowd might want to keep this one in their books.

Sets can be declared with **{}** containing the initial set values:

Code 2.21:

```
Set<int> integers = {1, 2, 3};
```

Note: Maps are also declared with braces, and not declaring maps correctly might lead to Dart interpreting your code as a Set.

Sets work similar to a list with the **add()**, **remove()** functions as well as the **forEach()**, **map(),** and further functions.

Code 2.22:

```
Set<int> demoSet = {1, 2, 3} ;

demoSet.add(4);
// Result: {1,2,3,4}

demoSet.add(1);
// Result: {1,2,3,4} (No extra 1)
```


Sets are used where unique items are required.

Maps

Maps allow us to map one element to the other. It will enable us to store key-value pairs. A List can be visualized as a mapping from the index to the object in the array. For example, if we made a list of alphabets, the mapping can be seen as:

0 -> "a"

1 -> "b"

2 -> "c" …and so on.

Similarly, a map allows us to define our own index. If we wanted to make a map of basic alphabets and common words associated with it:

"a" -> "Apple" "b" -> "Ball"

The map associated with it would be syntactically defined as:

Code 2.23:

```
Map<String, String> words = {
     "a": "Apple",
     "b": "Ball",
}
```

The **`<String, String>`** in the map declaration says we are going to associate a **`String`** index to a **`String`** value. To access a map, we use an index. In a normal List, the index is 0,1,2...; the user defines the index here.

Code 2.24:

```
String wordFromA = words["a"];
```

In other aspects, a map is similar to a list, where we have **`forEach()`**, **`map()`**, and so forth.

Code 2.25:

```
words.forEach((key, value) {

    print("The letter is $key");
    print("The word is $value");

});
```


We will revisit functions such as **`forEach()`** in chapter 4 (Functions and Classes).

Other types

Dart also has other data types like runes (used for Unicode characters) and symbols which are great to study. However, for brevity and conciseness, we do not go into depth as they are not used as often in Flutter.

Type inference

In all the examples given till now, the types were written out. However, *Dart provides a var keyword that offers type inference,* meaning writing types is unnecessary. The types were given till now just for better understanding.

So, rewriting some of the earlier examples with type inference:

Code 2.26:

```
// a is assigned an int type
var a = 3;

// b is assigned a double type
var b = 3.14;

// c is assigned a boolean type
var c = true;

// d is assigned a List<int> type
var d = [1, 2, 3];

// e is assigned a String type
var e = "Demo String";

// f is assigned a Map<String,String> type
var f = {"a": "Apple"};
```

Note: `var` infers type and assigns the type and does not make the variable's type changeable. Use dynamic for potentially changing type.

A thing to be careful about is Dart inferring dynamic types in collections. For example:

Code 2.27:

```
var list = [];

list.add(1);
```

This code assigns a type of **`List<dynamic>`** and not **`List<int>`** because the type is assigned when the value is assigned. Dart does not know what you intend to add to the list in the future, so it assigns a dynamic type. As an example:

Code 2.28:

```
var list = [1, 2, 3.14];
```

Here, the list is also a **`List<dynamic>`** as more than one type (integer, double) is

added to the list at initialization.

Public/private variables

Simply put, the **public** and **private** keywords do not exist in Dart. A variable is public by default; we need to add an underscore to the variable name to mark it private.

Code 2.29:

```
// This is public var
a = 42;

// This is private
var _b = 42;
```

In practice, this saves a tremendous amount of code, and code avoids becoming unnecessarily verbose. There are also a few other time-savers that Dart offers, for which we have a chapter on Operators later on. But for now, we must first cover a few fundamentals.

Null safety

While null safety was not originally built into the Dart language, it has become quite a useful addition (added in Dart SDK version 2.12). Many developers breathed a sigh of relief as it was one of the top things asked for in the Dart language before the Flutter team decided to deliver it to them.

So, what is *null safety*?

When a variable is not assigned a value, it is given a default value of null. This simply means that the variable does not hold any kind of value yet. So why is this dangerous? Most mobile development errors before null safety came in the form of errors caused by using a variable value that was not assigned yet. I can understand if you are confused about how experienced developers forget to give their variable's values – but I can assure you it is not due to a lack of trying.

In a real-world app, most variables are not assigned values in a linear way. A lot of asynchronous tasks (seen in *Chapter 6, Asynchronous Programming*) work in the background fetching data and then creating objects in the app. Inevitably, some variables are used before they are assigned and this leads to errors. Developers were fed up with this and wanted a better way to know if this error existed – mainly

because it was tough to spot this kind of error before running it. So, the question became – can we tell developers of a potential variable still being null without running the app every time? This is how null safety came to be.

You will see two kinds of variables in a Flutter app:

Code 2.30:

```
int a = 5;
int? a = 5;
```

As you can see, the second line has an added question mark which denotes that this variable can be null. The first line indicates that the variable cannot be null, no matter how hard you try.

This means that this code snippet is valid:

Code 2.31:

```
int? a = 5;
a = null;
```

But this is not:

Code 2.32:

```
int a = 5;

// Compile error
a = null;
```

This is not just for integers but for any type. If you add a question mark in front of any type, it becomes nullable, while a type without a question mark is non-nullable.

Note: This excludes the dynamic type, as it can be nullable without declaring it nullable. Using dynamic will not throw a compiler error, it is redundant.

But how exactly does this help?

When writing code, you expect some data to never be null. If something you did makes this data null, null safety will ensure this is caught and not allowed. If you make everything nullable and write an app like that, it is equivalent to not having null safety at all – so easy with the question marks.

Late variables

Additionally, if you have a variable that will be non-nullable but still cannot be assigned at initialization, we can mark it late, meaning this will be assigned late, but

cannot be null once assigned. This variable assigned before initialization will throw an error.

Code 2.33:

```
late int a;

// Somewhere later
a = 5;
```

If you notice, we did not need to provide a value for **a** when declaring it unlike other non-nullable variables. However, this is only temporary. Using this variable without initializing it will throw an error.

Converting nullable to non-nullable

Sometimes, an operation expects a non-nullable variable, but you are stuck with a nullable variable to do it. Let us take an example of a number squaring function:

Code 2.34:

```
void square(int a) {
    print(a * a);
}

int? a;

// Throws an error
square(a);
```

This throws an error because **square()** expects a non-nullable variable. There are two ways to do this. One is to have a null check when calling the function:

Code 2.35:

```
void square(int a) {
    print(a * a);
}

int? a;
```

```
if(a != null) {
    square(a);
}
```

Since we are checking if **a** is null, the compiler gives it effective non-null status. The other way (be much more careful with this) is the **!** operator:

Code 2.36:

```
void square(int a) {
    print(a * a);
}


int? a;
square(a!);
```

This converts the nullable variable to a non-nullable variable and allows the operation. However, you should be careful with this operator because this code will throw an error at runtime because **a** is still null in the given snippet – we are just forcing the operation to occur.

The critical thing to remember is to use this operator only when you are sure that the variable cannot be null at that point of execution.

Conclusion

In this chapter, we went into detail about data types and variable declarations in Dart. We also saw how null safety works and influences the declaration decisions of any variable. Now that we know how to work well with variables, we can move on to more advanced concepts in Dart.

In the next chapter, we will learn about conditionals and loops, which are essential tof building logic when writing code.

Questions

1. What are in-built data types in Dart?
2. How do you declare a variable as private in Dart?
3. What is null safety?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 3

Conditionals and Loops

Introduction

Conditions and loops allow us to manipulate the control flow of programs and lead to more complex types of logic. Generally, conditions offer certain blocks of code to get executed when a condition is met, avoiding the other blocks. On the other hand, loops allow the execution of the same block multiple conditions depending upon a fixed condition (assuming we do not like infinite loops, of course). Both conditionals and loops are vital to any programming language as they form the base of the control flow mechanism. The easier it is to implement, the easier the language becomes.

Structure

In this chapter, we will discuss the following topics:

- Conditionals
    - The if statement
    - The else block
    - The ternary operator
    - The switch statement

- Loops
    - The while loop
    - The for loop

- The for loop
- The for-in loop
- The do-while loop
- Iterable specific loops
- A little more about loops

Objectives

After reading this chapter, you should have a good understanding of how to implement logic in a Dart program using conditions and loops.

Conditionals

True to their name, conditionals usually allow us to execute certain blocks of code or statements when certain conditions are met. They enable us to decide between two or more directions a program can take, allowing us to make more complex logic chains. Here is how a conditional statement might be shown on a flow diagram:

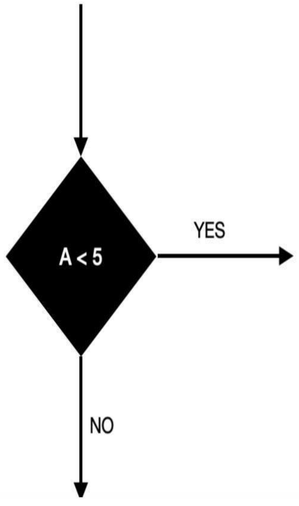


***Figure 3.1**: Diagram depicting a decision in program flow*

The placement of the condition depends upon the situation but let us look at both - the `if` condition and the ternary operator. Additionally, other helpful conditions will be shown briefly here, but in more detail in a later chapter.

The if statement

The **`if`** statement allows us to execute a code block based on a Boolean condition.

Here is what an example looks like:

Code 3.1:

```
var a = 5;

if ( a > 10 ) {

    print('A is greater than 10');

}
```

Output: **`No output`**

The else block

If the **`if`** block does not execute since the condition was not met, the **`else`** block can be added as a fallback block of code to be executed.

Code 3.2:

```
var a = 5;

if ( a > 10 ) {

    print('Execute this block when A is greater than 10');

} else {

    print(‘Execute this block when A is smaller than or equal to 10’);

}
```

Along with the **`else`** block, the **`else if`** block can also be used for certain conditions when the **`if`** condition is not met.

Code 3.3:

```
var a = 5;

if ( a > 10 ) {
```

```
    print('A is greater than 10');
} else if ( a > 5) {
   print('A is greater than 5 but less than 10');
} else {

    print('A is less than or equal to 5');

}
```

Output:

```
A is less than or equal to 5
```

The ternary operator

The ternary operator is an operator that can be used in two ways and often gives us a simple way to replace the if statement in code.

The ternary operator is written as:

Code 3.4:

```
Condition ? Statement1 : Statement2
```

Similar to the **if…else** structure, a ternary operator takes a condition and two statements. If the condition value is true, the first part is executed, failing which the second part gets executed.

An example of this is:

Code 3.5:

```
int a = 5;

a > 5 ? print('A is greater than 5') : print('A is smaller than or equal
to  5');
```

The ternary operator is also useful for other things. It also returns values:

Code 3.6:

```
bool condition = true;

int a = condition ? 1 : -1;
```


This cuts down on compared to the equivalent **if…else** implementation:

This cuts down on compared to the equivalent **if…else** implementation.

Code 3.7:

```
bool condition = true;

int a;

if (condition) {

  a = 1;

} else {

  a = -1;

}
```

The switch statement

When a large number of condition checks are involved with the same variable, **switch** statements make code much more concise than many **if…else** blocks. In a **switch** block, we define several cases that, execute the code associated with that particular case when met.

The general syntax is:

Code 3.8:

```
switch(variable) {

  case value1:
    // code here
    break;

  case value2:
    // code here
    break;

}
```

Let's take an example of a string variable that stores a color:

Code 3.9:

```
String color = 'red';

switch(color) {

 case 'blue':
   print('COLOR IS BLUE');
   break;

 case 'red':
   print('COLOR IS RED');
   break;

 case 'yellow':
   print('COLOR IS YELLOW');
   break;

 default:
   print('NO COMPATIBLE COLOR');
   break;
}
```

Here, we are examining the value of the variable **`color`**. We write cases in which we specify what happens if the given variable has the value of the case. So, if the variable has a value of **`yellow`**, the block after **`yellow`** is called.

The preceding example has the break written after each case code to avoid executing the next case block. A difference between Dart and other languages is that we must specify the behavior after the execution of each case code - such as telling it to break execution, continue (go to next **`case`** block), throw an exception, and so on.

The final default case written without the explicit **`case`** keyword is the block called when no other case matches. This is not compulsory; **`switch`** statements without a default case are allowed.

Switch statements are often used in handling all cases of **`enums`** (Most IDEs also allow auto-filling of all possible values of **`enums`** in switch blocks).

For example:

Code 3.10:

```
enum Color {
  red,
  blue,
  yellow
}

Color color = Color.red;

switch(color) {

 case Color.red:
   break;

 case Color.blue:
   break;

 case Color.yellow:
   break;
}
```

Loops

A loop is a control structure that allows you to repeat a block of code multiple times. There are two main types of loops in Dart: the for loop and the while loop. In a flow diagram, a loop may go back to an earlier part of the flow:

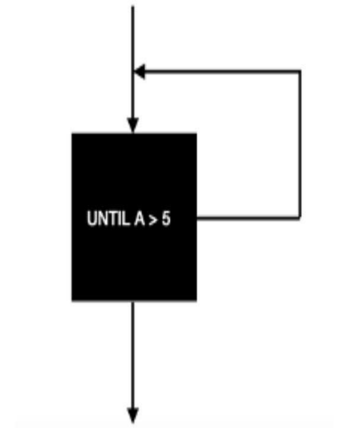


Figure 3.2: *Diagram depicting a loop in program flow*

This is useful for various conditions such as iterating over lists, running a task certain number of times, or until a condition is met. Loops must be well-defined for running since loops with uncertain conditions will crash a program by running infinitely. There are various types of loops in Dart and various ways to run them - which will be explored in this section.

First up, types of loops.

The while loop

The **while** loop allows us to execute a certain code block while a certain condition is true. The **while** loop is particularly prone to infinite loop errors since changing the value of the variable that defines the condition is not forced.

Here is how a basic **while** loop is structured:

Code 3.11:

```
int a = 0;

while ( a < 5 ) {

    // Execute code here
    a++;

}
```

Since the increment statement such as **a++** is not forced, **while** loops often cause errors.

Technically, **while** loops can also be run until broken out. The **break** keyword is used to break out of the loop at any time, regardless of the condition attached to it.

Code 3.12:

```
while (true) {

    // Execute code

    // This is usually done on a condition being met
    break;

}
```

Doing the loop this way may lead to unforeseen errors, so use it cautiously.

The for loop

The **for** loop allows for a far safer way of executing a loop by imposing a start condition, end condition, and an increment statement run after every loop. The code for a **for** loop is also more concise and readable than a **while** loop.

Here is an example of a basic **while** loop printing the first four natural numbers:

Code 3.13:

```
for ( int i = 1 ; i < 5 ; i++ ) {

    print(i);

}
```

Output:

```
1
2
3
4
```

The **for** loop takes four things:

1. Initializations before loop.
2. End condition to terminate loop.
3. Statements to execute after each cycle.
4. The code for each execution.

Since they are explicitly written but do not need several lines, **for** loops simplify code over **while** loops. The **for** loops can easily be used for iterating over loops using index:

Code 3.14:

```
var list = [1, 2, 3, 4, 5];

for ( int i = 0; i < list.length; i++) {

    print(list[i]);

}
```

Output:

```
1
2
3
4
5
```

There can also be multiple variables used within a **for** loop:

Code 3.15:

```
for ( int i = 1, j = 2, k = 3 ; i < 3 ; i++, j++, k++ ) {
    print(i);
    print(j);
    print(k);

}
```

Output:

```
1
2
3
2
3
4
```

This can be used for **for** loops with higher complexities. You can save space by doing multiple things when declaring the loop.

The for...in (for...each) loop

The **for** loop can easily be used for iterating over a list, as displayed in the preceding example. However, there is a better way to iterate over an object like, a list, map, or other **Iterable**. The **for…in** loop lets us directly iterate over every object in the **Iterable** without dealing with their index.

Here is the basic structure of the **for…in** loop:

Code 3.16:

```
var numList = [1, 2, 3, 4, 5];

for ( var number in numList ) {
```

```
    print(number);
}
```

Output:

```
1
2
3
4
5
```

The **for…in** loop makes it much more readable to iterate over objects like lists while removing the need for dealing with the index of the individual list items. This loop replaces a similar loop in languages like Java which would have had a syntax like this:

Code 3.17:

```
for ( int number : numList ) {
      // Execute code here
}
```

Changing the '**:**' to **in** makes a difference in terms of readability.

The main thing to note here is that we lose track of the index, and the **for…in** loop exists specifically for iterating over an object.

The do…while loop

The **do…while** loop is essentially a **while** loop but turned on its head. The **while** loops first checks if the condition supplied to it is true, following which it tries to execute the block of code for the first time. The **do…while** loop first runs the block of code supplied to it, after which it checks the condition for the first time. This allows us to guarantee at least one execution of the code block, no matter the condition.

The basic structure of the **do…while** loop is as follows:

Code 3.18:

```
do {
    // Execute code here
      execute();
```

```
} while (condition);
```

The equivalent while loop is:

Code 3.19:

```
execute();

while ( condition ) {

    execute();

}
```

Iterable specific loops

Iterable objects such as Lists, Maps, and Sets allow **`for…in`** loops, simplifying iteration over them. While this is an easy way to do it, they also offer other ways of iteration which involves higher order functions (explained more in the next chapter). In short, higher order functions are functions that take functions as parameters.

An example is the **`forEach()`** function which takes a function to be executed for each list element.

Code 3.20:

```
var list = [1 , 2 , 3 , 4 , 5];

list.forEach((e) {

    print(e);

});
```

Output:

```
1
2
3
4
```

```
5
```


Here, the **forEach()** function of the list accepts another function executed for all list elements. This is another concise way that iteration can be achieved over a list.

Similarly, other methods iterate over all elements of an **Iterable** for a specific purpose. The **map()** function also iterates over the **Iterable** object to convert one type of item to another and then returns a list with 1:1 mapping.

Code 3.21:

```
var list = [1, 2, 3, 4 ,5];

var stringList = list.map((e) => e.toString());
```

There are many more functions associated with **Iterable** objects that do similar things.

More about higher order functions in the next chapter.

A little more about loops

While the previous section discussed termination conditions for **for** loops, there are other ways to manipulate the execution of loops.

The **break** keyword allows us to break out of the loop even when the termination condition is not yet met:

Code 3.22:

```
for (int i = 0; i < 10; i++) {

  if( i == 5) {
    break;
  }

  print(i);

}
```

This example would only print the first five numbers since the 6th iteration breaks the loop.

Similarly, we have the **continue** keyword, which ends execution for the current iteration but does not end the loop – instead, it goes to the next iteration. Using the same example as earlier, we can see:

same example as earlier, we can see:

Code 3.23:

```
for (int i = 0; i < 10; i++) {

  if( i == 5) {
    continue;
  }

  print(i);

}
```

In this case, numbers from 0 to 9 would be printed except 5, since on that iteration, continue is called, which bypasses the **`print()`** method.

Conclusion

Most of the conditionals and loops in Dart retain the familiarity while making minor changes to make them more readable. Using the appropriate elements in your code is important to implement the intended logic.

In the next chapter, we go into all types of functions which have quite a few things to make Dart unique.

Questions

1. What are the types of conditional statements in Dart?
2. What are the types of loops in Dart?
3. What is the difference between the for loop and the for-in loop?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 4
Functions and Classes

Introduction

Although Dart makes no significant breaks from convention regarding functions and classes, there is still a marked difference between classes in Dart and other languages. Dart carries over a hierarchical system akin to Java, however, it adds things like mixins to allow a varied inheritance structure for classes. Since Dart was originally envisioned as a replacement for JavaScript, functions lean more towards the JavaScript style than the Java style. This chapter explores the subtleties of classes and functions in Dart and more about the syntactic sugar provided in both.

Structure

In this chapter, we will discuss the following topics :

- Starting with functions
- Structure of a Dart function
- Optional parameters

- Positional parameters
- Named parameters
- Passing default values to optional parameters

- Functions as first-class objects
- Defining function types in Dart
- Extension methods
- Starting with dart classes
  - Constructors in Dart
  - Named constructors
- Inheritance, interfaces, and mixins
  - Basic inheritance
  - Interfaces
  - Mixins

Objectives

The objective of this chapter is to explain the nuances of writing functions in Dart as well as writing and constructing class hierarchies. It also describes how to extend and reuse existing classes and code.

Starting with functions

A function is a block of code designed to do a specific task. It helps in the cleanliness of code, avoids repetition of the same code wherever the same task needs to be done, and enables us to change/refactor logic quickly, as code in multiple places does not need to be edited for the same effect. A function may accept particular input objects as well as process and return a certain object. Once a function is complete, it returns the object if declared. (Unless the function sets up a stream of events, it returns a stream we can subscribe to and cancel when complete).

Structure of a Dart function

The structure of a Dart function more or less aligns with the conventional norm for functions; however, the difference here comes in terms of flexibility offered by the function definition. This is what a function structure looks like:

function definition. This is what a function structure looks like:

Code 4.1:

```
return-type function-name (param1, param2, …) {
    // Function code
}
```


As an example,

Code 4.2:

```
int add (int a, int b) {
    return a+b;
}
```

Dart makes several things inside the function definition optional, including the return type, parameter types, and even the braces if we simply want to use the fat arrow operator.

For example, the same function can be written as:

Code 4.3:

```
add(a+b) => a+b;
```

The return and parameter types are unnecessary here, and the fat arrow operator returns the value by default. However, the code does not necessarily make it easier to understand the purpose or the logic of the code and should be used where appropriate.

Optional parameters

Functions also allow us to make parameters optional. This can be useful when we do not necessarily need that parameter or when we can assume reasonable default values for the said parameters.

We use two types of optional parameters depending on the use case.

Positional parameters

Positional parameters are optional parameters that must follow a specific order when being passed to the function. Here is how we define a function with positional parameters:

Code 4.4:

```
void foo(int a, [int? b, int? c]) {
```

```
void foo(int a, [int? b, int? c]) {
  // Function code
}

// Valid function calls:
foo(1);
```



```
foo(1, 2);
foo(1, null, 3);
foo(1, 2, 3);
```

An essential thing to understand is that all the arguments passed to the function must be the same order as defined (a -> b -> c). In Code 4.4, to pass the third argument, you must pass the second one. What this means is that you may have to pass in a null value to the function as shown in the valid function calls. This is not particularly convenient in Flutter as many widgets have tens of optional properties, for which positional parameters would be inconvenient. In these cases, named parameters come as a solution.

Named parameters

A function needs a bunch of specific parameters. These parameters are usually identified by the order in which they appear when a function is called. However, if we have many optional parameters that need to be passed, we need to use a different way to identify parameters: by naming them. Named parameters are optional parameters that do not restrict the ordering of parameters since the parameter value is given by name, not the index of the parameter. An example of such a function is:

Code 4.5:

```
void foo(int a, {int? b, int? c}) {
  // Code block
}

// Valid function calls:
foo(1);
foo(1, b: 2);
foo(1, b: 2, c: 3);
```

Here, curly braces are used to define named parameters. When calling the function, the parameters need to be called by name. Simply adding them to the appropriate

the parameters need to be called by name. Simply adding them to the appropriate index without naming them throws an error.

Code 4.6:

```
foo(1, c: 3, b:2);
```

This is an acceptable call to the function.

Passing default values to optional parameters

Optional parameters are often used when a default case is defined, and the user may tweak it if necessary. For example, Port **8080** may be the default connection port defined, but the user may change it if required. For this reason, we need a default value to set to parameters in case the user does not pass in anything (which passes all undefined arguments as null).

To set default parameters, we can set the default values in the function definition.

Positional:

Code 4.7:

```
void foo(int a, [int b = 10]) {
  // Code block
}


Named:
Code 4.8:


void foo(int a, {int b = 10}) {
  // Code block
}
```

Functions as first-class objects

Dart was created as a true object-oriented language, meaning functions are also objects of the type Function, just as **1** is an **int** and **Hello, World!** is a **String**. This implies functions can take the place of variables in most places, for example -

Functions can be passed as arguments to functions:

Code 4.9:

```
void foo(Function x) {
  x();
}
```

Here, the function **`foo()`** accepts a function **`x`** just like we would accept a normal object in a language like Java. We call the function like a regular function invocation.

This also allows us to define variables with functions as value and pass anonymous (lambda) functions.

Code 4.10:

```
var x = () {
  print("foobar");
};
```

If we tried to pass a function into the **`foo()`** function at the top, we could either do this:

Code 4.11:

```
foo(() {
    print("Invoke function");
});
```

Or do this:

Code 4.12:

```
foo(x);
```

Where **`x`** is the function we defined.

Defining function types in Dart

Often in Flutter, we have various types of callbacks when events occur. For example, we have **`VoidCallback`**, which does not give any value as a parameter, **`ValueChangedCallback`**, which provides a value when something changes, and so on. We can define function templates with return and parameter types called function types. The callbacks mentioned are examples of function types defined for specific use cases, such as button taps or value changes.

We may also want to create our own function types: we do this using the typedef keyword. Here is an example of declaring a function-type alias:

keyword. Here is an example of declaring a function type alias:

Code 4.13:

```
typedef CustomCallback = void Function(int a);
```

Or

Code 4.14:

```
typedef void CustomCallback(int a);
```

This is very useful when creating custom components (widgets) in Flutter since we may need to pass specific data back. Creating function types helps in that process.

Extension methods

Extension functions arrived first in Dart 2.7 and were previously a popular feature among the Kotlin crowd of Android developers. The feature was requested for Dart and arrived with excitement as extension functions allow developers to extend normal classes and add functions. This not only makes development easier, but also adds cleanliness when additional functions are required as the functions look and behave as part of the class itself and not as an external member.

To create an extension method, create an extension to a class using the **`extension`** keyword like this:

Code 4.15:

```
extension CustomList<E> on List<E> {

    List<E> demoFunction() {

    // Write logic here

    }

}
```

Since **`demoFunction`** behaves as an extension of the class itself, we can call it using an instantiation of the **`List`** class itself.

Code 4.16:

```
List a = [];

a.demoFunction();
```

If we did not have this functionality, we would have to create a separate function using the following:

using the following:

Code 4.17:

```
List<E> demoFunction(List x) {

    // Write logic

}
```

The problem with this approach is that this adds cognitive load for the developer to remember function names and add code structuring issues.

Another approach we could use is to extend the normal class and add our own functions inside the new class, but this makes us use a different class (example,

`CustomList`) over the normally defined classes (example, **`List`**), a restriction which extension methods do not share.

Starting with Dart classes

A class is a template for creating objects containing their own variables, functions, and more. It is a fundamental concept in object-oriented programming (OOP) and is used to create instances of objects that share common properties and behaviors. A class can inherit code from other classes (inheritance). The contents of this resultant class can also be inherited by other classes and so on.

Dart is an object-oriented language with mixin-based inheritance. All classes can have one superclass, and all classes extend from **Object**.

Constructors in Dart

Constructors are the first function that runs in a class and are generally used to initialize class elements. Constructors often take parameters to initialize variables inside the class in this pattern:

Code 4.18:

```
class DemoClass {

    late int a;
    late int b;

    DemoClass(int a, int b) {

        this.a = a;
```

```
        this.b = b;

    }

}
```

Dart offers a way to simplify this process by adding a way to assign values to variables directly:

Code 4.19:

```
class DemoClass {

    int a;
    int b;
```



```
    DemoClass(this.a, this.b);

}
```

Named constructors

Along with normal constructors, which are simple functions with the class name, Dart also named constructors, essentially, multiple variants of a constructor, each having a name to clarify the properties of the object being initialized. In most languages, we can have multiple constructors with different options for initialization. However, there is little clarity on the difference in what the constructors are doing differently. Named constructors help us give a way to solve this problem. Take an example out of the Dart documentation for this case: a class **`Point`** with an X and a Y coordinate.

Code 4.20:

```
class Point {

    int x;
    int y;

    Point (this.x, this.y);

}
```

If we want to add an additional blank constructor with a default point starting at the origin, named constructors do a better job:

Code 4.21:

```
class Point {
```

```
    late int x;
    late int y;

    Point (this.x, this.y);

      Point.origin () {

        this.x = 0;
        this.y = 0;

      }

}
```


We can initialize an object of this class by using either of the constructors. However, it is now clearer what each constructor does.

Inheritance, interface, and mixins

We can use multiple ways to handle the hierarchy of classes and how we structure our code. Defined here are some of the most prominent ones.

Basic inheritance

Dart allows every class to extend one other class directly other than the object using the **`extends`** keyword.

Code 4.22:

```
CustomClass extends NormalClass {

    // Class code

}
```

Superclass constructors can be invoked like this:

Code 4.23:

```
CustomClass extends NormalClass {

    CustomClass() : super();
```

```
}
```

We can also invoke named constructors in the same way:

Code 4.24:

```
class NormalClass {

      NormalClass.namedConstructor();

}

class CustomClass extends NormalClass {

      CustomClass() : super.namedConstructor();

}
```


Interfaces

We also use classes as interfaces in Dart without an explicit **`interface`** keyword. To implement an interface, we simply use the **`implements`** keyword on a class:

Code 4.25:

```
class DemoInterface {

    void doSomething() {}

}

class NormalClass implements DemoInterface {

    NormalClass.namedConstructor();

    @override
    void doSomething() {
      // TODO: implement doSomething
    }

}
```

Here, we create the interface similar to a normal class, and can implement it using the **`implements`** keyword, after which we need to override the methods in the interface.

Mixins

Mixins are an addition to the usual class-type features in usual languages. Mixins are a way to share class code across multiple hierarchies. This is especially useful when we need some code shared across multiple classes. We add mixins using the **with** keyword. We can use mixins to inherit code from multiple classes:

Code 4.26:

```
mixin DemoMixin {

    void foo() {}

}

class NormalClass {}

class CustomClass extends NormalClass with DemoMixin {
```



```
    void customFunction() {
        foo();
    }

}
```

The methods from **DemoMixin** can be used in any class that adds mixin.

Conclusion

Functions and classes are an integral part of any Flutter app and are essential to learn before we proceed to widgets. By the end of this chapter, we now have a good idea of the subtleties of writing functions and classes in Dart and can proceed to higher level concepts.

In the next chapter, we go into operators which help define and simplify various operations in Dart.

Questions

1. What is the difference between a positional and a named parameter?
2. What are the different ways to inherit properties and methods in Dart?

3. What is the difference between a normal method and an extension method?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 5
Operators

Operators in any programming language instruct the compiler to perform a certain operation. This operation may be logical, relational, or mathematical in

nature. Apart from the usual operators like the dot operator (used to access objects), Dart tries to provide several other types of operators for cleaner code, syntactic sugar, and, more recently, null safety. In languages without null safety, code is often littered with null checks. Common tasks are simplified in Dart using operators, making the code concise and easier to read. In Flutter, these operators come in handy as the widget structure becomes harder to read if more and more code is involved.

We will now explore a list of operators and examples regarding them.

Structure

In this chapter, the following topics will be covered:

- The ternary operator (**`A ? B : C`**)
- The **`??`** operator
- The **`?.`** operator
- The **`??=`** operator
- The cascade (**`..`**) notation

- The fat arrow (**`=>`**) notation
- The **`~/`** operator
- The spread (...) operator
- The **`?[]`** operator
- The null-aware index (**"`?[]`"**) operator
- The null-aware cascade (**"`?..`"**) operator

Objectives

Knowing Dart operators well leads to more accurate and concise code, improving readability and reducing unnecessary lines of code. When working with null values, it often does away with the null checks and other operations like assignments and function access. After studying this chapter, you should be able to use the various operators in the Dart language effectively.

The ternary operator (A ? B : C)

The ternary operator is present in a plethora of languages, and the objective of the

The ternary operator is present in a plethora of languages, and the objective of the operator is pretty simple: do something based on a condition. If condition **A** is met, return **B**. If it is not, return **C**. This is how it would be used:

Code 5.1:

```
int level = 50;

String expertise = level > 60 ? "Expert" : "Amateur";
```

B and **C** here may be statements or values to return. This can also be done with a simple **if…else**:

Code 5.2:

```
int level = 50;

String expertise = "";

if (level > 50) {
    expertise = "Expert";
} else {
    expertise = "Amateur";
}
```


It is not essential to return some kind of value, running statements based on a condition is equally valid. This could be demonstrated as:

Code 5.3:

```
int level = 50;

level > 60 ? printExpertInstructions() : printAmateurInstructions();
```

Also, if the ternary operator is used to return something, return is used like this.

Code 5.4:

```
return level > 60 ? "Expert" : "Amateur";
```

And NOT

Code 5.5:

```
level > 60 ? return "Expert" : return "Amateur";
```

The ?? operator

The **??** operator is a null-aware operator often used for reducing boilerplate code when it comes to potentially null values. It is primarily used as:

Code 5.6:

```
String name = person.name ?? "John Doe";
```

Here, if the name attribute of the person object is null, **John Doe** is assigned instead.

The equivalent code without the operator would be as follows:

Code 5.7:

```
String name = "";

if(person.name != null) {
    name = person.name;
} else {
    name = "John Doe";
}
```


Similarly, we can also do this with the ternary operator as:

Code 5.8:

```
String name = person.name != null ? person.name : "John Doe";
```

The operator may even have some abstract use cases, such as calling other functions when the main function returns null.

Code 5.9:

```
void main() {

  doA() ?? doB();

}

doA() {
```

```
  return null;

}

doB() {

  print("Backup function");

}
```

The **??** operator makes handling unexpected null values in Dart code much easier.

The ?. operator

We often deal with nested objects in code, and checking for null values in each can quickly become a nightmare. If we have an intricate and complex object that needs a value deep inside the main object, one null value can throw errors since the further chain is broken.

As an example:

Code 5.10:

```
String firstName = mainList.users.first.bio.name.firstName;
```


If the **bio** object is null, accessing an attribute throws a whole new error. Checking null values for each object can quickly become hard since each value must be checked in succession. You could argue that in an ideal system, this error would not occur. However, real-world systems are often not ideal, and any number of things can cause things not to be as expected. As such, we need our code to be resilient in the face of these things.

It would be helpful if null were returned instead of throwing an error when hitting a null in the attribute chain. This would help us handle nulls instead of handling errors. The **?.** operator does exactly this:

Code 5.11:

```
String? firstName = mainList?.users?.first?.bio?.name?.firstName;
```

In the newer example, if any **users**, **first**, **bio**, or **name**, were null, instead of trying to access the next object, null would simply return and be assigned to the **firstName** variable. Doing this without the operator would be unnecessarily verbose.

We can combine this operator with our previous **??** operator to have a placeholder

when something is null in the chain.

Code 5.12:

```
String firstName = mainList?.users?.first?.bio?.name?.firstName ?? "John";
```

Now, when any attribute returns null, the name **John** is assigned to the **firstName** variable, automatically dealing with nulls in one single line.

The ??= operator

The ??= operator simply means *if the left-hand side is null, carry out the assignment*. This will only assign a value if the variable is null.

Imagine a point class with an **X** and **Y** coordinate value that can be assigned:

Code 5.13:

```
Point? point;

// Assignment is carried out
point ??= Point(x: 1, y: 2);

// Assignment is not carried out since point is already set
point ??= Point(x: 2, y: 4);
```


Here, we try to initialize the variable point value. At first, the point is assigned a new object since the variable is null. When it is attempted for the second time, the new value is not assigned since the point variable is not a null value.

The **??=** operator only assigns a value if the variable is null. This spares us unnecessary code since we do not need to check if a value is null.

The cascade ("..") notation

The .. operator is known as the cascade notation. Cascade notation is a simple way to change the properties of an object, usually while creating it, rather than getting a reference to the object and changing properties one-by-one.

Imagine a situation where we first need to initialize the class and then set the **x** and **y** properties. This is what is often imagined:

Code 5.14:

Code 5.14:

```
Point p = Point();

p.x = 3;
p.y = 6;
```

The cascade notation gives us an easier way to do it without using the object again to set properties.

Code 5.15:

```
Point p = Point()
..x = 3
..y = 6;
```

Note that if we had just used `the.` operator, it would not return a **`Point`** object on the first line.

What the cascade notation essentially does is:

- Create the **`Point`** object to default values.
- Change any affected values by the cascade operations
- Return the original object (here, the instantiated **`Point`** object)

This is very useful when many properties need to be set, in the Builder pattern for example.

The best example for this comes out of the Dart documentation:

Code 5.16:

```
final addressBook = (AddressBookBuilder()
..name = 'jenny'
..email = 'jenny@example.com'
..phone = (PhoneBuilder()
..number = '1234567890'
..label = 'home')
.build())
.build();
```

Cascades help us modify objects without storing references to them for further modification. This may be useful in defining **`Paint`** objects for **`CustomPainter`**, which helps paint custom graphics on the screen.

The fat arrow (=>) operator

The `=>` operator is known as the fat arrow notation in Dart. It can be used in two ways: both have to do with defining functions.

The first way can be used as a shorthand for returning something.

So `=> x` simply means `{return x;}`

Taking two equivalent examples:

Code 5.17:

```
String demoFunction (String s) {

   return s.toLowerCase();

}

String demoFunction(String s) => s.toLowerCase();
```

However, the operator also works for a single statement even without returning anything.

Code 5.18:

```
void demoFunction(String s) => print(s);
```

Note that if the return type is void, even `=> s` will not return anything.

The ~/ operator

Dart also has a few operators for speeding up arithmetic operations. The `~/` operator divides and returns the floored (integer part) of the result. As an example of use:

Code 5.19:

```
int a = 3;

int b = 7;

int c = b ~/ a;

// Prints 2
print(c);
```

The spread ("...") operator

The `...` operator is referred as the spread operator.

If a List's elements need to be added to another list, we can work with `for` loops, or we can use the `.addAll()` method present in `Iterable` collections. In addition to this, another method that makes it easier to code with full lists is the spread operator. Spread operators `unpack` a list, meaning they insert the elements themselves into the list and not the list item.

Lists can be easily unpacked with this operator:

Code 5.20:

```
var demoList = [1,2,3,4,5];

var otherList = [
  6,
  7,
  8,
  ...demoList,
];

print(otherList);
```

This will add elements of `demoList` directly to the list instead of adding it as a `List` object.

The printed list output is: `6, 7, 8, 1, 2, 3, 4, 5`

> **Note: Without the spread operator, we would have added a list object directly into the list, effectively making the list: `6, 7 , 8, [1, 2, 3, 4, 5]`**

Using these operators vastly helps us simplify code by removing additional lines and checks as well as making it more concise and easier to understand. As we continue learning Flutter, larger files become easier to read and understand using the full suite of operators provided.

The null-aware index ("?[]") operator

Once null safety was added to Dart, a couple of operators were added, which are seen here. The first is often the more useful of the two – the null-aware index operator. This allows accessing a list index if the list isn't null – and returns null otherwise. This reduces the null check usually required when accessing lists.

Code 5.21:

```
void main() {

  List<int>? list;

  print(list[0]); // This gives an error

  print(list?[0]); // Prints "null"

}
```

This operator reduces the extra code required to check if the list is null before accessing an index which was more important after null-safety was implemented. This can also be combined with the **??** to add a default value when retrieving a list index.

Code 5.22:

```
void main() {

  List<int>? list;

  print(list?[0] ?? 3); // Prints "3"

}
```

The null-aware cascade ("?..") operator

The null-aware cascade implementation is similar to the null-aware index operator that adds to existing operators (index and cascade) by accessing/modifying the

variables only if they are not null. The normal cascade operator deals with invoking several operations on a single object. If this object turns out to be null, this will cause an error. To solve this, the null-aware cascade operator was introduced.

Code 5.23:

```
void main() {

  List<int>? list;

  list?..add(1)..add(2)..add(3);
```

```
}
```

Here, the list is null, but the code will not run into an issue since the null-aware cascade operator is used. Note that you do not use the null-aware cascade repeatedly since once the null check is performed, subsequent operations can assume the object is non-null.

Conclusion

In this chapter, we learned about various types of Dart operators and further strengthened our understanding of Dart. This will help us write more meaningful code with fewer lines of code going ahead. Next, we will look at asynchronous programming in Dart which helps us deal with more complex situations, such as networking and running tasks in parallel.

Questions

1. What is the purpose of the fat arrow notation in Dart?
2. What are the various null aware operators of Dart?
3. What benefits does the cascade operator offer?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 6

Asynchronous

Asynchronous Programming

Introduction

Asynchronous programming allows you to write code that can handle multiple tasks in parallel without restricting the execution of other parts of the application. This often leads to faster and more efficient code - especially when dealing with tasks like network communication or any file operations. Async programming also allows us to make the most of the computational power now available on mobile phones. Real-world app code differs from simple programs in the fact that there are a lot more uncertainties and variables to consider. Features such as network requests, native API calls, databases, and so on, do not adhere to the usual norms of programs since they depend on a service external to application memory. This leads to the necessity of the concept of asynchronous programming when creating any kind of application.

Asynchronous code runs parallel to the main execution and uniquely have the ability to *wait*. This makes it practical to handle services outside of developer control such as an API supplying data or a database performing a query. This chapter goes into the basics of writing asynchronous code as well as understanding fundamental Flutter concepts such as isolates.

Structure

In this chapter, we will discuss the following topics:

- The need of asynchronous programming
- Understanding the async/await structure
- Looking into futures

- Handling Futures
    - Delayed futures
    - FutureBuilder
- Streams
- Isolates

Objectives

After studying this chapter, you should be able to write programs with asynchronous code. You should also have a general understanding of the concept of isolates in Flutter.

The need of asynchronous programming

When writing a normal program in any language, each statement in it is processed in sequence. Most times, each statement takes a miniscule amount of time to be completed. In a majority of frameworks including Flutter, the UI and code run on the same thread, by default. However, that also makes it possible for a program waiting to complete one instruction for a long time to simply lock the UI. Fortunately, CPUs are fast enough to not let us notice the processing happening behind the scenes. However, some processes may need time beyond what is expected and running them on the main thread would lead to the app not responding and being forced to shut down. This may be on-device computation, fetching some data from a server, and so on. For these kinds of purposes, we cannot run the same code synchronously and write code in such a way that it runs in the *background* – that is, asynchronous to the main thread.

Each language has its own implementation of asynchronous programming - we will now explore the one Dart comes with.

Understanding the async/await structure

Let us imagine that an app needs to fetch news articles from the web. For this purpose, it needs to create a network request and fetch all the data for relevant news. However, when a network request is created, it can take any amount of time to finish. If the app runs the code for fetching the articles like a simple program would, it will block the UI and the app will freeze. The hypothetical code for this function

would look like this:

Code 6.1:

```
void fetchNewsTitles() {
    var titles = loadTitlesFromCloud();
    print(titles);
}
```

To convert this to a function that can run parallel to the main execution of the app, you can use the **async**/**await** keywords.

In simple terms:

async is used to mark that a function may need to run parallel to normal code.

- **await** is used to wait for an operation to complete.

We can add the **async** keyword to the function declaration to declare that this function may run asynchronously. When the execution reaches the **await** keyword, it starts running the function parallel to the main execution. We need to use await to wait for the news titles to be fetched from the network in the previous example. Hence, the new (syntactically correct) version of this becomes:

Code 6.2:

```
void fetchNewsTitles() async {
    var titles = await loadTitlesFromCloud();
    print(titles);
}
```

Looking into futures

Looking at *Code 6.2*, the function seems to run asynchronously. However, there is an obvious question to ask here: what if we need to return data from the same function?

One way to do this would be simply returning the list of articles:

Code 6.3:

```
List<String> fetchNewsTitles() async {
    var titles = await loadTitlesFromCloud();
    return titles;
```

```
}
```

There is a problem here, the function being **async** does not know when it is going to complete. Additionally, since the function creates a network request, there is a chance that it might simply fail. Put simply, normal return types do not work with **async** functions.

To solve this, Dart has the **Future** class. In a simple sense, it promises to return a specific kind of data type in the future.

Modifying the code from earlier, we get:

Code 6.4:

```
Future<List<String>> fetchNewsTitles() async {
    var titles = await loadTitlesFromCloud();
    return titles;
}
```

The function now promises to return the list of titles when execution is complete.

Handling futures

We now have a complete function for getting news titles from the network. However, since it is an **async** function, it cannot be called the usual way. Let us take an example to see why:

Code 6.5:

```
// Call the earlier function to fetch data
var data = fetchNewsTitles();
// Print the data
print(data);
```

The issue with this piece of code is that the 'data' variable cannot be assigned immediately since fetching news titles from the internet will likely take some time. Hence, when the program execution reaches the **print()** statement above, it will have nothing to print since the 'data' variable isn't defined yet. Fortunately, this is where the **Future** class helps us deal with the data received.

The **Future** class has a callback named **then()** which can be used when an asynchronous function completes:

Code 6.6:

```
fetchNewsTitles().then((list) {
```

```
    print(list);
});
```

This avoids the issue of execution since the callback will only be invoked once the function is complete. Anything inside the **then()** callback will have access to the fetched data.

Interestingly, the **then()** callback accepts a function with one parameter. Since **print()** is also a function with one parameter, we can print the list of titles with this line of code:

Code 6.7:

```
fetchNewsTitles().then(print);
```

In this example, we are passing in a function (**print**) as an argument to the **then()** function. We pass a function reference (**print**) and do not execute the function itself (**print()**). The function reference allows the function to be executed later, whereas actually executing the function will simply pass along its return value.

As mentioned before, there is also a chance that the network request simply fails. In that case, the **then()** callback is pointless since there is no data. The Future class also offers a **catchError()** callback which runs when any error occurs in the execution of the asynchronous function:

Code 6.8:

```
fetchNewsTitles().then((list) {
    // Prints list if successfully fetched
    print(list);
}).catchError((err) {
    // Prints error if fetch is not successful
    print(err);
});
```

In case an asynchronous function takes too long without failing or finishing, the **timeout()** callback comes in handy. The **timeout()** function takes a duration and is called when the set duration is reached in executing the function.

Overall, this helps us deal with all possible scenarios that happen with asynchronous functions.

Here is a complete example combining all three possibilities:

Code 6.9:

```
fetchNewsTitles().then((list) {
    // Prints list if successfully fetched
    print(list);
}).catchError((err) {
    // Prints error if fetch is not successful
    print(err);
}).timeout(Duration(seconds: 10), (val) {
    // Prints "timeout" is set duration is reached
    print("timeout");
});
```

Delayed futures

To take an action after a set delay, we use **`Future.delayed()`**. It takes a duration which starts when the line is reached and a **`then()`** callback can be attached to do something once the duration is finished:

Code 6.10:

```
Future.delayed(Duration(seconds: 1)).then(() {
    print("Delay Complete");
});
```

We can also use this as a **`sleep()`** function by simply awaiting a delayed function for a duration:

Code 6.11:

```
await Future.delayed(Duration(seconds: 1));
```

FutureBuilder

While this section is mostly about Dart, it makes sense to discuss one more topic here for completeness: the **`FutureBuilder`**.

When using a **`Future`** in Flutter, you can use the **`FutureBuilder`** Widget to build UI based on the result of the asynchronous function:

Code 6.12:

```
FutureBuilder(
```

```
    future: fetchNewsTitles(),
    builder: (context, snapshot) {
    // Check if async function is completed
    if (snapshot.connectionState == ConnectionState.done) {

      if (snapshot.hasError) {

        // Build UI for error

      } else if (snapshot.hasData) {

     // Build UI for fetched data

      }
    }
)
```

This makes it easy to build UI based on the result of an asynchronous function.

Streams

A Stream is an asynchronous flow of data which is listened by subscribing to it. Unlike Futures, which are one-time computations, Streams can constantly be fed with data. An example of a Stream is continuous data from a sensor on a phone.

To create a simple Stream, you can use the **async*** generator:

Code 6.13:

```
Stream<int> countStream(int end) async* {
  for (int i = 1; i <= end; i++) {
    yield i;
  }
}
```

The function creates a Stream and adds numbers to the end. The **async*** generator creates a Stream from the function. The **yield** keyword adds a number into the Stream.

The **`Stream`** class has a **`listen()`** function which allows you to provide a callback and do something when a new value is added to the Stream:

Code 6.14:

```
countStream.listen((data) {
    print(data);
});
```

There are also additional callbacks for when a Stream is done or has an error:

Code 6.15:

```
countStream.listen((data) {
    print(data);
}, onDone : () {
    // Completed
}, onError : (err) {
    // React to error
});
```

There are two main types of Streams: single-subscription and broadcast.

- A single-subscription Stream will buffer all the events received until there is a listener.
- A broadcast does not buffer events and simply passes along new events to all existing listeners.

Isolates

One important thing to remember is that Dart is a single-threaded language. Async functions also run on the same thread as the UI does. But what if we want to do more intensive tasks that may block it? We use isolates.

Note: Isolates are not needed for general Flutter development. For now, we will only explore the surface.

Each isolate consists of a thread and allocated resources. The main thread can also be called the main isolate. Since isolates are independent, there is a separate process to send and get messages from isolates. We communicate using ports by using **`SendPort`** to send messages and **`ReceivePort`** to receive them.

We use **`Isolate.spawn()`** to create new isolates which then occupy a thread of their own. Isolates also need to be destroyed when completed since they are computationally taxing. A simple example of spawning an isolate is:

Code 6.16:

```
// Spawning an isolate
var isolate = Isolate.spawn(heavyFunction, 'id');

// Killing an isolate
isolate.kill(priority: Isolate.immediate);
```

For most tasks in Flutter, we have no direct use of isolates as the normal async/await does tasks without a hitch and does not make the app jittery. However, for cases like image processing, we may just want another isolate to perform the heavy work for us and not interrupt the main isolate.

Conclusion

Comprehending asynchronous programming opens new paradigms in building an app. It is essential for everything from networking to databases. For advanced apps, isolates and parallel programming are often required and help speed up the app by leaps and bounds. Feel free to try working with these concepts yourself to get a better feel of how asynchronous programming is different from the usual programs you may be used to.

This marks the end of the Dart section of the book. We now proceed to Flutter development. In the next chapter, we go into the historical context and reasons for the development of Flutter itself. The chapter also dives into the features that make Flutter unique.

Questions

1. Why is asynchronous programming needed?
2. What is the use of the async keyword?
3. What does an isolate achieve that a simple asynchronous method cannot?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 7
Why Flutter?

Introduction

Given the ubiquity of mobile application development frameworks, it is easy to ask why anyone should also consider learning/switching to Flutter. This question is answered better by studying frameworks that existed previously around Flutter. Since developers have limited time on their hands and companies preferably want little change in their technology stack, there needs to be a solid value proposition that Flutter offers over other frameworks. In this chapter, we try to understand how Flutter came to be, by studying the drawbacks of other solutions and the unique features Flutter offers.

Structure

In this chapter, we will discuss the following topics:

- Another framework
- A look back in time
- Understanding mobile applications
- Hybrid vs. cross-platform
- The rise of hybrid frameworks

- Things we can do differently
- Enter Flutter
- Hot restart, hot reload
- Docs, support, and community

Objectives

After studying this chapter, you should understand why the Flutter framework was developed and how it differs from other popular frameworks. Features that make Flutter unique and the underlying technology are also explained.

Another framework

Flutter is the latest in a long line of mobile application development frameworks[1]. Native solutions, along with cross-platform development frameworks have been making the rounds ever since apps became the new hot thing:

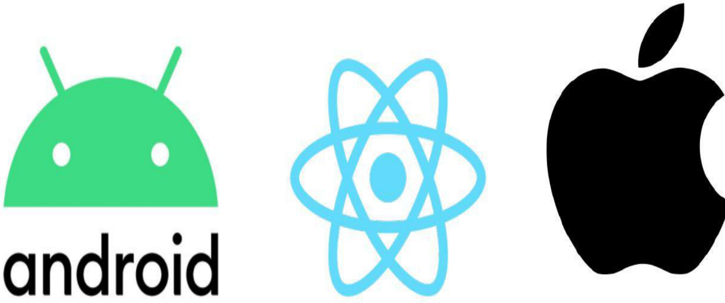


Figure 7.1: Common app development frameworks

Developers will probably know of other frameworks like PhoneGap, Ionic, Cordova, and React Native. Web developers might wonder if mobile is going the same route that web development has gone - multitudes of frameworks span the entire English vocabulary, and new ones are coming out every month. Android developers may be understandably confused by Google's decision to have multiple ways to develop Android apps – the native way and the Flutter way. This confusion is compounded by the fact that these methods may end up competing with each other in some

1 **Important**: While called a framework here to generalize it with its competitors, Flutter does not technically adhere to the strict definition of a framework. Rather, it can be called a UI toolkit, or an SDK since Flutter does not actually build the app from the bottom up and depends on the platform-tools to build the "shell" or container in which the Flutter engine runs and rendering is done on anything similar to a Canvas (in most cases).

ways. To understand why Flutter is needed even though more mature native app development solutions exist, we need to understand the problems in a true cross-platform (and even native) mobile app development. Only then can we truly understand and appreciate what Flutter as a framework fundamentally changes in the mobile development process. To understand these problems, let us walk down history lane and look at the earliest mobile development frameworks and how they evolved.

A look back in time

The web is quintessentially cross-platform. Since the old days of the web, most browsers have been able to render the same websites. Here is one of the first ones:

Figure 7.2: Netscape – one of the first web browsers

Netscape

Web pages written once run (mostly) look (mostly) the same. This is not to say that no setup is required for specific browsers, systems, or backward compatibility, but it mostly works fine. If you have seen the word *mostly* a lot in the last few lines, there is a reason: cross-platform is hard. The number of operating systems, hardware types, software installations, and languages complicates things. Supporting everything at once is a double-edged sword, you need to give up certain things to achieve it. Let us think about the web. Why does the Facebook mobile app exist when there is already a Facebook website that runs just fine on both Android and iOS? Before we look at why certain frameworks exist, let us look at why mobile apps exist at all.

Understanding mobile applications

If you have asked the question about what a mobile app does better compared to a website to a normal non-tech audience, the answers would probably be things like: *the app is smoother*, *there is less junk*, *it is easier to use*, *it has more features*, and so on

We can agree to the general answers, but the reasons are more important. The mobile application has lesser junk and a smoother UI for several reasons. The code that runs on the web simply has to run through more layers, while the native code running on the mobile app does not. However, there are many things that non-tech people miss out on, such as native API access, push notifications, GPU access, and so on. While there have been strides in getting access to some of these, it is still a restricted way of doing so: for example, getting notified by websites is managed by the browser itself. Websites cannot run tasks in the background or while the browser is closed. It is easily noticeable that websites on mobile are heavily dependent on the shell they run in and do not have an independent existence. While websites have come a long way, the latest succession being *Progressive Web Apps*, an attempt to bring some capability of native mobile apps to the web, the functionality of native apps is still unchallenged.

Now we are somewhere close to the main definition of native mobile apps.

Native mobile apps are programs that have the following:

- Easy access to device APIs such as camera, microphone, location, and so on.
- Capability to observe and react to system events
- Ability to run background tasks and push notifications
- GPU access for rendering

We will stick with this definition for now.

The first mobile apps

While there is significant and amazing work done in the early days of mobile phones with things like PalmOS and Symbian, the main topic in this section is the rapid change in development strategies after the release of the first iPhone and the Apple app store.

While there were smartphones before the iPhone, they were clunky, and using apps like Amazon on that kind of interface and screen size would make you wait until you got home and use your home desktop computer instead. So, developers focused on making utility apps and games instead of full-fledged apps like today. There were similar memory and computation constraints, but the interface outweighed them. The launch of the iPhone and similar phones made it possible for a new

paradigm of apps to be developed. But funnily enough, the first iPhone did not support third-party apps at launch! After developers requested access to an SDK, Apple recommended developers build web apps instead. While this changed over time, it is ironic that web apps were recommended before native apps.

Android and iOS soon launched native app development frameworks, and mobile application development became a lucrative business opportunity. Close to all large companies had (and often still have) mobile development divisions consisting of Android and iOS developers cranking out new updates.

However, a certain subset of companies was in a dilemma. They did not need access to the device APIs, or heavy computation and did not have much incentive to develop apps in general. A lot of these firms were from the field of e-commerce: listings and filters are not things that the full power of mobile apps is needed for. However, the mobile web at the time was also not as powerful to be a completely viable alternative. These companies needed something as close to a single codebase as possible while exploiting push notifications. This led to the development of the first hybrid frameworks to bridge this gap between platforms.

Hybrid vs. cross-platform

Before we go ahead, we need to distinguish between two words you might often read when talking about frameworks that span across platforms. Both have critical differences and can even make or break your product, so discussing it is relevant before learning examples of the individual types.

The central theme connecting hybrid and cross-platform app development frameworks is code shareability, and the way both these types achieve the same result is worlds apart.

Hybrid apps use a blend of mobile and web components to create the application, often using a web view to render the components on the screen. Since web components are inherently cross-platform, the code is shareable by default. Hybrid apps usually have a mobile app shell used to integrate features like push notifications. These are written in the native language of the respective platform. This mixing of the two realms makes it a *hybrid* rather than a native framework.

Cross-platform, on the other hand, usually makes no use of the web components and does not have a web view rendering UI. Cross-platform frameworks use a native rendering engine to render the elements on the screen. (This may or may not include the native mobile components that the OS provides, as we will see when we dive into React Native vs. Flutter) Cross-platform frameworks are usually more performant than their hybrid counterparts and offer better device API access.

While hybrid frameworks were considered an alternative to a certain set of apps, they were not considered a true replacement for the native Android and iOS frameworks. While they worked well for e-commerce style apps, and quite a lot are still in use today, they did not make much headway in general app development. Cross-platform frameworks, however, are changing this perception and fast, making apps nearly, or as performant as their native counterparts with drawbacks shrinking all the time.

The rise of hybrid frameworks

Attempting to solve the problem of having multiple codebases for the same project, a multitude of hybrid solutions arose and tried the same problem in similar but subtly different ways. Many of these frameworks are built upon older hybrid frameworks to implement better features and performance. For example, Ionic was built on top of Apache Cordova (formerly PhoneGap):

Figure 7.3*: Ionic – a popular hybrid app development framework*

These frameworks from 2010 to roughly before 2015 gave a cross-platform solution. Still, even while lacking functionality, it was more of an attempt to bring the web to mobile than making mobile code cross-platform.

Let us take a basic look at the architecture of a hybrid app of this type:

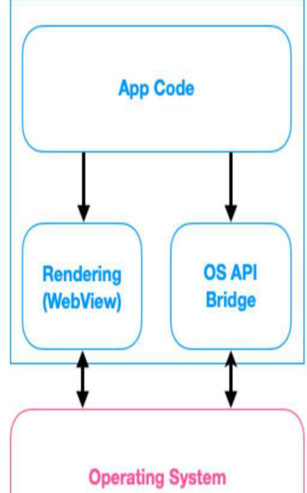


Figure 7.4: Simplified architecture of a hybrid app

While a native app does not have any wrappers for rendering such things, hybrid apps use a different degree of native and web components. Even with a large amount of optimization, this often does not match up with native components.

Things we can do differently

After perusing the pure web and hybrid approaches, maybe going through the web rendering is not the best and most performant way to render our UI. We can probably understand that at least for UI, the OS components, either offered by or rendered by the native platform offer the best performance. These mobile components mean the actual views of the operating system that offer: the components for displaying text (`TextView` on Android, `UILabel` / `UITextView` on iOS), images, lists, and so forth. However, there is no real way to write code that can run and create the respective component on each*. So let us try to imagine a potential solution to this problem.

Here is what our imaginary solution needs to do:

- Render natively
- Use mobile OS components
- Have full API access

Let us draw a flowchart for this:

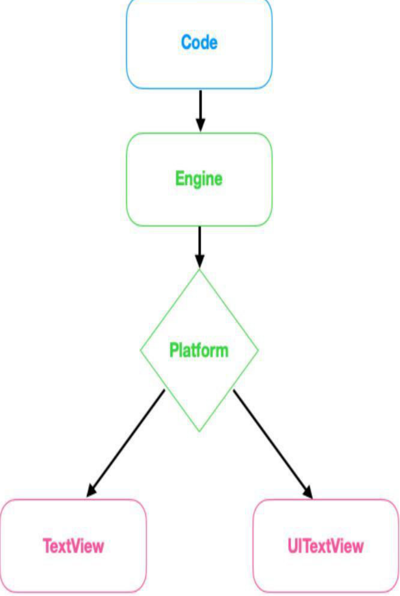


Figure 7.5: Imagining a better architecture than a hybrid app

Ultimately, we want each platform's output to be a mobile component. First, we use a language (This may or may not be one of the languages that the underlying

platforms use. Either way, the code is not directly going to be run through the native engines) defining a component that is common to both platforms. The goal is to pass this block through an engine where we write those outputs, the actual mobile component code: For example, we define a block called **Text**. This, when passed through the engine, outputs a **`TextView`** for Android and a **`UITextView`** for iOS. Now, the only added weight to the app is the intermediary engine, and the code going through the engine is truly cross-platform since it stays the same for either app.

If you had not guessed it yet, this foreshadowed the next framework on this journey: React Native.

> **Note: Here, *no real way* refers to a language not being able to be understood by all OS UI renderers as-it-is rather than using an intermediary language that is understood by a third-party engine in the app.**

Going towards true cross-platform: React native

React Native was launched in 2015 after Facebook CEO *Mark Zuckerberg* admitted betting on the web over mobile was a mistake and started a project to develop a framework of its own in 2012:

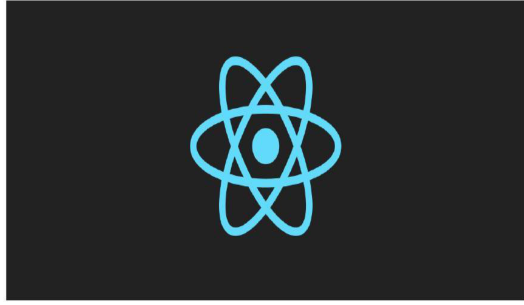


***Figure 7.6**: React Native – one of the most popular cross-platform frameworks*

React Native is used for a majority of Facebook's own projects, such as the Facebook app and Instagram. However, oddly enough, the entire Facebook or Instagram app is not written in React Native, only certain portions of it are. Nevertheless, React Native is an extremely powerful framework that appeals to a large audience - especially web developers looking to develop native apps. It has a large community and a huge number of packages and is considered a mature development framework. React Native was a significant departure from the normal way of approaching cross-platform and brought a slew of changes. However, for a few reasons seen later,

we will refrain from calling it a truly cross-platform framework and consider it somewhere between the hybrid and true cross-platform approaches.

How does React Native work?

React Native roughly follows the architecture of the hypothetical way to do cross-platform apps in the earlier section. Native mobile components are used on the respective mobile platforms quite interestingly. React Native uses a language called JSX which can be considered an extension to JavaScript so that HTML/XML elements can be included inside the code. This code is then parsed and converted into mobile aspects of the platform that the code is running on. To access the underlying platform, React Native uses a JavaScript bridge. This bridge is used for rendering as well as on-device API access:

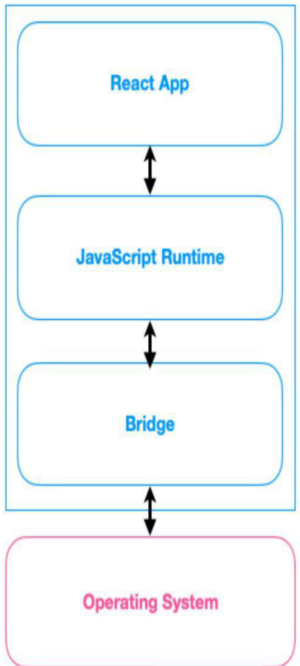


***Figure 7.7**: Simplified architecture of React Native*

This gives React Native access to native components for rendering and on-device APIs. Problem solved, right? Well, not quite. The native components used for rendering are more performant than an app rendering with a web engine. However, the concept of the JavaScript bridge proves to be a bottleneck. This bridge actually slows down the app itself, and now we are stuck in a Catch-22: Do we render on the web or use native components which prove hard to load all at once? This is the main problem that Flutter was developed out of.

Along with these problems, several other things also plagued React Native development. Developers with native knowledge were almost always required to develop apps since native modules were often needed. Along with this, making custom components often became harder than developers expected it to be. Several companies like Airbnb, tried going with React Native but switched back to native

later. It still is not the sunset for React Native, and around 2019, React Native started to change its architecture and streamline the framework: only time will tell how good it is at mitigating its problems.

Enter Flutter

Before we dive into the details, here is some background: Flutter is an open-source, cross-platform SDK released in end-2018 and maintained by Google. It was first known as Sky and then later named Flutter. It was developed by members of the Chrome team and not the mobile team, and it uses the Google graphics library known as **Skia**.

***Figure 7.8**: The Flutter Logo*

Flutter, on the face of it, sounds very promising: a framework that promises to develop apps for Android, iOS, Fuchsia[2], Web, and Desktop (MacOS, Linux, Windows) that have already proven to run on other platforms like Raspberry Pi, although official support is not in the works for the latter. This is appealing to beginners, developers, and companies alike for obvious reasons: learning one thing can help develop all kinds of products without reskilling required for any employee or having separate teams working on separate products. A single codebase can work wonders. Some development teams opt to maintain separate codebases for each platform even in a Flutter app. Since these applications are all written with Flutter, it is much easier to shift developers across these codebases. Flutter also offers faster development time than native development methods with features like hot reload, which allows for near instant reload times. Reloading the entire app in a few seconds is something that feels like a godsend to developers tired of Gradle build times. While this is often enough reason for several groups and companies to give it a spin at least, the underlying technology marks it as truly different from the other frameworks in the same space. Flutter takes a radically different approach to solve the problem described throughout the chapter. Since the solution is extensible, supporting new

platforms is easier than ever.

What does Flutter do differently?

2 Fuchsia is an operating system under development by Google that works jointly for mobile and desktop formats and is said to be developed to solve the shortcomings of Android and the fact that Google also does not have a viable alternative to desktop as of yet.

The problems plaguing the previous frameworks showed that neither web-based rendering nor direct native mobile components were truly effective in making cross-platform apps. The question is, what other alternative is there?

Every OS usually has a canvas-style component: a view where a user can paint their own points, lines, circles, squares, and by extension, UI:

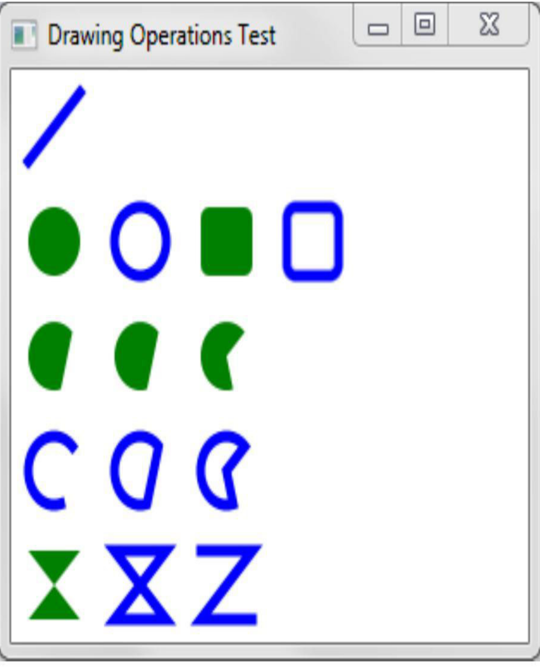


***Figure 7.9**: A figure depicting rudimentary canvas operations*

Flutter does not go with web-based rendering or native components, but paints the entire UI on the OS's canvas component. This is where the *Skia* graphics library mentioned earlier comes into play. The obvious question is: Does not the logic for painting all UI components add quite a lot of weight to the app itself? It obviously cannot be zero, but through clever optimization, a Flutter app is a few MB heavier than an Android app at the Hello World level. Still, this weight does not increase with more components, primarily because of the Dart VM and Flutter C/C++ engine. While this may rule Flutter out for making ultra-light apps, it is a small compromise to make in exchange for cross-platform support and ease of development since most apps still clock at 20 MB – 100 MB. With phones getting more space with newer

apps still clock at 20 MB – 100 MB. With phones getting more space with newer iterations, it is not as problematic as it would have been in earlier years.

Instead of JSX as the top-level language, Flutter uses Dart (or Dartlang). Dart is a language created by Google to create a JavaScript alternative, but it failed to match the expected response. When the creators of Flutter were analyzing the best language for Flutter, Dart matched all the expected specifications, and the use of Dart has been

skyrocketing since the launch of Flutter. Since we do not need native OS components now, the blocks (widgets), for say, *Text* does not need to be transformed and can directly be fed into the Flutter engine.

Flutter solving React Native's issues

The main issues plaguing React Native were excessive native calls and the JavaScript bridge. Since Flutter requires no native calls to render UI on the screen, calls are reduced excessively, and the equivalent *bridge* can only be used for device API calls. Flutter is optimized to run at 60 frames per second.

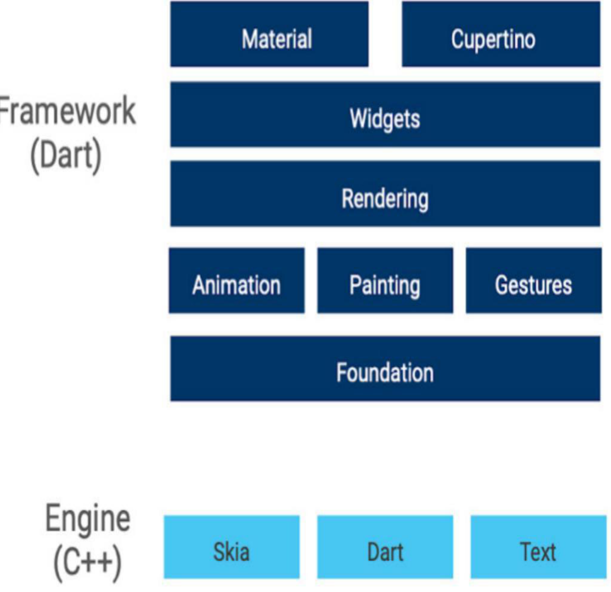


Figure 7.10*: The architecture of Flutter – from the official docs*

Flutter has in-built themes and elements like the Material design standard from Google and Cupertino elements from Apple. The layout consists of a tree of widgets (covered in depth in a later chapter) and is painted inside the app itself rather than relying on any native elements or web engine. *Figure 7.10* explains the Flutter

elements inside the app. The lower level is responsible for painting the final UI onto a canvas. At the same time, the framework works on breaking down components into smaller units, handling events such as taps and drags, and optimizing the app by refreshing elements only when needed.

The equivalent to the *bridge* in Flutter is the Platform Channel, which connects the app to the underlying platform:

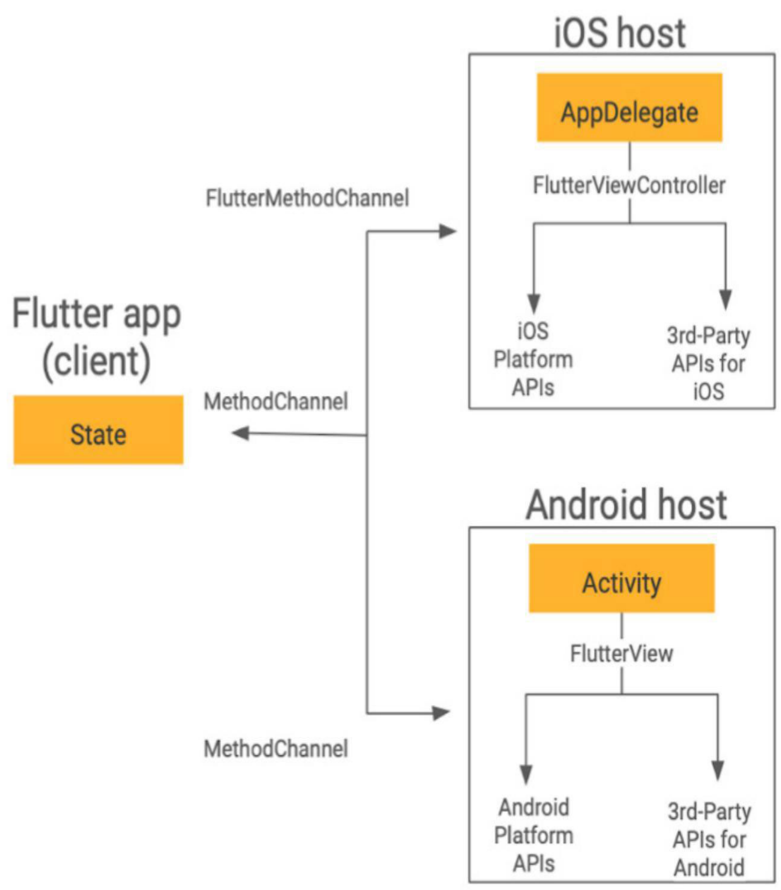


***Figure 7.11**: How Flutter interacts with native hosts – from the official docs*

It is important to remember that a Flutter app consists of a respective app on the platform and Flutter running inside it, that is, A Flutter project consists of separate modules for Android, iOS, Web, and so on. We can also write native code (if needed) and invoke it with the same platform channels defined. However, Flutter apps need

and invoke it with the same platform channels defined. However, Flutter apps need native code far fewer times than other frameworks. If all plugins exist, the native code only consists of a basic setup like setting up the **`AndroidManifest.xml`**, **`Info.plist`**, and Gradle files.

In summary, Flutter can offer blazing-fast UI by rendering all UI on a canvas with an engine in the app. It can also access native APIs through Platform Channels and run on various platforms that are either supported or under development. Platform Channels come as **`MethodChannels`** and **`EventChannels`** – the first is a single asynchronous call, while the latter is a continuous data stream. This allows all kinds of data to be sent from the platform to the application and vice versa.

However, with every platform, tech is often not the end of the story. A framework only exists if the people supporting all aspects of it exist. We need people to document the framework from the maintainers' side and write support posts and answers on forums like StackOverflow. The strength of this support and the community, in general, defines how easily new developers can join the ecosystem. Let's take a look at the community around Flutter.

Hot restart, hot reload

One of the most loved features of Flutter is the ability to make changes and have the app reload in less than a second. Coming from backgrounds like native Android development, this is revolutionary. This is because, in development mode, Flutter carries all code and assets, used or unused, into the APK or IPA, making the development mode app size as large as 100 MB or more. (No need to worry, this is only for development) When the app code is changed, the new Dart code is injected into the app, so there is no need to build the APK/IPA or equivalent file again, saving build time.

Another neat trick in Flutter is that compilation in development is **Just-In-Time** (**JIT**) to show changes instantly but switches to **Ahead-Of-Time** (**AOT**) in release to make app start-up time much quicker. The release mode build directly contains the native ARM code to make the app much faster.

Docs, support, and community

The Flutter team at Google has done exceptionally well in documenting every aspect of Flutter from the absolute basics to detailed technical overviews of the entire Flutter SDK. This is evident by their website, and surveys conducted quarterly by the Flutter team. However, this is complemented by a large number of developers writing explainers and detailed articles on every aspect of Flutter.

***Figure 7.12**: 'Flutter Community' – one of Flutter's most popular publications based on Medium.com*

Technical writing about Flutter mostly started on Medium and then branched out to many more platforms such as dev.to and similar. The preceding figure shows Flutter Community, the largest publication on Medium for Flutter, ranking even higher than the official Flutter publication. Being beginner-friendly, new technical writers join every day. To their credit, the members of the Flutter team often give shout-outs to great content and even feature recommended community-written content on their official website: **`flutter.dev`**.

Since Flutter is open source, the repository exists on GitHub and has more stars than the much older React Native repository. Although not a perfect metric, it shows the amount of interest given to the repository and ecosystem in general. The sheer amount of activity on the repository can be seen easily by the issues filed. There are multiple branches maintained, such as stable, master, beta, and dev, to give the correct amount of stability to each kind of project.

Regarding packages and plugins available, (find Flutter packages on the official source **`pub.dev`**) Flutter has some way to go before catching up to older frameworks. However, this is happening very quickly, and writing a package is not that different from writing an app, and developing your own is far easier than competing frameworks.

Conclusion

Understanding the history behind the development of a framework can allow a developer to utilize its power completely. This knowledge is important since companies may also need a set of reasons to expend resources converting/building their projects in a new framework.

Now that we have understood the actual need for Flutter to exist and how it solves the problems plaguing the cross-platform domain, in the next chapter, we start development in Flutter by first analyzing a Flutter project itself.

Questions

1. What are some other frameworks for developing mobile applications?
2. What is the difference between cross platform and native development?
3. What features does Flutter offer over native development?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 8
Installing Flutter

Now that the basics of Dart and the need for Flutter are clear, the next step is to start with proper Flutter development. To do this, Flutter and its related tools need to be installed on your development device. Flutter does not currently have its own IDE and uses other widely used IDEs such as Android Studio or VS Code. Note that while XCode is a necessary installation if you want to carry out iOS/macOS development, it is not needed every time since Android Studio itself can open and

run Flutter apps on the iOS simulator.

The thing to note here is that while Flutter itself can run on any device, creating builds for Apple devices can only be done through Apple machinery. While some aspects of Apple devices can be mocked through Android emulators, this does not fully test Apple features. Flutter development can currently be done on macOS, Windows, Linux, and ChromeOS.

Another thing to be mindful of is that Flutter underneath uses the usual package files such as the APK/IPA/, and so on. This means the tools or IDEs underneath must be present on the development device - such as XCode for iOS development.

This chapter focuses on getting Flutter development tools set up on your device.

Structure

In this chapter, we will discuss the following topics:

- Initial setup
- Installing for Windows
- Installing for macOS
- Installing for Linux
- IDEs for Flutter development
- Setting up Android Studio
- Setting up VS code
- Flutter extensions

Objectives

After studying this chapter, you should be able to install Flutter and Dart on your development device. This includes setting up your preferred IDE for development and any plugins and extensions that may speed up or simplify the development process for your Flutter application.

Initial setup

Initial setup

The recommended initial installation of Flutter consists of downloading the current stable version of Flutter from the **flutter.dev** website. After that, there are a few other steps, like adding a path variable so that *flutter* becomes a recognized command and setting up individual platform development.

This section details the steps for the three main operating systems used for Flutter development - Windows, macOS, and Linux.

Installing for Windows

Flutter's main website (**flutter.dev**) offers the latest installation bundle for Flutter, available on the installation page. To start with, download this ZIP file and unzip it in a location you can remember:

flutter_windows_2.10.4-stable.zip

Figure 8.1*: The button to download the stable version of Flutter for Windows*

Flutter has multiple branches, such as master, beta, and stable. You can also get the installation bundles for the other branches on a related page:

Beta channel (Windows)

Select from the following scrollable list:

Flutter version	Architecture	Ref	Release Date	Dart version
3.1.0	x64	bcea432	26/5/2022	2.18.0
2.13.0-0.4.pre	x64	25caf14	6/5/2022	2.17.0
2.13.0-0.3.pre	x64	5293f3c	28/4/2022	2.17.0
2.13.0-0.2.pre	x64	8662e22	20/4/2022	2.17.0
2.13.0-0.1.pre	x64	13a2fb1	14/4/2022	2.17.0
2.12.0-4.2.pre	x64	5c931b7	6/4/2022	2.17.0

Figure 8.2*: Listing of versions for the beta channel of Flutter for Windows*

Next, we add the flutter/bin folder to the **PATH** environment variable. This allows for the **flutter** command to be run from all places. To do this, select **Edit environment variables** from the **Start** menu, as shown in the following screenshot:

Figure 8.3: The edit environment variables option in the Start menu

You can then add the full path to the flutter/bin folder in the unzipped file:

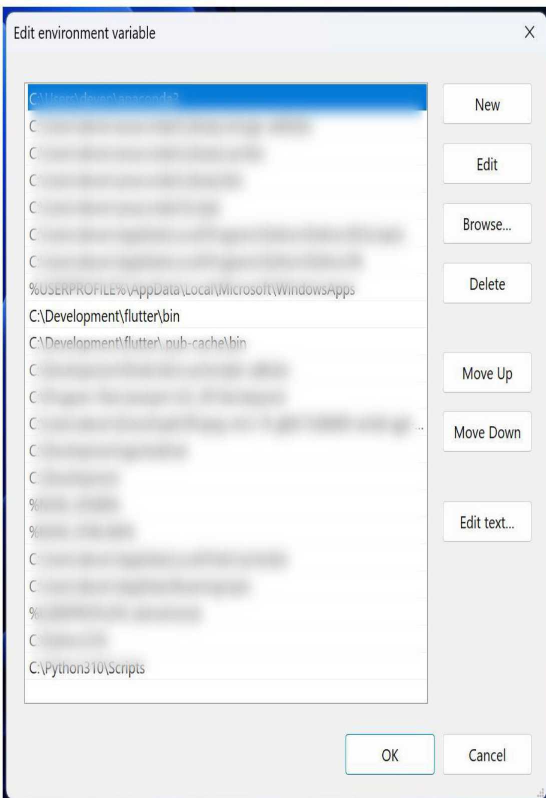


Figure 8.4: *Adding the bin folder path to environment variables*

Installing for macOS

Flutter's main website (**`flutter.dev`**) offers the latest installation bundle for Flutter, available on the installation page. To start with, download this ZIP file and unzip it in a location you can remember:

flutter_macos_2.10.4-stable.zip

Figure 8.5: The button to download the stable version of Flutter for macOS

Flutter has multiple branches, such as master, beta, and stable. You can also get the installation bundles for the other branches on a related page:

Beta channel (macOS)

Select from the following scrollable list:

Flutter version	Architecture	Ref	Release Date	Dart version
3.1.0	x64	bcea432	26/5/2022	2.18.0
3.1.0	arm64	bcea432	26/5/2022	2.18.0
2.13.0-0.4.pre	x64	25caf14	6/5/2022	2.17.0
2.13.0-0.4.pre	arm64	25caf14	6/5/2022	2.17.0
2.13.0-0.3.pre	x64	5293f3c	28/4/2022	2.17.0
2.13.0-0.3.pre	arm64	5293f3c	28/4/2022	2.17.0

Figure 8.6: *Listing of versions for the beta channel of Flutter for macOS*

Next up, we add the flutter/bin folder to the path. This allows for the **`flutter`**

command to be run from all places:

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

You also may need to add this to your **.zshrc** or **.bshrc** file, depending on if you use the **zsh** or **bash** shell.

Installing for Linux

Flutter's main website (flutter.dev) offers the latest installation bundle for Flutter, available on the installation page. To start with, download this zip file and unzip it in a location you can remember:

flutter_linux_2.10.4-stable.tar.xz

***Figure 8.7**: The button to download the stable version of Flutter for Linux*

With Linux, you can also use **snapd** to install Flutter:

```
sudo snap install flutter --classic
```


Flutter has multiple branches, such as master, beta, and stable. You can also get the installation bundles for the other branches on a related page:

Beta channel (Linux)

Select from the following scrollable list:

Flutter version	Architecture	Ref	Release Date	Dart version
3.1.0	x64	bcea432	26/5/2022	2.18.0
2.13.0-0.4.pre	x64	25caf14	6/5/2022	2.17.0
2.13.0-0.3.pre	x64	5293f3c	28/4/2022	2.17.0
2.13.0-0.2.pre	x64	8662e22	20/4/2022	2.17.0
2.13.0-0.1.pre	x64	13a2fb1	14/4/2022	2.17.0
2.12.0-4.2.pre	x64	5c931b7	6/4/2022	2.17.0

***Figure 8.8**: Listing of versions for the beta channel of Flutter for Linux*

Next, we add the flutter/bin folder to the **PATH** environment variable. This allows for the **flutter** command to be run from all places. Open or create the **rc** file for the shell your device is using and add this line:

```
export PATH="$PATH:[PATH_OF_FLUTTER_GIT_DIRECTORY]/bin"
```

IDEs for Flutter development

Which IDE to use for development can be quite a contentious topic between developers since most are familiar with one, and moving to another can be quite a significant context shift. The main IDEs for Flutter development are Android Studio, VS Code, and IntelliJ IDEA. Since IntelliJ IDEA and Android Studio are based on the same base IDE, the extensions section ahead only discusses Android Studio and VS Code.

Setting up Android Studio

Android Studio is the most popular IDE for Flutter. It might be easier to use if you are from an Android Developer background or have experience working with JetBrains IDEs. To get started with Android Studio, go to the Android Studio download website (**https://developer.android.com/studio**) and download the latest Android Studio build:

Figure 8.9*: The website view when downloading Android Studio*

Follow the Android Studio installation steps, and install all necessary Android components since it is necessary to run Android apps on Flutter. You can also install preview builds for Android Studio, however, check compatibility before you do.

After installation is complete, install the Flutter and Dart plugins from **Preferences | Plugins**.

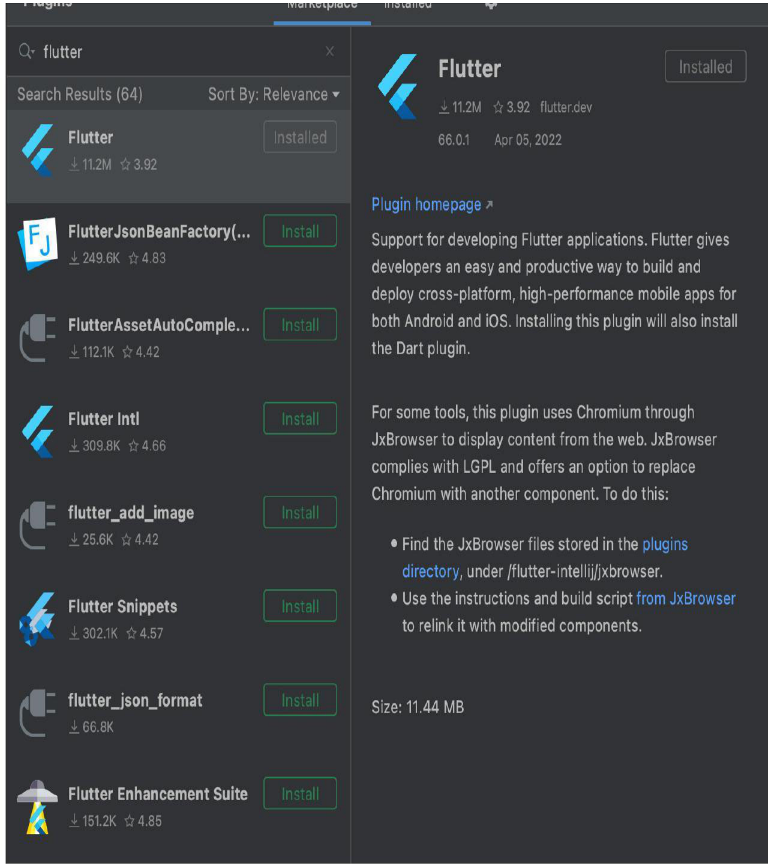


Figure 8.10: A view of the JetBrains plugins marketplace

Downloading the Flutter plugin should also trigger a question about downloading the Dart plugin, so you may not need to install it separately.

Once the plugins are setup, there are various changes for Flutter apps around the IDE - such as new project creation options for Flutter apps:

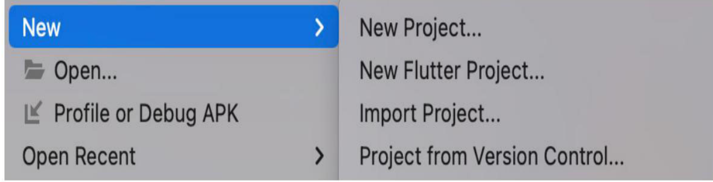
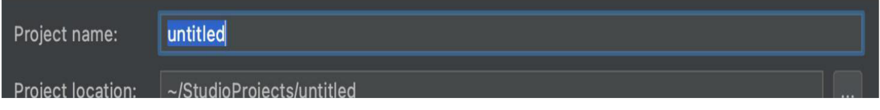


Figure 8.11: The new options for creating projects

This gives a new window with configuration options for Flutter apps, including the platforms targeted by the created app:



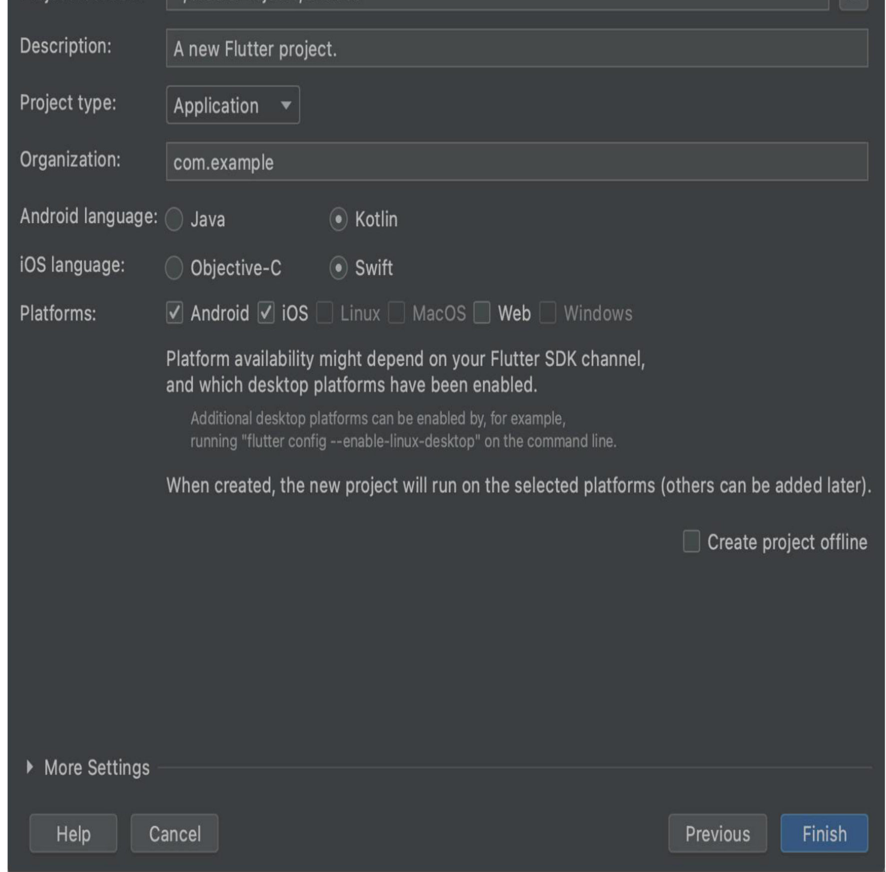


Figure 8.12: Options are shown when creating a new project

Additionally, it also adds other emulator options for Flutter, such as Chrome, iOS simulator, and so on, which are usually not accessible from Android Studio:

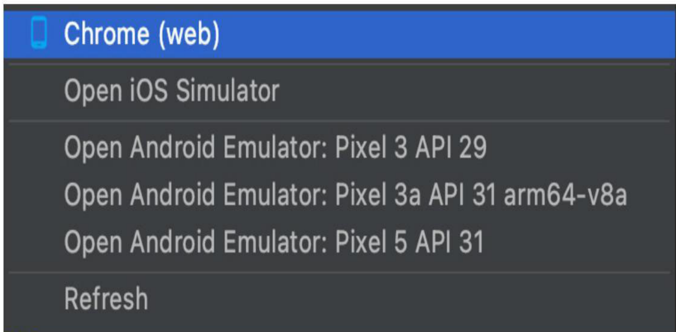


Figure 8.13: Options for emulators/simulators in Android Studio

There are other additions, such as the ability to create Dart files from the options menu directly.

Following these instructions should allow you to develop Flutter apps with Android

Studio on your development device. There may be additional steps for the platforms, such as installing XCode for iOS development and the Android SDK for Android development. Please note that you may also have to accept Android licenses using the command: flutter doctor --android-licenses.

Setting up VS Code

VS Code is another popular IDE for Flutter (and many others) development. It is a lightweight IDE and supports all kinds of extensions. To get VS Code, go to VS Code's download site (**https://code.visualstudio.com/Download**) and download the IDE for your OS:

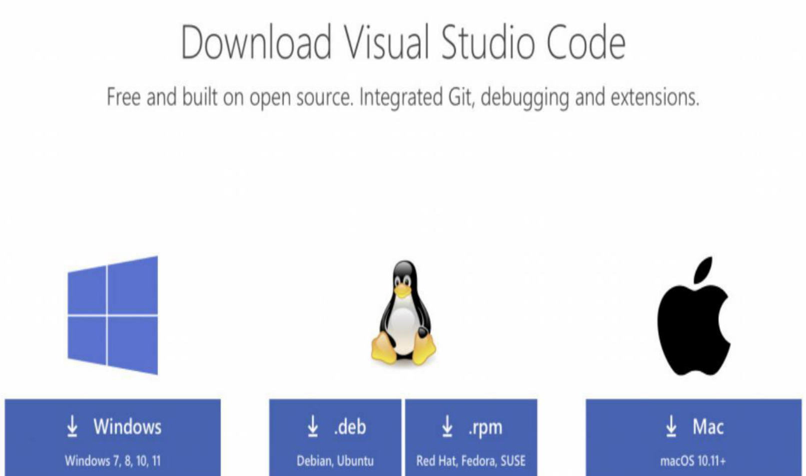


Figure 8.14: Download view when downloading VS Code

Once opened, you need to open the **Command Palette**:

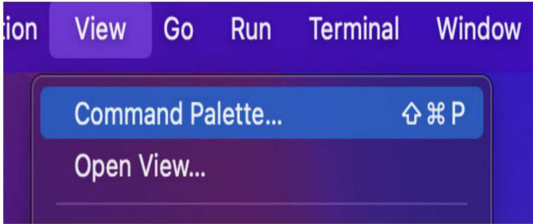
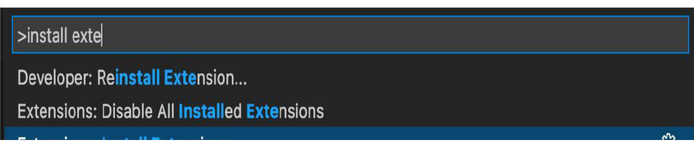


Figure 8.15: Instructions to open the Command Palette

And then go to install extensions so that we can install additional plugins:



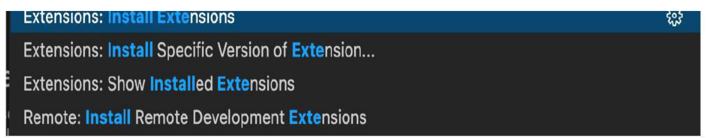


Figure 8.16: Opening the Install Extensions menu

The Flutter plugin is the official way to get started with Flutter development in VS Code:

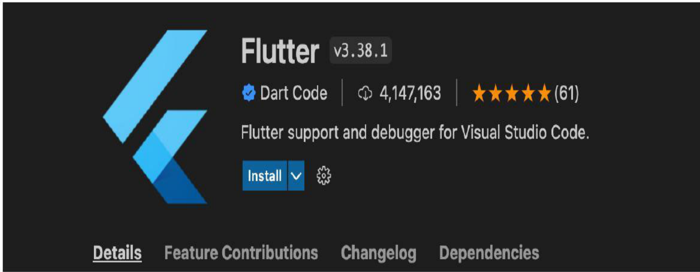


Figure 8.17: The official Flutter plugin for VS Code

This also installs the Dart plugin, which is needed as well. VS Code should now be installed and ready for Flutter development.

Flutter extensions

Each IDE offers plugins that simplify Flutter development in various ways, from starter code to easy shortcuts. The following sections describe the best plugins/ extensions for VS Code and Android Studio in detail.

VS Code extensions

VS Code contains extensions that allow adding extra functionality to the editor. You can find extensions for all kinds of languages and frameworks; however, we focus on the most useful extensions for Flutter.

There are a lot of great extensions that will save developers a lot of time. However, not all good extensions can be listed here, but it is a good idea to look for yourselves. Let us start by looking at popular VS Code extensions for Flutter.

Awesome Flutter snippets

Awesome Flutter Snippets is one of the most popular VS Code extensions for Flutter, which is an easy way to eliminate the boilerplate code associated with a lot of Widgets in Flutter. You can find it on the VS Code Extensions marketplace:

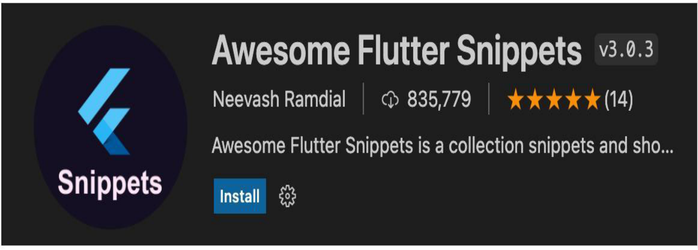


Figure 8.18: The Awesome Flutter Snippets plugin

This is similar to the *stful* and *stless* shortcuts available by default, but extends it to a large number of Widgets that involve a lot of boilerplate code.

Here are a few examples of the shortcuts available with the plugin:

listViewB	ListView.Builder
listViewS	ListView.Separated
gridViewB	GridView.Builder
gridViewC	GridView.Count
gridViewE	GridView.Extent
customScrollV	Custom ScrollView

Figure 8.19: Shortcuts available with the plugin

This is important since Widgets with builder methods slow down developers and increase the time required.

Pubspec assist

Adding and updating dependencies can be time-consuming since the accurate name and version number needs to be looked up for every single one. The Pubspec Assist plugin provides an easy-to-add, update, and sort dependencies from **`pub.dev`**. You can find it on the VS Code Extensions marketplace:

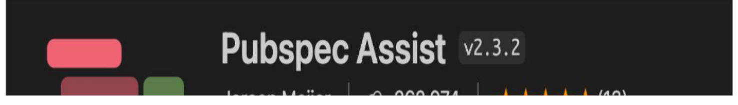


Figure 8.20: The Pubspec Assist plugin on the marketplace

You can then use the commands available via the plugin:

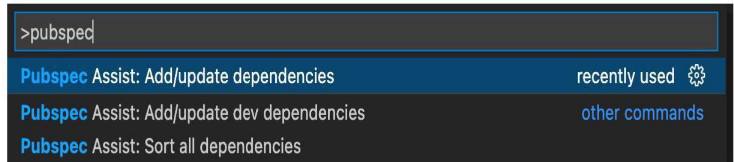


Figure 8.21: Commands available with the plugin

Rainbow brackets

Since Flutter code can be quite indented since Widgets are nested, it can be hard to tell the start and end of brackets. The Rainbow Brackets plugin changes the color of brackets for indentation. You can find it on the VS Code Extensions marketplace:

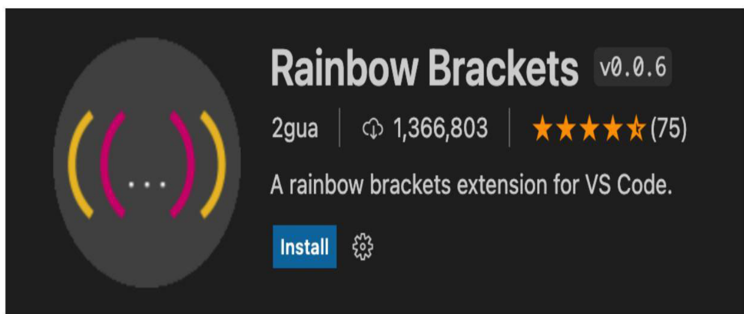


Figure 8.22: The Rainbow Brackets plugin on the marketplace

An example of this is shown in *figure 8.23*:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
        textTheme: GoogleFonts.openSansTextTheme(),
      ), // ThemeData
      home: HomeScreen(),
    ); // MaterialApp
  }
}
```

Figure 8.23: View of the editor with the plugin enabled

Now, onto the equivalents for Android Studio.

Android Studio extensions

Android Studio contains plugins that allow adding extra functionality to the editor. You can find plugins for all kinds of languages and files; however, we focus on the most useful extensions for Flutter.

There are a lot of great extensions that will save developers a lot of time, and not all good extensions can be listed here, but it is a good idea to take a look for yourselves. Let us start by looking at popular Android Studio plugins for Flutter.

Flutter Pub Version Checker

The Flutter Pub Version Checker plugin shows if the version in the **`pubspec.yaml`** file is up to date and also supplies actions to allow updating them. Here is what the plugin looks like on the marketplace:

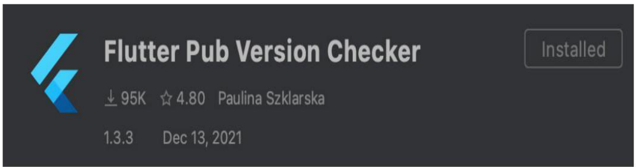


Figure 8.24: The Flutter Pub Version Checker plugin on the JetBrains marketplace

Here is what it looks like in action:

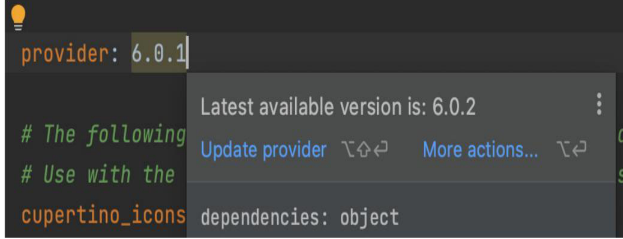


Figure 8.25: Version info available with the plugin

You can also update the dependencies in the file:

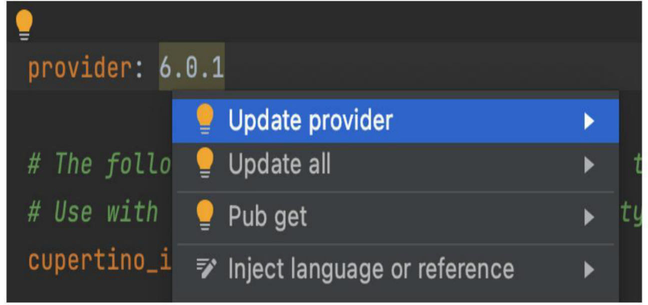


Figure 8.26: Options available with the plugin

Flutter Snippets

Flutter Snippets is a popular Android Studio plugin for Flutter which is an easy way to eliminate the boilerplate code associated with a lot of Widgets in Flutter. This is similar to the *stful* and *stless* shortcuts available by default but extends it to a large number of Widgets that involve a lot of boilerplate code. Here is what the plugin looks like on the marketplace:

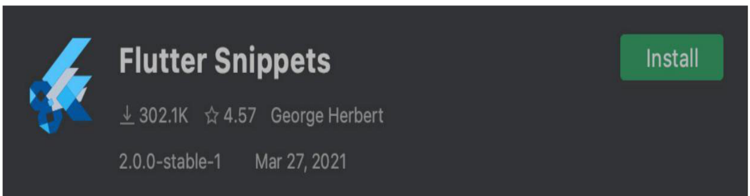
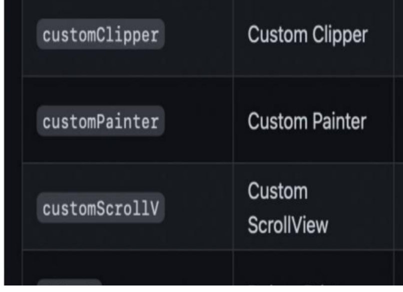


Figure 8.27: The Flutter Snippets plugin on the JetBrains marketplace

Here are a few examples of the shortcuts available with the plugin:



debugP	Debug Print
dis	Dispose
futureBldr	Future Builder
initS	InitState
layoutBldr	Layout Builder

Figure 8.28: *Examples of Flutter snippets shortcuts*

This is important since Widgets with builder methods slow down developers and increase the time required.

Rainbow brackets

Since Flutter code can be quite indented since Widgets are nested, it can be hard to tell the start and end of brackets. The Rainbow Brackets plugin changes the color of brackets for indentation. Here is what the plugin looks like on the marketplace:

Figure 8.29: *The Rainbow Brackets plugin on the JetBrains marketplace*

The editor brackets are now paired by color and are easier to group:

```
return Scaffold(
  appBar: AppBar(
    title: Text(widget.title),
  ), // AppBar
  body: Center(
```

```
child: Column(
  mainAxisAlignment: MainAxisAlignment.center,
  children: <Widget>[
    const Text(
      'You have pushed the button this many times:',
    ), // Text
    Text(
      '$_counter',
      style: Theme.of(context).textTheme.headline4,
    ), // Text
  ], // <Widget>[]
), // Column
), // Center
```

Figure 8.30: *View of the editor with the plugin enabled*

Conclusion

If done correctly, you should now be able to develop Flutter applications on your development device. There are various other extensions to make Flutter development simpler for you.

In the next chapter, we dive into a Flutter project and study the project structure and the responsibilities carried by each file and folder.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 9

CHAPTER 9

Flutter Project Structure and Package Ecosystem

Introduction

Before diving into real-world Flutter code, we need to understand how Flutter apps are structured, what each file does, and the best practices associated with common tasks. This will help you avoid headaches months down the development pipeline. A mobile app project is rarely completely independent – it uses external sets of code called **plugins** or **packages**, which usually adds specific common functionality such as asking for permissions, opening the camera, etc. This chapter goes through these topics so that you gain a better understanding of the composition of a Flutter project.

Structure

In this chapter, we will cover the following points:

- Beginning the Flutter journey
- Breaking down UI and logic files
    - o Understanding lib and `main.dart`
- Understanding `pubspec.yaml`
    - o What is a YAML file?
    - o Setting basic project metadata

    - o Versioning and environment
    - o Adding dependencies (Packages and plugins)
    - o Understanding Flutter's package repository: `pub.dev`

- Packages outside **`pub.dev`**
- Package vs. plugin
- Versioning dependencies
- Dependency overrides
- Developer dependencies
- Adding assets
- Specifying **`publish_to`**

- The test folder
- Importing packages
- README.md and LICENSE
- Important files to remember for a Flutter project
  - AndroidManifest
  - build.gradle
  - Info.plist
  - AppDelegate

Objectives

After reading this chapter, you will be able to understand the purpose of all the files in a Flutter project. You will also understand how to add external code to your Flutter project using a plugin/package.

Beginning the Flutter journey

All Flutter projects run as part of a native app on each platform and render their UI without using any native mobile views. Since Flutter needs a native app as a *container*, we need files for building each platform we intend to support. Let us take

a look at the basic files for an app that supports Android, iOS, Web, Linux, and macOS:

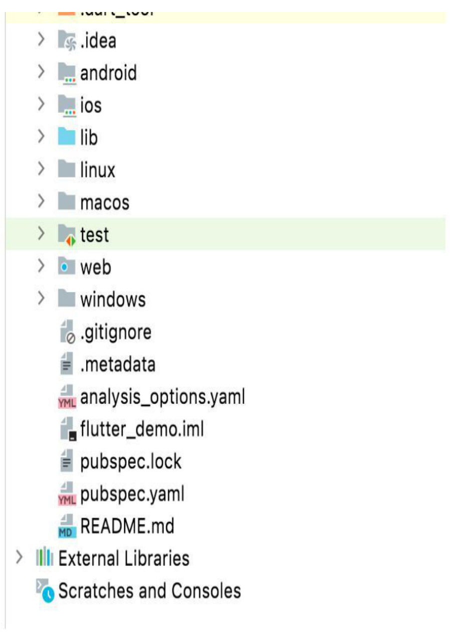


***Figure 9.1**: Basic structure of a Flutter project*

Here, we can observe that the main Flutter project contains android, ios, linux, macOS, and web files. Note that these directories are full-fledged native projects with the appropriate setup required for the respective platform and may need to be modified in some cases. When creating basic projects, you will not need to modify these folders - and most projects will only need to modify them in cases of permissions, dependencies, or slight tweaks. In Android and iOS, some code and dependencies that may exist in normal native projects can be removed since Flutter does not need the default mobile views offered by the respective frameworks.

For the most part, you will not need to implement features using native code such as Kotlin/Swift. However, if you intend to implement features reliant on native APIs, you may need some knowledge of the native language yourself. These bits of native code can be called from Flutter directly through links called Platform Channels.

This section will also go through some important files to keep in mind for mobile development.

Breaking down UI and logic files

Coming from a background such as native Android development, developers may be used to structures like this:

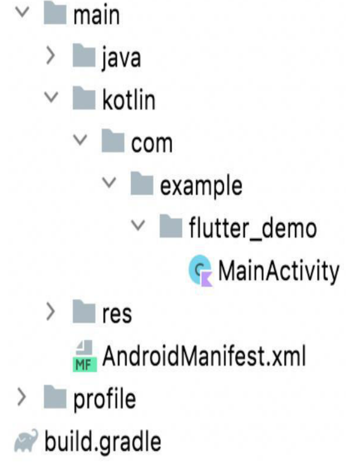


Figure 9.2: *Default Android native project structure*

Traditionally, frameworks like native Android write UI via a layout XML file and logic through a Java/Kotlin file. This caused the problem of context-switching since the logic may need to reference a view in the UI – but they do not exist in the same file. This required additional work to avoid the problem altogether with solutions like **`ViewBinding`** or packages like ButterKnife. Later on, Android added Jetpack Compose - which is a declarative UI solution allowing Android developers to develop apps similar to how Flutter apps are made.

Flutter avoided the aforementioned problem by not separating the two file types and putting logic and UI in a single Dart file. This file contained a **`build()`** function to build a layout composed of a tree of widgets. For worries about mixing business logic with UI, Flutter offers state management and dependency injection solutions such as BLoC, Provider (third-party), and so on.

Understanding lib and main.dart

The Dart code of the app resides in the lib folder at the root of the project. The **`lib`** folder, by default, contains **`main.dart`**, which includes the main function which initializes and runs our app:

Code 9.1:

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


The general naming convention for folders and files is snake case (**`folder_name`**).

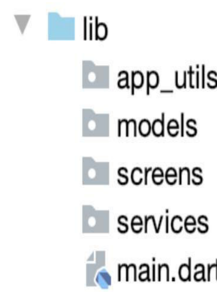


***Figure 9.3**: Example of folder and file naming scheme in Dart*

Understanding pubspec.yaml

The **`pubspec.yaml`** is an important file for configuring an app, package, or plugin written in Flutter. It contains asset declarations, dependencies (**`package imports`**), versioning and build number, constraints, environment used, metadata about the app or package, and so on.

You will find this file at the root level of your project:

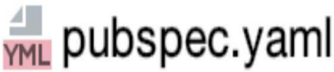


***Figure 9.4**: The pubspec.yaml file*

What is a YAML file?

YAML stands for **YAML Ain't Markup Language** (originally **Yet Another Markup Language**) - another recursive acronym in the programming world. YAML files are often used as configuration files but are designed to be more readable than other files used for similar purposes.

YAML uses Python-style indentation to indicate nesting:

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

***Figure 9.5**: Example of nesting in a YAML file*

Keeping this strictly related to Flutter **pubspec**:

- '**#**' is used to start comments
- '**-**' is used to indicate a single element of a list.

Setting basic project metadata

The **pubspec** file defines basic metadata such as:

- **name**: The name of the project. This also shows up on things related to the app, such as recent apps' names.
- **description**: Description of the project. This is considered a package description for packages and is also the default GitHub description.
- **homepage**: This optional field points to a homepage for the project.
- **repository**: Optional field that points to the repository for the project.
- **documentation**: Optional field that points to the documentation for the project.
- **issue_tracker**: Optional field that points to the **issue_tracker** for the project.

The default Flutter project has only these fields for metadata (fields from this list):

```
name: flutterdemo
description: A new Flutter application.
```

***Figure 9.6**: Project metadata in pubspec.yaml*

Versioning and environment

Flutter sets version numbers and build numbers in the **pubspec.yaml** using two values separated by a '**+**.' You can use the version tag to indicate build version:

```
version: 1.0.0+1
```

***Figure 9.7**: A version number declared in the pubspec.yaml file*

The build number is read by both app stores, and a build. The same build number cannot be uploaded twice. In Android, the version number is read as **versionName**, and the build number is read as **versionCode**. In iOS, this is **CFBundleShortVersionString** and **CFBundleVersion,** respectively.

When a version number and build number are set in the **`pubspec.yaml`** and need to be overridden, the **`--build-name`** and **`--build-number`** flags can be used as:

Code 9.2:

```
flutter build apk -build-name=1.0.1 -build-number=2
```

Adding dependencies (packages and plugins)

Packages and plugins are essential to any SDK since they allow extending an app's functionality and importing blocks of code without having to do everything from scratch. We specify the dependencies of a Flutter project inside the **`pubspec.yaml`** under dependencies:

```
dependencies:
  http: ^0.13.5
  shared_preferences: ^2.0.15
```

***Figure 9.8**: Example of adding dependencies in a Flutter project*

Before we get into more detail about packages and versioning, it is essential to know more about the primary source of packages in Flutter: **`pub.dev`**.

Understanding Flutter's package repository: pub.dev

Pub.dev is the primary host for all Flutter packages and is open for all to publish their packages. If a dependency is specified with a version number, Flutter tries to download it from **`pub.dev`** as default. Each package usually has installation instructions, a repository, examples, and documentation.

You can search for all the packages available on the home page:

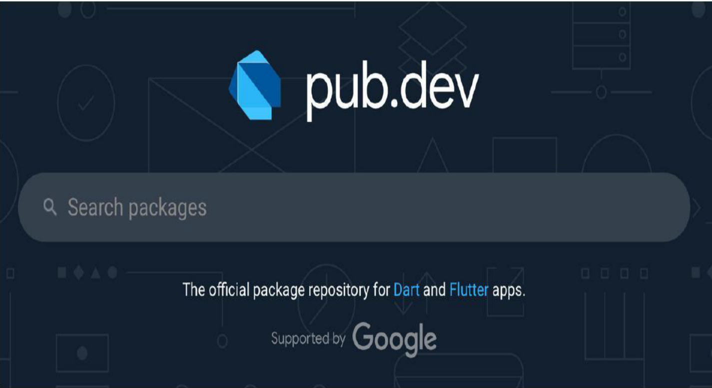


***Figure 9.9**: pub.dev – Flutter's main package repository*

Before using a package, you need to check the health and score of the package. For an example, here is the **battery_plus** package used to get details about the device battery:

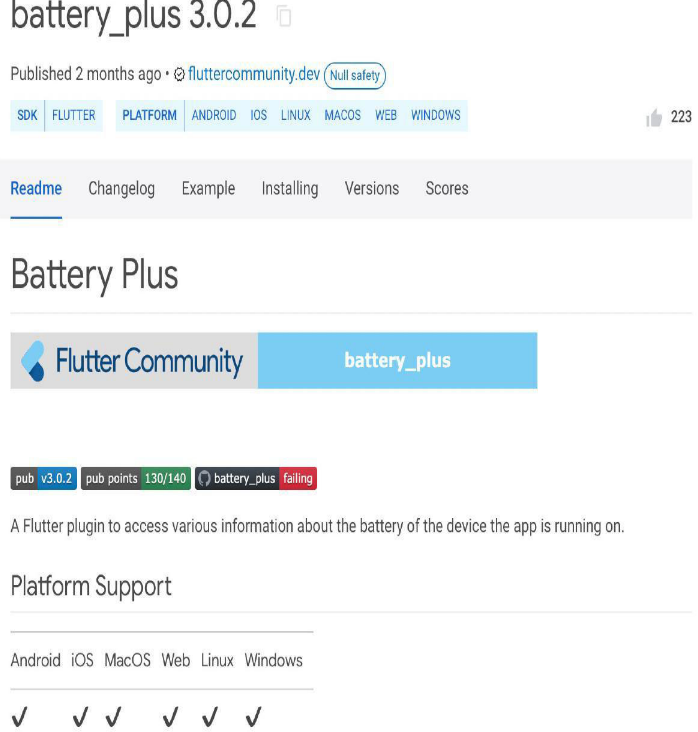


Figure 9.10: battery_plus – a commonly used Flutter package

An important criterion to judge packages is by the package score assigned by pub. It is computed by adding up factors such as static checks, file conventions, platform support, support for null safety, and so on. Here is a look at the **battery_plus** score:

Figure 9.11: Score assigned to the battery_plus plugin

Packages outside pub.dev

However, fetching a package from a custom URL is also possible, making it easier for proprietary code or custom implementations that do not need to be published to `pub.dev`.

```
dependencies:
  plugin1:
    git:
      url: git://github.com/flutter/plugin1.git
```

***Figure 9.12**: Example of adding a dependency from a git repository instead of pub.dev*

A dependency can also accept a URL that points to a Git repository to fetch a package. If a repository contains multiple packages and a path needs to be specified, you can add a path like this:

```
dependencies:
  package1:
    git:
      url: git://github.com/flutter/packages.git
      path: packages/package1
```

***Figure 9.13**: Example of adding a package from a monorepo hosted on GitHub*

At this point, it is also important to know the difference between a package and a plugin.

Package vs plugin

A package in Flutter is a Flutter dependency that contains pure Dart code and does not contain any native code. This might be, for example, a new kind of button or any visual component written with pure Flutter code.

A plugin, on the other hand, is a dependency that contains native code and is often used for exposing native APIs to the Flutter app above. An example of this might be the camera plugin used to gain access to the camera.

Versioning dependencies

There are multiple ways to specify the version for a package in **`pubspec.yaml`**. The easiest one is saying any version compatible with the project can be used:

```
dependencies:
  battery: any
```

***Figure 9.14**: Specifying that any version can be fetched for the dependency*

The second one is to use a certain version number:

```
dependencies:
  battery: 1.0.1
```

***Figure 9.15**: Specifying a version number for the dependency*

The next type is specifying above, lower, or range:

```
dependencies:
  battery: ">0.5.0"
```

***Figure 9.16**: Specifying a bound for the version number of the dependency*

The allowed types are >, >=, <, <=. Note that **`>`** is not a recognized YAML character. Hence, the version number is a string here. Strings in place of normal versions are also allowed. We can also combine operators to create a bound for versions: **`>=0.5.0 < 1.0.0`**.

The last type allowed, and often the most used, is the **caret** syntax:

```
dependencies:
  battery: ^0.5.0
```

***Figure 9.17**: Dependency versions with semantic versioning*

The caret syntax relies on semantic versioning to decide what range of packages to use. Breaking changes often carry a version increase: package versions < 2.0.0 are assumed to be compatible with package numbers like 1.4.5 in the same N.X.X version. So, ^1.4.5 is treated as >=1.4.5 but <2.0.0.

This changes when the package version is <1.0.0: every second digit changes are assumed to be semantically incompatible, and hence, ^0.5.0 is treated as >= 0.5.0 but < 0.6.0.

Dependency overrides

In some circumstances, we may need to override the versions of certain dependencies without changing the versions of the normal dependencies. This may also be to temporarily use an online version of a package instead of the **`pub.dev`** version. This is done by settings the values in the **`dependency_overrides`** field:

```yaml
dependencies:
  battery: 1.0.0

dependency_overrides:
  battery: 0.6.0
```

Figure 9.18: Overriding dependency versions with overrides

Developer dependencies

Some dependencies may only need to be used in the development, not production. For these scenarios, the `dev_dependencies` field comes in handy.

```yaml
dev_dependencies:
  battery: 1.0.1
```

Figure 9.19: Adding a dependency for development mode

Note that dependency overrides work just the same in `dev_dependencies` as they do with normal dependencies.

Adding assets

All assets used inside the Flutter application must be declared inside the `pubspec.yaml` file. This is done inside the Flutter section of the `pubspec` since the commands here are specific to Flutter and not Dart.

First, we need to create a folder to hold our assets. For this example, let us create a folder called `images`.

We can then declare the assets inside the folder using a hyphen and the file name. As written previously, a hyphen denotes a single list element.

```yaml
assets:
  - images/a_dot_burr.jpeg
  - images/a_dot_ham.jpeg
```

Figure 9.20: Adding assets to a Flutter project

Fun fact: The file names given by default here reference the play, *Hamilton*. The Hamilton app was one of the first full production apps to be written with Flutter, due to several examples and easter eggs that refer to the same musical.

If the app has many images or audio assets, declaring them all is quite a hardship. To avoid this, we can simply declare the folders containing our assets with a **/** at the end, and all the assets inside are then available for use in the app.

```
assets:
  - images/
```

Figure 9.21: Adding a directory of assets to a project

This imports all assets under the folder **images**.

Specifying publish_to

Pub.dev is the primary resource of Flutter and Dart packages. However, to publish to an internally hosted pub server, the **publish_to** command comes to use.

By default, it is set to none for Flutter projects to not accidentally upload it to **pub.dev**

```
publish_to: 'none'
```

Figure 9.22: Specfiting that the project should not be published to pub.dev

The test folder

The **test** folder holds all the tests for the Flutter app. Flutter has multiple types of tests, such as widget and integration tests, as shown in the following screenshot::

```
testWidgets('Counter increments smoke test', (WidgetTester tester) async {
  // Build our app and trigger a frame.
  await tester.pumpWidget(MyApp());

  // Verify that our counter starts at 0.
  expect(find.text('0'), findsOneWidget);
  expect(find.text('1'), findsNothing);

  // Tap the '+' icon and trigger a frame.
  await tester.tap(find.byIcon(Icons.add));
  await tester.pump();

  // Verify that our counter has incremented.
  expect(find.text('0'), findsNothing);
  expect(find.text('1'), findsOneWidget);
});
```

Figure 9.23: Default test in the project

The default test is a widget test on the counter app that Flutter creates when a project is created. We will go into testing in more detail in a later chapter.

Importing packages

The import keyword is used to import a package in a Dart file. For example, to import the battery package which supplies information about the battery and charging stats, we would do this at the top of the dart file:

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

Figure 9.24: Importing a package into a file

Often, we need a shorthand for packages to use inside the file. For example, let us try importing **dart:math** which is provided by default by Dart:

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

Figure 9.25: Importing a package with the as keyword

Everything inside the **dart:math** package is now accessible through **math.X**. This shorthand is used for multiple things, such as improving code readability, (**math.pi** gives a better understanding of where the constant **pi** comes from rather than just **pi**) and to solve conflicts where two imports may supply the same values. For example, if **pi** is defined in **math** as well as another import, it is essential to solve the clash.

Going along a similar line, assume we have pi coming in from two packages and **dart:math** has a lower accuracy, so we need to hide it. In cases like these, we can solve this problem with this method:

```
import 'dart:math' hide pi;
```

Figure 9.26: Hiding specific data with the hide keyword

This hides the constant **pi** defined inside the **dart:math** package. On the other side, we may only like to expose a few things from a package and leave out the rest. We can do this as:

```
import 'dart:math' show pi;
```

Figure 9.27: Importing specific data with the hide keyword

This only exposes **pi** from the package and hides everything else. These keywords can also be combined:

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

Figure 9.28: Combining previous import approaches

README.md and LICENSE

README.md is a markdown file meant to provide more information about the project for others to understand its purpose better and use it in apps. It is the main information displayed in a GitHub project or **pub.dev** package when uploaded. Better information gives a more precise idea to a contributor/user about the use of the project.

LICENSE is a file that is not provided by default but is helpful for others to understand the permissions and liability of use for the project. It is also required for upload to **pub.dev** and is helpful for an open-source project.

Important files to remember in a Flutter project

This section discusses some important files to remember apart from the dart files for Flutter development for Android and iOS.

Android manifest

The Android manifest summarizes all the data the OS requires while installing the app. It holds all the permissions the app needs, activities, services, default screen (not for a Flutter app), and so on. The manifest needs to be edited for adding permissions and sometimes an activity in case of things like Android background notifications:

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.flutter_demo">
   <application
        android:label="flutter_demo"
        android:name="${applicationName}"
        android:icon="@mipmap/ic_launcher">
        <activity
            android:name=".MainActivity"
            android:exported="true"
            android:launchMode="singleTop"
            android:theme="@style/LaunchTheme"
            android:configChanges="orientation|keyboardHidden|keyboard|screenSize|smallestScreenSize|locale|
            android:hardwareAccelerated="true"
            android:windowSoftInputMode="adjustResize">
            <!-- Specifies an Android theme to apply to this Activity as soon as
                 the Android process has started. This theme is visible to the user
                 while the Flutter UI initializes. After that, this theme continues
                 to determine the Window background behind the Flutter UI. -->
            <meta-data
              android:name="io.flutter.embedding.android.NormalTheme"
              android:resource="@style/NormalTheme"
              />
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <!-- Don't delete the meta-data below.
             This is used by the Flutter tool to generate GeneratedPluginRegistrant.java -->
        <meta-data
            android:name="flutterEmbedding"
            android:value="2" />
    </application>
</manifest>
```

***Figure 9.29**: A look at the AndroidManifest file*

Find it under **`android/app/src/main`**. There is also often another manifest in **`src/debug`**. Do not add permissions there as the release mode build will not carry over the same permissions and the app will simply crash.

build.gradle

There are both app level and module level **`build.gradle`** files in an Android app. For the most part, only the module level file will need to be edited. The outer gradle file will be needed for app releases and adding sources for downloading dependencies.

will be needed for app releases and adding sources for downloading dependencies. Some Flutter dependencies may need you to add dependencies in the module level gradle file.

Here is a look at the gradle file:

```
apply plugin: 'com.android.application'
apply from: "$flutterRoot/packages/flutter_tools/gradle/flutter.gradle"

android {
    compileSdkVersion 28

    lintOptions {
        disable 'InvalidPackage'
    }

    defaultConfig {
        // TODO: Specify your own unique Application ID (https://developer.android.com/studio/build/application-id.html).
        applicationId "dev.joshi.flutterdemo"
        minSdkVersion 16
        targetSdkVersion 28
        versionCode flutterVersionCode.toInteger()
        versionName flutterVersionName
    }

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

Figure 9.30: A look at the build.gradle file

Find this file under: **`android/app/build.gradle`**

Info.plist

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>CFBundleDevelopmentRegion</key>
    <string>$(DEVELOPMENT_LANGUAGE)</string>
    <key>CFBundleExecutable</key>
    <string>$(EXECUTABLE_NAME)</string>
    <key>CFBundleIdentifier</key>
    <string>$(PRODUCT_BUNDLE_IDENTIFIER)</string>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundleName</key>
    <string>flutterdemo</string>
    <key>CFBundlePackageType</key>
    <string>APPL</string>
    <key>CFBundleShortVersionString</key>
    <string>$(FLUTTER_BUILD_NAME)</string>
    <key>CFBundleSignature</key>
```

```
<string>????</string>
<key>CFBundleVersion</key>
<string>$(FLUTTER_BUILD_NUMBER)</string>
<key>LSRequiresIPhoneOS</key>
<true/>
<key>UILaunchStoryboardName</key>
<string>LaunchScreen</string>
<key>UIMainStoryboardFile</key>
```

***Figure 9.31**: Info.plist which stores metadata about an iOS/macOS project*

The **`Info.plist`** holds several things required for iOS such as descriptions of permissions, metadata, and often some experimental keys for Flutter. This file can be viewed in a better format in XCode. Flutter packages such as image pickers often need you to add permission descriptions to this file.

Find this under: **`ios/Runner/Info.plist`**

AppDelegate

The **`AppDelegate`** is not used as often as the other files, but it is needed at times to add elements like API keys for services such as Google Maps, or iOS configuration at times.

```
import UIKit
import Flutter

@UIApplicationMain
@objc class AppDelegate: FlutterAppDelegate {
  override func application(
    _ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?
  ) -> Bool {
    GeneratedPluginRegistrant.register(with: self)
    return super.application(application, didFinishLaunchingWithOptions: launchOptions)
  }
}
```

***Figure 9.32**: The AppDelegate.swift file in the iOS project*

Find this under: **`ios/Runner/AppDelegate.swift`**

The extension for the file changes depending on swift support being enabled for the project. Though nothing is said about Objective-C/Swift support in the official docs on the time of writing, enabling Swift support gives fewer headaches in the longer term as some backward compatibility issues exist when Swift plugins are used with Objective-C projects.

Conclusion

After exploring the file structure for Flutter projects, it is prudent to go into the *widgets* mentioned throughout the earlier sections in more detail and understand how Flutter UI works internally.

In the next chapter, we will learn about Widget – the building blocks of a Flutter application and understand their types and uses in a Flutter project.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 10
Diving into Widgets

Introduction

Widgets form the basic components of the Flutter UI structure. They represent the building blocks of a Flutter app, and hence having an in-depth understanding of how widgets work makes a better Flutter developer. However, a person beginning Flutter development may not need to know the full depth of the working of widgets - you do not need to know how an internal combustion engine works to drive a car. For this reason, this chapter will is split into two sections: one explaining Flutter Widget's working enough to develop for Flutter. The second will explain the technical mechanism underlying widgets and how Flutter's multiple rendering layers work to give a smooth, optimized development experience.

Structure

Structure

In this chapter, we will cover the following topics:

- What is a widget?
- Understanding composition
- Refreshing UI: `setState()`
- Stateful vs. stateless widgets

- Underlying differences and performance differences
- The layers beneath Flutter
- Material, Cupertino, and more
- The widgets layer
- Understanding `RenderObjects`
- About Elements

Objectives

After studying this chapter, you will know about Widgets – the basic building blocks of Flutter. While Flutter Widgets are often interchangeable, they have many nuances, including their types, properties, performance implications, and more. Various design systems for different platforms also need to be considered.

What is a widget?

In a rudimentary sense, consider a Widget the Flutter equivalent of a Lego brick. In this analogy, Lego bricks are small elementary pieces that connect to make a larger construction, which is a Flutter app. However, *everything is a widget* goes deeper than this. Widgets are not only the UI elements on the screen. Widgets can represent UI elements, layouts, and even components that build on an event, usually supplied by a Future or Animation.

The philosophy of everything being widgets are tied to individual components being interchangeable rather than just being small cogs in a bit system. In the following figure, we see a tree of Widgets that compose a Flutter application:

Scaffold
Center
Column
Text
Text
AppBar
Text
FloatingActionButton
Icon

Figure 10.1: A look at the Widget Tree

Understanding composition

Composition, at its very core, creates more complex objects from a set of simple things. Flutter uses aggressive composition in which more widgets are progressively made of more basic widgets until we reach *leaf* widgets: components that do not have children.

Let us take an example of a vertical column of multiple **`Text`** objects:

```
return Column(
  children: [
    Text("Text 1"),
    Text("Text 2"),
    Text("Text 3"),
  ],
);
```

Figure 10.2: An example of composition

We will go into specific widgets in more detail later, but here, we have used two basic widgets: **`Text`** and **`Column`**.

- A **`Column`** takes the widgets as children and displays them vertically, one below the other.
- A **`Text Widget`** displays text with optional properties like text styling. The thing to notice here is that the **`Text`** widgets are passed as children to the **`Column`** which expects widgets. Hence the **`Column`** can take any other widgets as children: even other **`Columns`**.

Here is what the layout looks like:

Text 1
Text 2
Text 3

Figure 10.3: Three text widgets in a column

In the above image, the **`Column Widget`** provides structure to the children widgets but does not have any visual component otherwise to allow you to see the **`Column`** itself. Similarly, Flutter widgets consist of other widgets and so on. Widgets like **`Text`** end this tree.

Refreshing UI: setState()

Upon some events, such as loading data, we need to build the elements on the screen again to reflect new information that needs to be displayed (that is, new state). To understand this better, we need to understand Flutter a bit better:

All widgets in Flutter are immutable.

Widgets' properties cannot be changed; hence, we must build the entire UI (or the subtree to be changed) again. In frameworks like native Android, the properties of elements on the screen could easily be changed by obtaining a reference to the view on the screen. While it is possible to obtain a reference to the **`Widget*`** using keys, it is not useful for manipulation properties, and is used for other purposes entirely. (While this is a simplistic explanation of the processes at play, the actual Widget tree consists of three different trees that go through several layers to render the UI that we intended finally. Read the next section of this chapter for a more detailed explanation of the same.)

Now, if we cannot directly get a handle on individual elements on the screen, how do we change what is displayed?

The answer lies in the difference between imperative and declarative frameworks. While we would find the reference in an imperative UI, we need to store the changeable value in a variable or put in a literal for elements that do not change. For example, let us use the **`Text Widget`** and see how we would need to structure it to change text upon some kind of event:

```
var text = "Hello World";

return Text(text);
```

Figure 10.4: Separating variable storing state and widget

Note that these two statements belong in two different blocks of code, this is just a simplification.

To change the text displayed, we need to change the value of the text variable. Let us do this when an event happens. (Example, user tapping on a button).

```
text = "Text Changed";
```

Figure 10.5: Changing state of any event

However, simply changing the variable does not work. We must also let Flutter know that we intend to rebuild the UI. So, the correct way to rebuild the UI and display the text changed is:


```
setState(() {
  text = "Text Changed";
});
```

Figure 10.6: Using the setState() function to update screen

Using **setState()** makes Flutter rebuild the UI, and hence our changed text is displayed. Rebuilding the UI is far less expensive in Flutter than in other imperative frameworks due to some clever optimization which saves time on rebuilding the tree that was not changed.

setState() is a function that takes another function as a parameter (higher-order function). However, this does not mean we must strictly put all our changes in this function.

```
text = "Text Changed";

setState(() {
});
```

Figure 10.7: An example of an empty setState() function

This works just as well.

The point of **setState()** having a closure (accepting a function) is, it encourages better practices when rebuilding, avoids certain errors when the function runs asynchronously, and the state changes are neatly encapsulated. Underneath **setState()** is a more basic **markNeedsBuild()** function, which performs the actual refresh and was earlier also available to Flutter developers to use freely, resulting in errors. This is why Flutter added **setState()** as a function on top.

Stateful vs. stateless widgets

UI = f (state)

UI is a function of state made popular by the React framework. Since Flutter is heavily inspired by React, a similar philosophy carries over. This method dictates that the user interface displayed on the screen is a function that depends on the state. Hence, the state determines the UI.

What is state?

The state is everything the app holds in memory while running, including variables, themes, user data, etc. But we can split the types of state into two types:

- Ephemeral state
- App state

The Ephemeral state is usually local to a single app component, that is, a single *page* or widget. The *page* is in quotes since in Flutter, there is no strictly defined *app page*. This is because almost every Widget can be a page on its own, and there is no defined *activity* like in native Android development. Every Widget is closer to an Android fragment than an activity.

To explain this, let us take a simple example from the Flutter docs - imagine we have a simple app with a **`BottomNavigationBar`** such as this:

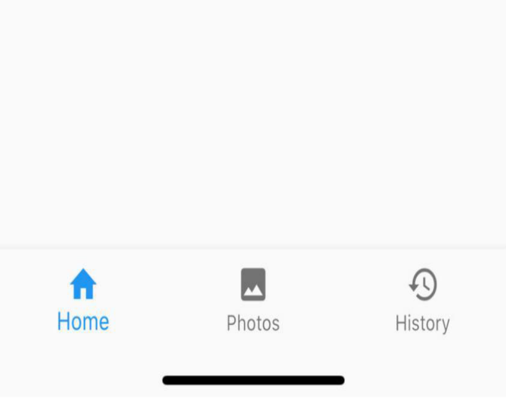


Figure 10 8: *Example of a BottomNavigationBar*

We may need to store which of the tabs is open. Let us have a simple variable to store this. The simplified code looks like this:

Code 10.1:

```
class MyHomePage extends StatefulWidget {

    MyHomePage({Key? key}): super(key: key);

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

}

class _MyHomePageState extends State<MyHomePage> {

    int _tabNumber = 0;

// ...
```


Here we store the tab clicked in a **`_tabNumber`** variable, ignore the rest for now. The main thing to note is that for most purposes, we do not need to access this tab number in any other widget or part of the code except the build function of our page. Hence, this is an *ephemeral* state – a state which does not last beyond the widget or a short time.

The opposite of this is app state: A state which may need to be accessed throughout the app in various places. An example of this is user data. We may need to know user details on the profile page while making purchases, to know user preferences, and so on. App state is managed by various methods, but Stateful/Stateless widgets deal with where an ephemeral state exists.

An easy rule to remember

Before we go into detail about what stateless and stateful widgets are, here is a simple rule to remember:

If your Widget needs to refresh/rebuild itself, use a stateful widget. Otherwise, stick to a **`StatelessWidget`**.

That is, stateful widgets can call **`setState()`** from the inside, whereas stateless widgets cannot. A page with a counter would be a stateful Widget as it may have a text displaying a count that needs a refresh on counter update. A profile page may be stateless as its information may not need to change. These are all basic examples, and they often go the other way as there are multiple implementations for doing the same thing, but all in due time.

StatelessWidgets

StatelessWidgets

As obvious from the name, **StatelessWidgets** have no state of their own. As seen before, a state is anything stored in memory. However, that does not mean it cannot have data. For example, stateless widgets can accept user data and display it just fine. Stateless refers to the inability to mutate or modify that data from inside the Widget.

Here is a very basic profile page built using a **StatelessWidget**:

Code 10.2:

```
class ProfileScreen extends StatelessWidget {

    final UserData data;

    const ProfileScreen({Key? key, required this.data}) : super(key: key);

    @override
```



```
    Widget build(BuildContext context) {
        return Container(
            child: Column(
                children: [
                    Text(data.email),
                    Text(data.username),
                ],
            ),
        );
    }
}
```

Here, first, we define a **StatelessWidget** called **ProfileScreen** which accepts an object of a class **UserData** which presumably contains all the information pertaining to the user. The **build()** function is responsible for building the UI.

Note that the **StatelessWidget** can be rebuilt from above in the Widget hierarchy: that is, if a **StatelessWidget** is instantiated inside a stateful widget, it can be rebuilt.

StatefulWidgets

Let us look back to a sentence earlier in this chapter: *All widgets are immutable.* If a widget is immutable, how can it contain state - which would be mutable since state

can change, by definition.

Flutter solves this problem by having the state **tied** to a Widget instead of being an integral part of it. When the Widget tree is rebuilt, the state is simply tied back to the same Widget instead of being destroyed and recreated - which would be expensive. Therefore, Flutter rebuilds are much cheaper than other frameworks. Hence, stateful widgets are two blocks: the Widget and the state.

Let us take the same profile page example (even though it does not necessarily exhibit all the advantages of using a stateful widget) to look at the differences between the two types:

Code 10.3:

```
class ProfileScreen extends StatefulWidget {

    final UserData data;

    const ProfileScreen({Key? key, required this.data}) : super(key: key);

    @override
```



```
    ProfileScreenState createState() => _ProfileScreenState();

}

class ProfileScreenState extends State<ProfileScreen> {

    @override
    Widget build(BuildContext context) {
        return Container(
        child: Column(
            children: [
                     Text(widget.data.email),
                     Text(widget.data.username),
            ],
        ),
        );
    }
}
```

The main thing to notice is that, now there are two classes instead of one: for the Widget and one for the state.

The Widget instantiates a state class and the state. There is no real need to worry

about remembering this syntax as most IDEs auto-create a basic Widget which can then be modified. Since the data is passed in the constructor of the Widget class and not the state class, we can access this using the widget property.

Stateful widgets can rebuild the Widget using the **`setState()`** function upon which any changes to the state are reflected.

Underlying differences and performance differences

It is easy to assume that there is a significant difference in performance between the stateful and stateless widgets since one carry state and has associated operations to perform whereas the other does not necessarily have this issue since no state exists. However, this is not true.

While there is a (very) small performance gain by using **`StatelessWidgets`**, they contain most of the elements like **`setState()`** inside them which are simply not available to the developer. This makes them like **`StatefulWidgets`** and does not make for a very large performance difference. Yet, using more **`StatelessWidgets`**

may be a sign of a good app structure given that the Widgets are more likely to be broken down well and serve a singular purpose.

State management is a term often used in large apps. It refers to how an app handles and stores the aforementioned state of the app. Simply put, **`setState()`** is usually not a very good way to rebuild the UI since it rebuilds everything in the subtree. The UI should ideally be rebuilt only when an associated event occurs and only the affected component should ideally change, leaving the rest untouched. Using stateless widgets enforces the developer to make better design choices and manage state better. There are various state management choices which are available as packages hosted on **`pub.dev`**. To name a few in no particular order: Provider, BLoC, MobX and many more. There is an endless supply of state management solutions in Flutter, and it is wise for a developer to try them for themselves rather than relying on opinions given by other people.

The content of the chapter till this point is sufficient to go ahead to the next chapter and begin with the basics of Flutter development. Further sections in this chapter go into detail about the underlying working of Flutter as well as the inbuilt design systems. Extra knowledge never goes to waste.

The layers beneath Flutter

Flutter is a complex toolkit which has several layers to help perform tasks like rendering UI, connecting to device APIs, and so on. Here is a look at the internal structure:

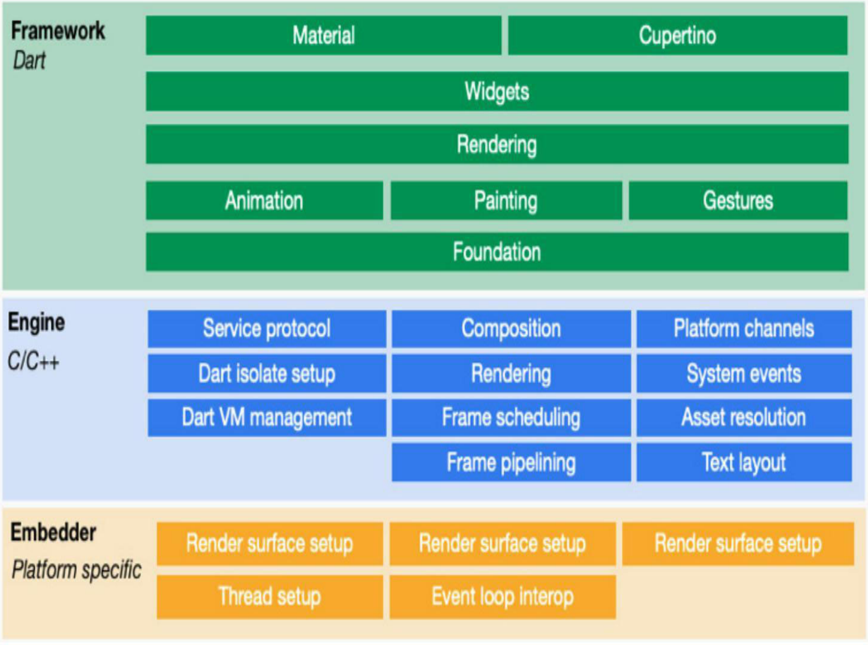


***Figure 10.9**: Layers beneath Flutter*

The preceding figure depicts the layers of the Flutter framework that deal with the UI and interaction. There are also layers beneath this that deal with the Flutter engine which is written in C/C++.

The framework layer consists of three fundamental layers which progressively turn user-defined widgets into objects which can be painted onto the device canvas. These layers are:

- The widget layer (dealing with widgets)
- The rendering layer (dealing with **`RenderObjects`**)
- **`dart:ui`** (animation, painting, gestures)

There are three types of trees in the app (not corresponding with the layers):

- The Widget tree
- The Element tree
- The RenderObject tree

These three trees combine to optimize memory use and rebuild time for UI components. In short, the **`Widget`** tree is the tree of widgets that the user defines (**`Columns`**, **`Containers`**, and so on). The **`RenderObject`** tree contains the objects that physically render the components in question. The Element tree (explained in detail in the *About Elements* section) contains elements which bind widgets and **`RenderObjects`** since it is expensive to instantiate new **`RenderObjects`**. These three trees make Flutter UI rendering performant and allow quick rebuilds.

Let us explore each layer and tree in more detail.

Material, Cupertino and more

It is easy to observe the differences between the app interfaces in android and iOS. Certain qualities feel similar between all android and all iOS apps, since all platforms offer development based on design guidelines of the platform in question.

As an example, here is Google's standard design system – Material Design:

***Figure 10.10**: Example of Material Design*

Material design was a standard Google design that was revealed to the world in 2014. It allowed components to mimic real-world objects with surfaces, edges, motion, and shadows. Apps like Gmail, Google Drive, and YouTube strictly conform to these standards and hence, there is a certain familiarity between these apps and their usage. When users started complaining that all apps looked the same with material design, Google revamped their design and allowed better customization and released plugins that integrated with Sketch and similar tools.

Material design has a defined set of styled components such as **`BottomNavigationBars`**, **`FloatingActionButtons`**, **`AppBars`**, and so on. It also has a certain way of navigation and a specific way that the components fit together. For example, the app drawer draws over the main app content that is not in the same plane. An iOS design illustration can be seen in the following screenshot:

***Figure 10.11**: iOS design example*

The iOS design guidelines are referred to as the *Cupertino* design style (named after the city). iOS components and interfaces are very recognizable, and it is easy to tell the difference between an iOS app and otherwise. iOS used to rely on a design type called skeuomorphism which used to model things on how their real-world counterparts looked. Hence, the YouTube logo was literally a TV. After iOS 7, iOS components are flat and translucent, having their own way of navigation between them.

Flutter recognizes these design styles and provides inbuilt components from both Cupertino and Material design styles. Along with these, other things such as Cupertino icons are also in the app. At the top level, an app needs to define if it follows Material or Cupertino style like this:

Code 10.4:

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

```
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
        visualDensity: VisualDensity.adaptivePlatformDensity,
      ),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
```

Similarly, we can also define the app to use Cupertino theme instead of material:

Code 10.5:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return CupertinoApp(
      title: 'Flutter Demo',
      theme: CupertinoThemeData(),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
```

The theme object changes with Cupertino theme since the same principles do not carry over to the other. There are also defined components for each theme, for example, this is a Material **`AlertDialog`**:

AlertDialog

This is a demo alert dialog.
Would you like to approve of
this message?

Approve

Figure 10.12: Example of Material AlertDialog

Here is a Cupertino **AlertDialog**:

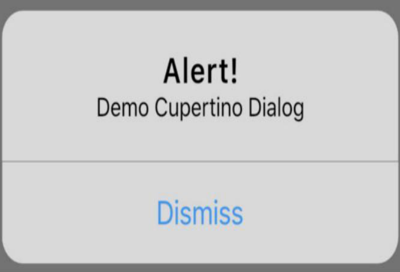


Figure 10.13: Example of Cupertino AlertDialog

Note that since widgets are freely compatible with other widgets, it is allowed to have Cupertino widgets in Material apps and vice-versa.

Bonus: WidgetsApp

There is a third, or rather a super-class of both Material and Cupertino app types which allows more customization called **WidgetsApp**:

Code 10.6:

```
class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return WidgetsApp(
     title: 'Flutter Demo',
     home: MyHomePage(title: 'Flutter Demo Home Page'),
     color: Colors.black,
   );
 }
}
```

There is more setup needed for basic functions, but some applications may need another implementation outside Cupertino/Material.

Material and Cupertino forms the highest layer of the Flutter stack. Below this level, we see the widgets layer.

The Widgets layer

The Flutter UI is composed of widgets which may be in turn composed of other Widgets. Flutter creates a Widget tree to represent how different widgets are connected and hierarchy is maintained as shown in the following screenshot:

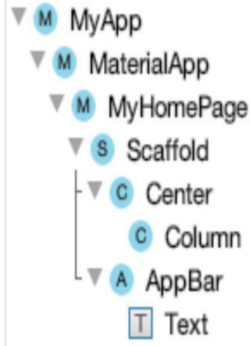


***Figure 10.14**: Example of the Widget tree*

However, widgets are more of a blueprint to what should be built as the UI. They do not carry any functions to render by themselves. This makes widgets easy to dispose of and rebuild as they are not mutable and do not carry a lot except the information required to render them.

All widgets produce equivalent **`RenderObjects`** to render them on the screen.

Understanding RenderObjects

`RenderObjects` are the buildings that are erected from the blueprints provided by widgets. However, constructing buildings is expensive and time-consuming, so ideally, we do not want to build the entire city again if one building needs to be slightly altered.

`RenderObjects` carry the functions to render widgets using the properties provided to them - thus forming a corresponding render tree of **`RenderObjects.`** This is similar to how widgets have a Widget tree:

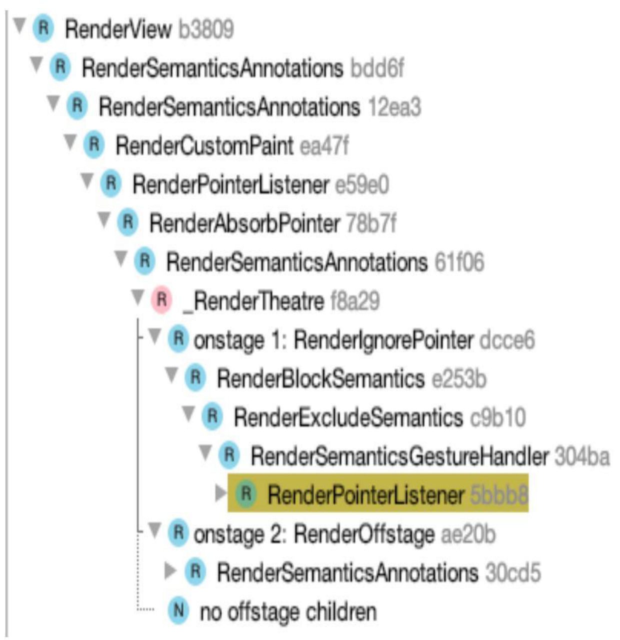


***Figure 10.15**: Example of the RenderObject tree*

While the Widget tree contains relatively obvious components (Widgets), the render tree consists of a lot of components that underlie the widgets themselves.

Let us take an example of a **`RenderObject`** with the Text widget. A **`Text Widget`** is responsible for displaying and styling text on the screen. Rendering the actual text is quite complex and is underneath most layers in the Flutter framework stack but this example is dealing with the process from **`Text Widget`** to the **`RenderObject`**.

The **`Text Widget`** underneath creates **`RichText`**, which is a **`Widget`** that allows more customization over **`Text`** and can also be used directly inside the app as a standalone widget which accepts **`TextSpans`**.

```
Widget result = RichText(
  textAlign: textAlign ?? defaultTextStyle.textAlign ?? TextAlign.start,
  textDirection: textDirection, // RichText uses Directionality.of to obtain a default if this is null.
  locale: locale, // RichText uses Localizations.localeOf to obtain a default if this is null
  softWrap: softWrap ?? defaultTextStyle.softWrap,
  overflow: overflow ?? defaultTextStyle.overflow,
  textScaleFactor: textScaleFactor ?? MediaQuery.textScaleFactorOf(context),
  maxLines: maxLines ?? defaultTextStyle.maxLines,
  strutStyle: strutStyle,
  textWidthBasis: textWidthBasis ?? defaultTextStyle.textWidthBasis,
  textHeightBehavior: textHeightBehavior ?? defaultTextStyle.textHeightBehavior
```

```
textHeightBehavior: textHeightBehavior ?? defaultTextStyle.textHeightBehavior,
  text: TextSpan(
    style: effectiveTextStyle,
    text: data,
    children: textSpan != null ? <InlineSpan>[textSpan] : null,
  ),
);
```

Figure 10.16: A look into the Text Widget

RichText in turn creates a RenderObject called RenderParagraph which is responsible for displaying a paragraph of text as shown in the following screenshot:

```
@override
RenderParagraph createRenderObject(BuildContext context) {
  assert(textDirection != null || debugCheckHasDirectionality(context));
  return RenderParagraph(text,
    textAlign: textAlign,
    textDirection: textDirection ?? Directionality.of(context),
    softWrap: softWrap,
    overflow: overflow,
    textScaleFactor: textScaleFactor,
    maxLines: maxLines,
    strutStyle: strutStyle,
    textWidthBasis: textWidthBasis,
    textHeightBehavior: textHeightBehavior,
    locale: locale ?? Localizations.localeOf(context, nullOk: true),
  );
}
```

Figure 10.17: A look into the RichText Widget

This **`RenderParagraph`** is the **`RenderObject`** responsible for displaying text which then uses a **`TextPainter`** object to paint the text on the canvas depending on the properties passed to it. All widgets either directly extend a **`Widget`** (For example, Center extends an Align Widget setting the alignment to center) or create a **`RenderObject`** of some kind.

To summarize, widgets provide configuration and lead to **`RenderObjects`** painting the Widget on the screen. However, if this is the entire process, it does not make sense to make widgets separate from **`RenderObjects`**. If both are created and destroyed on rebuild, it would be easier to have a **`RenderObject`** tree directly.

This is where elements step in.

About Elements

Elements form the third tree in the Flutter framework. This tree is not below the **RenderObject** but rather between widgets and **RenderObjects**. As you may recall, **RenderObjects** are hard to instantiate, destroy and rebuild - hence, we should do these as few times as possible. Widgets are easy to manipulate, create and destroy - so we should expect this to happen. Since **RenderObjects** represent widgets on the UI, we need something else to optimize the rebuilds in the **RenderObject** tree that the widgets cause. Here is where the element tree steps in.

When an app is built, the three trees (Widget, **Element**, **RenderObject**) are built, and **Elements** lie in between the Widget and **RenderObject** trees:

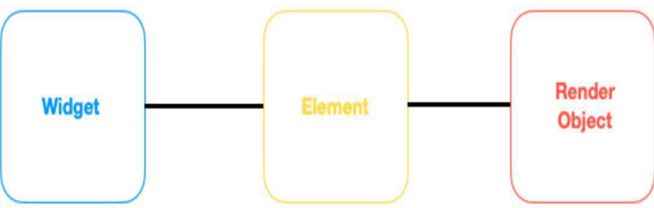


***Figure 10.18**: A diagram representing the connections between Widgets, Elements, and RenderObjects*

Now the **Widget** has certain properties like size or text to render, which it passes to the eventual **RenderObject** created. When a rebuild is triggered, the widget-element connection is destroyed. However, the element and **RenderObject** are not affected. When the new **Widget** tree is created, the properties of the existing elements are compared to the widgets in the **Widget** tree. *Creating the Widget tree* is not equivalent to a rebuild. This only implies the creation of the first tree and a rebuild is the complete process that ends in creating **RenderObjects** that reflect the **Widget** tree correctly. When the **Widget** and element have the same properties, the **RenderObject** does not need to be created again or modified. This saves us an expensive operation and the only change was in the **Widget** tree - which is inexpensive compared to a change in the other trees as illustrated in the following figure:

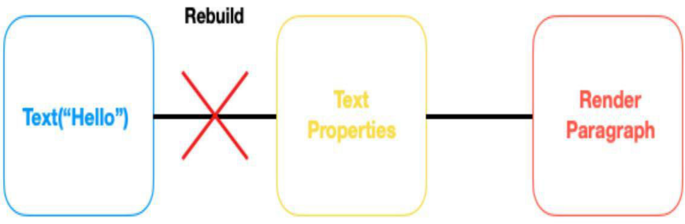


***Figure 10.19**: How a rebuild affects the trees*

Any widgets in the tree that do not have a counterpart in the existing element tree

then instantiate their own element and **RenderObject**. This process is why calling **setState()** which results in the entire tree being *rebuilt* does not actually rebuild all the trees and is efficient overall.

Due to this process and the **Widget**, element and **RenderObject** tree, Flutter apps reduce rebuild times substantially.

Conclusion

In this chapter, we started from the basics of **Widgets** and ended up covering the internals including the basics of **Elements** and **RenderObjects** – which will also be seen later in detail in *Chapter 16, Advanced Flutter – Under the Hood*. All Flutter sections until now were mostly theoretical and did not dive deep into the code itself, further chapters will be code heavy and will be purely about Flutter components or app development.

In the next chapter, we will go into common widgets used in Flutter apps in detail to gain a Widget repertoire, so to speak.

Questions

1. What is a Widget in Flutter?
2. What are the types of Widgets in Flutter?
3. What is the difference between a Stateful and Stateless Widget?
4. Is there a performance difference between the two kinds of Widgets?
5. How is a Widget different from **RenderObjects** and **Elements**?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 11
Basic Widgets and Layouts

Introduction

To start building a Flutter layout, we need to know the basic Flutter Widget library which consists of visual elements like text, images, and so on, and layout Widgets such as rows and columns. Additionally, more complex Widgets like the Scaffold give us a predefined structure for a general page in an app – which makes layout much easier by automatically placing navigation bars, app bars, and so on, in the correct position. These *simple* Widgets also offer quite a lot of customization options which can be leveraged for building the layout you need. In this chapter, we will focus on learning about these Widgets in more detail and building our first layout.

Structure

In this chapter we will cover the following topics:

- Approaching Flutter layouts
- Text
- Buttons
- Column and row
- Icon

- Padding
- Container
- Stack
- AppBar
- Scaffold
- Bottom
- ListView
- Understanding the basic Counter app

Objectives

The main purpose of this chapter is understanding the basic Widgets in Flutter and learning how they fit together to make a layout. The basic counter app example that Flutter provides as the starting app for all projects was chosen as an example. This app covers a number of ideas, including basic layout, rebuilds, and other related topics.

Approaching Flutter layouts

Complex Flutter layouts consist of many nested widgets which are in turn made of smaller, more basic widgets. To build full-blown apps, it is necessary to have a good grasp of the most essential widgets and their properties and customization. Since Flutter widgets are interchangeable, there is no need to study normal and layout widgets separately, hence, they will be listed in terms of complexity rather than grouping them according to function. Simple Widgets take a set number of children Widgets and display them according to the configuration defined inside the **`build()`** method. However, a Widget that displays a list or grid of items cannot function the same way since there can be any number of items inside them – which may also change over time.

There is also another class of widgets, namely Sliver widgets, which operate inside scrollable areas that will not be covered in this chapter but will have some explanation in further chapters. Sliver allow custom scrolling effects to be implemented but most things can also be implemented with normal widgets.

Let us start with the most basic of widgets. Since we cannot cover all possibilities with any particular widget, this chapter covers the most used properties.

Text

The Text widget allows displaying text on the screen and additionally allows customization of how the text looks and behaves. Here is how we can add a basic **`Text Widget`**:

Code 11.1:

```
Text("Hello World!")
```

The output of the text assumes a default text size based on accessibility settings on the phone:

Figure 11.1*: Basic Text Widget implementation*

The Text widget has a **`String`** of text as a positional property and then additional optional properties for customization (as seen in the Dart section).

TextStyle

Most of the styling properties are set with an object of **`TextStyle`** passed to the style parameter of text.

Changing font size, font weight and font family

We can change font size by specifying the **`fontSize`** parameter and **`fontWeight`** for font weight.

Code 11.2:

```
Text(
  'Hello, World!',
  style: TextStyle(
    fontSize: 26.0,
    fontWeight: FontWeight.bold,
  ),
)
```

This results in:

***Figure 11.2**: Text Widget with size and weight modified*

For changing the font, we first need to add the font in the assets in **`pubspec.yaml`**. (Alternatively, use the **`google_fonts`** package to make it even easier) After that, we can simply name the font family to use like this:

Code 11.3:

```
Text(
  'Hello, World!',
  style: TextStyle(
    fontFamily: "Raleway",
  ),
)
```

Text color

Changing text color is done via the color property:

Code 11.4:

```
Text(
  'Hello, World!',
  style: TextStyle(
    color: Colors.red,
  ),
)
```

Other customizations

We can specify a distance between letters and words to space them out:

Code 11.5:

```
Text(
```

```
  'Hello, World!',
```



```
  style: TextStyle(
    letterSpacing: 5.0,
    wordSpacing: 5.0,
  ),
)
```

This gives the following output:

Figure 11.3: Text Widget with letter and word spacing modified

We can also add aspects like text shadows:

Code 11.6:

```
Text(
  'Hello, World!',
  style: TextStyle(
    fontSize: 26.0,
    fontWeight: FontWeight.bold,
    shadows: [Shadow(blurRadius: 10.0, offset: Offset(5.0, 5.0))],
  ),
),
```

Here, the shadows parameter expects a list of shadows and the offset inside the shadow defines where the shadow lies with respect to the text itself. The output is:

Figure 11.4: Text Widget with shadows added

There are several other properties to modify the display of text inside **`TextStyle`** which are left out for brevity but do give them a try.

Max lines and overflow

Outside the **`TextStyle`** passed to the Text, **`maxLines`** and overflow are two important properties of the Text widget. The **`maxLines`** property defines the maximum lines that the text is allowed to render while the **`overflow`** property defines how the text should behave in case the text cannot fit in the space allocated to it.

For example, let us take a large chunk of text and allow only one line:

Code 11.7:

```
Text(
  'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua.',
  maxLines: 1,
)
```

This only displays one line:

Lorem ipsum dolor sit amet, consectetur adipiscing

***Figure 11.5**: Text Widget with maxLines set to 1*

To the user, it does not suggest that there is any text afterward. To solve this, we use the **`overflow`** property:

Code 11.8:

```
Text(
  'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua.',
  maxLines: 1,
  overflow: TextOverflow.ellipsis,
)
```

Now, in a better look:

Lorem ipsum dolor sit amet, consectetur adipiscin...

Figure 11.6: Text Widget with overflow implemented

Here, we specified the text to show ellipses when the text overflows the assigned size. There is also **`TextOverflow.clip`**, which does the same thing as without the ellipsis, and **`TextOverflow.fade`**, which fades out while the text is going down:

Lorem ipsum dolor sit amet, consectetur adipiscing
elit, sed do eiusmod tempor incididunt ut labore et

Figure 11.7: An example of TextOverflow.fade

Buttons

Buttons are elements that can be pressed and usually triggers some action. There are different kinds of buttons in Flutter and base buttons which can be customized to a higher degree. Buttons usually have two things: a child inside it and a callback when the button is pressed (usually defined as an **`onPressed`** parameter). Since buttons accept a widget, they can contain a plethora of things inside them: a text, an icon or even a mini app! (Everything is a widget!)

There are multiple types of buttons that are separated by platform (Material/Cupertino), elevation (raised/flat), and so on.

Some types are mentioned in this section.

TextButton

Buttons which have no elevation are called **`TextButtons`**. This means that it is difficult to distinguish the boundary of the button as shown here:

Tap here

Figure 11.8: Basic implementation of the TextButton

Buttons of a similar type are used in iOS. However, clicking on **`TextButton`** gives a material splash - iOS buttons do not. The code for all buttons is mostly similar, here is the code for a **`TextButton`**:

Code 11.9:

```
TextButton(
```

```
TextButton(
  onPressed: () {
    // Do something here
```



```
  },
  child: Text("Tap here"),
),
```

ElevatedButton

The **ElevatedButton** adds an elevation to the button itself and clearly separates itself from the surrounding:

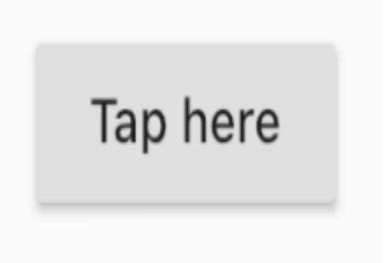


Figure 11.9: Basic implementation of the ElevatedButton

Otherwise, the code remains similar:

Code 11.10:

```
ElevatedButton(
  onPressed: () {
    // Do something here
  },
  child: Text("Tap here"),
),
```

OutlinedButton

The **OutlinedButton** can be defined as a **TextButton** with an outline:

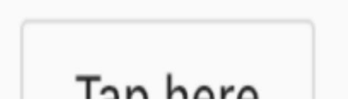


Figure 11.10: Basic implementation of an OutlinedButton

Everything else more or less remains the same.

Code 11.11:

```
OutlinedButton(
  onPressed: () {
    // Do something here
  },
  child: Text("Tap here"),
),
```

We will leave other button types (**`MaterialButton`**, **`CupertinoButton`**, **`IconButton`**) out for simplicity but the base concepts remain the same.

Button properties

Buttons offer customization in similar ways across all button types. Following sections cover few properties of the button:

Color

Button color is defined according to the **`color`** property, which we can supply a color via the defined colors list or a custom color using the **`Color`** class:

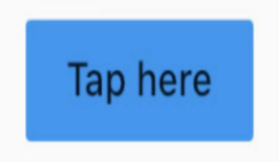


Figure 11.11: A Button with color changed

The color property can be set like this:

Code 11.12:

```
ElevatedButton(
  onPressed: () {
    // Do something here
  },
```

```
  child: Text("Tap here"),
  style: ElevatedButton.styleFrom(
    primary: Colors.blue,
  ),
)
```

Button shape

Buttons by default have a rectangular perimeter. The **shape** property allows us to set other shapes such as a rounded rectangle instead of a normal one like this:

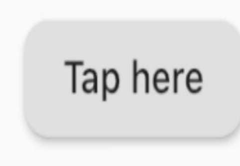


Figure 11.12: A Button with shape changed

The **shape** property can be set like this:

Code 11.13:

```
ElevatedButton(
  onPressed: () {
    // Do something here
  },
  child: Text("Tap here"),
  style: ElevatedButton.styleFrom(
    shape: RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(10.0),
    ),
  ),
),
```

There are other types of borders such as **BeveledRectangleBorder**:

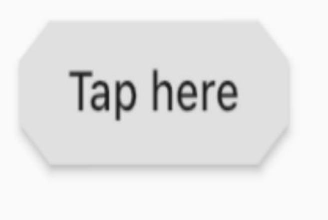


Figure 11.13: A Button with a BeveledRectangleBorder

You can also try **StadiumBorder** or create your own border by extending **ShapeBorder** and let your imagination run free.

Column and row

Rows and columns are self-explanatory when it comes to their function: they allow a horizontal and vertical layout of children (widgets) respectively. Any children supplied to a column will be laid out from top to bottom, one below the other.

Here is a basic example:

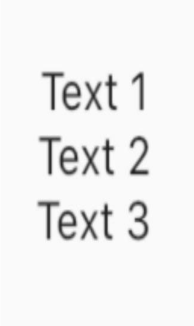


***Figure 11.14**: A basic implementation of a Column*

However, while it is easy to understand the relation between elements defined by the `Column` (the next widget is below the current widget), it is imperative to define the spacing of the same elements with respect to the screen.

For example, here is a `Column` with the same elements that is wrapped by a `Center` widget:

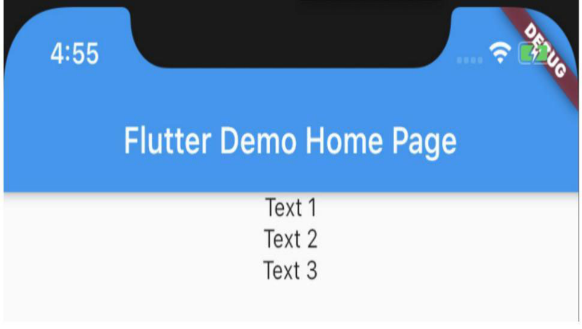


***Figure 11.15**: A Basic implementation of a Column with elements centered*

Here is the code that makes this possible:

Code 11.14:

```
Center(
  child: Column(
    children: <Widget>[
      Text("Text 1"),
```

```
      Text("Text 2"),
      Text("Text 3"),
    ],
  ),
),
```

By default, all three would be on the left side, the **Center** widget centers the child that is given to it, hence, our column to the center. However, the main thing we are interested in is that all three elements are one below the other and there is no specific spacing in the column. The **mainAxisAlignment** solves this issue. Refer to the following figure:

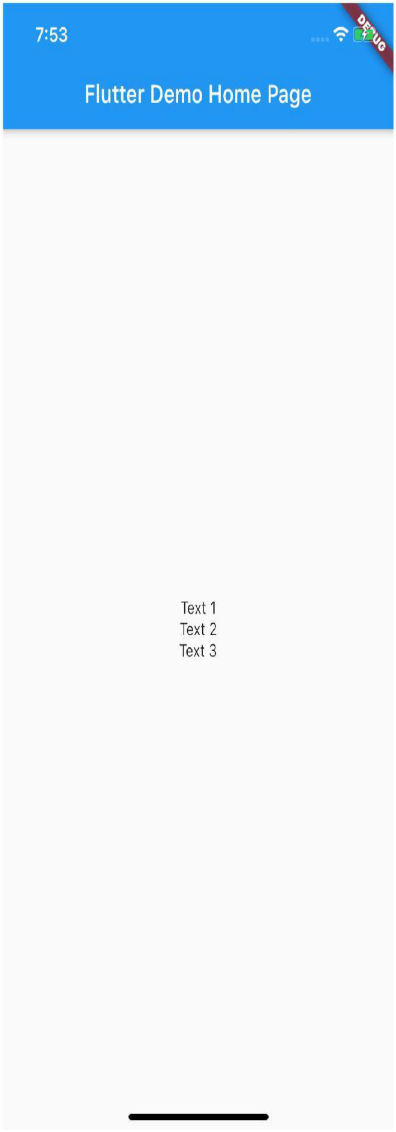


***Figure 11.16**: Basic implementation of Column with mainAxisAlignment set to center*

The **mainAxisAlignment** property defines how the widgets inside the column are spaced with respect to each other and the screen itself. For example, the image above shows how using the **MainAxisAlignment.center** works. The **Main Axis** in a **Column** is the vertical axis and the **Cross axis** is the horizontal one. Hence, a **Column** allows us to set how widgets behave both in a horizontal and vertical sense.

For example, the **spaceEvenly** type spaces all elements evenly across the axis:

Code 11.15:

```
Column(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: [
    Text("Text 1"),
    Text("Text 2"),
    Text("Text 3"),
  ],
)
```

In the following figure, the three texts are spaced evenly across the vertical axis – which is the main axis for a **Column**:

Figure 11.17: Implementation of Column with mainAxisAlignment set to spaceEvenly

Rows: columns - but horizontal

Rows work the same way as columns in all respects except that the main axis is horizontal, and the cross axis is vertical. In *figure 11.18*, the three texts are spaced evenly across the horizontal axis – which is the main axis for a **`Row`**:

Figure 11.18: Basic implementation of Row with mainAxisAlignment set to spaceEvenly

Code 11.16:

```
Row(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: [
    Text("Text 1"),
    Text("Text 2"),
    Text("Text 3"),
  ],
)
```

Icon

An icon allows us to display any of the inbuilt Flutter icons or (with some effort) custom defined and imported icons.

Code 11.17:

```
Icon(
  Icons.add,
)
```

Which gives:

Figure 11.19: Basic implementation of add icon

The list of all material icons is visible through the **Icons** class. Similarly, the icons for the Cupertino design system can be found through the **CupertinoIcons** class.

Code 11.18:

```
Icon(
  CupertinoIcons.game_controller_solid,
)
```

Which results in:

Figure 11.20: Basic implementation of Cupertino Icon

Icon customization

The main things to customize in icons are the size and color through fields of the same name:

Code 11.19:

```
Icon(
  CupertinoIcons.game_controller_solid,
  size: 60.0,
  color: Colors.blue,
)
```

Figure 11.21: Icon with color changed

Padding

In the simplest sense, the **Padding** widget adds space around a widget. It allows us to add distance between a widget and the parent.

Let us take an example with a **Text** widget and a **Container** (seen later):

***Figure 11.22**: Text Widget inside a Container without Padding*

Code 11.20:

```
Text(
  "Title",
  style: TextStyle(color: Colors.white, fontWeight: FontWeight.bold),
),
```

The **Text** is too close to the corner to look appealing. Here is where **Padding** can help us:

Figure 11.23: Text Widget inside a Container with Padding

Code 11.21:

```
Padding(
  padding: const EdgeInsets.all(8.0),
```



```
  child: Text(
    "Title",
    style: TextStyle(color: Colors.white, fontWeight: FontWeight.bold),
  ),
),
```

The padding argument allows us to supply the amount of distance between the child and parent. However, we can notice that a direct value is not passed, rather an **`EdgeInsets`** object is used.

An **`EdgeInsets`** object allows us to define space between the parent and child in all four directions or two axes. Here, **`EdgeInsets.all()`** passes the distance required for all directions.

`EdgeInsets.symmetric()` can be used if we need to define distance along an axis. (Distance on left and right-hand side or distance between top and bottom will be the same) **`EdgeInsets.only()`** can be used to define distance only in one direction (top, bottom, left or right).

Container

A **`Container`** widget is a convenience widget which allows us to perform several operations such as padding, alignment, and so on, on a widget without adding several layers of nesting to get the same result.

Adding size and color

A **`Container`** allows us to set a size and background color to it with the width, height, and color properties.

Code 11.22:

```
Container(
  color: Colors.blue,
  width: 100.0,
  height: 100.0,
)
```

Figure 11.24: A basic blue Container

Alignment and padding

The **Container** widget makes it easy to add padding and alignment without either of the respective widgets. Let us try to align an icon to the top-center inside a colored **Container**:

Code 11.23:

```
Container(
  color: Colors.blue,
  width: 100.0,
  height: 100.0,
  alignment: Alignment.topCenter,
  child: Icon(Icons.call, color: Colors.white,),
)
```

In *figure 11.25*, the icon is shown at the top of the **Container** since the alignment is set to **Alignment.topCenter**:

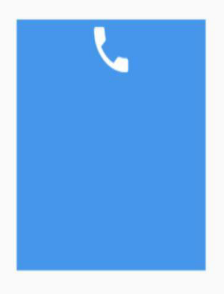


Figure 11.25: Container with child alignment set to top center

Here, we use the **alignment** property to align the child widget inside the **Container**. Similarly, we can add padding to the same code to create some distance between the child icon and the container:

Code 11.24:

```
Container(
```

```
  color: Colors.blue,
  width: 100.0,
  height: 100.0,
  alignment: Alignment.topCenter,
  child: Icon(Icons.call, color: Colors.white,),
  padding: EdgeInsets.only(top: 16.0),
)
```


You can notice that here, **`EdgeInsets.only()`** is used to add padding only at the top.

Decoration

Containers also allow decorations such as gradients, shadows (like the text shadow seen earlier) and background colors and images. For brevity, let us see how to add gradients to the **`Container`**. The other uses can mostly be covered by other widgets as well:

Gradients allow us to show a smooth change between two or more colors, like this:

***Figure 11.26**: Container with a linear gradient*

To define a gradient, we need two main things:

- Which colors to use.
- At what point does the gradient start converting from one color to another and when it stops.

To achieve this effect, we use a **`LinearGradient`**, passing it to the **`gradient`** property of the decoration.

Code 11.25:

```
Container(
  width: 100.0,
```

```
  height: 100.0,
  decoration: BoxDecoration(
    gradient: LinearGradient(
      colors: [
        Colors.blue,
        Colors.red,
      ],
```



```
      stops: [0.0, 1.0],
    ),
  ),
)
```

The colors property defines the colors used, and the stops define when the transitions between colors start and end. The **`Container`** also allows for some other possibilities such as transformations, but those will be covered in the chapters dealing with animations and transformations.

Stack

A **`Column`** arranges widgets in the Y (vertical) direction. A row does it in the X (horizontal) direction. A **`Stack`** arranges widgets in the Z direction: Meaning from the perspective of the user, they are overlaid on top of the others. This is useful for a lot of cases where widgets need to exist on top of other widgets, such as a title on top of an image.

Basic implementation

Here, we use three containers of different colors and sizes that are stacked on top of each other.

Code 11.26:

```
Stack(
  children: [
    Center(
      child: Container(
        width: 100.0,
        height: 100.0,
```

```
        color: Colors.blue,
      ),
    ),
    Center(
      child: Container(
        width: 75.0,
        height: 75.0,
        color: Colors.green,
```



```
      ),
    ),
    Center(
      child: Container(
        width: 50.0,
        height: 50.0,
        color: Colors.red,
      ),
    ),
  ],
)
```

The output will look like the following figure:

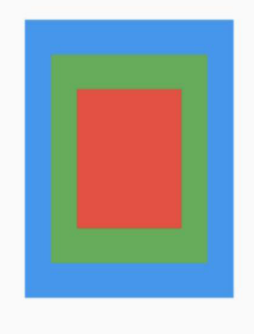


***Figure 11.27**: Stack with three Containers overlayed*

In a **Stack**, the widget added later in the children list is displayed above the others. The main thing to remember here is, if a **Text** is added first and then an image, the **Text Widget** will not be displayed since the image covers the text. Here, the red container is added later in the children. If it was placed first, it would be covered up

by the green and blue Containers.

Positioned

A thing to note in such an implementation is that while in a column or row, the widgets occupy the same space (horizontally or vertically), the elements of a **`Stack`** can occupy the entire area of the screen and position themselves independently.

Positioned is another widget that is primarily used with stacks to position children with relation to the full size of the screen - it positions them with respect to the screen since they can occupy the entire size of the screen. With **`Positioned`**, we can define

the space between a side (left, right, top, or bottom) and the child. For example, let us define a blue container at a distance of 50.0 from the left and top sides.

Code 11.27:

```
Positioned(
  top: 50.0,
  left: 50.0,
  child: Container(
    width: 100.0,
    height: 100.0,
    color: Colors.blue,
  ),
),
```

In *figure 11.28*, a **`Container Widget`** is positioned 50 units from the top and left as defined by the **`Positioned Widget`**:

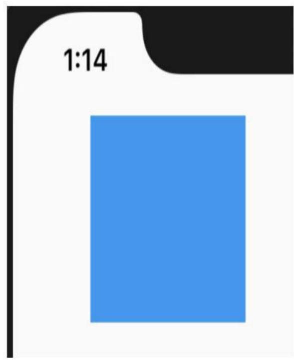


***Figure 11.28**: Container positioned 50 units away from top and left boundary of a Stack*

This shows up on the top left of the screen. This is useful when defining widgets with respect to the screen instead of defining them with respect to each other.

AppBar

An **AppBar** is the toolbar we see at the top of most apps which provides context about what the screen does and allow access to elements like app drawers and menus. An **AppBar** widget in Flutter allows us to easily create these drawers and actions.

Code 11.28:

```
AppBar(
  title: Text("Demo Page"),
),
```


Which creates the most basic **AppBar**:

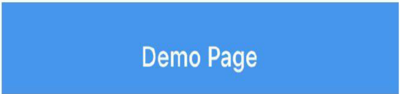


Figure 11.29: Basic implementation of an AppBar

The text might change position depending on the device that the app is running on. iOS convention is the title stays at the center.

Title, size, color

The most basic properties of an **AppBar** are the title displayed, the height, and the color of the background.

These can be set as:

Code 11.29:

```
AppBar(
  title: Text("Changed Text"),
  toolbarHeight: 100.0,
  backgroundColor: Colors.red,
),
```

Which gives a larger **AppBar** with a different background color:

Figure 11.30: An AppBar with height and color changed

Leading and actions

When an app navigates to a different screen, Flutter already adds a back button by default to the **AppBar**. However, we can also customize these icons at the beginning and end of the app bar via the leading and actions properties.

Code 11.30:

```
AppBar(
  title: Text("Demo Text"),
  leading: IconButton(icon: Icon(Icons.arrow_back), onPressed: (){}),
  actions: [
    IconButton(icon: Icon(Icons.more_vert), onPressed: (){})
  ],
),
```

This creates a back button at the start and a menu icon at the end. However, there can be additional icons at the end if needed.

Figure 11.31: An AppBar with actions added

CenterTitle

While Android prefers the title of the **AppBar** to the left, iOS prefers it centered. For this, the **centerTitle** parameter helps us.

Code 11.31:

```
AppBar(
  title: Text("Demo Text"),
```

```
  centerTitle: false,
),
```

This enforces the title to move to the start instead of centering title in all cases as shown in the following figure:

***Figure 11.32**: An AppBar with centerTitle disabled*

Scaffold

The **`Scaffold`** widget is a convenient way to set up the general notion of an app screen and helps us define many things such as **`AppBars`**, **`FloatingActionButtons`**, **`BottomNavigationBars`** and more. It is helpful to think of this as a chassis for a

car, which provides the basic structure for it and allows an appropriate space for attaching components (widgets). Most app screens will have a Scaffold as the root widget or close to the root widget.

Code 11.32:

```
Scaffold()
```

This creates… well, nothing yet. So, let us add more code and look at the results.

AppBar

We can easily define an **`AppBar`** inside a **`Scaffold`** using the **`appBar`** property.

Code 11.33:

```
Scaffold(
  appBar: AppBar(
    title: Text("Demo Screen"),
  ),
)
```

In the following figure, an **`AppBar`** is added to a Scaffold which has a title defined:

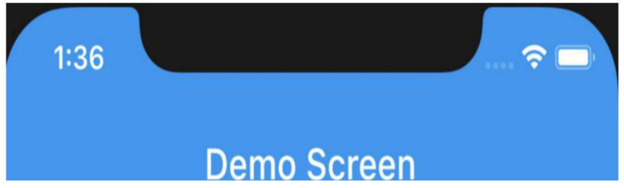


Figure 11.33: AppBar defined inside Scaffold

FloatingActionButton

A **FloatingActionButton** (FAB) is a distinguished button which usually carries out the primary action in a page. (As an example, create a note on a To-Do list app) The Scaffold widget automatically positions the FAB for us and allows more advanced behavior such as defining how the FAB and **BottomNavigation** overlap.

Code 11.34:

```
Scaffold(
  floatingActionButton: FloatingActionButton(
    child: Icon(Icons.add),
```



```
    onPressed: () {},
  ),
)
```

The preceding code creates a button like this on the lower right-hand corner:

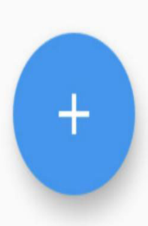


Figure 11.34: FAB defined inside the Scaffold

Bottom

The **bottom** property allows us to create navigation elements or widgets that permanently rest at the bottom of our screens. Let us look at one such possibility:

BottomNavigationBar

A **BottomNavigationBar** provides easy access to many related screens while keeping the **BottomNavigationBar** on the screen.

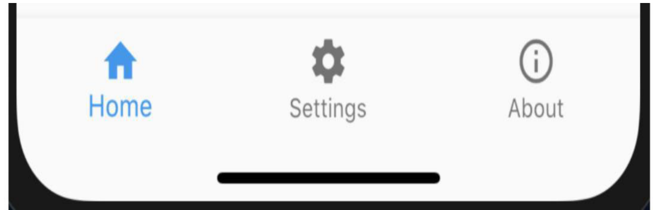


Figure 11.35: BottomNavigationBar inside a Scaffold

This is done via the **`bottomNavigationBar`** property of the Scaffold:

Code 11.35:

```
Scaffold(
      bottomNavigationBar: BottomNavigationBar(items: [
      BottomNavigationBarItem(
        icon: Icon(Icons.home),
        label: "Home",
      ),
      BottomNavigationBarItem(
```

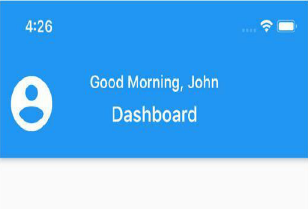


```
        icon: Icon(Icons.settings),
        label: "Settings",
      ),
      BottomNavigationBarItem(
        icon: Icon(Icons.info_outline),
        label: "About",
      ),
   ]),
)
```

Body

The body is the main content contained in the screen. The Scaffold also allows us to define if the body extends below the **`AppBar`** and **`BottomNavigation`**. This is where we show the main content on the screen (for example: the notes in a notes app).

A simple example of a complete app page using the concepts till now is:



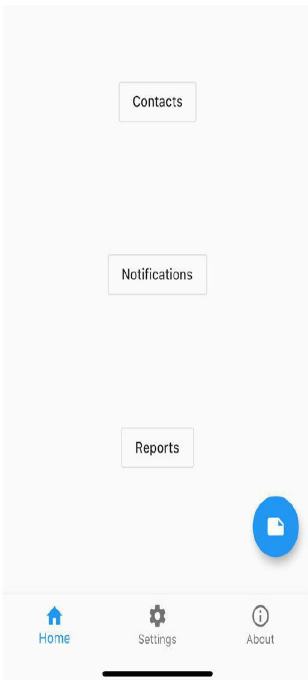


Figure 11.36: Body items defined inside a Scaffold

Source code:

Code 11.36:

```
Scaffold(
  appBar: AppBar(
    title: Column(
      children: [
        Text(
          "Good Morning, John",
          style: TextStyle(fontSize: 16.0),
        ),
        Padding(
          padding: const EdgeInsets.all(8.0),
          child: Text("Dashboard"),
        )
      ],
    ),
    toolbarHeight: 100.0,
```

```
  leading: IconButton(
    icon: Icon(
      Icons.account_circle,
      color: Colors.white,
      size: 60.0,
    ),
  ),
),
bottomNavigationBar: BottomNavigationBar(
  items: [
   BottomNavigationBarItem(
     icon: Icon(Icons.home),
     label: "Home",
   ),
   BottomNavigationBarItem(
     icon: Icon(Icons.settings),
     label: "Settings",
   ),
```



```
   BottomNavigationBarItem(
     icon: Icon(Icons.info_outline),
     label: "About",
   ),
  ],
),
floatingActionButton: FloatingActionButton(
  onPressed: () {},
  child: Icon(Icons.note),
),
body: Center(
  child: Column(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    children: [
      OutlineButton(
        onPressed: () {},
        child: Text("Contacts"),
      ),
```

```
          OutlineButton(
            onPressed: () {},
            child: Text("Notifications"),
          ),
          OutlineButton(
            onPressed: () {},
            child: Text("Reports"),
          ),
        ],
      ),
    ),
  )
```

ListView

Lists are an important aspect of every app which consist of a number of similar items. It is important for lists to be efficient since they potentially contain many components and need to be properly managed.

Lists are defined with the **`ListView`** widget. Although Flutter allows infinite lists, the most practical lists need two things: an item count and a builder. A builder is a function which builds a widget for a particular index in the list.

The code for displaying a list simply displaying some text and the position of the item is:

Code 11.37:

```
ListView.builder(
  itemBuilder: (context, index) {
    return Text("Position $index");
  },
  itemCount: 10,
),
```

Which gives a basic list:

Position 0
Position 1

Position 2
Position 3
Position 4
Position 5
Position 6
Position 7
Position 8
Position 9

***Figure 11.37**: Scrollable List of items*

Here, the **itemBuilder** function is called for each index to build the item at that specific position. The only thing that changes is the value of the index. This also allows immense flexibility in the items that are created and do not need to stick to a specific type of layout for each item in the list.

It is also possible to make this list infinite by completely removing the **itemCount** property. However, in most practical apps, lists have a defined number of items such as notes in a To-do app.

Scroll direction

By default, **ListView** scrolls vertically. To change that, the **scrollDirection** property is used as:

Code 11.38:

```
ListView.builder(
  itemBuilder: (context, index) {
    return Text("Position $index");
  },
  itemCount: 10,
  scrollDirection: Axis.horizontal,
),
```

This changes the scroll direction to horizontal:

Position 0Position 1Position 2Position

***Figure 11.38**: Horizontal List implementation*

Scroll physics

The difference in scrolling a list in android and iOS is noticeable. In android, the list goes to the end and stays there, while in iOS, the list overshoots and goes back to the end causing a rubber band like scrolling effect. Flutter allows us to change the scroll physics on any platform but defaults to the appropriate behavior on all platforms, so the default app does not feel unnatural.

We can change the scroll physics of the list via the physics parameter. Let us try setting the parameter in such a way that the list never scrolls:

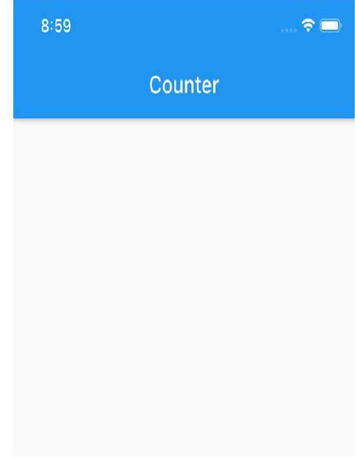
Code 11.39:

```
ListView.builder(
  itemBuilder: (context, index) {
    return Text("Position $index");
  },
  itemCount: 10,
  physics: NeverScrollableScrollPhysics(),
),
```

Here, **`NeverScrollableScrollPhysics()`** completely blocks scrolling. Similarly, some other types are **`ClampingScrollPhysics`** (default on Android, stops on end), **`BouncingScrollPhysics`** (default on iOS, bounces back on end).

Understanding the basic Counter app

A default Flutter project comes along with a Counter app. Since the basic widgets are completed, let us try to recreate a similar app. Here is what we will try to recreate:



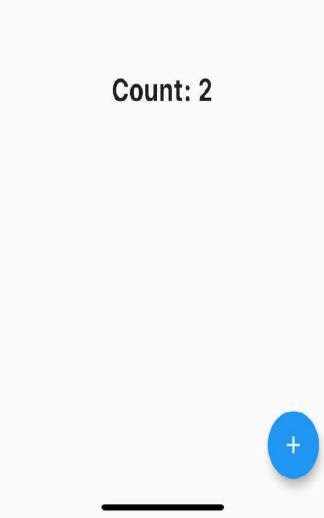


Figure 11.39: Basic Counter App UI

Let us take a look at the components of the app:

- A **Scaffold** for holding all components.
- The **AppBar** says **Counter**.
- The **Text widget** at the center displays the count.
- A **FloatingActionButton** with an icon which increments count.

Let us follow these steps one by one:

1. Adding a basic **Scaffold** to define a page layout:

   Code 11.40:

```
Scaffold()
```

2. Adding an **AppBar** on the top of the page with the text **Counter** on it:

   Code 11.41:

```
Scaffold(
  appBar: AppBar(
    title: Text("Counter"),
  ),
)
```

3. Here, we define a variable named **_count** in the state for storing the counter count. We also make some modifications to the text style since the text appears to be large and bold.

Code 11.42:

```
int _count = 0;

Scaffold(
    appBar: AppBar(
      title: Text("Counter"),
    ),
    body: Center(
      child: Text(
        "Count: $_count",
        style: TextStyle(
          fontSize: 26.0,
          fontWeight: FontWeight.bold,
        ),
      ),
    ),
  );
```

4. We use **`setState()`** to rebuild our UI after a state change. Here, when the **`FloatingActionButton`** is clicked, we update the value of the **`_count`** variable inside the **`setState()`** callback.

Code 11.43:

```
floatingActionButton: FloatingActionButton(
        onPressed: () {
          setState(() {
            _count++;
          });
        },
        child: Icon(Icons.add),
      ),
```

The full source code finally becomes:

Code 11.44:

```
Scaffold(
      appBar: AppBar(
        title: Text("Counter"),
      ),
      body: Center(
        child: Text(
          "Count: $_count",
          style: TextStyle(
            fontSize: 26.0,
            fontWeight: FontWeight.bold,
          ),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          setState(() {
            _count++;
          });
        },
        child: Icon(Icons.add),
      ),
    )
```


Conclusion

Now that we have defined a few basic widgets and how layout works in Flutter, we will move on to something new before we go on to creating more complex apps. Although there was quite a lot of information about widgets, the number of actual available widgets is vast and demands more exploration.

In the next chapter, we will learn about connecting to the network and making API calls in Dart. This will allow creating more complex and feature-rich applications in Flutter.

Questions

1. What is the use of a **`Scaffold`**?

2. What is the difference between a `Stack` and a `Row`?
3. What are the different types of buttons in Flutter?
4. What is the difference between a `Column` and a `ListView`?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 12

Networking in

Flutter

Introduction

In today's digital age, mobile devices have become an integral part of our daily lives. With the growing demand for mobile apps, it has become essential for developers to have a solid understanding of internet networking to build robust and reliable mobile apps. Internet networking is the backbone of modern mobile apps, as it enables them to connect with various services and exchange data with servers. Without a proper understanding of internet networking, developers may face challenges in building scalable and efficient mobile apps that can deliver seamless user experiences. Therefore, learning internet networking is crucial for developers who aspire to build successful mobile apps in today's competitive market.

Structure

In this chapter, we will discuss the following topics:

- Connecting to the internet
- The TMDB API
- The HTTP packages
- Understanding data models

- Modifying the fetch function
- Ways to create data models
- Problems with building UI from data in API calls
- Building UI from network data

Objectives

After completing this chapter, you should be able to set up a network connection from your Flutter app and communicate with services on the internet. You will also understand the basic workings of an API, which is an essential aspect to understand when creating any application.

Connecting to the internet

Fetching data from the internet is an important part of most mobile apps. Since network operations are carried out frequently and in several ways, it is good to know ways to not only connect to the internet but also to display the data acquired by doing it. There are several things to do when an app uses data off the internet:

- The app needs to connect to the appropriate API endpoint / server to fetch data off it.
- The data needs to be converted into usable objects in the code.
- Deciding when to fetch network data to display in the application.

For this chapter, we will use the example of **The Movies DataBase** (**TMDB**) API (**https://www.themoviedb.org**) for fetching a list of movies and displaying cards with the poster of the movies. This will involve making the network calls to fetch data, writing data models, and then efficiently building a UI based on it.

The TMDB API

Networking in Flutter often involves connecting to a REST API to fetch certain kinds of data. REST APIs are an easy way to get data from the web. They usually contain several endpoints through which we can fetch specific categories of data. In the example for this chapter, we will be using the TMDB API to create an app that fetches real-world data for display.

The TMDB API allows us to fetch all kinds of movie data, such as trending movies, new releases and movie trailers. It is a great starting point for using REST APIs in Flutter apps since the API allows variety without adding unnecessary complexities, as shown in the following figure:

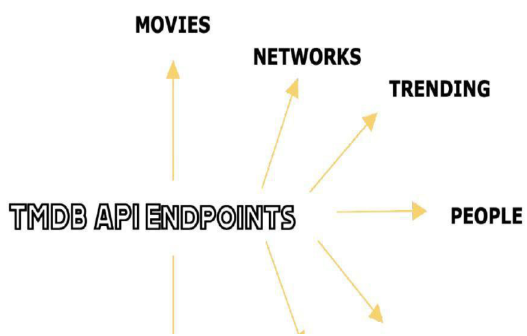
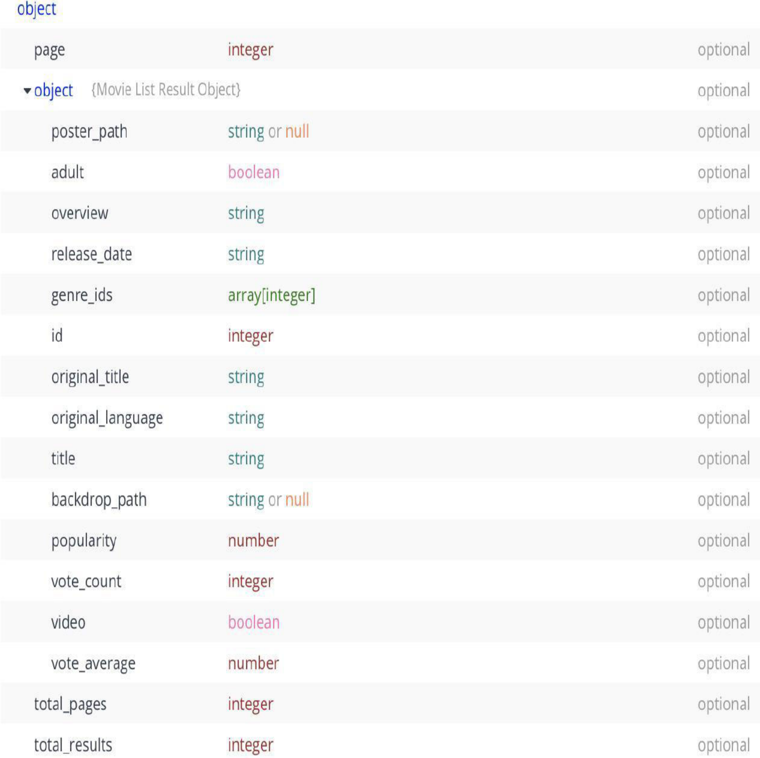


Figure 12.1: The various endpoints of the TMDB API

The preceding figure shows some of the things the TMDB API offers. We will be using the trending movies endpoint to display movie cards to the users. Before using the TMDB API, make sure you get an API key from the TMDB developers' website, which attributes your API usage to your account. We will be supplying this API key and request the trending movies endpoint for data.

Every endpoint of the API provides a response with different properties. Let us take a look at the Trending Movies response:



object

Property	Type	
page	integer	optional
▾object {Movie List Result Object}		optional
poster_path	string or null	optional
adult	boolean	optional
overview	string	optional
release_date	string	optional
genre_ids	array[integer]	optional
id	integer	optional
original_title	string	optional
original_language	string	optional
title	string	optional
backdrop_path	string or null	optional
popularity	number	optional
vote_count	integer	optional
video	boolean	optional
vote_average	number	optional
total_pages	integer	optional
total_results	integer	optional

Figure 12.2: Properties of the API response

The response contains a list of movies and other properties like **`total_results`**. An individual movie contains things like the movie poster, the title, language, popularity, and so on, which we can display when constructing the data model (more about this in the *Understanding Data Models* section ahead) and the respective UI elements.

The HTTP packages

The HTTP packages

HyperText Transfer Protocol (**HTTP**) is a mechanism for data transfer between web servers and clients. HTTP uses a request-response model for getting and supplying data between server and client. The HTTP Dart package allows us to easily exchange data with an API/back end of our choice. There are also other packages that extend the functionality of the HTTP package, which will be discussed later in this section.

You will find the 'http' package on **pub.dev**:

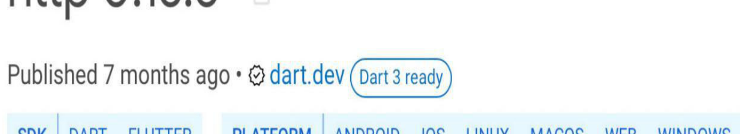


Figure 12.3: The http package for Flutter

HTTP uses several methods, such as the following:

- `GET`: Used for retrieving data from the server
- `POST`: Used for supplying data to the server
- `PUT`: Used to modify a certain resource on the server
- `DELETE`: Used to delete data on the server
- `HEAD`: The same as `GET` without the response data

There are also other methods that are not as important here. Note that these are conventions, and some request methods may break the usual norms and might do something unexpected on the server side.

As we only intend to retrieve data, we only need to use the `GET` request, nothing else. We do need to supply the API key to use the data, but this does not need a separate `POST` request; we will see how the URL is constructed to use the API key.

To make the `GET` request for the retrieval of the trending movies, we need to construct the appropriate URL. Let us take a glance at what that looks like:

```
URL = Base API URL + Endpoint + API key as parameter
```


First, use the base URL:

Code 12.1:

```
var baseUrl = "https://api.themoviedb.org/3";
```

Each API has a base URL, which can be found on the home page of the API documentation.

documentation.

Then, the endpoint documentation always has the endpoint:

Get Trending
GET /trending/{media_type}/{time_window}

***Figure 12.4**: The API endpoint to use for fetching trending movies in TMDB*

This also allows us to specify the media type and the period for which we are trying to find trending movies (as an example, movies trending over the last week or movies trending over the last month). Our media type is movie and for now, set the time window to the day's trending movies using the following command:

Code 12.2:

```
String trendingEndpoint = "/trending/movie/day";
```

Let us create a variable to store our API key:

Code 12.3:

```
String apiKey = "YOUR_API_KEY";
```

Finally, add it all up:

Code 12.4:

```
String url = baseUrl + trendingEndpoint + "?api_key=$apiKey";
```

Here, the API key is added as an extra parameter with the request URL, which allows TMDB to identify the request as valid and attribute usage to the responsible account.

Now, let us use HTTP to get data from the URL:

Code 12.5:

```
import 'package:http/http.dart' as http;


String baseUrl = "https://api.themoviedb.org/3";


String trendingEndpoint = "/trending/movie/day";
```

```
String apiKey = "YOUR_API_KEY";

void getTrendingMovies() async {

    String url = baseUrl + trendingEndpoint + "?api_key=$apiKey";

    var response = await http.get(Uri.parse(url));

}
```

This is an example of fetching API data via a **Dart** function. Note that we are not returning any data and will come back to this example in a while. The **HTTP** package is imported as 'http' here, which is optional. If the import is unnamed, we can simply use the **get()** function directly.

One thing to note is that the function is marked **async**, meaning it may need to work asynchronously. This is essential to any function that may need to fetch network data since fetching the data takes an indefinite amount of time. The **await** keyword tells the function that it needs to wait for the operation to finish before moving on to the next line. This is used for the network call since we do not know how long it needs.

There are several things to note here:

- Marking the function **async** does not make it asynchronous by default. The function only goes asynchronous when the first **await** is reached. Till then, the function remains synchronous.
- Async functions do not run on a separate thread; rather, they use a clever way to run on the same thread as synchronous functions.
- Functions marked async either have a **void** return type or return a **Future** in which they return a type but not immediately.

Before we move on to the next stages of building, it is important to know that there are also alternatives to the HTTP package. One of these is Dio, a popular package for network communication:

Figure 12.5: The dio package for extending http functionality

Dio is an HTTP client for Flutter supporting global configuration, interceptors, request cancellation, file uploading/downloading, timeout, custom adapters, and more.

This allows us to extend HTTP functionality and add new features to our app easily.

Understanding data models

Before we look at data models, let us take a look at what we currently receive from the network call:

Code 12.6:

```
void getTrendingMovies() async {

    String url = baseUrl + trendingEndpoint + "?api_key=$apiKey";

    var response = await http.get(Uri.parse(url));

    print(response.body);

}
```

This response contains multiple things, such as the data in **JavaScript Object Notation** (**JSON**) and the response status code that lets us know whether the response was successful (2xx) or there was any kind of error vis-a-vis authentication, data access, and so on (Codes such as 3xx, 4xx, 5xx).

JSON is a lightweight format often used to carry data from a web server to a client. The received data is a string that we can decode to a JSON map. While this is usable on its own, it may need to be used frequently throughout the app. It is not advisable to use maps of data since the variable names add to cognitive overhead. To solve this issue and utilize data more easily, we use data models.

Data models are class representations of the supplied data, essentially converting the map of data to actual variables in the class. This reduces user errors when using fields in the class while making the code more understandable. Data models usually also contain functions to convert the JSON data to a class and vice versa.

As an example of a data model, let us take the trending movies response that we get via the TMDB API. Note that while data models can be large, they usually do not need to be created manually, and creation can be automated.

Here is an example of a model that represents the response given back by the trending movies API endpoint of TMDB:

Code 12.7:

```
import 'dart:convert';

TrendingResponseModel trendingResponseModelFromJson(String str) =>
TrendingResponseModel.fromJson(json.decode(str));

String trendingResponseModelToJson(TrendingResponseModel data) => json.
encode(data.toJson());

class TrendingResponseModel {
    TrendingResponseModel({
        required this.page,
        required this.results,
        required this.totalPages,
        required this.totalResults,
    });

    int page;
    List<Result> results;
    int totalPages;
    int totalResults;

    factory TrendingResponseModel.fromJson(Map<String, dynamic> json) =>
TrendingResponseModel(
        page: json["page"],
        results: List<Result>.from(json["results"].map((x) => Result.
fromJson(x))),
        totalPages: json["total_pages"],
        totalResults: json["total_results"],
    );

    Map<String, dynamic> toJson() => {
        "page": page,
        "results": List<dynamic>.from(results.map((x) => x.toJson())),
        "total_pages": totalPages,
        "total_results": totalResults,
    };
}
```

```
class Result {
    Result({
        this.adult,
        required this.backdropPath,
        required this.genreIds,
        required this.id,
        required this.originalLanguage,
        this.originalTitle,
        required this.overview,
        required this.posterPath,
        this.releaseDate,
        this.title,
        this.video,
        required this.voteAverage,
        required this.voteCount,
        required this.popularity,
        this.firstAirDate,
        this.name,
        this.originCountry,
        this.originalName,
    });

    // .. Define variables

    factory Result.fromJson(Map<String, dynamic> json) => Result(
        //.. Creates model variables from JSON
    );

    Map<String, dynamic> toJson() => {
        //.. Creates JSON from model variables
    };
}

enum OriginalLanguage { EN, ZH }
```

```
final originalLanguageValues = EnumValues({
    "en": OriginalLanguage.EN,
    "zh": OriginalLanguage.ZH
});

class EnumValues<T> {
    Map<String, T> map;
    late Map<T, String> reverseMap;

    EnumValues(this.map);

    Map<T, String> get reverse {
        reverseMap = map.map((k, v) => MapEntry(v, k));
        return reverseMap;
    }
}
```

Modifying the fetch function

Now that we have a data model, let us take our **fetch** function for the API and convert it to use data models instead:

Code 12.8:

```
void getTrendingMovies() async {
    String url = baseUrl + trendingEndpoint + "?api_key=$apiKey";
    var response = await http.get(Uri.parse(url));
    print(response.body);
}
```

First, let us convert the response to a **TrendingMoviesResponse** data model:

Code 12.9:

```
void getTrendingMovies() async {
    String url = baseUrl + trendingEndpoint + "?api_key=$apiKey";
```

```
    var response = await http.get(Uri.parse(url));

    var trendingMovieModel = trendingResponseModelFromJson(response.body);
```



```
    return trendingMovieModel;

}
```

Here, we used the **fromJson()** function to convert the **HTTP** response to a data model.

Now, the last thing we need to do is to change the **return** type of the function as the current function returns the model but does not declare the **return** type.

Code 12.10:

```
Future<TrendingResponseModel> getTrendingMovies() async {

    String url = baseUrl + trendingEndpoint + "?api_key=$apiKey";

    var response = await http.get(Uri.parse(url));

    var trendingMovieModel = trendingResponseModelFromJson(response.body);

    return trendingMovieModel;

}
```

Here, we have to wrap our **TrendingResponseModel** with a **Future<>** since the return is not immediate.

Ways to create data models

As can be seen from the size, manually writing data models is a tedious process and is not feasible with the tens or hundreds of models an app may potentially need. Hence, we need an easy way to generate models or even automate creation of a large set of them.

QuickType

QuickType is an easy way to generate data models by simply providing an example JSON response.

Figure 12.6: The quicktype.io logo

The input is just the example response from the trending endpoint we received from the TMDB API:

Name: TrendingResponseModel
Source type: JSON

```json
{
  "page": 1,
  "results": [
    {
      "adult": false,
      "backdrop_path": "/mY8oTSMlVHgkBtKsvanKnvT8rY9.jpg",
      "genre_ids": [
        10749,
        18,
        9648,
        53
      ],
      "id": 505379,
      "original_language": "en",
      "original_title": "Rebecca",
      "overview": "After a whirlwind romance with a wealthy wid
      "poster_path": "/bSKVKcCXdKxkbgf0LL8lBTPG02e.jpg",
      "release_date": "2020-10-16",
      "title": "Rebecca",
      "video": false,
      "vote_average": 6.4,
      "vote_count": 23,
      "popularity": 32.635,
      "media_type": "movie"
    },
```

Figure 12.7: The source JSON to input into quicktype

The entire model is generated as output. Keep in mind that some things like enums are guessed by the generator and may be incompatible with the actual field values.

This is the partial output of the response:

```dart
// To parse this JSON data, do
//
//     final trendingResponseModel = trendingResponseModelFromJson(jsonString);

import 'dart:convert';

TrendingResponseModel trendingResponseModelFromJson(String str) => TrendingResponseModel.fromJson(json.dec

String trendingResponseModelToJson(TrendingResponseModel data) => json.encode(data.toJson());

class TrendingResponseModel {
    TrendingResponseModel({
        required this.page,
        required this.results,
        required this.totalPages,
        required this.totalResults,
    });

    int page;
    List<Result> results;
    int totalPages;
    int totalResults;

    factory TrendingResponseModel.fromJson(Map<String, dynamic> json) => TrendingResponseModel(
        page: json["page"],
        results: List<Result>.from(json["results"].map((x) => Result.fromJson(x))),
```

```
        totalPages: json["total_pages"],
        totalResults: json["total_results"],
    );

    Map<String, dynamic> toJson() => {
        "page": page,
        "results": List<dynamic>.from(results.map((x) => x.toJson())),
        "total_pages": totalPages,
        "total_results": totalResults,
    };
}
```

Figure 12.8: The resultant model class given by the service

Code generation

While QuickType is a convenient and easy way to generate data models, it is a third-party service and becomes infeasible with larger apps. What we need now is a reliable way to generate data models that can deal with the complexity of large-scale apps while also making writing data models relatively effortless. This is where code generation comes in.

Code generation allows us to create data models using generation libraries that require us to write a basic class with annotations for fields, and the libraries take care of the rest of the work. One of the main code generation libraries is **json_serializable**.

An example out of the **json_serializable** docs to create a basic model is as follows:

Code 12.11:

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

part 'example.g.dart';

@JsonSerializable(nullable: false)
class Person {
  final String firstName;
  final String lastName;
  final DateTime dateOfBirth;

  Person({required this.firstName, required this.lastName, required this.
dateOfBirth});

  factory Person.fromJson(Map<String, dynamic> json) =>
_$PersonFromJson(json);

  Map<String, dynamic> toJson() => _$PersonToJson(this);
}
```

To create a model using code generation, we need to mark a class with the **@**

JsonSerializable annotation, which says that the class is used for generating a model.

Two functions, i.e., converting the model to and from JSON, also need to be included.

Finally, to generate the models, run the following command in the terminal: **flutter pub run build_runner build**

While data models here are talked about in context of JSON, they are also used when data needs to be persisted offline. The same models can be used with databases like SQLite to store data offline to provide persistence and continuity before online data is fetched.

Problems with building UI from data in API calls

Let us take a look at the first line of the function we used to fetch the API data:

Code 12.12:

```
Future<TrendingResponseModel> getTrendingMovies() async {

    //...

}
```

The function used is marked asynchronous, meaning it takes an indefinite amount of time to complete. This does not work well with how we build Flutter UIs.

Let us look at a build function building a simple page:

Code 12.13:

```
build() {
  return Scaffold(
    appBar: AppBar(),
    body: ListView(),
  );

}
```

We need to fetch the data to display it in **`ListView`**. If it was a simple synchronous function, this would have worked:

Code 12.14:

```
build() {

  var data = getTrendingMovies();

  return Scaffold(
    appBar: AppBar(),
    body: ListView(),
  );

}
```

However, this does not work since **`getTrendingMovies()`** does not complete immediately; and if this was valid, the UI may freeze up since it is waiting for the API data to build the screen. Here's the crux of the problem: how do we make the synchronous **`build()`** function and the asynchronous **`getTrendingMovies()`** work well together?

Building UI from network data

Let us see how to solve the sync - async problem for fetching and displaying API data. First, let us look at what we want to build:

***Figure 12.9**: A simple ListView displaying received movies*

Here, we have a simple page with a **`ListView`** for displaying the details of the movie items received.

Since our problem is regarding fetching network data in the **`build()`** function, we can create a workaround by calling it in **`initState()`** instead, which is called when the page is first created:

Code 12.15:

```
List<Movie> movies = [];

@override
void initState() {
  super.initState();
  fetchTrendingMovies().then((response) => setState(() {
   movies = response?.movies ?? [];
  }));
}
```

The **`then()`** function provides us with a callback that fires once data is created. Here, we can simply store the movie data in a list and then display the data in a **`ListView`** like this:

Code 12.16:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Trending Movies'),
    ),
    body: ListView.builder(
      itemBuilder: (context, position) {
        return Card(
          child: ListTile(
            title: Text(movies[position].originalTitle),
            subtitle: Text(movies[position].overview),
          ),
        );
      },
      itemCount: movies.length
    ),
  );
}
```

While this method works, it does not use a declarative style like the rest of the Flutter widgets. This makes it an outlier, and if a page contains various async calls, it quickly makes it unmanageable.

Another method to use a declarative style of code comes in the form of **FutureBuilder**.

FutureBuilder is a widget that builds a subtree (widgets) based on the data received via a future. The structure of a **FutureBuilder** is as follows:

Code 12.17:

```
FutureBuilder(

  future: _yourFutureFunction(),

  builder: (context, snapshot) {
    // Build UI here
  },

)
```

FutureBuilder calls our future and provides us data via the snapshot. When the data is yet to be received, **snapshot.data** is null, and we can check on the availability of data via **snapshot.hasData**.

Our **build()** function in the **FutureBuilder** case would be as follows:

Code 12.18:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Trending Movies'),
    ),
    body: FutureBuilder(
      future: fetchTrendingMovies(),
      builder: (context, snapshot) {
        List<Movie> movies = snapshot.data;

        if (!snapshot.hasData) {
          return Center(
            child: CircularProgressIndicator(),
          );
        }

        return ListView.builder(
          itemBuilder: (context, position) {
            return Card(
              child: ListTile(
```

```
              title: Text(movies[position].originalTitle),
```



```
              subtitle: Text(movies[position].overview),
            ),
          );
        },
        itemCount: movies.length,
      );
    },
  ),
 );
}
```

Here, we do not need to call async functions separately; all the UI and async network calls meld into one unit.

One thing to note is that **`FutureBuilder`** should be used to display data only and should not be mixed with other async calls since the future fires every rebuild.

Conclusion

This chapter discussed how to fetch network data in a Flutter app since most apps need to interact with the web in some way. Along the way, it also introduced ways to deal with asynchronous functions.

Since we have now dealt with the concept of data models, the next chapter introduces dealing with data storage in Flutter, which will help us store user preferences or store data offline to make apps work offline.

Questions

1. What is a data model?
2. What methods does HTTP allow?
3. Why is a method communicating with the network always async?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 13
Local Data Persistence

Introduction

In most app development cycles, the first part after the planning stages is usually developing core functionality, UI, back-end features, and so on. (For some people, this may also be writing a LOT of tests instead, but for the next statement to function, let us stick with what I said.) The next part of development is usually bug fixes, optimizing features, and making additions that could save cost and/or improve user experience. Often, this comes in the form of offline data persistence - storing app data on the device. This could be in a million different ways - storing songs always played by the user, storing the user preference between Celsius or Fahrenheit (or Kelvin if they only deal with absolute scales), storing a list of messages, and so much more. Data persistence could be added for enhancing user experience (lower loading times and intermediary screens), for performance if repeated network data does not need to be fetched again, and so much more.

Whatever the reason, any major app has some kind of data storage on the device. For creating high-quality apps over a large time scale, learning to store data locally and knowing the different ways to store it are essential skills for any app developer. In this chapter, we intend to learn a few ways to do exactly this. However, I do

In this chapter, we intend to learn a few ways to do exactly this. However, I do recommend checking other approaches since new packages that deal with storing data pop up all the time and some may even fit your specific needs better.

Structure

In this chapter, we will discuss the following topics:

- Getting started
- Types of data
- A note on multi-platform support
- SharedPreferences
- sqflite
- Hive

Objectives

After studying this chapter, you should understand the intricacies of local data persistence in a Flutter app. You should also be able to implement data persistence using a few common methods and packages, and have a basic understanding of the other methods.

Getting started

Most apps require network calls to fetch data to display in the UI. To enhance user experience, some apps require/add offline support. This may be because some operations do not require network calls, like adding notes to a note app for example, to cache data received and make the app usable offline, or to avoid the pesky loading times at the start of the app. Local data persistence is the ability to persist, that is, save data offline (locally).

There are several ways to save data, which depend on the type of data needed to be saved and the type of database needed to save it. In this chapter, we intend to dig deeper into the topic of data persistence in Flutter.

Types of data

Not all data is created equally in an app. How the data is going to used defines how it is going to be stored. Depending on the types of data, our implementation

changes. This is not always easy to define; apps can also completely store larger data online and only lighter data offline. In these types of cases, implementing full-blown database logic might seem like a stretch.

Primarily, there are two types of data:

- Data that exists independently of other data and consists of simple values
- A larger table of complex data which is related to each other

This is not necessarily intuitive with the given definition, so a better example is needed. For understanding various types of data apps may need to store, let us look at two examples.

Settings / preferences

Almost all apps contain a settings screen with preferences in them, something reminiscent of the following screen:

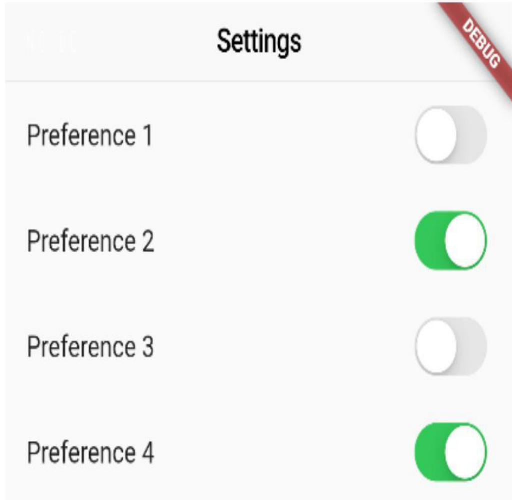


***Figure 13.1**: The general layout of a preferences screen in an app*

Preference or settings data can be as simple as the date format being DD/MM/YYYY or MM/DD/YYYY. This is not usually data required online on the server side but is required to show dates in the app.

One thing to note is that we generally know the types and number of preferences or settings displayed in the app. Also, we generally do not need any complex filters/sorting or any kind of complex queries on the stored data.

App feed

Ever since the pandemic hit, doom scrolling levels have hit all-time highs. Having feeds in the app boosts user retention massively; however, feeds are usually data heavy and not as easy to fetch again from a server.

Feeds are usually something like the following figure which demonstrates a basic Twitter clone made in Flutter:

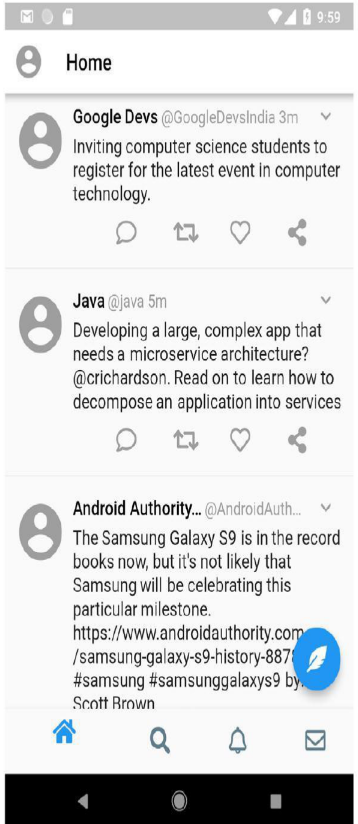


Figure 13.2: An activity feed in an app

A feed consists of a series of complex posts, usually by different users or in different categories, that are mixed to give the user a continuously flowing stream of data (Instagram, Facebook, Twitter, and most other apps have some kind of feed feature).

Feed posts contain complex data from several categories: the main feed content (picture + text), likes/reactions to the posts, comments on the post that have their own type of data, such as the information of the person who commented, and so on.

A more sophisticated method of storage is required for feeds, one that allows us to

add heavily nested, (mostly) structured data easily and has the ability to sort and filter to some extent.

A note on multi-platform support

Several people are first attracted to Flutter looking at its multi-platform app development capability. However, being cross-platform is only a consequence of how Flutter deals with UI. Since databases are stored natively, they need to be supported

for all platforms because the low-level code that deals with them is written natively by the developers of the respective plugins, not in Dart.

Before choosing any persistence solution, confirm that it supports all platforms that your application is aiming for. As more and more platforms join the Flutter ecosystem, there may be a few lags, but development for plugins is a constant process and solutions will arise in due time. And if they don't, plugin/package development in Dart is a breeze compared to some other frameworks.

SharedPreferences

As the name suggests, `SharedPreferences` allow you to store simple data by virtue of key-value pairs. This allows for storage of things like settings, preferences in the app, or any data that is preferably not associated with anything but itself.

Even if an app does not intend to store any kind of heavy data on device, most, if not all, serious apps must use `SharedPreferences` or something similar. This may be for many reasons:

- Identifying whether it is the first install of the app
- Identifying whether the tutorial is to be shown for every page
- Storing basic info about local users and so on

Note that `SharedPreferences` are visible with root access; do not store any sensitive data like passwords on it. If absolutely needed, ensure that said data is encrypted.

Adding SharedPreferences to your app

`SharedPreferences` are related to the first type of data discussed, which is mostly concerned with smaller values with little to no relation with any other data.

To add `SharedPreferences`, we need a Flutter plugin since their implementation also requires native code, which we preferably want to avoid. Luckily, the Flutter team has us covered: the `shared_preferences` plugin (found on `pub.dev`) is one of the most popular Flutter plugins.

First, we add the plugin as a dependency to our **pubspec.yaml** file:

Code 13.1:

```
dependencies:
  shared_preferences: ^2.0.18
```

After this, you can run the commands **flutter pub get** or **flutter packages get** on the terminal or through the IDE to retrieve the packages.

Using SharedPreferences

SharedPreferences are a key-value store, which means all elements stored inside can be identified by one key. However, this does not allow you to query your data in any sort of fancy way without writing that implementation yourself. No two pieces of data are linked to each other in this type.

To use **SharedPreferences** , we first initialize a **SharedPreferences** instance:

Code 13.2:

```
SharedPreferences prefs = await SharedPreferences.getInstance();
```

In databases, we have four kinds of operations, commonly referred to as CRUD: Create, Read, Update, and Delete.

We will see each one in the following section.

Create

This is how you can set a key-value pair in **SharedPreferences**:

Code 13.3:

```
String key = 'key1';
int value = 3;


prefs.setInt(key, value);
```

We use the **setInt()** function to set an integer value for the key. Similar functions exist for strings, string lists, and so on.

(Obviously, this function can be run without the local variables here, but they are given to add clarity).

Read

We use the same key that we stored the value with, to retrieve it again:

Code 13.4:

```
String key = 'key1';

int? value = prefs.getInt(key);
```

The **getInt()** function works to get the value corresponding to the key, with corresponding functions for other data types.

Additionally, if by any chance there is nothing stored in **prefs** under that key, we can use the **??** operator to specify a default value like this:

Code 13.5:

```
String key = 'key1';

int value = prefs.getInt(key) ?? 0;
```

This works since **getInt()** returns a null if the value does not exist.

Update and delete

Update and delete do not have special functions in this type of storage since the data contained is stored in a relatively simple format.

Hence, we can use the set function to update a value:

Code 13.6:

```
String key = 'key1';
int updatedValue = 3;

prefs.setInt(key, value);
```

We can use the remove() method to delete the value:

Code 13.7:

```
String key = 'key1';

prefs.remove(key);
```

sqflite

sqflite

SQLite storage involves relational data rather than simple key-value pairs, like those used in **`shared_preferences`**. The data involved in SQLite may need more complex querying or manipulation operations.

Unlike **`SharedPreferences`**, which are a key-value store, most apps have more complex data that needs a better form of storage. SQLite is a structured database that allows normal SQL queries and data format. The **`sqflite`** plugin allows us to create and manipulate SQLite databases on platforms (except web at the time of writing).

Adding sqflite to your app

To add the **`sqflite`** plugin to your app, add it to **`pubspec.yaml`** first:

Code 13.8:

```
dependencies:
  sqflite: ^2.0.1
  path_provider: ^2.0.13
```

Creating a database

Since **`sqflite`** is an SQLite implementation, we create the database like a normal SQL database (which we would not cover in extreme detail here since SQL has tons of resources already). We also need a directory to store the database that we get through the **`getApplicationsDocumentsDirectory()`** function.

Here's how an example database initialization would look:

Code 13.9:

```
Directory documentsDirectory = await getApplicationDocumentsDirectory();

var path = join(documentsDirectory.path, "DemoDB.db");

return await openDatabase(path, version: 1, onOpen: (db) {
     // Any operation on open
}, onCreate: (Database db, int version) async {
     await db.execute("CREATE TABLE Person ("
         "id INTEGER PRIMARY KEY,"
```

```
        "name TEXT,"
        "age INTEGER"
        ")");
});
```

We get a path to create the database by using the storage path we retrieved and appending the database name to it. In this case, we are creating a new table called **Person**, which has an ID, name, and age.

We use the **openDatabase()** function to create the function if not created, and supply a version number. When the version number changes, we can also decide if we need to change the database in any way, in case of either an upgrade or a downgrade. This is important since a new version number may add or remove columns in the

tables we create, and modifying the existing tables to be compatible is critical to not breaking any production code:

Code 13.10:

```
return await openDatabase(path, version: 1, onOpen: (db) {
    // Any operation on open
}, onCreate: (Database db, int version) async {
    // Create database here
}, onUpgrade (Database db, int oldVersion, int newVersion) {
    // Write code to deal with upgrades here
}, onDowngrade(Database db, int oldVersion, int newVersion) {
    // Write code to deal with downgrades here
}
);
```

CRUD operations

Let us look at the operations we need to modify the SQLite database. Most of these operations can also be done via raw SQL queries, which is quite a cumbersome method. Using proper models and defined CRUD methods in the package can help remove some of this extra code.

Create

Since we have already investigated creating tables, let us look at adding rows into an already created table using a raw query:

Code 13.11:

```
var result = await db.rawInsert(
      "INSERT Into Person (id,name,age)"
      " VALUES (${yourId},${yourName},${yourAge})");

return result;
```

This ideally should be done with a model class, as shown here:

Code 13.12:

```
var result = await db.insert('Person', person.toJson());

return result;
```


Read

We need to query the tables we created to get data from them:

Code 13.13:

```
var result = await db.query("Person", where: "id = ?", whereArgs: [id]);

return result.isNotEmpty ? Person.fromJson(result.first) : null;
```

Update

To update values in the table, we can use the **`update`** function:

Code 13.14:

```
var result = await db.update("Person", person.toJson(), where: "id = ?",
whereArgs: [person.id]);

return result;
```

Delete

Delete is similar to the other functions:

Code 13.14:

```
db.delete("Person", where: "id = ?", whereArgs: [id]);
```

Hive

Hive

Till now, we have looked at two types of databases: one involving simple key-value pairs, and the other involving complex tables of data. `Hive` is a third kind of approach: a NoSQL database that primarily deals with key-value pairs. `Hive` also has built-in encryption that provides additional security and is hence, more secure than normal `SharedPreferences`.

While `Hive` does have the ability to bunch objects together, there are no advanced query mechanisms, making it less suitable for more complex use cases. `Hive` stores objects under 'boxes', which are essentially containers that can be used to separate different kinds of data.

Note: There do exist some features that allow us to relate multiple objects (such as HiveList), but this is relatively less powerful than a normal SQL database. Hive, on the other hand, claims to be much faster from the benchmarks on its official documentation.

Adding Hive to your app

This is how you can add `Hive` to your app:

Code 13.15:

```
dependencies:
  hive: ^[version]
  hive_flutter: ^[version]

dev_dependencies:
  hive_generator: ^[version]
  build_runner: ^[version]
```

Basic data storage

Like the `shared_preferences` package from earlier, we can store simple key-value pairs in `Hive`. Boxes contain values, which are loaded into memory when the boxes are opened.

To start with, we define a box:

Code 13.16:

```
var box = await Hive.openBox('demoBox');
```

We can now add key-value pairs to the box:

Code 13.17:

```
box.put('demoKey', 'Demo Value');


box.put('demoList', [1, 2, 3, 4, 5]);


box.put('demoInteger', 3);
```

`Hive` also has lazy boxes that do not load all values from memory when open; rather, they do it lazily.

We can use them as follows:

Code 13.18:

```
var lazyBox = await Hive.openLazyBox('demoBox');


var value = await lazyBox.get('demoLazyKey');
```


`Hive` also contains encrypted boxes that allow for data encryption:

Code 13.19:

```
var encryptionKey = Hive.generateSecureKey();


// Please store this encryption key somewhere, for example using a
package like flutter_secure_storage since this will be required to
unencrypt the values in the box.


var encryptedBox = await Hive.openBox('demoEncrypedBox',
encryptionCipher: HiveAesCipher(encryptionKey));


encryptedBox.put('encryptedKey', 'encryptedValue');
```

Storing objects in Hive

Sometimes, we need to store more complex objects than simple primitive values in the database. We take advantage of **`TypeAdapters`** for this. **`Hive`** has generative tools written for this:

Code 13.20:

```
@HiveType(typeId: 1)
class Person extends HiveObject {
```

```
@HiveField(0)
  int id;

@HiveField(1)
  String name;

@HiveField(2)
  int age;

  Person({
    required this.id,
    required this.name,
    required this.age,
  });

}
```


After this, we need to run code generation, which generates the respective adapter:

Code 13.21:

```
flutter packages pub run build_runner build
```

This generates a class called **PersonAdapter**. After this, we need to register the adapter so that **Hive** can use it.

Code 13.22:

```
void main() async {

  await Hive.initFlutter();
  Hive.registerAdapter(PersonAdapter());

  await Hive.openBox<Person>('demoList');
  runApp(MyApp());

}
```

Hive also has some neat features, like listening to changes on the objects:

Code 13.23:

```
var box = Hive.openBox('personBox');
```

```
ValueListenableBuilder(
 valueListenable: box.listenable(),
   builder: (context, Box<Person> box, _) {
     return ListView.builder(itemBuilder: (_, index) {
       return Text(box.getAt(index)!.name);
     }
   );
 },
)
```

Conclusion

While a good-looking UI is significantly important to a Flutter app, there are several factors that determine whether the user experience matches up. Setting up local data persistence allows for faster loading, better performance, fewer API calls, and much more. Knowing how to implement data persistence is critical to becoming a

top-tier Flutter developer. It is also helpful to know some other powerful ways of storage, such as Drift (earlier named Moor), which is a SQL-based database with cross-platform functionality. Better and faster options will always come up at some point, so staying up to date with available options is important.

In the next chapter, we will discuss concepts like theming, navigation, and state management, which are important when creating larger applications in Flutter.

Questions

1. What is local data persistence?
2. What are the benefits of having local data storage?
3. How can we implement data persistence in Flutter?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 14

Theming, Navigation, and State Management

Creating complex Flutter apps can be an experience worlds apart from the basic principles taught in several courses and general literature. This is in part because a large production has several parts working in unison and needs to be scalable in the long term since the features, architecture, team working on the project, general priorities, and more can change over the period of development and after the first release is published.

When going beyond the simple one-page demo apps taught in courses, there are several things to master to create a large-scale app in production. One is theming, which creates themes that are applied throughout the app and reduces the duplication of code everywhere. The second is navigation, which structures the screens of the app (and more), and the third is state management, which manages data in the overall app independent of the widgets currently shown. These three are critical aspects of Flutter app development, without which most apps would have significant amounts of code or dismal levels of performance.

This chapter explores these three topics in detail as they are of great importance to building a full production-level Flutter app. There are also other aspects, such as testing and deploying the app, which will be discussed later.

Here is the structure of the chapter ahead:

Structure

In this chapter, we will discuss the following topics:

- Adding theming
- Creating and adding themes
- Adding dark mode
- Understanding navigation
- Navigator methods
- Introduction to state management
- The problems with setState()
- InheritedWidget
- Provider
- Riverpod
- bloc/flutter_bloc

Objectives

After completing this chapter, you should be familiar with the various components of a complex Flutter app. Theming reduces code duplication and maintenance time by storing theme data at a centralized location, and navigation describes how app pages are structured. State management manages app data across screens; knowing how to implement various state management solutions is important for building large applications.

Adding theming

If you are new to the concept of theming in apps, try to imagine what the word 'theme' itself means to you. Most people imagine themes as common colors or accents popping up within a certain object. Theming is similar, but one layer deeper.

Imagine that there is a design given to you to implement for a Flutter app in which most text is colored in a certain way, the buttons have a certain color, and so on. The most obvious way to do this is to define colors when implementing the UI.

Consider the following example:

Code 14.1:

```
ElevatedButton(
 onPressed: () {
   // Your action here
 },
 child: Text(
   'Click here',
   style: TextStyle(
     color: Colors.white,
     fontSize: 24.0,
     fontWeight: FontWeight.w300,
     fontFamily: 'YOUR_FONT',
   ),
 ),
 style: ButtonStyle(
   backgroundColor: MaterialStateProperty.all(Colors.amber),
 ),
);
```

This is already a lot of code for defining a single button. The issue with this now is that throughout the app, there will be many buttons, a lot of text, and so on. If we end up defining the same color and styles everywhere, it would be a lot of code duplicated for very little gain.

However, the bigger problem arises when this theme needs to be changed. If the code must be changed, let alone while running, the styles need to be changed throughout the app. Here, we see the definition and purpose of theming: centralizing the colors and styles used in the app and making them much easier to change either in development or in runtime.

Creating and adding themes

To start on the theming journey, you first need to create a centralized theme for your app. If you have ever seen the Flutter starter counter app, you may have seen this bit of code:

Code 14.2:

```
return MaterialApp(
  title: 'Flutter Demo',
  theme: ThemeData(
```

```
  // This is the theme of your application.
  //
  // Try running your application with "flutter run". You'll see the
  // application has a blue toolbar. Then, without quitting the app, try
  // changing the primarySwatch below to Colors.green and then invoke
  // "hot reload" (press "r" in the console where you ran "flutter run",
  // or simply save your changes to "hot reload" in a Flutter IDE).
  // Notice that the counter didn't reset back to zero; the application
  // is not restarted.
  primarySwatch: Colors.blue,
  ),
  home: const MyHomePage(title: 'Flutter Demo Home Page'),
);
```

Here, the theme parameter in **MaterialApp** defines the theme for the app. More accurately, this parameter creates a **Theme** Widget, which is an **InheritedWidget** that passes the theme down the tree. To do this yourself, you can also define a **Theme** Widget in your widget tree.

Note: The MaterialApp technically adds an AnimatedTheme widget to animate any changes in theme instead of showing abrupt changes from one color or style to the other.

Looking more into what can be added to the theme, there is an abundance of properties, some of which are listed here:

Code 14.3:

```
ColorScheme? colorScheme,
Color? colorSchemeSeed,
Brightness? brightness,
MaterialColor? primarySwatch,
Color? primaryColor,
Color? primaryColorLight,
Color? primaryColorDark,
Color? scaffoldBackgroundColor,
Color? bottomAppBarColor,
Color? cardColor,
Color? disabledColor
```

```
Color? disabledColor,
```



```
Color? secondaryHeaderColor,
Color? backgroundColor,
Color? dialogBackgroundColor,
Color? indicatorColor,
Color? hintColor,
Color? errorColor,
Color? toggleableActiveColor,
// TYPOGRAPHY & ICONOGRAPHY
String? fontFamily,
Typography? typography,
TextTheme? textTheme,
TextTheme? primaryTextTheme,
IconThemeData? iconTheme,
IconThemeData? primaryIconTheme,
// COMPONENT THEMES
AppBarTheme? appBarTheme,
// …
// AND MORE
```

Using these in the theme data defines the default styles, colors, and so on that the app uses.

There are some things, however, that don't have a default value, for example, the app has no clue if the **`Text`** widget is supposed to be a title, a subtitle, and so on. To define this, we need to get the theme of the app and apply the style ourselves:

Code 14.4:

```
// In top of tree
MaterialApp(
 title: 'Flutter Demo',
 theme: ThemeData(
   primarySwatch: Colors.blue,
   textTheme: TextTheme(
     headline1: TextStyle(
       fontWeight: FontWeight.bold,
       fontSize: 26.0,
     ),
   ),
```

```
 ),
 home: const MyHomePage(title: 'Flutter Demo Home Page'),
)

// In any descendant Widget
var textStyle = Theme.of(context).primaryTextTheme.headline1;

Text(
 'This text has the headline1 style',
 style: textStyle,
),
```

In this example, the primary text theme of the app defined in the **MaterialApp** is retrieved down the tree in a separate widget. This style is then applied to the required Widget. This obviously needs some extra effort to apply the theme, but it is very easy to modify during development or at runtime.

Adding dark mode

The moment an application is feature complete, the most asked-for feature in recent times is dark mode. Dark mode also makes an app easily usable at night-time with low brightness since the colors don't become blinding. This is especially important for apps with bright colors. With the advent of OLED and other technologies, darker themes become naturally better to look at.

Let's look at how to create a dark theme for your app.

Like the earlier examples used the theme parameter in the **MaterialApp**, there is also a **darkTheme** parameter. Alongside this, there is a **themeMode** parameter to define which theme is currently active. Here's what a code snippet looks like:

Code 14.5:

```
MaterialApp(
 title: 'Flutter Demo',
 theme: ThemeData(
   primarySwatch: Colors.blue,
   backgroundColor: Colors.white,
   // ...
```

```
  ),
  darkTheme: ThemeData(
    primarySwatch: Colors.blueGrey,
```



```
    backgroundColor: Colors.black,
    // ...
  ),
  themeMode: ThemeMode.dark,
  home: const MyHomePage(title: 'Flutter Demo Home Page'),
)
```

You can also use **`ThemeMode.system`** to follow the system theme instead of defining it yourself. Overall, Flutter offers a pretty easy way to switch between light and dark themes. However, if you need more themes, you may need to define your own and pass along the instance to the **`MaterialApp`**.

Understanding navigation

Apps are built with multiple screens, and defining their relationship and ensuring easy movement between them is critical to building a good app. Unlike other ways to build apps, Flutter does not have defined pages in an app; everything is a widget. Hence, it is imperative to know how to use navigation in an app. This consists of defining routes corresponding to every screen in the app.

To navigate between screens, Flutter has a widget called the **`Navigator`**. **`MaterialApp`** wraps the app with a default **`Navigator`** that is used when the navigation methods defined in the next section are used. If you do not use **`MaterialApp`**, you may need to use your own **`Navigator`**. In navigation, the screens are called routes. While there is no need to define specific routes in an app, it is recommended for apps since it provides a better structure and tidier code.

There are two ways to define route names in the **`MaterialApp`** widget: with the routes parameter and with the **`onGenerateRoute`** parameter. Both do the same thing overall, but the **`onGenerateRoute`** parameter allows the addition of any required business logic before navigating to a page.

Code 14.6:

```
MaterialApp(
  title: 'Flutter Demo',
  // Accepts a static map
  routes: {
    // Cannot pass arguments in routes
    '/': (context) => DefaultPage(),
```

```
  '/page1': (context) => Page1(),
  '/page2': (context) => Page2(),
  '/page3': (context) => Page3(),
```



```
 },
 // Can pass parameters
 onGenerateRoute: (settings) {
   final route = settings.name;
   final args = settings.args;
   // Return appropriate page according to route and args
 },
 initialRoute: '/',
 home: const MyHomePage(title: 'Flutter Demo Home Page'),
)
```

The **initialRoute** parameter defines the first loaded route of the app.

There are two **Navigator** types, the original **Navigator** and **Navigator** 2.0, both of which currently coexist.

Navigator methods

The **Navigator** class has several defined static methods that help navigate to and from different pages. The **Navigator** itself defines a stack of pages onto which we can push new pages. The top page is the currently shown page. In a simplistic model, there would have only been a push (go to new page) and pop (go to the page before) method. However, in a complex app, the navigation stack needs to be manipulated in complex ways without having multiple operations. As an example, once a user logs in, we replace the login page with the home page or similar. The equivalent of this would be pushing a new page and removing an element at the bottom of the stack, which is why we need more complex operations.

In this section, we will look at methods defined by the **Navigator**. Note that there are also equivalent methods for named routes that you can use, which are omitted here for brevity.

Push page

Pushing a page refers to pushing a new page on the navigation stack. This keeps the old page below the new page in the stack, and popping the new page reveals the old page.

page.

Here's the example code:

Code 14.7:

```
Navigator.push(
 context,
```



```
 MaterialPageRoute(
   builder: (context) => NewPage(),
 ),
);
```

`Navigator.push()` needs the context argument to retrieve the **`Navigator`** from the above the **`Widget`** in the tree. This **`BuildContext`** is used to find the app's **`Navigator`** by using an **`InheritedWidget`** which shall be seen later in this chapter. If a named route isn't used, a **`MaterialPageRoute`** needs to be created.

Pop page

Once a page is pushed onto the navigation stack, it can be popped from the stack, in which case the page below the popped page becomes the active page:

Code 14.8:

```
Navigator.pop(context);
```

Additionally, we can supply data back when popping the page, which is useful if the function that pushed the page onto the stack needs any kind of data back. As an example, a dialog can supply back selected data:

Code 14.9:

```
Navigator.pop(context, resultData);
```

When you are working with navigation, make sure the stack doesn't ever become empty by popping all pages.

Push replacement

Often, the current page needs to be replaced with a new one. If we pop and push another page to do this, it may cause a weird transition. Using the **`pushReplacement()`** function makes it appear as if a new page is being pushed onto the stack, but it also removes the original page from the stack.

Code 14.10:

```
Navigator.pushReplacement(
 context,
```

```
 MaterialPageRoute(
   builder: (context) => NewPage(),
 ),
);
```


Push and remove until

The **pushAndRemoveUntil()** function is a more powerful version of the push replacement function, which can remove more than one or even all the routes on the stack before it. Here's an example where the new page removes all the routes before it on the stack:

Code 14.11:

```
Navigator.pushAndRemoveUntil(
 context,
 MaterialPageRoute(
   builder: (context) => NewPage(),
 ),
 (Route<dynamic> route) => false,
);
```

Pop until

This function is similar to the previous one, with the exception that no new page is pushed, and pages are removed until a certain page is reached.

Code 14.12:

```
Navigator.popUntil(context, ModalRoute.withName('/oldScreen));
```

Introduction to state management

A mobile application has two kinds of state: ephemeral state and app state. Ephemeral state exists within a certain widget and does not persist across multiple screens. App state is needed across more than one widget. A common example of app state is a shopping cart that needs to exist, regardless of whether you are scrolling across items, in the item detail screen, or on the checkout screen. This data should not be stored within a screen or passed around when navigation is done.

How to manage app data is quite a serious question since a significant part of

the architecture of an app often depends on it. There are many solutions to state management in Flutter, and the number is increasing by the day. This chapter explores many of the most popular and tested ones.

A thing to note here is that there is no best method to manage state. *What's the best state management method* is heard a bit too often in most talks and conferences. Go with the solution most appropriate for your problem. If it is a small app with minimal rebuilds, even **`setState()`** works just fine. Going into larger apps, the more developed solutions pay dividends after implementation.

The Problems With setState()

The **`setState()`** function is used by **`StatefulWidget`** to notify the **`Widget`** state of changes that were made. This rebuilds the widget to reflect the changes. If state is modified but the function is not called, the UI is not updated.

An example of the function's use would be as follows:

Code 14.13:

```
var count = 5;


// On trigger
setState(() {
    count++;
});
```

While we use the closure supplied to the function to update the count value here, we can also do it before calling **`setState()`**, which works the same. This works because calling **`setState()`** marks the current frame as dirty. This was originally done with a **`markNeedsBuild()`** function, which could be simply called without passing a closure.

Code 14.14:

```
int count = 3;


// On button tap (or other change event)
count++;
markNeedsBuild();
```

However, this was removed since the new syntax makes it less error-prone and bunches up state mutations so that you are less likely to accidentally remove them. There are no real changes underneath, but it still calls **`markNeedsBuild()`**:

Code 14.15:

```
@protected
void setState(VoidCallback fn) {
  assert(fn != null);
  assert(() {
    if (_debugLifecycleState == _StateLifecycle.defunct) {
      throw FlutterError.fromParts(<DiagnosticsNode>[
        ErrorSummary('...'),
```



```
        ErrorDescription(
          '...',
        ),
        ErrorHint(
          '...',
        ),
        ErrorHint(
          '...',
        ),
      ]);
    }
    if (_debugLifecycleState == _StateLifecycle.created && !mounted) {
      throw FlutterError.fromParts(<DiagnosticsNode>[
        ErrorSummary('...'),
        ErrorHint(
          '...',
        ),
      ]);
    }
    return true;
  }());
  final Object? result = fn() as dynamic;
  assert(() {
    if (result is Future) {
      throw FlutterError.fromParts(<DiagnosticsNode>[
        ErrorSummary('...'),
        ErrorDescription(
```

```
        '...',
      ),
      ErrorHint(
        '...',
      ),
    ]);
  }
  // We ignore other types of return values so that you can do things like:
  //   setState(() => x = 3);
```


Here's a summary of what the function does internally:

- Verifies that the closure given to the function isn't null
- Checks if the lifecycle state is valid and the widget is mounted
- Makes sure the closure passed into the **`setState()`** function is not returning a **`Future`**
- If everything looks good, calls **`markNeedsBuild()`** to rebuild the **`Widget`**

While Flutter optimizes the rebuild on its own, calls to **`setState()`** do rebuild the entire **`Widget`** instead of simply rebuilding specific parts of the UI. When complex layouts are being built, this can drag down performance if not used appropriately.

Underneath, Flutter does not rebuild the actual **`RenderObject`** every time (read *Chapter 16, Advanced Flutter: Under the hood* to understand these better); it only recreates the **`Widget`** tree, which is immutable and only holds configuration. For many use cases, **`setState()`** does quite well and doesn't rebuild everything. However, whenever larger, more complex widgets that can't use the inbuilt optimization for rebuilds come in, **`setState()`** starts lagging behind the other methods.

To solve these issues, let's look at other, better methods of state management.

InheritedWidget

Since Flutter's widgets are structured in a tree, it is extremely useful to pass data down the tree. If all the data had to be passed manually using parameters, constructors would grow too large, and creating a tree would get impractical very quickly.

Since the objective of state management is to store app data independent of widgets, it makes sense to store it above the **`Widget`** tree entirely and pass down the data to descendants. This is where the *inherited* part of the **`InheritedWidget`** comes from. There are various **`InheritedWidgets`** used directly by users often without realizing it, such as the **`Navigator`** and **`Theme`**.

Let's see what an **InheritedWidget** looks like:

Code 14.16:

```
class DemoWidget extends InheritedWidget {

 const DemoWidget({
   Key? key,
   required this.value,
   required Widget child,
 }) : super(key: key, child: child);
```



```
 final String value;

 static DemoWidget of(BuildContext context) {
   final DemoWidget? result =
     context.dependOnInheritedWidgetOfExactType<DemoWidget>();
   assert(result != null, 'No DemoWidget found in context');
   return result!;
 }

 @override
 bool updateShouldNotify(DemoWidget old) => value != old.value;

}
```

Any **InheritedWidget** needs to extend the **InheritedWidget** class. It is also a general pattern to add an **.of()** static function so that the nearest **InheritedWidget** of the same type can be found up the tree. To find the nearest **Widget**, the **context.dependOnInheritedWidgetOfExactType()** function is used. There is also a **updateShouldNotify()** function, which is used to determine when the listeners of this **Widget** should be notified of the changes made to it.

Most times, **InheritedWidgets** are not written directly; other packages like **Provider**, which are a simplification and feature extension of them are used.

Provider

Provider is one of the most popular state management solutions in Flutter. It is an

extension of **InheritedWidget**, which adds simplicity and extra functionality. This extra functionality includes the ability to select changes to listen to, easy ways to deal with multiple types of data, lazy loading, and more.

To create a data model to work with the **Provider** package, **ChangeNotifier** is a class that is often used. **ChangeNotifier** is a **Flutter** class, not one from the package. It allows us to listen to changes easily.

Let's create a simple arithmetic model with a single result:

Code 14.17:

```
class ArithmeticModel extends ChangeNotifier {

 int result = 0;
```



```
 void add(int a) {
   result = result + a;
   notifyListeners();
 }

 void subtract(int a) {
   result = result - a;
   notifyListeners();
 }

 void multiply(int a) {
   result = result * a;
   notifyListeners();
 }

 void divide(int a) {
   result = result ~/ a;
   notifyListeners();
 }

}
```

Notice the **notifyListeners()** function that updates listeners about any changes to the data in the **ChangeNotifier**.

On the **Provider** side, we use **ChangeNotifierProvider** to initialize a **ChangeNotifier** above the tree:

Code 14.18:

```
ChangeNotifierProvider(
      create: (context) => ArithmeticModel(),
      child: MyApp(),
    ),
```

If there are multiple providers to add, you can also use the **MultiProvider** widget instead:

Code 14.19:

```
MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (context) => Model1()),
        ChangeNotifierProvider(create: (context) => Model2()),
        ChangeNotifierProvider(create: (context) => Model3()),
        …
      ],
      child: MyApp(),
    )
```

There are multiple ways to read/watch changes in the **Provider**. The **Consumer** widget allows you to listen to changes in the **Provider** :

Code 14.20:

```
Consumer<ArithmeticModel>(
  builder: (context, model, child) {
    return Text("Value: ${model.result}");
  },
)
```

Similar to the **InheritedWidget**, the **Provider** widget also has the **.of()** static method to retrieve the nearest **Provider** up the tree:

Code 14.21:

Code 14.21:

```
Provider.of<ArithmeticModel>(context, listen: false)
```

There are extension functions on **BuildContext**, making it easy to read:

Code 14.22:

```
final value = context.watch<ArithmeticModel>();
```

Provider also contains the **Selector** widget, which allows the selection of only the changes we need to listen to; all other changes are ignored:

Code 14.23:

```
Selector<ArithmeticModel, int>(
 selector: (_, provider) => provider.value,
 builder: (context, value, child) {
   return YourWidget();
```



```
 },
);
```

Along with these features, **Provider** allows caching other results like futures and streams using **FutureProvider** and **StreamProvider**.

Riverpod

Riverpod is a newer package, also made by the creator of the **Provider** package. **Riverpod** claims to solve several issues present with the **Provider** package on the basis that while **Provider** is a simplified, extended version of InheritedWidget, **Riverpod** is built completely differently. If you are in any way familiar with the **Provider** package, using **Riverpod** feels familiar, yet quite different in terms of code structure.

Riverpod itself does not even have a dependency on Flutter itself. There are three different packages associated with **Riverpod**; in this chapter, we focus on the **flutter_riverpod** package for brevity, but do look at the other **Riverpod** packages (**riverpod** and **hooks_riverpod**) on **pub.dev** to understand the full picture.

First of all, **Riverpod** allows globally defining providers; in general practice, this is actually a bad idea with most things, but **Riverpod** is not affected by it.

Here we're defining our arithmetic model again, but with **Riverpod**:

Code 14.24:

```
final arithmeticProvider = StateNotifierProvider<ArithmeticModel,
int>((ref) {
```

```
  return ArithmeticModel();
});

class ArithmeticModel extends StateNotifier<int> {
  ArithmeticModel() : super(0);

  void add(int a) {
    state = state + a;
  }

  void subtract(int a) {
    state = state - a;
  }
```

```
  void multiply(int a) {
    state = state * a;
  }

  void divide(int a) {
    state = state ~/ a;
  }

}
```

The **Provider** we just defined can be a global variable with no issue, which is quite a significant shift from the **Provider** package.

However, to achieve all this functionality, the package introduces two new types of widgets substituting for **StatelessWidget** and **StatefulWidget: ConsumerWidget** and **ConsumerStatefulWidget**.

Code 14.25:

```
class DemoPage extends ConsumerWidget {
  const DemoPage({Key? key}): super(key: key);

  @override
   Widget build(BuildContext context, WidgetRef ref) {
    // use ref to listen to a provider
    final value = ref.watch(arithmeticProvider);
    return Text('$value');
  }
}
```

As you can see, the **build()** method now supplies an additional widget reference through which we can read or watch the **Provider** created.

Here is the corresponding **ConsumerStatefulWidget**:

Code 14.26:

```
class DemoPage extends ConsumerStatefulWidget {
  const DemoPage({Key? key}): super(key: key);

  @override
  DemoPageState createState() => DemoPageState();
```

```
}

class DemoPageState extends ConsumerState<DemoPage> {
  @override
  void initState() {
    super.initState();
    ref.read(arithmeticProvider);
  }

  @override
  Widget build(BuildContext context) {
    final value = ref.watch(arithmeticProvider);
    return Text('$value');
  }
}
```

There are similar providers in **Riverpod** like the **Provider** package, however it uses them in an entirely new way. Next up, we look at the BLoC pattern and more specifically, the **flutter_bloc** package, to implement it.

bloc/flutter_bloc

flutter_bloc is an implementation of the BLoC pattern in Flutter. BLoC or Business Logic Components is a pattern that was created by Google in the early days of Flutter to separate business logic from UI. The BLoC pattern is a relatively verbose way to separate business logic from the app and represent the app data as a series of sinks and streams.

While still relevant, this was even more relevant when **AngularDart** was maintained since **AngularDart** and Flutter had different UI implementations but the business logic could be written in Dart itself.

There are two packages involved in the bloc implementation. One is the **bloc** package that provides the basic building blocks like cubits. The other is **flutter_bloc**, which provides an easy way to read and pass cubits down the tree using the **Provider** package internally.

Let's look at the bloc package itself. This does not stick to the original model of sinks and streams introduced by the Bloc pattern; it introduces a similar concept using Cubit and Bloc.

The **Cubit** is a simple class that takes an initial value of the type specified and emits states of modified values:

Code 14.27:

```
class ArithmeticCubit extends Cubit<int> {

  ArithmeticCubit() : super(0);

  void add(int a) => emit(state + a);

  void subtract(int a) => emit(state - a);

  void multiply(int a) => emit(state * a);

  void divide(int a) => emit(state ~/ a);

}
```

On the other hand, the Bloc (the class in the package, not the pattern) takes in an event and emits a new state based on the event that occurred. Here's a counter example from the documentation:

Code 14.28:

```
abstract class CounterEvent {}

class CounterIncrementPressed extends CounterEvent {}

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0) {
    on<CounterIncrementPressed>((event, emit) {
      emit(state + 1);
    });
  }
}
```

The **flutter_bloc** package also provides additional builders and selectors (similar to **Provider**) using Blocs:

Code 14.29:

```
BlocBuilder<BlocA, BlocAState>(
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
You can also select certain states to update the UI:
Code 14.30:
BlocSelector<BlocA, BlocAState, SelectedState>(
  selector: (state) {
    // return selected state based on the provided state.
  },
  builder: (context, state) {
    // return widget here based on the selected state.
  },
)
```

The package also uses **Provider** to have **BlocProviders** and **MultiBlocProviders** like the **Provider** package:

Code 14.31:

```
MultiBlocListener(
  listeners: [
    BlocListener<BlocA, BlocAState>(
      listener: (context, state) {},
    ),
    BlocListener<BlocB, BlocBState>(
      listener: (context, state) {},
    ),
    BlocListener<BlocC, BlocCState>(
      listener: (context, state) {},
    ),
  ],
  child: ChildA(),
)
```

Overall, both packages give a solid-state management solution that is easy to test

since the business logic is well separated from the UI.

Conclusion

Being well versed in theming, navigation, and state management promises helps build apps in tidier ways, with less code, in a scalable manner, and much faster. Properly following these methods allows for less code throughout the app without reducing quality. There is also a lot of community conversation around these concepts, including heavy debate over the best methods of course. Job interviews also often focus on various state management solutions since they expect familiarity of the candidate with the state management solution that the company uses so that they can get started quickly.

In the next chapter, we will learn to make a good-looking app by adding animations.

Questions

1. How do you add themes to a Flutter application?
2. What is the `Navigator`, and what responsibilities does it have?
3. Why is state management required, and what are the popular packages used?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 15
Advanced Flutter - Animations

Introduction

The core of a mobile application is always the logic and functionality that it provides to the user. However, in today's time, multiple applications can do the same things and have the same feature set. Animations help provide the user experience to set an application apart from all its competitors. Animations help add fluidity and visual flair to the app. Alongside the looking-good part, animations can add meaning to every interaction that the app provides; for example, clicking on a delete button may move an element across the screen into a red zone or a recycle bin icon, indicating quite well as to what element was and what action was carried out. A combination of these two factors allows an app to go from looking like a hobby project to a genuinely well-written professional app (even if the underlying logic is EXACTLY the same).

In this chapter, you will be introduced to the different kinds of animations in Flutter and understand the different components associated with them. Alongside this, you will learn about implicit animations that add basic animations to a Flutter app with barely any extra code.

Structure

Structure

In this chapter, we will discuss the following topics:

- What is an animation?

- About the Flutter animation framework
- Basic building blocks of animations
- Tween
- AnimationController
- Animation
- Creating a basic animation from scratch
- AnimatedBuilder
- TweenAnimationBuilder
- ImplicitAnimations
- Widgets that use implicit animations

Objectives

After studying this chapter, you should understand the basic concepts of animations in Flutter and the various ways to implement them. Best practices for animations are also explained in this chapter.

What is an animation?

Before you take the leap to creating your own animations, it helps to have a solid technical understanding of what you are trying to achieve. The following diagram has examples of the changes you often see as animations: the first being color transitions and the second being changes in size:

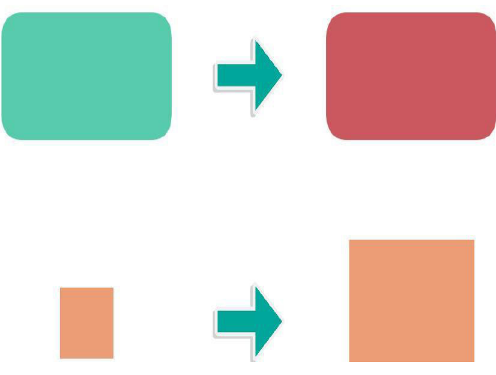


Figure 15.1: Example changes duration an animation

Every app has changes in state: a list might have items added to it, new pages may open and close, or items may change position on the screen. To provide a smooth

user experience, animations show a gradual nature in these changes rather than an abrupt one. This helps users understand their interaction with the app better. For example, a splash on the button helps them identify the click on the button. A transition on the page starting from the component clicked on helps convey what caused the transition. Overall, animations help add more meaning to the interaction between the phone and the user. The app also feels smoother and more polished with animations instead of having abrupt changes that seem to appear out of nowhere.

There are also other kinds of animations that don't necessarily have a direct underlying meaning but are simply there to look good. Having good animations helps retain a user in the app. Think of this like a mirror and elevator music being added to an elevator; they may not have a direct functional purpose vis-a-vis going up and down floors, but they help make the users of the elevator calmer and patient while they are in it.

While most people have a feel of what an animation is, it's hard to pin down exactly what the proper definition is.

About the Flutter animation framework

Flutter has several ways of building animations, some easy and fixed, some complex and more customizable. Unlike aspects such as networking (and sometimes asynchronous tasks), animating visible elements tends to be very different across mobile frameworks, and most concepts don't necessarily carry over. So, to understand the Flutter way of animation better, we're going to understand the basic components of an animation, after which we will also look at the types of animations (which may or may not need these components, but it is better to study the basics before we go into out-of-the-box solutions).

Basic building blocks of animations

In this section, we will look at what makes a Flutter animation tick…literally. After understanding the individual components, we can combine them to create a few basic animations. Let's get started with a concept that is common to most animations, whether you are using a proper video-editing / animation tool or creating an animation with Flutter: a `Tween`.

Tween

The word *Tween* is a shortened form of *in-between*. Before we understand the process or why it's named that, let's think of what an animation's states are. An animation has a start and an end. But what that means isn't properly defined; an object might start from the color red and go to blue. It might translate from the top-left corner of

the screen to the bottom-right corner. In short, any property of the object that the animation is associated with might change in the eventual transition.

Without animations, we might have just changed the object from red to blue instantly. However, the beauty of animations is that we show a gradual change. To figure out this gradual change (from red to blue or top-left to bottom-right), we need to calculate, for example: *what is the color when the animation is halfway between the red to blue transition*, or *what is the position of the object when it is halfway from top-left to bottom-right*.

While I say *halfway*, what we are more looking for is a way to calculate an equation for the change, say at time 0.0, the color is red, and at 1.0 it is blue; what is the color value at any time x in between 0 and 1?

If the foreshadowing has not been completely clear, `Tween` helps us calculate the value in between two states when the animation is running. We provide the type of animation to the `Tween`, tell it what the initial and end conditions are, and it gives us the values in between the initial and end states.

Let's look at a basic code example:

Code 15.1:

```
var tween = Tween(begin: 0.0, end: 100.0);


// prints "50"

print(tween.lerp(0.5));
```

Here, we set the initial value to 0.0 and end at 100.0. This could potentially be useful when you need to translate a widget 100 units in any direction. Since the animation runs from time 0.0 to 1.0 (this indicates the completion of the animation, NOT the number of seconds), we can get half completion at 0.5.

The question here: What is "lerp"?

Lerp stands for linear interpolation. The *interpolation* part refers to getting a value in

between the middle and end states of the **Tween**. The *linear* refers to something else entirely: it assumes that the value goes from the beginning to the end uniformly: 10 at 0.1, 20 at 0.2, 30 at 0.3, and so on. If we want to do something fancy, we can start by slowly changing the value and then increase the speed; the lerp won't do the trick.

The good news is that Flutter does allow us to do this, but we need to work with **Animation** objects (the Flutter class of animations) to do this. We will get to this in the later part of the chapter.

In the earlier example, we talked about having an animation where a color is changed from red to blue. To do this with **Tween**s, we can do one of two things:

Code 15.2:

```
var colorTween = Tween<Color?>(begin: Colors.red, end: Colors.blue);


print(colorTween.lerp(0.5).toString());
```

Alternatively, Flutter provides a more direct way to do this using **ColorTween**, which is a **Tween** made specifically for handling colors.

Code 15.3:

```
var tween = ColorTween(begin: Colors.red, end: Colors.blue);


print(tween.lerp(0.5).toString());
```

Point to remember: A Tween allows us to get intermediary values between two values: start/begin and end.

AnimationController

With **Tween**s, we could control the beginning, ending, and intermediary values of an animation, so what else do we really need? Well, an animation has a specific duration; transition taking a second is far different from it taking a hundred. Another thing is that we need something to trigger the start of an animation, maybe even reverse or stop it.

Speaking theoretically, we can probably implement this ourselves, but it's too big a hassle. If you are implementing an animation in your app, it is likely not the only animation in the app. So, setting up a mechanism to track what percentage of completion (say 45% done) an animation is at, if it's running or not, etc. is quite tedious.

An **AnimationController** is a Flutter class that helps us control our animations, i.e., start, stop, check state, and even *fling* them, referring to starting an animation with

start, stop, check state, and even *fling* them, referring to starting an animation with a set velocity and sprint simulation. Here's the irony: the **AnimationController** itself is an animation going from 0.0 to 1.0 referring to the progress of our animation.

Hence, the **AnimationController** has a property named 'value' that we can access, which gives us the overall progress of the animation:

```
var animationCompletion = animationController.value;
```


To declare an **AnimationController**, we do two things. First, we add a very specific mixin to the widget:

Code 15.4:

```
class _DemoState extends State<Demo> with SingleTickerProviderStateMixin
{
    // Widget code
}
```

Secondly, if you plan to have multiple controllers in the same class, we add the following:

Code 15.5:

```
class _DemoState extends State<Demo> with TickerProviderStateMixin {
    // Widget code
}
```

Then we create the actual **AnimationController**:

Code 15.6:

```
late AnimationController _controller;

  @override
  void initState() {
    super.initState();
    _controller = AnimationController(
      vsync: this,
      duration: Duration(seconds: 1),
    );
  }
```

One parameter is obvious: the *duration* is the duration that we want our animation to run for. The other one, not so much: what is **vsync**? Is it related to the mixin we needed to add to the widget?

In short, **AnimationController** requires a **Ticker**. The **vsync** parameter provides it a **TickerProvider** (the mixin we added to the widget state is a **TickerProvider**, which is why we can use **this** as the argument when creating a controller), which – get this – provides a **Ticker**. A **Ticker** sets the refresh rate for the animation (hence the **ticks**), but along with that, the **SingleTickerProviderStateMixin** mixin handles the animation state since it can pause the **Ticker** in certain cases – for

example, a new page is pushed on to the current page, making the current widget invisible. You can theoretically create your own **TickerProvider**, but this approach would make you manually pause and resume **Ticker**s when they are not shown.

Controlling an animation with the controller is pretty straightforward:

Code 15.7:

```
// Start an animation
_controller.forward() ;

// Reverse an animation
_controller.reverse() ;

// Reset the animation to the beginning
_controller.reset() ;

// Stop the running animation
_controller.stop() ;

// Start and repeat the animation
// You can decide if you want to reverse the animation or go immediately
to the beginning once the animation is done through the 'reverse'
_controller.repeat() ;

// Animate to a certain value of completion - for example animate to 0.5
(50% completion)
_controller.animateTo() ;
```

Now, we have two objects: the **Tween** that holds the values we want to animate to

and from, and the **AnimationController**, which sets the animation in motion. But how do we connect these two things? Enter **Animation**, the class that actually holds the animation values and connects **Tween**s and animation controllers.

Animation

All the earlier times when the word *animation* was written, it meant the concept of animating some kind of state in the app. Alongside this, there is also a **Flutter** class called **Animation** that ties together a **Tween** and **AnimationController**.

To summarize, a **Tween** holds values of the state change involved in the animation. The **AnimationController** holds the progress/state of the animation and the

ability to 'drive' the animation. To run an animation, we need to connect these two to get the current value of the animation at the current progress. For this, we use the **`Animation`** class.

The question now is, "Why do we need anything other than **`AnimationController`** and **`Tween`** at all?" As seen in *Code 15.1*, we can use a **`Tween`** to calculate the current animation value. However, as we discussed, **`lerp()`** gives values that are linearly extrapolated. Most animations do not run linearly; they run in a fancy way, for which we use a **`CurvedAnimation`**. The **`Animation`** also avoids the need to get the progress using the **`AnimationController`** value every time. Additionally, the **`Tween`**, being separate, allows the state change values to be separated from the actual animations created from the expected changes.

To create an animation, we do as follows:

Code 15.8:

```
var tween = Tween(begin: 0.0, end: 100.0);
late Animation animation;
late AnimationController controller;

@override
void initState() {
  super.initState();
  controller = AnimationController(duration: Duration(seconds: 1),
vsync: this);
  animation = tween.animate(controller);
}
```

`Tween`s have an **`animate()`** method that links the **`Tween`** value to the controller and creates an **`Animation`** object. Now, when we start the animation using the controller, we can get the current value of the animation through the following:

Code 15.9:

```
var currentValue = animation.value;
```

In its current state, the animation is still a linear animation. To create a **`CurvedAnimation`**, we can do the following:

Code 15.10:

```
var tween = Tween(begin: 0.0, end: 100.0);
late Animation animation;
late AnimationController controller;
```

```
@override
void initState() {
  super.initState();
  controller = AnimationController(duration: Duration(seconds: 1),
vsync: this);
  animation = tween.animate(CurvedAnimation(parent: controller, curve:
Curves.easeInOut));

  var currentValue = animation.value;
}
```

This allows us to create more natural feeling animations; linear animations look a bit awkward when it comes to real-world apps.

Creating a basic animation from scratch

Let's assume that we want to create a basic animation from scratch. Since the components till now have been detailed separately, it is a bit tricky to understand how to use them in unison in an actual app. For this example, we will create a square of size 10 at the center of the screen and expand it to be size 100 in a bouncy kind of fashion. Follow these steps to do so:

1. **Creating the basic layout**

   We need to create a screen with a square at the center of the screen with size 10. The basic code for this would look as follows:

   Code 15.11:

```
class DemoPage extends StatefulWidget {
 const DemoPage({Key? key}) : super(key: key);

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

class _DemoPageState extends State<DemoPage> {
 @override
 Widget build(BuildContext context) {
```

```
    return Scaffold(
      appBar: AppBar(
        title: const Text('Animation Demo'),
      ),
      body: Center(
        child: Container(
          width: 10.0,
          height: 10.0,
          color: Colors.green,
        ),
      ),
    );
  }
}
```

This gives us the following output:

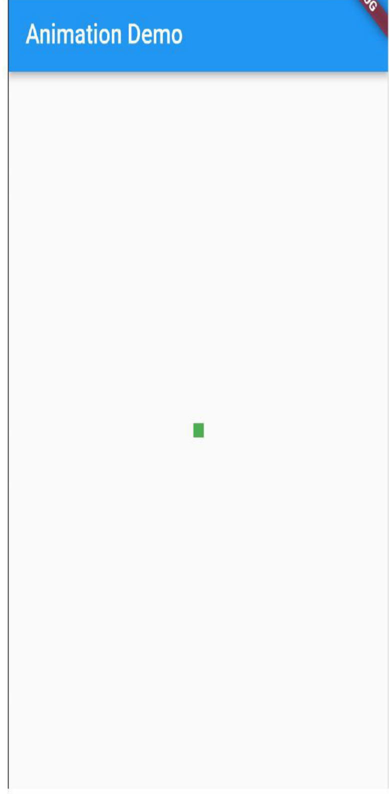


Figure 15.2: *Initial UI before animation*

The **`Center`** widget centers the child widget, and the **`Container`** widget is used to draw the actual square and give it a color.

2. **Creating the Tween**

   In this example, we are animating from a value of 10 to a value of 100. We can use the base **`Tween`** class for this and use double values:

   Code 15.12:

```
var sizeTween = Tween<double>(begin: 10.0, end: 100.0);
```

3. **Creating the AnimationController**

   To create an **`AnimationController`**, we need to do take steps: one is to add the **`SingleTickerProviderStateMixin`** to our widget, and the second is to actually declare the **`AnimationController`** itself and initialize it.

   Code 15.13:

```
class _DemoPageState extends State<DemoPage> with
SingleTickerProviderStateMixin {
  var sizeTween = Tween<double>(begin: 10.0, end: 100.0);
  late AnimationController animationController;


  @override
  void initState() {
    super.initState();
    animationController = AnimationController(
      duration: const Duration(seconds: 1),
      vsync: this,
    );
  }


  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Animation Demo'),
      ),
```

```
      body: Center(
        child: Container(
          width: 10.0,
          height: 10.0,
          color: Colors.green,
        ),
      ),
    );
  }
}
```

We'll also start it immediately after creating it:

Code 15.14:

```
@override
void initState() {
  super.initState();
  animationController = AnimationController(
    duration: const Duration(seconds: 1),
    vsync: this,
  );

  animationController.forward();
}
```

4. **Creating the Animation**

   We can now link our **`Tween`** and **`AnimationController`** together to create the **`Animation`**.

Code 15.15:

```
var sizeTween = Tween<double>(begin: 10.0, end: 100.0);
late AnimationController animationController;
late Animation sizeAnimation;

@override
```

```
void initState() {
  super.initState();
  animationController = AnimationController(
    duration: const Duration(seconds: 1),
    vsync: this,
  );

  sizeAnimation = sizeTween.animate(animationController);

  animationController.forward();
}
```

5. **Using the animation value in the UI**

   While we now have a potential animation that will run from 10 to 100, we need to tell the UI to use the animation value instead of the default value **10** that we have provided in the starter code.

   Code 15.16:

```
var sizeTween = Tween<double>(begin: 10.0, end: 100.0);
late AnimationController animationController;
late Animation sizeAnimation;

@override
void initState() {
  super.initState();
  animationController = AnimationController(
    duration: const Duration(seconds: 1),
    vsync: this,
  );

  sizeAnimation = sizeTween.animate(animationController);

  animationController.forward();
```

```
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Animation Demo'),
    ),
    body: Center(
      child: Container(
        width: sizeAnimation.value,
        height: sizeAnimation.value,
        color: Colors.green,
      ),
    ),
  );
}
```

However, if we run this, it will not actually update the UI. This is because even though we are using the animation value, we are never actually telling the UI to update, by using **`setState()`** for example.

Hence, for the final step, we will add a listener to the animation, which updates the UI when the animation itself updates:

Code 15.17:

```
@override
void initState() {
  super.initState();
  animationController = AnimationController(
    duration: const Duration(seconds: 1),
    vsync: this,
  );
```

```
    sizeAnimation = sizeTween.animate(animationController);

    animationController.addListener(() {
      setState(() {});
    });

    animationController.forward();
  }
```

And hence, we have our working animation:

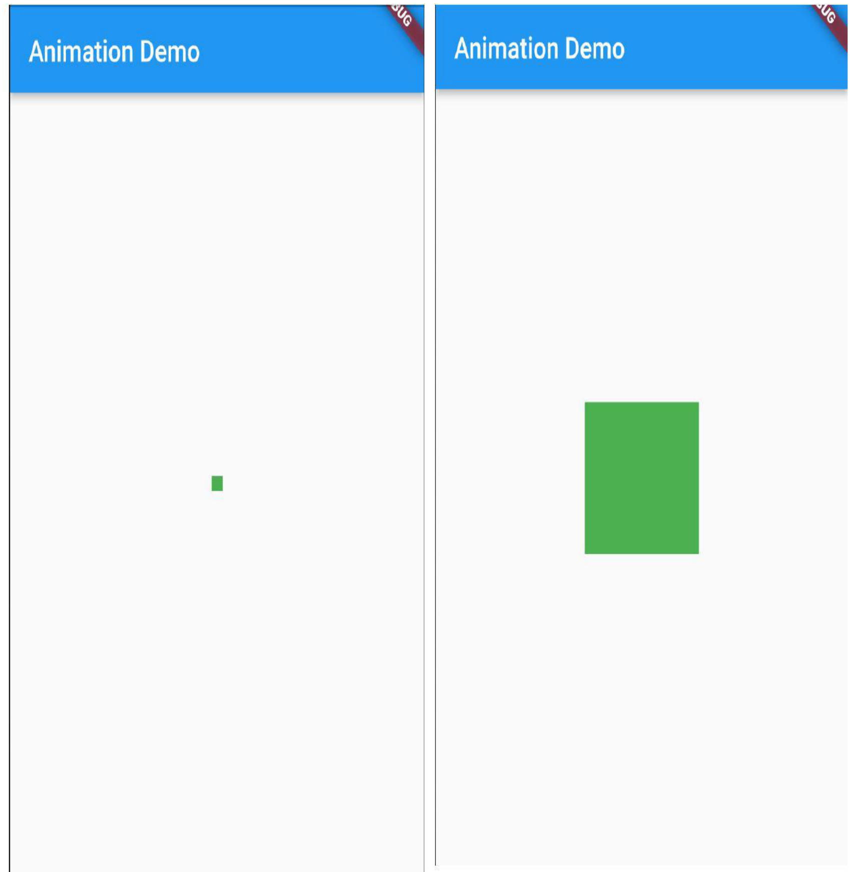


Figure 15.3: Before and after animation

While using **setState()** works, this is not actually the best way to do it. Enter **AnimatedBuilder**.

AnimatedBuilder

The crucial flaw of **`setState()`** is that it rebuilds the entire screen. Flutter, to its credit, does quite a lot of optimization to make sure even a complete rebuild does not rebuild all existing UI elements. So, even **`setState()`** does not truly update the entire screen; however, it does run the entire **`build()`** function again.

What we want specifically is only the part we are concerned about - the square at the center of the screen - to be actually rebuilt; we can leave the other widgets like the external Scaffold untouched. **`AnimatedBuilder`**, like most of Flutter's builder widgets, builds on an update of something, an animation in this case. This only rebuilds the builder function of the **`AnimatedBuilder`** and NOT the entire widget's build function.

We can add an **`AnimatedBuilder`** to the code, like this:

Code 15.18:

```
class _DemoPageState extends State<DemoPage>
    with SingleTickerProviderStateMixin {
  var sizeTween = Tween<double>(begin: 10.0, end: 100.0);
  late AnimationController animationController;
  late Animation sizeAnimation;

  @override
  void initState() {
    super.initState();
    animationController = AnimationController(
      duration: const Duration(seconds: 1),
      vsync: this,
    );

    sizeAnimation = sizeTween.animate(animationController);

    animationController.forward();
  }

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

```
    return Scaffold(
      appBar: AppBar(
        title: const Text('Animation Demo'),
      ),
      body: Center(
        child: AnimatedBuilder(
          animation: sizeAnimation,
          builder: (context, wid) {
            return Container(
              width: sizeAnimation.value,
              height: sizeAnimation.value,
              color: Colors.green,
            );
          }
        ),
      ),
    );
  }
}
```

The benefit of doing this is also that we can now remove the listener we added to update the UI by calling **`setState()`**, as the builder is now doing the update.

TweenAnimationBuilder

The sections till the previous one define what are called 'explicit' animations, where you create all the components required for the animations and implement the required functionality. However, a lot of animations created have similar characteristics: they run once between two values when the widget is shown. **`TweenAnimationBuilder`** helps with these kinds of animations and reduces a lot of effort needed to create a normal animation.

Additionally, anyone who is newly introduced to the Flutter animation framework does not necessarily need the entire set of functionalities that it provides, and it can often be a bit confusing to start out with, especially if you are hearing about **`Tween`**s for the first time.

To start out, let's use the same example that we used earlier to understand the difference between **TweenAnimationBuilder** and normal animations.

This is how we would add an animation for a square going from size 10.0 to size 100.0:

Code 15.19:

```
class DemoPage extends StatefulWidget {
  const DemoPage({Key? key}) : super(key: key);

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

class _DemoPageState extends State<DemoPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Animation Demo'),
      ),
      body: Center(
        child: TweenAnimationBuilder<double>(
          duration: const Duration(seconds: 1),
          tween: Tween<double>(begin: 10.0, end: 100.0),
          builder: (context, value, wid) {
            return Container(
              width: value,
              height: value,
              color: Colors.green,
            );
          }
        ),
      ),
    );
```

```
  }
}
```

This is a quite clean example; here are a few things to note:

- No **AnimationController**
- No **Animation**
- No **Mixins**
- Handles rebuilds on its own

The **TweenAnimationBuilder** handles animation state internally, and we only have to provide a **Tween** and duration to it.

We can also provide a curve and get notified when the animation ends through the widget:

Code 15.20:

```
TweenAnimationBuilder<double>(
  duration: const Duration(seconds: 1),
  tween: Tween<double>(begin: 10.0, end: 100.0),
  builder: (context, value, wid) {
    return Container(
      width: value,
      height: value,
      color: Colors.green,
    );
  },
  curve: Curves.bounceInOut,
  onEnd: () {
    // Do something when animation ends
  },
),
```

The question is: if this is so easy, why don't we do all animations like this by default?

While **TweenAnimationBuilder** can be an extremely useful tool at times, it is limited in some sense: control over animations is quite limited. While we can technically rerun the animation by providing a separate **Tween** to the widget, this is not as

straightforward a mechanism as using an **AnimationController**. Also, it is mainly

used for dealing with a singular animation, so coordinating multiple animations or chaining is more than a bit tricky. Hence, explicit animations still should be the go-to choice for fairly complex animations.

However, the purpose of **TweenAnimationBuilder** is clear: to make a certain kind of animations easy when it deals with the properties of a single widget. Going into implicit animations, this becomes even easier since there are inbuilt widgets that allow easy animations.

Implicit animations

A lot of widgets in Flutter have some kind of visual properties, for example, height, width, color, etc. In the life of an animation, several of these properties may change: a button may go from green to red indicating something missing, a box containing some text may expand in size, and so on. When these kinds of changes are represented using direct instantaneous changes, it isn't a good look. However, if these changes are carried out through animations, the color of the button slowly turning red or the box gradually expanding to fill the space, it feels visually much better to the user.

However, with the explicit animations that we have seen, it may feel a bit cumbersome to implement all of these for the change of a single property. **TweenAnimationBuilder** may fit the purpose in some cases but definitely not all of them. In this case, it would be really helpful to have widgets that animate on change of their properties, for example, a **Container** that expands when its width and height are increased in a gradual way. Thankfully, this is exactly what Flutter provides. Since we don't explicitly define these animations, these come under a category called **implicit animations**.

We will take one example in this section and then cover a few other implicitly animated widgets in the next section.

Let's take **AnimatedContainer** as an example:

Code 15.21:

```
class AnimatedDemo extends StatefulWidget {
  const AnimatedDemo({Key? key}) : super(key: key);

  @override
  _AnimatedDemoState createState() => _AnimatedDemoState();
}
```

```
class _AnimatedDemoState extends State<AnimatedDemo> {
```



```
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: AnimatedContainer(
          duration: const Duration(seconds: 1),
          color: Colors.green,
          width: 10.0,
          height: 10.0,
        ),
      ),
    );
  }
}
```

At first glance, an **AnimatedContainer** isn't very different from a normal one, barring the duration parameter, which tells the widget how long the animation to the new value lasts.

When we now change the color, width, height, etc., the **AnimatedContainer** will now animate to the new value - go smoothly from width 10.0 to width 100.0 for example - without any kind of the aforementioned animation components.

Obviously, at the moment, we've defined literal values, so let's make this a bit more real:

Code 15.22:

```
class AnimatedDemo extends StatefulWidget {
  const AnimatedDemo({Key? key}) : super(key: key);

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

class _AnimatedDemoState extends State<AnimatedDemo> {

  var width = 10.0;
```

```
  var width = 10.0;
  var height = 10.0;
```



```
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: AnimatedContainer(
          duration: const Duration(seconds: 1),
          color: Colors.green,
          width: width,
          height: height,
        ),
      ),
    );
  }
}
```

Now when we change the width and height variables, the **Container** will smoothly move to that value. This helps create the most common animations with a **Container** quite easily, without any explicit animations being needed.

Additionally, we can add any required curve to the same animation:

Code 15.23:

```
AnimatedContainer(
  duration: const Duration(seconds: 1),
  color: Colors.green,
  width: width,
  height: height,
  curve: Curves.easeInOut,
),
```

A few widgets that use implicit animations

These are a few examples of other implicitly animated widgets. This is definitely not an exhaustive list; you should definitely go and check out the complete list. This is just here to give you an idea of the other capabilities of implicitly animated widgets.

AnimatedOpacity

The **Opacity** widget allows us to set an opacity for the given child - 0 making the child invisible and 1 making it completely visible. **AnimatedOpacity** does the same - and also allows animating to the new opacity when changed. This is extremely helpful for creating fade-in or fade-out effects since it gradually changes opacity.

The code works almost exactly like the normal **Opacity** widget:

Code 15.24:

```
class AnimatedDemo extends StatefulWidget {
  const AnimatedDemo({Key? key}) : super(key: key);

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

class _AnimatedDemoState extends State<AnimatedDemo> {

  var opacity = 1.0;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: AnimatedOpacity(
          opacity: opacity,
          duration: const Duration(seconds: 1),
          child: Container(
            color: Colors.green,
            width: 100.0,
```

```
            height: 100.0,
          ),
        ),
      ),
    );
```



```
  }

  void fadeOut() {
    opacity = 0.0;
  }
}
```

We can write a function that simply changes the opacity given and automatically creates a fade-out effect.

AnimatedPositioned

This is one of the nicer widgets when it comes to creating real motion in the app using animations. A **`Positioned`** widget gives a position to the child widget in a **`Stack`**. This is mostly using its position relative to one of the sides - *my widget should be 15 units away from the top and 25 from the left-hand side of the screen.* A**`nimatedPositioned`**, as you may be able to guess, moves the widget across the screen to the new position that we give it. This can help create neat effects and animations, and it doesn't even take a lot of effort.

Consider the following example:

Code 15.25:

```
class AnimatedDemo extends StatefulWidget {
  const AnimatedDemo({Key? key}) : super(key: key);

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

class _AnimatedDemoState extends State<AnimatedDemo> {
  var top = 15.0;
```

```
  var left = 25.0;

  @override
  Widget build(BuildContext context) {
   return Scaffold(
      body: Stack(
        children: [
```



```
          AnimatedPositioned(
            duration: const Duration(seconds: 1),
            top: top,
            left: left,
            child: Container(
              color: Colors.green,
              width: 100.0,
              height: 100.0,
            ),
          )
        ],
      ),
    );
  }

  void moveContainer() {
    top = 150.0;
    left = 250.0;
  }
}
```

To move our evergreen container across the screen, we simply change the top and left values.

AnimatedCrossFade

We often have some kind of progress indicator in our apps for a while when loading; and after loading, we switch the loading indicator with the list or whatever it is we loaded. A nicer effect to create is a crossfade, a nice fading in and out between two widgets depending on a certain condition. This creates continuity between the two elements and appears nicer. Another example of where to use this is when there is a button that switches to a loading indicator when a process is ongoing.

button that switches to a loading indicator when a process is ongoing.

An example of an **AnimatedCrossFade** is as follows:

Code 15.26:

```
class AnimatedDemo extends StatefulWidget {
  const AnimatedDemo({Key? key}) : super(key: key);

  @override
```



```
  _AnimatedDemoState createState() => _AnimatedDemoState();
}

class _AnimatedDemoState extends State<AnimatedDemo> {
  bool isFirstState = true;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: AnimatedCrossFade(
        duration: const Duration(seconds: 1),
        crossFadeState: isFirstState ? CrossFadeState.showFirst :
CrossFadeState.showSecond,
        firstChild: Container(
          color: Colors.green,
          width: 10.0,
          height: 10.0,
        ),
        secondChild: Container(
          color: Colors.red,
          width: 100.0,
          height: 100.0,
        ),
      )
    );
  }
```

```
  void changeState() {
    isFirstState = !isFirstState;
  }
}
```

Here, in this admittedly lazy example, we change between a red and green container depending on the state provided to the widget. The **`crossFadeState`** property helps define what state the widget is in, and the **`firstChild`** and **`secondChild`** parameters define the two children that fade in and out.

These were a few examples of implicitly animated widgets in Flutter that make animation much easier than other methods but are a bit restricted in their application. There are many more widgets like these to explore, and the possibilities are endless.

Conclusion

With an abundance of apps flooding the app stores every day, animations are what separate a pretty good app from a great one. While they obviously can't make up for any missing underlying functionality, they can be the tiebreaker between apps. In this chapter, we learned how to add animations to our apps and give them that visual je ne sais quoi they need.

In the next chapter, we will go into the depths of the Flutter framework and understand the fundamentals that make up widgets themselves.

Questions

1. What are the components of an animation in Flutter?
2. How are implicit animations different from explicit animations?
3. What is the purpose of the **`TweenAnimationBuilder`**?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 16

Advanced Flutter - Under the Hood

While a simple internet search would reveal a plethora of Flutter knowledge on the basic levels like using Widgets in Flutter, building complex UI, state management solutions, HTTP working and so much more, there seems to be a lack of resources explaining how all of this translates into the actual visual representation found on the screen. This is partially because the framework is designed to hide clever ways of optimization that Flutter uses away from developers so that it does not have to be manually implemented in every app. Knowing this process gives greater clarity into the Flutter build and painting process, which makes it easier for

developers to understand the kind of optimizations needed in their app.

This chapter explores in depth not only Widgets in Flutter but also their underlying components, mainly **`Elements`** and **`RenderObjects`**, which do most of the heavy lifting in the actual rendering process.

Here is the structure of the chapter ahead:

Structure

In this chapter, we will discuss the following topics:

- Understanding Flutter as a UI toolkit
- Concerns about the Flutter approach

- The Trees of Flutter
- Understanding RenderObjects
- Types of RenderObjects
- Understanding Elements
- RenderObjectWidgets
- Breaking down Widgets
- Breaking down Opacity
- Breaking down Text

Objectives

After studying this chapter, you should be able to understand the underlying processes used when user-defined Widgets are to be converted to the actual UI defined on the screen. You should also gain a better understanding of the internals of Widgets themselves and their contents.

Understanding Flutter as a UI toolkit

It should be pretty clear to anyone using Flutter that Flutter does not exist as an independent app-building framework. Whenever Flutter apps are built, the end results still create APK/AAB or IPA files for Android and iOS, respectively. These also use the underlying tooling provided by those frameworks and hence, Gradle builds do not vanish into thin air when Flutter apps are built. They do not, however,

have to be repeated since the new Dart code can just be injected into the app after the first build is completed.

The purpose of this section is to not only set expectations when using Flutter but also to make clear what domain Flutter excels in. Flutter has always been referred to as a *UI toolkit*, one that simply does the job of painting pixels on a screen. However, what does that mean in terms of building apps?

If you have ever upgraded to a new Android or iOS version and found out that the same apps look different since UI components originating from the OS have been upgraded, you partially see the original issue facing native development; they depend on the OS version, UI components and interactions from the OS itself. Flutter was built to solve this issue: to eliminate the dependence of the mobile app on the OS version. But the question still remains, 'how did Flutter solve this?'.

The surprising answer here is that Flutter essentially paints the app onto a canvas. Instead of importing a button from the OS and letting the OS define look and feel,

it defines the instructions to paint the button onto a canvas. It does this with all Widgets in the Flutter library. Going underneath - probably a few layers - the canvas paint instructions are quite clear. This is also seen in the breakdown of a few Widgets later in the chapter. This is where things get clearer: the entire Flutter app itself is painted onto a canvas that the device provides. The uniqueness of Flutter comes from the fact that this nearly eliminates the dependence on the OS for UI purposes while obviously retaining it for other purposes like accessing cameras, sensors, etc. Let's look at the caveats of this approach next.

Concerns about the Flutter approach

There is no perfect approach to building any tool. With the solutions that Flutter adopts come some problems that do not necessarily occur with native app development. One of them is that extra code needs to be added into the release builds of the apps since the app logic isn't the only thing needed now; the UI painting logic and Flutter engine itself is also included. While this doesn't have a massive impact in today's world, apps meant to be tiny in size cannot be made from this approach.

Additionally, there needs to be a considerable level of optimization since painting every frame isn't light in any respect. This is handled by Flutter using multiple trees with different responsibilities and rebuild cost, as we will see in the next section. However, the benefits generally outweigh the drawbacks of this system. It is important to be mindful of the needs of the app you build, but this approach works quite well for most of them.

The Trees of Flutter

Even though Flutter is described as a place where everything is a Widget, it is more a description of the user-facing APIs than what goes on inside to actually convert code to UI. Flutter contains a fairly complex (but not that difficult to understand) system that does this.

Here's the thing about Widgets: they're more about configuration than about painting the objects we describe. In short, they're like an order we place in a restaurant: we describe what we want and how we want it. However, the actual dish is made elsewhere and with completely different stuff. This is why Widgets are so inexpensive to build: they really don't have that much value; they just contain some data describing your preferences.

We've talked about the *Widget tree* in *Chapter 10, Diving Into Widgets*, and *Chapter 11, Basic Widgets and Layouts*. The 'Widget Tree' term is referring to how Widgets are structured: we start from one parent and then the children of that parent can have their own children and so on. Each Widget describes some configuration of its own. However, there are two more trees involved in creating the end UI: the **`Element`** tree

and the **`RenderObject`** tree. These are generally not things that you directly interact with. However, every single Widget ends with all leaves becoming some type of **`RenderObject`**.

The key thing to understand here is that simply painting something on a screen is not our primary objective. We also need to make it easy and performant to rebuild the UI whenever needed. If we rebuilt and repainted the entire UI with every frame, it would be horrendously slow. Perhaps one day we can look back and not care about these things, like how we now admire developers who worked with kilobytes of RAM to our gigabytes, but that's certainly not today. Performance matters.

Understanding RenderObjects

As you can likely tell from the name, **`RenderObject`** is the actual object responsible for rendering a Widget on the screen. Both layout and painting need to be managed while creating one, and this is quite tricky to do yourself, so a somewhat high level of abstraction is required.

Unlike Widgets, which are easy to destroy and recreate, a **`RenderObject`** is fairly expensive to deal with; this is why we try to do this as few times as possible. The focus in this area is to reuse or modify the existing **`RenderObjects`** where possible to take the least performance hit when building a new layout. This is the fundamental reason why Flutter has three trees: they make sure **`RenderObjects`** need to be destroyed and recreated as few times as possible (obviously, Widgets are far easier to write, but the base reason is still performance).

Theoretically, you can write a whole layout with `RenderObjects` directly, but this would mean developers have to do performance optimizations themselves, and we all know that's a slippery slope.

Types Of RenderObjects

When implementing your own `RenderObject`, there are multiple classes you can extend to build the Widget you desire. While Flutter has a direct `RenderObject` class, it is not implemented unless there is a very specific need (for example, not wanting to use the Cartesian coordinate system), and you need to implement a new protocol.

Most times, we deal with three classes that extend `RenderObject` themselves: `RenderBox`, `RenderProxyBox`, and `RenderShiftedBox`.

> **Note: When dealing with Slivers in lists, use `RenderSliver` instead.**

- **`RenderBox`:** This is a subclass of `RenderObject` that allows layout using a 2D cartesian coordinate system. This allows full control of sizing within the

constraints provided, and the more complex `RenderObjects` need to use `RenderBox`.

- **`RenderProxyBox`:** It is used when the child of the object has a similar size but a few properties are to be changed. An example is the `Opacity` Widget, which needs to change the opacity of the Widget, but the size stays the same.
- **`RenderShiftedBox`:** This is similar to `RenderProxyBox` but allows the child to have translation in the required direction, not assuming it exists at (0,0). This is useful for some applications, such as a Widget like `Padding` for example.

Understanding Elements

While it is fair to say that everyone who uses Flutter uses Widgets and people who have special use cases or need great performance use `RenderObjects`, elements are not usually needed by most developers. So, what is an `Element` exactly? In short, it is a wrapper around the Widget supplied by the user, which lives in its own *Element tree*. This wrapper contains the instantiation of the Widget itself and a way to update it.

When we create a `StatefulWidget` or `StatelessWidget`, we indirectly use the corresponding `StatelessElement` and `StatefulElement`.

Let's look at the source code of a `StatelessWidget`:

Code 16.1:

```
abstract class StatelessWidget extends Widget {

  const StatelessWidget({ Key? key }) : super(key: key);

  @override
  StatelessElement createElement() => StatelessElement(this);


  @protected
  Widget build(BuildContext context);

}
```

We see that apart from the normal constructor and **`build()`** method, the only other method is **`createElement()`**, which creates the **`StatelessElement`**. This **`StatelessElement`** is mutable and holds the instantiation of the Widget provided. Since it is mutable, the Widget can be replaced. This is more apparent looking at the source of the **`StatelessElement`** itself:

Code 16.2:

```
class StatelessElement extends ComponentElement {

/// Creates an element that uses the given widget as its configuration.
  StatelessElement(StatelessWidget widget) : super(widget);

  @override
  StatelessWidget get widget => super.widget as StatelessWidget;

  @override
  Widget build() => widget.build(this);

  @override
  void update(StatelessWidget newWidget) {
    super.update(newWidget);
    assert(widget == newWidget);
    _dirty = true;
    rebuild();
  }

}
```

RenderObjectWidgets

Converting a normal Widget built using compositing other Widgets to one that directly creates a **`RenderObject`** can massively reduce the performance overhead caused by the Widget - especially if it is used a lot. To do this, there are two things that needed to be known. One is creating the **`RenderObject`** itself, the second is using a **`RenderObjectWidget`** to add the **`RenderObject`** to the Widget tree.

The **`RenderObjectWidget`** allows instantiating a **`RenderObject`** and adding it to the Widget tree. While there is a base **`RenderObjectWidget`** class, we use one of its three subclasses to create our Widget . These three types are classified according to the number of children the Widget has.

The types are as follows:

- **`LeafRenderObjectWidget`**: A Widget that has no children; an example is the **`Image`** Widget, which takes in image location/URL and properties but no children

- **`SingleChildRenderObjectWidget`**: A Widget that has a single child; an example is the **`SizedBox`** Widget, which takes in a single child
- **`MultiChildRenderObjectWidget`**: A Widget that takes in multiple children; an example is the **`Column`** Widget

Breaking down Widgets

There are many types of Flutter Widgets. Some may consist of other Widgets, but the ones which do not almost always end in **`RenderObjects`**. Most user-made Widgets create a higher-level Widget with a mishmash of pre-existing Widgets. However, breaking down the pre-existing Widgets show a pattern of how Widgets are built from fundamental components and also how they translate Widget information - which is just the configuration for the **`RenderObject`** - into the paint instructions for the underlying canvas.

The sections ahead break down two well-known Widgets into their basic components. Let's start with the **`Opacity`** Widget.

Breaking down Opacity

The **`Opacity`** Widget changes the opacity (visibility) of the child Widget provided to it. This is the perfect Widget to explore initially since there is not an immense amount of configuration carried down the tree. It also mimics the child Widget size, so it is easy to use the **`RenderProxyBox`** class instead of the more complex **`RenderBox`**.

First, here's a basic example of the use of the **`Opacity`** Widget itself:

Code 16.3:

```
Opacity(
  opacity: 0.5,
  child: Container(
    color: Colors.blue,
    width: 100,
    height: 100,
  ),
)
```

The child Widget in this case is a **`Container`** of size 100x100. The opacity of this Widget is reduced by half - 1.0 is fully visible, while 0.0 is invisible - but still added to layout.

Looking inside the Widget, we find the following:

(Simplified code)

Code 16.4:

```
class Opacity extends SingleChildRenderObjectWidget {

 const Opacity({
   Key? key,
   required this.opacity,
   this.alwaysIncludeSemantics = false,
   Widget? child,
  }) : assert(opacity != null && opacity >= 0.0 && opacity <= 1.0),
      assert(alwaysIncludeSemantics != null),
      super(key: key, child: child);

  final double opacity;

  final bool alwaysIncludeSemantics;

  @override
  RenderOpacity createRenderObject(BuildContext context) {
    return RenderOpacity(
      opacity: opacity,
      alwaysIncludeSemantics: alwaysIncludeSemantics,
    );
  }

  @override
  void updateRenderObject(BuildContext context, RenderOpacity
renderObject) {
    renderObject
      ..opacity = opacity
      ..alwaysIncludeSemantics = alwaysIncludeSemantics;
  }

  // …
}
```

First, the **Opacity** Widget is a **SingleChildRenderObjectWidget**; this makes sense since it accepts a single child Widget and modifies its opacity. Second, it creates a **RenderObject** named **RenderOpacity** that does the actual job of modifying the opacity.

It is clear to see here that the Widget itself accepts configuration, and the main task is left up to **RenderObject**. Since **RenderObject** is mutable, there are both create and update functions for it.

Next, let's look inside the **RenderOpacity** class:

(Simplified code)

Code 16.5:

```
class RenderOpacity extends RenderProxyBox {

  RenderOpacity({
    double opacity = 1.0,
    bool alwaysIncludeSemantics = false,
    RenderBox? child,
  }) : super(child);

  int _alpha;

  double get opacity => _opacity;
  double _opacity;
  set opacity(double value) {
    if (_opacity == value)
      return;
    final bool didNeedCompositing = alwaysNeedsCompositing;
    final bool wasVisible = _alpha != 0;
    _opacity = value;
    _alpha = ui.Color.getAlphaFromOpacity(_opacity);

    markNeedsPaint();
  }

  @override
  void paint(PaintingContext context, Offset offset) {
```

```
    if (child != null) {
      if (_alpha == 0) {
        // No need to keep the layer. We'll create a new one if necessary.
        layer = null;
        return;
      }
      assert(needsCompositing);
      layer = context.pushOpacity(offset, _alpha, super.paint, oldLayer:
layer as OpacityLayer?);
    }
  }


}
```

The **RenderOpacity** Widget extends the **RenderProxyBox** since it does not need to resize the Widget and wants a layout exactly like the child Widget would.

The **paint()** method paints the Widget itself. As you can notice, if no child is provided, it simply does not paint anything. If there is a child, the opacity effect is created by pushing a new opacity layer onto the canvas. If opacity is 0, there is again no need to create this layer, which is shown in the **paint()** method of Code 16.5. The **markNeedsPaint()** function is the equivalent of **setState()**, but for the paint layer. The **setState()** call underneath is a call to **markNeedsBuild()** as well.

Since the **Opacity** widget does not draw anything on the canvas itself, it does not have any explicit canvas instructions that are seen across many Widgets.

Now, let's look at something that deals with these canvas instructions: **Text**.

Breaking down Text

The **Text** Widget is, in a way, both simpler and more complex than most people realize. The simplicity is in the actual painting of the text on the screen, the complexity comes from actually laying out and styling it. Basically, the **Text** Widget is an amazing abstraction for rendering and laying out text in a reasonable manner. It would be surprisingly easy to paint the actual text on a canvas but supremely difficult to add all the parameters by yourself.

The Widget itself underneath uses another Widget called **RichText**, as you can see from the simplified **build()** method of the **Text** Widget here:

Code 16.6:

```
@override
Widget build(BuildContext context) {
  final DefaultTextStyle defaultTextStyle = DefaultTextStyle.of(context);
  TextStyle? effectiveTextStyle = style;
  if (style == null || style!.inherit)
    effectiveTextStyle = defaultTextStyle.style.merge(style);
  if (MediaQuery.boldTextOverride(context))
    effectiveTextStyle = effectiveTextStyle!.merge(const
TextStyle(fontWeight: FontWeight.bold));

  Widget result = RichText(
    textAlign: textAlign ?? defaultTextStyle.textAlign ?? TextAlign.
start,
    textDirection: textDirection,
    locale: locale,
    softWrap: softWrap ?? defaultTextStyle.softWrap,
    overflow: overflow ?? effectiveTextStyle?.overflow ?? defaultTextStyle.
overflow,
    textScaleFactor: textScaleFactor ?? MediaQuery.
textScaleFactorOf(context),
    maxLines: maxLines ?? defaultTextStyle.maxLines,
    strutStyle: strutStyle,
    textWidthBasis: textWidthBasis ?? defaultTextStyle.textWidthBasis,
    textHeightBehavior: textHeightBehavior ?? defaultTextStyle.
textHeightBehavior ?? DefaultTextHeightBehavior.of(context),
    text: TextSpan(
      style: effectiveTextStyle,
      text: data,
      children: textSpan != null ? <InlineSpan>[textSpan!] : null,
    ),
  );

  return result;
}
```

The main thing to note from the preceding code snippet is the amount of configuration passed down from the **Text** Widget to the **RichText** Widget. The reason is that the **RichText** Widget is freely available to use, but the **Text** Widget abstracts several details, so the responsibility is not passed on to the developer. Keep this in mind since even more abstraction is coming up.

Underneath, this is what the **RichText** Widget looks like:

(Simplified code)

Code 16.7:

```
class RichText extends MultiChildRenderObjectWidget {

  RichText({
    Key? key,
    required this.text,
    // many other properties
  }) : super(key: key, children: _extractChildren(text));

  static List<Widget> _extractChildren(InlineSpan span) {
    int index = 0;
    final List<Widget> result = <Widget>[];
    span.visitChildren((InlineSpan span) {
      if (span is WidgetSpan) {
        result.add(Semantics(
          tagForChildren: PlaceholderSpanIndexSemanticsTag(index++),
          child: span.child,
        ));
      }
      return true;
    });
    return result;
  }

  //…

  @override
  RenderParagraph createRenderObject(BuildContext context) {
```

```
    assert(textDirection != null || debugCheckHasDirectionality(context));
    return RenderParagraph(text,
      textAlign: textAlign,
      textDirection: textDirection ?? Directionality.of(context),
      softWrap: softWrap,
      overflow: overflow,
      textScaleFactor: textScaleFactor,
      maxLines: maxLines,
      strutStyle: strutStyle,
      textWidthBasis: textWidthBasis,
      textHeightBehavior: textHeightBehavior,
      locale: locale ?? Localizations.maybeLocaleOf(context),
    );
  }

  @override
  void updateRenderObject(BuildContext context, RenderParagraph
renderObject) {
    assert(textDirection != null ||
debugCheckHasDirectionality(context));
    renderObject
      ..text = text
      ..textAlign = textAlign
      ..textDirection = textDirection ?? Directionality.of(context)
      ..softWrap = softWrap
      ..overflow = overflow
      ..textScaleFactor = textScaleFactor
      ..maxLines = maxLines
      ..strutStyle = strutStyle
      ..textWidthBasis = textWidthBasis
      ..textHeightBehavior = textHeightBehavior
      ..locale = locale ?? Localizations.maybeLocaleOf(context);
  }
}
```

First off, the obvious question is whether the **Text** Widget should be leaf **RenderObject** since there is no child or children parameter; why isn't it? The answer here is that the Widget converts the given input into Widget spans, and these are used as effective children. This can be seen in the **_extractWidgets()** method after the constructor.

Next, the **RenderObject** responsible for the **Text** Widget is finally seen: the **RenderParagraph**. All the properties taken as parameters for now are passed down to the **RenderParagraph** where the actual text painting happens. Next, let's look at this **RenderObject**.

While the code itself is quite complex and too large to show here, the following are some of the important parts of the **RenderParagraph**:

Code 16.8:

```
class RenderParagraph extends RenderBox
   with ContainerRenderObjectMixin<RenderBox, TextParentData>,
            RenderBoxContainerDefaultsMixin<RenderBox, TextParentData>,
                 RelayoutWhenSystemFontsChangeMixin {
  // …
}
```

Since the **Text** needs to be laid out, the **RenderObject** itself extends **RenderBox**, not **RenderProxyBox**, which can size itself to the child Widget. There are also additional mixins that perform various functions like redoing the layout when the system fonts change. The others are for handling multiple children while working with a **RenderBox**.

RenderParagraph itself contains complex functions for sizing and laying out the text, but the final painting is done via a **TextPainter** that the class creates:

Code 16.9:

```
_textPainter = TextPainter(
  text: text,
  textAlign: textAlign,
  textDirection: textDirection,
  textScaleFactor: textScaleFactor,
  maxLines: maxLines,
  ellipsis: overflow == TextOverflow.ellipsis ? _kEllipsis : null,
  locale: locale,
```

```
  strutStyle: strutStyle,
  textWidthBasis: textWidthBasis,
  textHeightBehavior: textHeightBehavior,
)
```

This is where we finally see the instructions to paint the text on the screen:

Code 16.10:

```
void paint(Canvas canvas, Offset offset) {
  final double? minWidth = _lastMinWidth;
  final double? maxWidth = _lastMaxWidth;
  if (_paragraph == null || minWidth == null || maxWidth == null) {
    throw StateError(
      'TextPainter.paint called when text geometry was not yet
calculated.\n'
      'Please call layout() before paint() to position the text before
painting it.',
    );
  }

  if (_rebuildParagraphForPaint) {
    Size? debugSize;
    assert(() {
      debugSize = size;
      return true;
    }());

    _createParagraph();
    // Unfortunately we have to redo the layout using the same constraints,
    // since we've created a new ui.Paragraph. But there's no extra work
being
    // done: if _needsPaint is true and _paragraph is not null, the previous
    // `layout` call didn't invoke _layoutParagraph.
    _layoutParagraph(minWidth, maxWidth);
    assert(debugSize == size);
  }
  canvas.drawParagraph(_paragraph!, offset);
}
```

Underneath, the text is painted by the **canvas.drawParagraph()** method in the **paint()** method. The fun fact about this is that this method is easily available to users through methods like using a **CustomPaint** Widget. However, the issue is that a large amount of code is required to push this bit of text to the screen. As a final example, here is a piece of code that needs to be added to render basic text on to the screen using a **CustomPainter** instead of a Widget:

Code 16.11:

```
void paint(Canvas canvas, Size size) {
    var center = size / 2;
    var style = TextStyle(color: Colors.white);

    final ui.ParagraphBuilder paragraphBuilder = ui.ParagraphBuilder(
        ui.ParagraphStyle(
          fontSize:   style.fontSize,
          fontFamily: style.fontFamily,
          fontStyle:  style.fontStyle,
          fontWeight: style.fontWeight,
          textAlign: TextAlign.justify,
        )
    )
      ..pushStyle(style.getTextStyle())
      ..addText('Demo Text');
    final ui.Paragraph paragraph = paragraphBuilder.build()
      ..layout(ui.ParagraphConstraints(width: size.width));
    canvas.drawParagraph(paragraph, Offset(center.width, center.height));
  }
```

This is the absolute bare minimum for rendering a bit of code using this method. That's what Flutter does best: abstract away the parts that would have made developers' lives difficult otherwise.

Conclusion

To become a good developer with any framework, it is necessary to know it inside

out. Knowing about the underlying mechanisms of Flutter makes life as a Flutter developer much easier, from understanding bugs that pop up to easily answering

interview questions that may stump others. Regardless of the uses and applications of this knowledge, since Flutter is relatively unique in mobile frameworks for the mentioned methods of rendering and optimization, it provides unique appreciation for it.

In the next chapter, we will add tests to our Flutter app. Tests increase confidence in the code written since any changes that would normally break the code would have a harder time going through to production. Additionally, testing is an essential skill to work in any large-scale project.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 17

Writing Tests in Flutter

A Flutter app contains various kinds of code: code to create layouts and UI, code for network calls that fetch and post data, code to store data in databases, and much more. It is a complex mixture of various components and dependencies, which, in unison, create a full application for the user. However, this interaction between various aspects of the app is often relatively fragile, and a change in one part of the app may break local or global functionality critical to the app's purpose.

There is an abundance of tales of something going wrong with an app after a few seemingly insignificant changes, mainly when this has to do with authentication, user data, and/or especially payments. Any failure in these systems can cause users to lose trust within the app or the company. Payment systems, in particular, need to be robust since any bug can either lead to the company losing money or the users deciding to not invest in the app anymore.

Testing is a great way to make sure all the modules of your app work as they are expected to. Strictly speaking, tests are not a required part of an app; the app does not depend on the existence of tests to run. However, on a longer timescale and with

multiple people on a single project, it is difficult for a single person to judge whether a code change will cause only the intended effect. In effect, tests are there to ensure that nothing slips through the cracks and causes long-lasting or permanent damage.

In this chapter, we will go into the various kinds of tests in Flutter and implement them to create a more robust app.

Structure

In this chapter, we will discuss the following topics:

- The different types of testing
- Setting up tests
- Unit tests
- Widget tests
- Integration tests
- Golden tests

Objectives

After studying this chapter, you should be able to write all the types of tests in Flutter that aid in creating more robust and scalable apps. This includes creating golden tests and the associated images to make UI pixel perfect.

The different types of testing

Testing as a concept has various scopes, ranging from testing individual units of logic to testing the entire app as a whole. This ensures that the individual components of the app behave as expected individually and also when connected. To achieve this, different kinds of testing are needed to help different scopes:

- **Unit testing**: These test individual units of logic (such as a function or a class) and are relatively easy to write. However, since the scope is relatively localized and many conditions cannot be mocked, it does not have a great impact on the overall testing for the app itself, except for verifying the correctness of the individual classes or functions.
- **Widget testing**: These test out individual components, which are Widgets in Flutter. They are still relatively quick to run and allow mocking more things and simulating conditions and interactions. They make sure an individual Widget is performing as expected

Widget is performing as expected.

- **Integration testing**: These tests run on an actual device and carry out a sequence of events to make sure everything works well together. However, these tests are quite slow and heavy to run as compared to the other types.
- **Golden testing**: Golden testing is a great way to ensure that all UI components are pixel perfect and are not affected by changes. This requires generating golden files that match all future iterations to ensure that the UI is unaffected by the code differences.

Setting up tests

Tests, by default, are outside of the cozy `lib` folder in their own test folder:

***Figure 17.1**: The Flutter tests folder*

Unit and Widget tests run in the Flutter test environment and don't actually run on a physical/simulated device, so keep in mind the restrictions that may come along with being executed as such. Integration tests work directly on an emulator.

Exploring setUp() and tearDown()

Before we start testing, we need to know more about one convenience Flutter testing gives us: the ability to set stuff up before a test and clean up after.

We do this via the four functions provided to us:

Code 17.1:

```
void main() {
  setUpAll(() {
    // This is called once before ALL tests
  });

  setUp(() {
    // This is called once before EVERY test
  });

  tearDownAll(() {
    // This is called once after ALL tests
  });
```

```
  });

  tearDown(() {
    // This is called once after EVERY test
  });

  testWidgets(
    'Test description',
```



```
    (WidgetTester tester) async {
      // Write your test here
    },
  );
}
```

The **`setUpAll()`** and **`tearDownAll()`** functions are called just once each before and after execution of tests, respectively. The **`setUp()`** and **`tearDown()`** functions are called before and after every test; they help us set up or clean up things.

One important thing to remember is that the same functions are called for every test.

Exploring test variants

Sometimes, one test needs to be run for several values, each requiring its own setup and cleaning but with the same test code. As an example, let's say we have three colors that we must run the same test with. Let's make an **`enum`** out of this:

Code 17.2:

```
enum WidgetColor {
  red,
  blue,
  green,
}
```

Next, we create a test variant that allows us to run the same test for multiple values:

Code 17.3:

```
class ColorVariant extends TestVariant<WidgetColor> {
  @override
  String describeValue(WidgetColor value) {
```

```
    // TODO: implement describeValue
    throw UnimplementedError();
  }

  @override
  Future<Object> setUp(WidgetColor value) {
    // TODO: implement setUp
    throw UnimplementedError();
  }
```



```
  @override
  Future<void> tearDown(WidgetColor value, covariant Object memento) {
    // TODO: implement tearDown
    throw UnimplementedError();
  }

  @override
  // TODO: implement values
  Iterable<WidgetColor> get values => throw UnimplementedError();
}
```

We see our familiar **setUp()** and **tearDown()** functions, albeit with different params so that we can run setup for each value; however, the critical thing here is the values getter.

Now, we can add the **WidgetColor** values to the variant:

Code 17.4:

```
class ColorVariant extends TestVariant<WidgetColor> {
  @override
  String describeValue(WidgetColor value) {
    return value.toString();
  }

  @override
  Future<Object> setUp(WidgetColor value) {
    // Do setup here
  }
```

```
  @override
  Future<void> tearDown(WidgetColor value, covariant Object memento) {
    // Do teardown here
  }

  @override
  // TODO: implement values
  Iterable<WidgetColor> get values => WidgetColor.values;
}
```


This variant can now run the test for all the **`WidgetColor`** values. We can now supply this to our test via the **`variant`** parameter:

Code 17.5:

```
void main() {
  testWidgets(
    'Test description',
    (WidgetTester tester) async {
      // Write your test here
    },
    variant: ColorVariant(),
  );
}
```

Running this one test will run it three times for all **`WidgetColor`** values:

```
D:\AndroidStudioProjects\flutter_app>flutter test
00:01 +3: All tests passed!
```

Figure 17.2: *Running working tests in Flutter*

Adding timeouts for a test

This is slightly trickier, since if you investigate, there are two parameters for setting timeouts for a test:

Code 17.6:

```
testWidgets(
```

```
    'Test description',
    (WidgetTester tester) async {
      // Write your test here
    },
    timeout: Timeout(Duration(minutes: 1)),
    initialTimeout: Duration(seconds: 15),
  );
```

Without going into the specifics of why it is this way, here's a gist of what is happening: The **`initialTimeout`** is the main timeout applied and can be increased, but it cannot be more than the timeout parameter.

To increase the timeout, we can use the following:

Code 17.7:

```
testWidgets(
    'Test description',
    (WidgetTester tester) async {
      // Write your test here

      // To increase timeout
      tester.binding.addTime(Duration(seconds: 5));

    },
    timeout: Timeout(Duration(minutes: 1)),
    initialTimeout: Duration(seconds: 15),
  );
```

This adds time to the initial timeout, but if we add more time than the timeout parameter, it won't be counted.

Unit tests

Unit tests test individual units of logic, such as functions or classes in an app. They are also not associated with any UI dependencies. A simple example of unit tests is a calculator:

Code 17.8:

```
class Calculator {
```

```
  double add(double a, double b) {
    return a + b;
  }

  double subtract(double a, double b) {
    return a - b;
  }

  double multiply(double a, double b) {
    return a * b;
  }
```



```
  double divide(double a, double b) {
    return a / b;
  }

}
```

We can now write a unit test using the **`test()`** function:

Code 17.9:

```
void main() {

  test('Calculator addition test', () {
    final calc = Calculator();

    var res = calc.add(2,3);

    expect(res, 5);
  });

}
```

The **`expect()`** function checks whether the value matches with the expected value. Tests can also be grouped together using the **`group()`** function:

Code 17.10:

```
void main() {
  group('Calculator Tests', () {

    test('Calculator addition test', () {
      final calc = Calculator();

      var res = calc.add(2,3);

      expect(res, 5);
    });

    test('Calculator subtraction test', () {
      final calc = Calculator();
```



```
      var res = calc.subtract(5,2);

      expect(res, 3);
    });

  });
}
```

Widget tests

A Widget test is usually written for one of two things:

- Checking whether visual elements are present
- Checking whether the interaction with visual elements gives the correct result

Let's go with the second type since the first type can be added in as well. To do this, we usually follow a few steps in the test:

1. Set up requisites and create (pump) a Widget to test with.
2. Find the visual elements on the screen via some kind of property (such as a key).
3. Interact with the elements (such as a button) using the same identifier.

4. Verify that the results match what was expected.

Let's look at the structure of a test:

Code 17.11:

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

import 'package:flutter_app/main.dart';

void main() {
  testWidgets(
    'Test description',
    (WidgetTester tester) async {
      // Write your test here
```

```
    },
  );
}
```

The basics look pretty simple:

- The **`main()`** function seems to have tests inside it.
- The **`testWidgets()`** function seems to hold a test.
- Inside **`testWidgets()`**, we have a description for the test itself and a way to write the actual code for the test.

Creating (pumping) a widget to test

To test a Widget, we need a Widget, obviously. Let's look at the default test in the test folder:

Code 17.12:

```
void main() {

  testWidgets(
    'Test description',
    (WidgetTester tester) async {
      // Write your test here
    },
  );

}
```

You may have noticed the **`WidgetTester`** object provided to us in the callback where we write our test. It's time to put that to use.

To pump a new Widget to test, we use the **`pumpWidget()`** method:

Code 17.13:

```
testWidgets(
    'Test description',
    (WidgetTester tester) async {
      // Write your test here

      await tester.pumpWidget(
```

```
        MaterialApp(
          home: Scaffold(
            appBar: AppBar(),
          ),
        ),
      );
    },
  );
```

(Don't forget the await or the test will complain beyond belief.)

This inflates a Widget for us to test with. We will go into more detail about the **`WidgetTester`** in just a bit, but there's something else we need to know better first.

Understanding Finders

While the first step involves instantiating a Widget for testing, the second step on our quest to write a test involves finding a visual element we want to interact with; it can be a button, text, and so on.

So how do we find a Widget? We use a finder. (You can find elements as well, but I digress.)

That's easy enough to say, but in reality, you need to identify something that's unique to the Widget: the type, the text, the descendants, or its ancestors, etc.

Let's look at some common and some of the more interesting ways to find a Widget:

find.byType()

Let's take an example of finding a **`Text`** Widget:

Code 17.14:

```
testWidgets(
    'Test description',
    (WidgetTester tester) async {
      // Write your test here

      await tester.pumpWidget(
        MaterialApp(
          home: Scaffold(
            appBar: AppBar(),
```

```
          body: Center(
            child: Text('Hi there!'),
          ),
        ),
      ),
    );

    var finder = find.byType(Text);
  },
);
```

Here, we use a predefined instance of the **CommonFinders** class called find to create a finder. The **byType()** function helps us identify **ANY** Widget of a particular type. So, if two texts existed in the Widget tree, BOTH would be identified. If you intend to find a particular **Text** Widget, consider adding a key or using the next type.

find.text()

To find a specific **Text** Widget, use **find.text()**:

Code 17.15:

```
testWidgets(
  'Test description',
  (WidgetTester tester) async {
    // Write your test here

    await tester.pumpWidget(
      MaterialApp(
        home: Scaffold(
          appBar: AppBar(),
          body: Center(
            child: Text('Hi there!'),
          ),
        ),
      ),
    );

    var finder = find.text('Hi there!');
  },
);
```

This also works for any **EditableText**, such as a **TextField** Widget:

Code 17.16:

```
testWidgets(
    'Test description',
    (WidgetTester tester) async {
      // Write your test here

      var controller = TextEditingController.
fromValue(TextEditingValue(text: 'Hi there!'));

      await tester.pumpWidget(
        MaterialApp(
          home: Scaffold(
            appBar: AppBar(),
            body: Center(
              child: TextField(controller: controller,),
            ),
          ),
        ),
      );

      var finder = find.text('Hi there!');
    },
  );
```

find.byKey()

One of the most common and easiest ways to find a Widget is by just attaching a key to it:

Code 17.17:

```
testWidgets(
    'Test description',
    (WidgetTester tester) async {
      // Write your test here

      await tester.pumpWidget(
```

```
      MaterialApp(
        home: Scaffold(
          appBar: AppBar(),
          body: Center(
            child: Icon(
              Icons.add,
              key: Key('demoKey'),
            ),
          ),
        ),
      ),
    );

    var finder = find.byKey(Key('demoKey'));
  },
);
```

find.descendant() and find.ancestor()

This is a more interesting type where you can find a descendant or ancestor of a Widget matching a particular property (which we again use a finder for).

Suppose we want to find an icon that is a descendant of a **Center** Widget that has a key; in this case, we can do the following:

Code 17.18:

```
testWidgets(
  'Test description',
  (WidgetTester tester) async {
    // Write your test here

    await tester.pumpWidget(
      MaterialApp(
        home: Scaffold(
          appBar: AppBar(),
          body: Center(
            key: Key('demoKey'),
            child: Icon(Icons.add),
```

```
          ),
        ),
      ),
    );

    var finder = find.descendant(
      of: find.byKey(Key('demoKey')),
      matching: find.byType(Icon),
    );
  },
);
```

We specify here that the Widget we intend to find is a descendant **`of`** the **`Center`** Widget and matches properties that we identify with a finder again.

The **`find.ancestor()`** call is mostly similar, but the roles are reversed since we are trying to find a Widget above the widget identified by the **`of()`** parameter.

If we were trying to identify the **`Center`** Widget, we would do the following:

Code 17.19:

```
testWidgets(
  'Test description',
  (WidgetTester tester) async {
    // Write your test here

    await tester.pumpWidget(
      MaterialApp(
        home: Scaffold(
          appBar: AppBar(),
          body: Center(
            key: Key('demoKey'),
            child: Icon(Icons.add),
          ),
        ),
      ),
    );
```

```
var finder = find.ancestor(
```

```
        of: find.byType(Icon),
        matching: find.byKey(Key('demoKey')),
      );
    },
  );
```

Understanding the WidgetTester

The **WidgetTester** is the main class used in Widget tests:

Code 17.20:

```
testWidgets(
    'Test description',
    (WidgetTester tester) async {
      // Write your test here
    },
  );
```

The **WidgetTester** allows us to interact with the test environment. Widget tests don't exactly run how they would run on a device since the asynchronous behavior in a test is mocked rather than real. There are also other differences to note.

The **setState()** function does not work as usual in a Widget test.

While **setState()** marks the Widget to be rebuilt, it does not actually rebuild the tree in a widget test. So how do we do it? Let's look at the pump methods.

TL;DR: **pump()** triggers a new frame (rebuilds the Widget), **pumpWidget()** sets the root Widget and then triggers a new frame, and **pumpAndSettle()** calls **pump()** until the Widget does not request new frames anymore (usually when animations are running).

A bit about pumpWidget()

As we saw earlier, **pumpWidget()** was used to set the root Widget for testing. It calls **runApp()** using the Widget provided and calls **pump()** internally. If the function is called again, it rebuilds the entire tree.

A bit about pump()

We need to call **pump()** to actually rebuild Widgets that need to be rebuilt. Let's say we have a basic counter Widget like this:

Code 17.21:

```
class CounterWidget extends StatefulWidget {
  @override
  _CounterWidgetState createState() => _CounterWidgetState();
}

class _CounterWidgetState extends State<CounterWidget> {
  var count = 0;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: Text('$count'),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.add),
          onPressed: () {
            setState(() {
              count++;
            });
          },
        ),
      ),
    );
  }
}
```

The Widget simply stores a count and updates it when a **`FloatingActionButton`** is hit, like the default counter app.

Let's try testing the Widget by finding the add icon and pressing it to verify that the count turns to **1**:

Code 17.22:

```
testWidgets(
    'Test description',
    (WidgetTester tester) async {
      // Write your test here
```

```
      await tester.pumpWidget(CounterWidget());

      var finder = find.byIcon(Icons.add);
      await tester.tap(finder);

      // Ignore this line for now
      // It just verifies that the value is what we expect it to be
      expect(find.text('1'), findsOneWidget);
    },
  );
```

This doesn't quite work:

```
00:01 +0 -1: Some tests failed.
```

Figure 17.3: Output of failing tests in Flutter

The reason is that we rebuild the **`Text`** Widget that displays the count using **`setState()`** in the Widget, but that does not rebuild the Widget. In addition, we need to call the **`pump()`** method:

Code 17.23:

```
testWidgets(
    'Test description',
    (WidgetTester tester) async {
      // Write your test here
      await tester.pumpWidget(CounterWidget());

      var finder = find.byIcon(Icons.add);
      await tester.tap(finder);
      await tester.pump();

      // Ignore this line for now
      // It just verifies that the value is what we expect it to be
      expect(find.text('1'), findsOneWidget);
    },
  );
```

If you need to schedule a frame after a specific duration, **`pump()`** also takes a duration, which schedules a rebuild AFTER that duration:

Code 17.24:

```
await tester.pump(Duration(seconds: 1));
```

Going to pumpAndSettle()

The **`pumpAndSettle()`** method is the **`pump()`** method but called until no new frames are scheduled. This helps finish all animations.

It has similar duration and engine **`phase`** parameters but also adds a **`timeout`** parameter for capping how long it can be called.

Code 17.25:

```
await tester.pumpAndSettle(
        Duration(milliseconds: 10),
        EnginePhase.paint,
        Duration(minutes: 1),
      );
```

Interaction with the environment

`WidgetTester` allows us to add complex interactions beyond the usual finder + tap interactions as well. Let's look at a few things you can do.

The **`tester.drag()`** function allows the creation of a drag from the middle of a Widget that we identify with a finder by a certain offset. We can specify the direction of the drag by specifying the x and y slopes of the direction:

Code 17.26:

```
var finder = find.byIcon(Icons.add);
var moveBy = Offset(100, 100);
var slopeX = 1.0;
var slopeY = 1.0;

await tester.drag(finder, moveBy, touchSlopX: slopeX, touchSlopY:
slopeY);
```

We can also create a timed drag using **`tester.timedDrag()`**:

Code 17.27:

```
var finder = find.byIcon(Icons.add);
var moveBy = Offset(100, 100);
var dragDuration = Duration(seconds: 1);

await tester.timedDrag(finder, moveBy, dragDuration);
```

If you don't want to use finders and just want to drag from one position on the screen to another instead, use **`tester.dragFrom()`**; it allows you to start the drag from a position on the screen.

Code 17.28:

```
var dragFrom = Offset(250, 300);
var moveBy = Offset(100, 100);
var slopeX = 1.0;
var slopeY = 1.0;

await tester.dragFrom(dragFrom, moveBy, touchSlopX: slopeX, touchSlopY:
slopeY);
```

Integration tests

While unit tests test out individual units of logic and Widget tests test out single Widgets, integration tests are used for the entire app in a sense, as we can test out a full flow of actions that may involve many Widgets and a lot of logic. Integration testing is significantly slower and heavier than other types since it is directly carried out on an emulator.

Integration testing is incredibly useful for testing out full app flows like login, payments, profile changes, and more. This ensures that these flows aren't broken with code changes. Many a time, individual Widgets may work fine, but problems begin on the scale of the entire app. Integration testing makes these kinds of flow issues much harder to miss.

To begin with integration testing, add the dependency to **`pubspec.yaml`**:

Code 17.29:

```
dev_dependencies:
```

```
integration_test:
  sdk: flutter
```


Also create a new folder called **integration_test** and add an **app_test.dart** file.

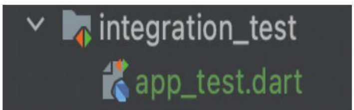


Figure 17.4: Adding integration tests to your Flutter app

At the top of the file, we add this line to make sure it runs on the emulator:

Code 17.30:

```
IntegrationTestWidgetsFlutterBinding.ensureInitialized();
```

After this, we can use the **testWidgets()** function to get started with integration testing:

Code 17.31:

```
import 'package:your_app/main.dart' as app;

void main() {
  IntegrationTestWidgetsFlutterBinding.ensureInitialized();

  group('demo group', () {

    testWidgets('demo login flow with login button tap',
        (WidgetTester tester) async {
      app.main();
      await tester.pumpAndSettle();

      expect(find.text('Sign Up'), findsOneWidget);

      final Finder loginButton = find.byTooltip('Login');

      await tester.tap(loginButton);

      await tester.pumpAndSettle();

      expect(find.text('Enter Email'), findsOneWidget);
    });
```

```
  });
}
```


In this test, we first import the **main()** function of the app to initialize the app itself. We verify that the login screen is present by checking whether the **Sign Up** text is present. After verifying this, the finder tries to tap the login button. After this, we verify that the screen displaying the email text field shows up.

Like this, app flows can be automated and tested, and confidence in the code can be retained without re-running expensive manual testing.

Golden tests

Most apps are now designed to high design standards, which need to be maintained throughout the development of an app. Golden tests primarily focus on creating and maintaining pixel-perfect UI in an application. This is done by instantiating a Widget similar to a normal Widget test and then generating golden files that are used to compare the original design to all the future iterations of code.

To write a golden test, we start like a normal Widget test: by pumping a Widget using **WidgetTester**:

Code 17.32:

```
void main() {

  testWidgets('Demo golden test', (WidgetTester tester) async {

    await tester.pumpWidget(YourWidget());

    await expectLater(find.byType(YourWidget),
                      matchesGoldenFile('demo_golden.png'));
  });

}
```

After this, we use the **expect** function (**expectLater** in this case since the function matching returns a future) to compare the Widget to the golden file. The **matchesGoldenFile()** function matches the Widget to the golden file.

However, to do this, the golden files required must be generated. To do this, you can

However, to do this, the golden files required must be generated. To do this, you can run this command to generate the files required when the UI is finalized:

Code 17.33:

```
flutter test --update-goldens
```


This runs through the tests and generates the files required for any golden tests identified by the **`matchesGoldenFile()`** function.

These tests now guarantee that the UI is not disrupted with future changes, and the same does not need to be visually verified; this can save a lot of time over the long run.

Conclusion

While not strictly necessary, testing provides an extra level of checks before pushing any new code or publishing the app to the respective app stores. This also adds development speed over time since any changes can be pushed with confidence if no tests are broken. Most large organizations also have testing knowledge as a mandatory part of interviews since they most likely have a large team for a single project, and testing becomes essential knowledge.

In the next chapter, we will look outside our own app and explore the popular external packages we can use.

Questions

1. Why is testing needed in a Flutter application?
2. What is the difference between integration and unit test?
3. How does a golden test differ to an integration test?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 18

Popular Flutter Packages

Introduction

Various kinds of logic need to be written to fulfill an application's intended purpose. This can be logic intended to fetch network data, represent the fetched data with UI, log a user in, and so on. Additionally, there are some tasks like picking a photo, reading sensor data, logging operations being done inside the app code, and more. While most of these operations will be defined by the developers, some parts are not doable without very complex and/or native code. As an example, cameras/sensors cannot be accessed directly via Dart, so we write plugins to fetch the data natively and provide it to us in Dart. These packages/plugins are written by Google or developers in the Flutter community and can greatly simplify developer workload. However, packages are not only for native code. They can be extensions or improvements over existing Flutter components like images, video players, and so on. They can also be

prebuilt UI, such as dialogs, snack bars, and animated components, which may take large amounts of code to build otherwise.

Utilizing Flutter packages efficiently allow us to create our apps with the least amount of effort and without reinventing the wheel every time. Since most Flutter apps rely on authentication, camera, file picking, and more, packages are essential to build apps. However, there are also drawbacks to relying on external code, which we will discuss towards the end of this chapter.

Note: There are two words used here: *package* and *plugin*. Package usually refers to a Dart-only library of code, whereas plugin refers to a library where both Dart and native code exist. Later, both started being referred to as packages to avoid unnecessary complexity; however, this is something to be mindful of when creating your own library.

Structure

In this chapter, we will discuss the following topics:

- dio
- url_launcher
- file_picker
- image_picker
- geolocator
- connectivity_plus
- sensors_plus
- google_maps_flutter
- animated_text_kit
- cached_image_network
- chewie
- auto_size_text
- flame
- Drawbacks of using packages

Objectives

In this chapter, we will try to cover the popular and essential Flutter packages that most apps may need to use. We will briefly look at the purpose of the package and explore some code related to using it. Since this is a large and non-exhaustive list, readers are recommended to give the packages a try themselves to understand the full set of nuances that the packages hold.

dio

Flutter offers a **`http`** package by default, which handles a lot of the usual HTTP calls and configuration. However, for more complex apps, developers usually need more; this is where **`dio`** comes in.

To start out with **`Dio`**, we first initialize the **`Dio`** class. Using this, we can perform our standard HTTP requests:

Code 18.1:

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


var dio = Dio();


var getResponse = await dio.get('...');


var postResponse = await dio.post('...', data: {});
```

The **`dio`** class offers advanced features, such as the following:

- **Interceptors**: Allows you to execute code before, after, and on error when executing certain or all queries:

  Code 18.2:

  ```
  dio.interceptors.add(InterceptorsWrapper(
      onRequest:(options, handler){
       // Called before request is sent
      },
      onResponse:(response,handler) {
       // Called after request is sent
      },
      onError: (DioError e, handler) {
         // Called on error
      }
  ));
  ```

- **Global config**: Sets configurations for all queries
- **FormData**: Allows for sending larger files by splitting them into multiple parts:

  Code 18.3:

```
var file = await MultipartFile.fromFile('...',filename: '...');

var formData = FormData.fromMap({
  'file': file,
```



```
});

var response = await dio.post('...', data: formData);
```

- **Request cancellation**: Uses a token to cancel HTTP requests:

  Code 18.4:

```
var token = CancelToken();

dio.get(url, cancelToken: token)
   .catchError((DioError err){

   });

token.cancel('cancelled');
```

- **File downloading**: Self-explanatory:

  Code 18.5:

```
var response = await dio.download('...', '...');
```

- **Timeout**: Timeout HTTP requests:

  Code 18.6:

```
BaseOptions options = BaseOptions( baseUrl: "...",
receiveDataWhenStatusError: true, connectTimeout: 6000,
receiveTimeout: 6000,);
```

```
var dio = new Dio(options);
```

… and more.

url_launcher

As the name suggests, the **url_launcher** package helps launch URLs in a Flutter app. This not only means web links; it also opens SMS and email apps, dials a phone number, and more.

A few examples of its capabilities are provided in the following sections.

Web link

To open a link, we can use the **launch()** function of the **url_launcher** package:

Code 18.7:

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

const String _url = 'https://flutter.dev';

void launchUrl() async {

   var result = await launch(_url);

}
```

Mail

To open a mailing app, we need to form the URL in a specific format that contains several aspects of the email. We can then pass this to the same **launch()** function as a normal URL:

Code 18.8:

```
const String _url = 'mailto:<email
address>?subject=<subject>&body=<body>';

void launchMail() async {
```

```
  var result = await launch(_url);

}
```

Phone

You can construct a URL to open a phone number using the **tel** prefix and adding a phone number:

Code 18.9:

```
const String _url = 'tel:<phone number>';

void launchPhone() async {
```



```
  var result = await launch(_url);

}
```

SMS

You can construct a URL to send an SMS using the **sms** prefix and adding a phone number:

Code 18.10:

```
const String _url = 'sms:<phone number>';

void launchSMS() async {

  var result = await launch(_url);

}
```

This plugin can also be used for other purposes like opening files on a desktop. It is used in most consumer-facing Flutter apps and hence, is quite important to know.

file_picker

Accessing stored files on the phone is essential to a lot of functions: picking images or

video for posting to feeds, uploading documents to a form, and more. **file_picker** allows us to pick one or more files of various types.

Picking single file

To pick a single file, use the **pickFiles()** method:

Code 18.11:

```
FilePickerResult? result = await FilePicker.platform.pickFiles();

if (result != null) {
  File file = File(result.files.single.path);
}
```


Picking multiple files

Picking multiple files is like picking a single file, albeit with the **allowMultiple** parameter:

Code 18.12:

```
FilePickerResult? result = await FilePicker.platform.
pickFiles(allowMultiple: true);
```

Pick certain types of files

You can also add allowed extensions to the **pickFiles()** method, which only allows picking files of a certain type:

Code 18.13:

```
FilePickerResult? result = await FilePicker.platform.pickFiles(
  type: FileType.custom,
  allowedExtensions: ['jpg', 'pdf', 'doc'],
);
```

image_picker

image_picker is similar to the previous package but focuses on the image picker instead of a more generic file picker.

Picking a single image

The **pickImage()** method from the **ImagePicker** class picks a single image. The source parameter determines whether the image is from the camera or the gallery.

Code 18.14:

```
final ImagePicker _picker = ImagePicker();

final XFile? image = await _picker.pickImage(source: ImageSource.
gallery);
```

Picking multiple images

The **pickMultiImage()** method from the **ImagePicker** class picks multiple images. The source parameter determines whether the image is from the camera or the gallery:

Code 18.15:

```
final ImagePicker _picker = ImagePicker();

final List<XFile>? images = await _picker.pickMultiImage();
```

Capture image or video

You can also pick videos using the package:

Code 18.16:

```
final ImagePicker _picker = ImagePicker();

final XFile? photo = await _picker.pickImage(source: ImageSource.camera);

final XFile? video = await _picker.pickVideo(source: ImageSource.camera);
```

geolocator

Most large-scale apps need access to location, whether it is for targeted content, local information, or delivery details. The easier it is to fetch information about a user's geolocation, the easier it is for developers to deal with the actual logic to implement in the app. 'Geolocator' simplifies getting a user's location on multiple platforms.

Getting location

You can get the user's current location using the **getCurrentPosition()** method. You can also supply the precision required:

Code 18.17:

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


var position = await Geolocator.getCurrentPosition();
```

This is a bit simplified since getting location requires a lot of permissions and previous setup on the native side.

Getting last known location

If the GPS is not available, you can also get the last known location of the user:

Code 18.18:

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


Position? position = await Geolocator.getLastKnownPosition();
```

Listening to location

To get a continuous stream of location updates from the user, you can use the **getPositionStream()** method:

Code 18.19:

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


final LocationSettings locationSettings = LocationSettings(
  accuracy: LocationAccuracy.high,
  distanceFilter: 100,
);
StreamSubscription<Position> positionStream = Geolocator.
getPositionStream(locationSettings: locationSettings).listen(
    (Position? position) {
        print(position == null ? 'Unknown' : '${position.latitude.
toString()}, ${position.longitude.toString()}');
```

```
});
```

connectivity_plus

The **connectivity_plus** plugin focuses on fetching data about a device's network connectivity. This can be used to distinguish between Wi-Fi and cellular, and also to listen to connectivity updates.

Check connection status

The **checkConnectivity()** method checks for network connectivity on the device. Note that this is not necessarily internet connectivity, just whether it is connected to a network.

Code 18.20:

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

var connectivityResult = await (Connectivity().checkConnectivity());
```


Check WiFi vs cellular

The connectivity result from the previous section can be used to check whether the device is connected to a WiFi network or a mobile network:

Code 18.21:

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

var connectivityResult = await (Connectivity().checkConnectivity());
if (connectivityResult == ConnectivityResult.mobile) {
  // Connected to a mobile network.
} else if (connectivityResult == ConnectivityResult.wifi) {
  // Connected to a wifi network.
}
```

Listen to connection status

The package also offers a way to listen to changes in connectivity using a **Stream** named **onConnectivityChanged**:

Code 18.22:

```
Connectivity().onConnectivityChanged.listen((ConnectivityResult result)
{
  // Do something on result
});
```

sensors_plus

A phone usually comes bundled with various kinds of sensors, many of which we use daily without realizing. We use the magnetometer to find the direction on a map, the accelerometer to get the orientation of the device, a gyroscope to drive a virtual car around by rotating the device, and so on. When it comes to building apps, we often need some kind of sensor data to work with. Since integrating sensors usually involves native code, doing it ourselves is tricky. **`sensors_plus`** does this task for us and allows us to use sensor data when needed.

Listening to accelerometer events

The package provides several **`Streams`**, such as **`accelerometerEvents`**, which notify you of the accelerometer events associated with the device.

Code 18.23:

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

accelerometerEvents.listen((AccelerometerEvent event) {
  print(event);
});
```

Listening to user accelerometer events

The accelerometer usually measures gravitational acceleration in addition to any acceleration caused by the user. The **`userAccelerometerEvents`** stream provides the values excluding gravity, hence giving us the actual acceleration, which is usually what we need:

Code 18.24:

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

userAccelerometerEvents.listen((UserAccelerometerEvent event) {
  print(event);
});
```

```
});
```

Listening to magnetometer events

The magnetometer allows measuring the magnetic field around the device. This is useful in various situations since it can measure the heading of the device relative to the Earth's magnetic field. To listen to magnetometer events, you can listen to the corresponding **Stream** in the package:

Code 18.25:

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


magnetometerEvents.listen((MagnetometerEvent event) {
  print(event);
});
```

Listening to gyroscope events

The gyroscope measures changes in device orientation. This is useful for many use cases one of which is Augmented Reality (AR). AR adds objects on top of the real world. These objects need to be moved according to device movement. The

gyroscope gives us this movement. You can use the corresponding **Stream** for the package to listen to these events:

Code 18.26:

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


gyroscopeEvents.listen((GyroscopeEvent event) {
  print(event);
});
```

google_maps_flutter

Integrating Google maps is almost unavoidable for a majority of apps. Hence, having good support for adding a map widget and the usual feature set like markers, different views, live user location, and so on is crucial. The **google_maps_flutter** plugin is the plugin developed by Google to do this.

Note: *Plugin,* since the Google map used in the package is not actually written

in Dart but a native map, is converted into a Flutter widget to be used in an app.

While there is a fair amount of setup involved, the usage of the map is easy on its own:

Code 18.27:

```
class MapDemo extends StatefulWidget {
  const MapDemo({Key? key}) : super(key: key);

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

class _MapDemoState extends State<MapDemo> {
  late GoogleMapController controller;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: GoogleMap(
        initialCameraPosition: CameraPosition(...),
```



```
        onMapCreated: (c) {
          controller = c;
        },
      ),
    );
  }
}
```

We need to provide an initial position to the **GoogleMap** widget, which can have a latitude, longitude, zoom for the map, and so on.

animated_text_kit

We discussed the advantages of animations at length in the animation chapter; this plugin allows us to create fancy text animations with little effort.

A few examples of the animations supported are given in the following sections.

Rotate

This animation cycles through different texts in the same area:

Code 18.28:

```
DefaultTextStyle(
      style: const TextStyle(
        fontSize: 40.0,
        fontFamily: 'Horizon',
      ),
      child: AnimatedTextKit(
        animatedTexts: [
          RotateAnimatedText('AWESOME'),
          RotateAnimatedText('OPTIMISTIC'),
          RotateAnimatedText('DIFFERENT'),
        ]
        onTap: () {
          print("Tap Event");
        },
      ),
    ),
```


Scale

The animation cycles through the given text and the transition is carried through by scaling the different texts:

Code 18.29:

```
DefaultTextStyle(
    style: const TextStyle(
      fontSize: 70.0,
      fontFamily: 'Canterbury',
    ),
    child: AnimatedTextKit(
      animatedTexts: [
        ScaleAnimatedText('Think'),
        ScaleAnimatedText('Build'),
```

```
      ScaleAnimatedText('Ship'),
    ],
    onTap: () {
      print("Tap Event");
    },
  ),
),
```

Fade

The animation cycles through the given text and the transition is carried through by scaling the different texts:

Code 18.30:

```
DefaultTextStyle(
  style: const TextStyle(
    fontSize: 32.0,
    fontWeight: FontWeight.bold,
  ),
  child: AnimatedTextKit(
    animatedTexts: [
      FadeAnimatedText('do IT!'),
      FadeAnimatedText('do it RIGHT!!'),
```



```
      FadeAnimatedText('do it RIGHT NOW!!!'),
    ],
    onTap: () {
      print("Tap Event");
    },
  ),
),
```

Typewriter

There is a typewriter effect when displaying any text where each character is displayed one by one; here, the animation cycles through the different **`animatedTexts`**.

Code 18.31:

```
DefaultTextStyle(
```

```
    style: const TextStyle(
      fontSize: 30.0,
      fontFamily: 'Agne',
    ),
    child: AnimatedTextKit(
      animatedTexts: [
        TypewriterAnimatedText('Discipline is the best tool'),
        TypewriterAnimatedText('Design first, then code'),
        TypewriterAnimatedText('Do not patch bugs out, rewrite them'),
        TypewriterAnimatedText('Do not test bugs out, design them out'),
      ],
      onTap: () {
        print("Tap Event");
      },
    ),
  ),
```

cached_network_image

Loading images can often be quite network heavy, and it can also cause performance degradation of the app. Hence, caching the image when retrieved is an important step to making a performant app and not adding on to the user's data bill.

Code 18.32:

```
CachedNetworkImage(
  imageUrl: "http://via.placeholder.com/200x150",
  imageBuilder: (context, imageProvider) => Container(
    decoration: BoxDecoration(
      image: DecorationImage(
          image: imageProvider,
          fit: BoxFit.cover,
          colorFilter:
              ColorFilter.mode(Colors.red, BlendMode.colorBurn)),
    ),
  )
```

```
  ),
  placeholder: (context, url) => CircularProgressIndicator(),
  errorWidget: (context, url, error) => Icon(Icons.error),
),
```

Thankfully, there are no extra steps involved in caching the images; the package does it for you. There are also additional builders when it comes to adding a placeholder and in the case of errors.

chewie

The default video player that comes bundled with Flutter is often seen as limited since it lacks a lot of controls and also the UI that is expected from users. This leaves developers with a significant amount of work to do when it comes to building a decent video player. **`Chewie`** builds on top of the default video player for Flutter and adds significant missing aspects. Note that you will still need to use aspects of both the default video player and the **`chewie`** package.

Here's how we would set up and create the main widget for playing a video from the internet:

Code 18.33:

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

final videoPlayerController = VideoPlayerController.network(
    '...');

await videoPlayerController.initialize();
```



```
final chewieController = ChewieController(
  videoPlayerController: videoPlayerController,
  autoPlay: true,
  looping: true,
);

final playerWidget = Chewie(
  controller: chewieController,
);
```

There are many things we expect from a mature video solution, including subtitles:

Code 18.34:

```
ChewieController(
  videoPlayerController: _videoPlayerController,
  autoPlay: true,
  looping: true,
  subtitle: Subtitles([
    Subtitle(
      index: 0,
      start: Duration.zero,
      end: const Duration(seconds: 10),
      text: 'Hello from subtitles',
    ),
    Subtitle(
      index: 1,
      start: const Duration(seconds: 10),
      end: const Duration(seconds: 20),
      text: 'Whats up? :)',
    ),
  ]),
  subtitleBuilder: (context, subtitle) => Container(
    padding: const EdgeInsets.all(10.0),
    child: Text(
      subtitle,
      style: const TextStyle(color: Colors.white),
    ),
```

```
  ),
);
```

auto_size_text

The yellow and black overflow lines are often hated sights when it comes to the process of Flutter app development. They indicate that your widgets exceed the space given to them. While sometimes this is fine, it's almost always not when it comes to rendering text. When text overflows, it makes the user miss potentially important content in the app. Here, **`auto_size_text`** helps us avoid this issue by automatically sizing the text to fit the available space.

The usage of the package is fairly straightforward:

Code 18.35:

```
AutoSizeText(
  'The text to display',
  style: TextStyle(fontSize: 20),
  maxLines: 2,
)
```

We can also set max and min sizes for the text:

Code 18.36:

```
AutoSizeText(
  'A really long String',
  style: TextStyle(fontSize: 30),
  minFontSize: 18,
  maxLines: 4,
  overflow: TextOverflow.ellipsis,
)
```

There are several other options when it comes to customizing how the widget handles sizing or overflow:

Code 18.37:

```
AutoSizeText(
  'A String tool long to display without extreme scaling or overflow.',
  maxLines: 1,
  overflowReplacement: Text('Sorry String too long'),
)
```

We can also use **RichText**, which allows us to create multiple text spans with different sizes and styles:

Code 18.38:

```
AutoSizeText.rich(
  TextSpan(text: 'A really long String'),
  style: TextStyle(fontSize: 20),
  minFontSize: 5,
)
```

flame

While we have mostly been discussing plugins apps we absolutely need till now, this is something a bit different. **flame** is Flutter's most popular game engine and supports all the complex components required for game development, such as collision detection, gestures, and audio.

A Flame game is made up of components that we can define. Let's look at the example given by the Flame team:

Code 18.39:

```
class Square extends PositionComponent {
  static const speed = 0.25;
  static const squareSize = 128.0;

  static Paint white = BasicPalette.white.paint();
  static Paint red = BasicPalette.red.paint();
  static Paint blue = BasicPalette.blue.paint();

  @override
  void render(Canvas c) {
    c.drawRect(size.toRect(), white);
    c.drawRect(const Rect.fromLTWH(0, 0, 3, 3), red);
    c.drawRect(Rect.fromLTWH(width / 2, height / 2, 3, 3), blue);
  }

  @override
  void update(double dt) {
    super.update(dt);
```

```
    angle += speed * dt;
    angle %= 2 * math.pi;
  }

  @override
  Future<void> onLoad() async {
    super.onLoad();
    size.setValues(squareSize, squareSize);
    anchor = Anchor.center;
  }
}
```

This creates a square component. It has an **onLoad()** similar to the **initState()** function of Flutter, the **render()** method where we initialize any resources or draw anything on the screen required, and the **update()** method that is used for value changes on every frame, such as position and velocity.

We can define the game itself using the **FlameGame** class:

Code 18.40:

```
class MyGame extends FlameGame with DoubleTapDetector, TapDetector {
  bool running = true;

  @override
  Future<void> onLoad() async {
    await super.onLoad();
    add(
      Square()
        ..x = 100
        ..y = 100,
    );
  }

  @override
  void onTapUp(TapUpInfo info) {
    final touchPoint = info.eventPosition.game;

    final handled = children.any((c) {
```

```
      if (c is PositionComponent && c.containsPoint(touchPoint)) {
        remove(c);
        return true;
      }
      return false;
    });

    if (!handled) {
      add(Square()..position = touchPoint);
    }
  }

  @override
  void onDoubleTap() {
    if (running) {
      pauseEngine();
    } else {
      resumeEngine();
    }

    running = !running;
  }
}
```

Here, **TapDetector** and others are gesture detector mixins.

To run the game, we use a **GameWidget**:

Code 18.41:

```
void main() {
  runApp(
    GameWidget(
      game: MyGame(),
    ),
  );
}
```

The Flame engine is quite complex, and this was a relatively simple example given in the package; that said, it is good to know there is a mature game development option for Flutter.

Drawbacks of packages

While initially there may seem no harm to importing and using a package that does most of the heavy lifting for us, there are concerns over time. A package is usually maintained by a single person or a small group that concentrates on it in their free time. This usually means that a package has a good chance of getting less attention over time by the maintainers since they are usually doing it for free and likely have other work to focus on as well. While there is no intention to start a debate regarding the open-source model here, this means that choosing a package after looking at a previous track record of maintenance is usually a good idea. Otherwise, if a package is broken after some Flutter changes, it may take a lot of time or may require you to create a fork and maintain it yourself to fix it.

Additionally, being too reliant on packages is a bit risky since it denies you customizability and likely locks you into the package framework. Long story short, be mindful of which packages you pick to go into your app; look for good track record and generally, look for a package with a relatively sharp focus.

As a last note, if you are creating an app, you can fork a package and use it temporarily if the above-mentioned maintenance scenario appears. However, if you maintain a package, this is not the case, and you need to either publish your own fork or wait for the maintainer to publish a new version of the package. So, if you are creating an app, be wary; if you are creating a package, check twice before you add a package.

Conclusion

In this chapter, we reviewed popular packages in the Flutter ecosystem and look at some pros and cons of depending on these. Using packages well guarantees that you do not spend unnecessary time building parts of the app you don't need to. Additionally, it is quite helpful to know the latest packages being used, whether you are trying to build out an indie project, complete your job tasks, or applying for an interview.

In the next chapter, we will look at deploying the apps that you build with your newfound Flutter knowledge.

Questions

1. What are some packages that enhance network features in Flutter?
2. What is a good way to cache network images?
3. What are a few packages to get user location in Flutter?
4. What are the advantages and drawbacks of using packages?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 19
Deploying Applications

When all is said and done in terms of design, code, and execution, it is time to release the application created on the platforms it was designed for. While it is relatively easy for normal single-platform app development frameworks, Flutter is designed to release apps across multiple operating systems. For this reason, the deployment instructions are different for each platform since underneath, the app must be deployed how a normal application on the respective platform would. However, since the actual development of the application does not need much interaction with the underlying platform, this is a relatively minor speedbump, especially since the process for deploying to any platform is mostly a one-time learning experience and does not change much over time.

There are additional tools available for automating these releases, which helps avoid the manual tasks involved in the app deployment process.

Structure

In this chapter, we will discuss the following topics:

- Deploying to the Google Play Store
- Deploying to the Apple App Store
- Deploying to Web

- Deploying to macOS
- Deploying to Linux
- Deploying to Windows

Objectives

A Flutter developer's goal is to publish their application and make a difference in the world. After studying this chapter, you should know how to deploy a Flutter application across all currently supported platforms. Deploying an application is a relatively long process, and knowing all potential pitfalls can help you save time.

Versioning your application

Before you upload your application to any platform, the version number of the application needs to be verified. This helps platforms differentiate builds, and identify updates to an application and display it accordingly. The version consists of a version number and a build number separated by a plus sign, for example, **`1.0.0+1`**.

All platforms except Linux use the **`pubspec.yaml`** file to define the version number. As seen later in the deploying to Linux section, it uses the **`snapcraft.yaml`** file instead to set the version of the build.

The version and build number are automatically applied by Flutter when building the application to the respective aspects in the platform. For example, an Android application substitutes the **`versionName`** and **`versionCode`** properties. Both the version and number can also be overridden when building an application, using the **`-build-name`** and **`-build-number`** flags.

Deploying to the Google Play Store

The most popular place to deploy Android apps is the Google Play Store. When running apps normally on an emulator or device during development, Flutter builds a debug version of your application. Therefore, the debug build is usually quite

hefty since it carries all resources required without much optimization.

Releasing your app to any Android app store requires the release build of the application. Before you do, however, check the resources on the Android side that may need to be managed. As examples, check whether the Android app icon and splash screen is the one you want it to be.

You can check out the icons in the **`android/app/src/main/res`** folder:

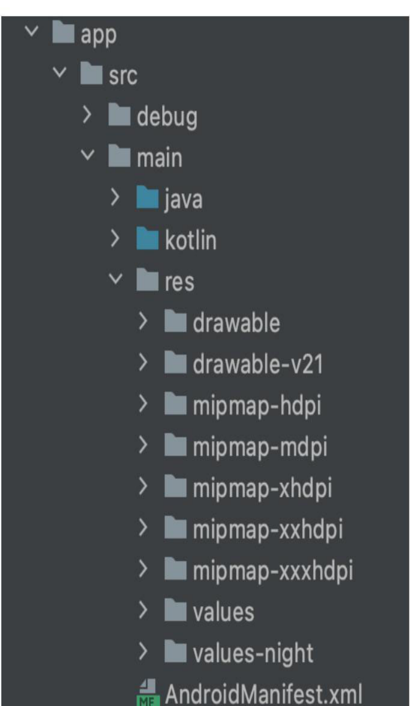


Figure 19.1*: Android app icons stored in the res folder*

You also need to check whether the app name is what you need the user to see in the **`AndroidManifest.xml`** file by checking the **`android:label`** attribute:

Code 19.1:

```
<application
    android:label="demo_starter"
    android:name="${applicationName}"
    android:icon="@mipmap/ic_launcher">
...
```

Next, the build can't be uploaded to the store without signing it first. For signing the build, we need to generate a keystore.

Run this command for Mac/Linux:

Run this command for Mac/Linux:

Code 19.2:

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

And this for Windows:

Code 19.3:

```
keytool -genkey -v -keystore c:\Users\USER_NAME\upload-keystore.jks
-storetype JKS -keyalg RSA -keysize 2048 -validity 10000 -alias upload
```

Next, the app needs the details for using this keystore. Create an **android/key.properties** file that contains the info about your keystore:

Code 19.4:

```
storePassword=<Your keystore password>
keyPassword=<Your key password>
keyAlias=upload
storeFile=<Location of keystore>
```

The **android/app/build.gradle** file is where the keystore info needs to be used. Read the properties file before the android block:

Code 19.5:

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

Next, add **signingConfigs** to the file to pass along the information contained in the file:

Code 19.6:

```
signingConfigs {
       release {
           keyAlias keystoreProperties['keyAlias']
           keyPassword keystoreProperties['keyPassword']
           storeFile keystoreProperties['storeFile'] ?
file(keystoreProperties['storeFile']) : null
           storePassword keystoreProperties['storePassword']
       }
   }
buildTypes {
       release {
```

```
            signingConfig signingConfigs.release
        }
    }
```

You can now build the app into either an APK or an **`appbundle`**. The APK is a fully built installable package file, while the **`appbundle`** defers to the Google Play Store for building the final release.

Use this command for building an APK:

Code 19.7:

```
flutter build apk --release
```

And this for building the **`appbundle`**:

Code 19.8:

```
flutter build appbundle
```

You can also split the APK across the **Application Binary Interfaces** (**ABI**) that it will be built for using:

Code 19.9:

```
flutter build apk --split-per-abi
```

The Google Play Store only accepts **`appbundles`**, while the APK can be uploaded on other app stores.

Deploying to the Apple App Store

Unlike Android, there aren't many alternatives to deploying to iOS; it only has the Apple App Store. The build tools and IDEs like XCode also exist only on Apple software, so creating an Apple build manually requires an Apple device capable of running Apple's build tools.

Apple is relatively straightforward compared to Android since there is no need to sign builds yourself (note that you can manually sign your builds, but XCode can automatically do this for you). However, building for iOS also requires an Apple developer account. Follow these steps to deploy to the Apple app store:

1. First, we need to register an App ID to identify the app itself:

Register a new identifier

App IDs
Register an App ID to enable your app, app extensions, or App Clip to access available services and identify your app in a provisioning profile. You can enable app services when you create an App ID or modify these settings later.

Figure 19.2: Registering a new App ID

2. After this, using the App ID, we can register the app:

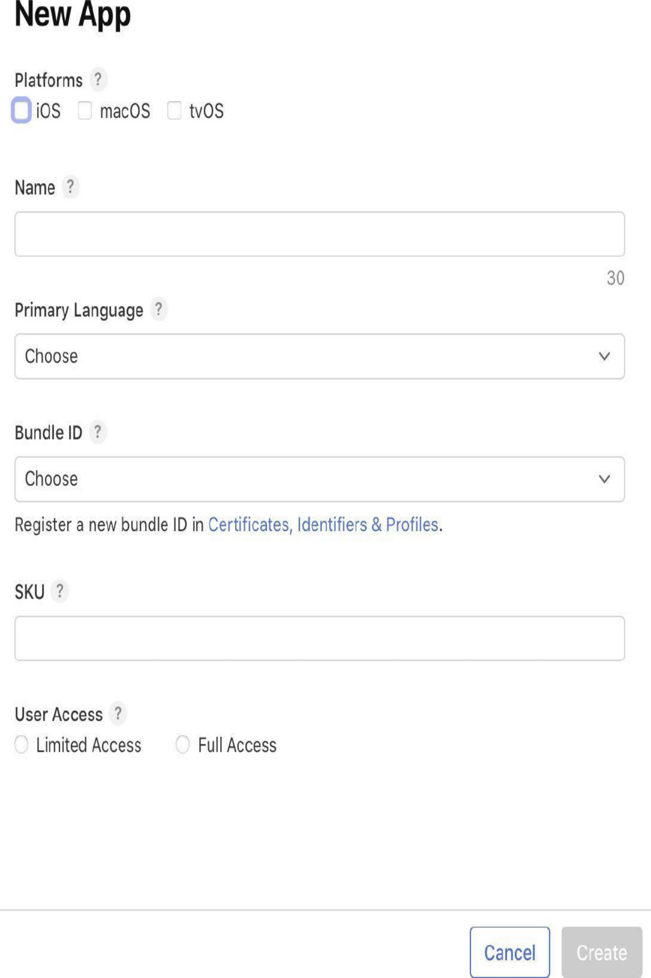


Figure 19.3: Submitting information about an iOS/macOS application

3. In the iOS project, the **`Info.plist`** file contains various kinds of data about the app:

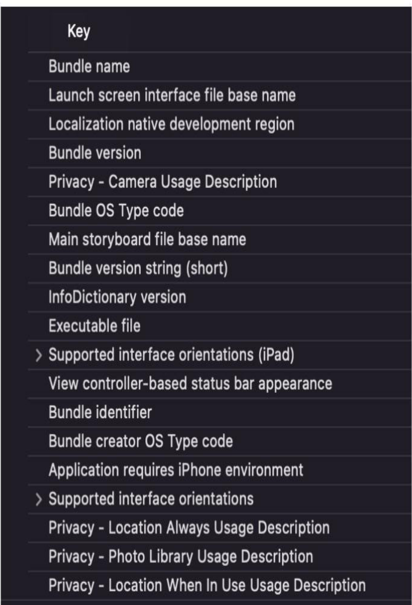


***Figure 19.4**: Looking into the Info.plist file*

4. The app icons are stored in the **`ios/Runner/Assets.xcassets`** folder, and you can modify icons through the same folder:

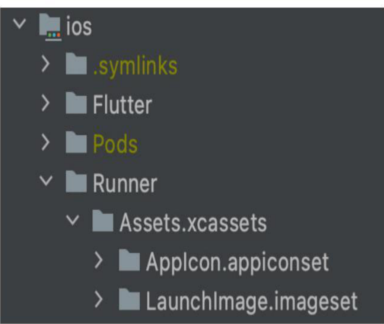


***Figure 19.5**: iOS app icon folder*

5. To upload the build onto the **`TestFlight`** / app store, run one of two commands:

   Code 19.10:

   ```
   flutter build ios
   ```

   Or:

   Code 19.11:

```
flutter build ipa
```


6. After running the first command, you need to create an **Archive**:

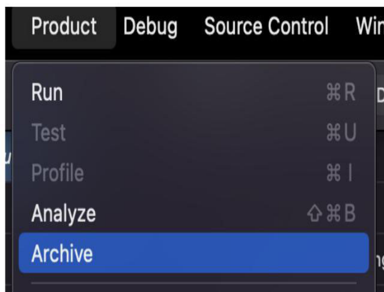


Figure 19.6: Creating an archive for the application

7. This creates a build that you can finally upload:

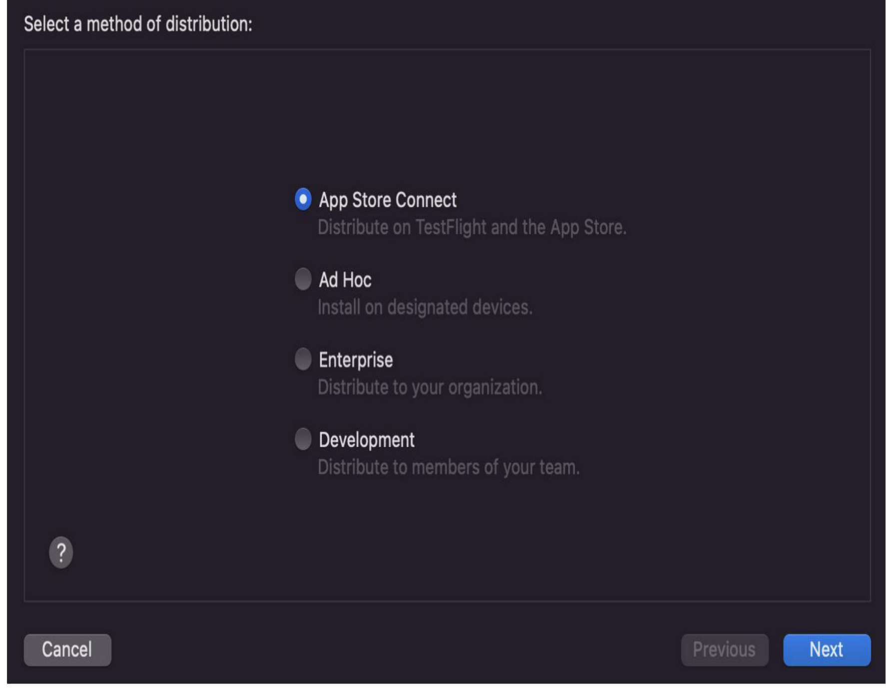


Figure 19.7: Uploading the archive to the app store for iOS

If you run the second command while building the iOS build, you need to validate the created archive and then distribute the application.

Deploying to Web

Web was the first platform after Android and iOS to be trialed as a target for Flutter apps. The main thing to know about Flutter Web is that it is designed to create Web

apps. The main thing to know about Flutter Web is that it is designed to create Web Apps, not static websites. This means that SEO and other concerns arise if it is taken out of its intended targets.

The next thing to consider with the Web is the renderer to use. By default, Flutter runs the **`HTML`** renderer on mobile and the **`CanvasKit`** renderer on desktop for web. The **`HTML`** renderer is smaller in size and closer to the normal web components than **`CanvasKit`**, but it does not necessarily maintain consistency with Flutter on mobile.

To build for the Web, we run the following:

Code 19.12:

```
flutter build web --release
```

This creates a build in the **`build/web`** folder:

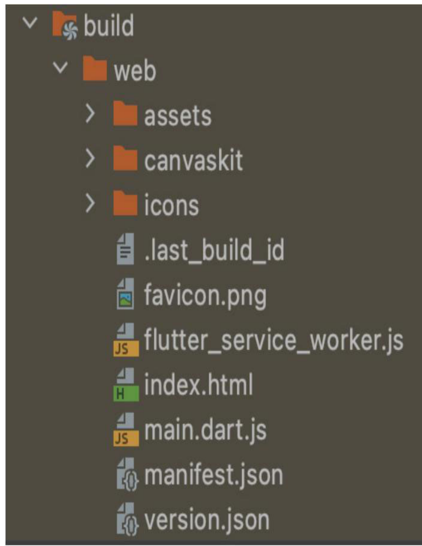


Figure 19.8: *The build folder for a Flutter Web project*

You can also specify the renderer to use with a flag:

Code 19.13:

```
flutter build web --web-renderer canvaskit
```

This build can now be uploaded to hosting services like Firebase.

Uploading to Firebase

In this section, we will look at hosting the Flutter web project that we just created onto Firebase web hosting.

Firebase is a Google **Backend-as-a-Service** (**BaaS**) solution and is a well-tested back-

end solution for Flutter offering all kinds of services from authentication to real-time NoSQL databases and web hosting. It also has Flutter packages/plugins for most, if not all, its services and integrates quite well into the Flutter fold.

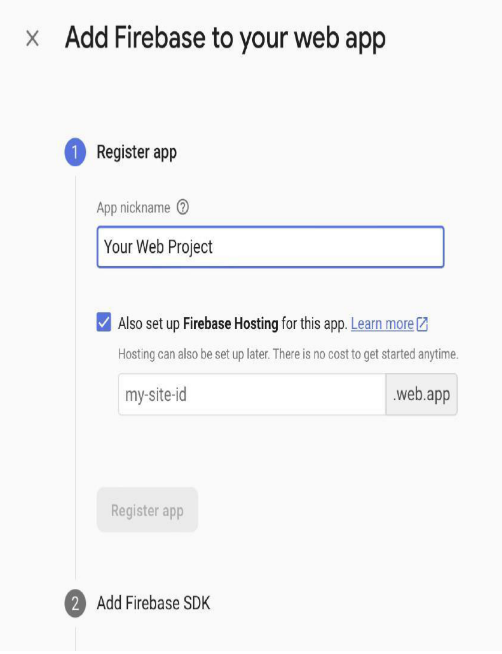
Step 1: Create a project

The first step is creating a Firebase project at **`console.firebase.google.com`**. This also allocates a Google Cloud bucket underneath.

Figure 19.9: Creating a new Firebase project

Step 2: Register a web project

The next step is to register a web project, which sets up the Firebase details for the project and initializes hosting for the project:



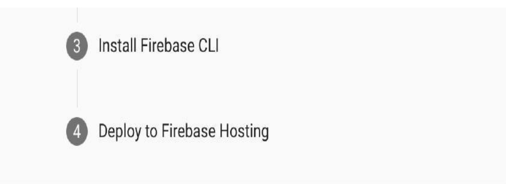


***Figure 19.10**: Registering a new web app for Firebase*

Step 3: Install Firebase CLI and initialize hosting

Next, we need to install the Firebase CLI through **`npm`**:

Code 19.14:

```
npm install -g firebase-tools
```

Then, we need to initialize a Firebase project in our main Flutter project using the following:

Code 19.15:

```
firebase init
```

This asks what we intend to set up; it's just hosting in this case. Other use cases also include setting up cloud functions for the project.

```
? Which Firebase features do you want to set up for this directory? Press Space to select features, then Enter to confirm your choices.
d <enter> to proceed)
)O Realtime Database: Configure a security rules file for Realtime Database and (optionally) provision default instance
 O Firestore: Configure security rules and indexes files for Firestore
 O Functions: Configure a Cloud Functions directory and its files
 O Hosting: Configure files for Firebase Hosting and (optionally) set up GitHub Action deploys
 O Hosting: Set up GitHub Action deploys
 O Storage: Configure a security rules file for Cloud Storage
 O Emulators: Set up local emulators for Firebase products
(Move up and down to reveal more choices)
```

***Figure 19.11**: Setting up Firebase features for the web project*

Step 4: Build and deploy

We can now build the app itself:

Code 19.16:

```
flutter build web -release
```

And then release it using the Firebase CLI:

Code 19.17:

```
firebase deploy -only hosting
```

This deploys the Flutter app to a predefined or new URL, which will be mentioned in the terminal once the deployment finishes.

Deploying to macOS

Coming to the first desktop platform for Flutter, the macOS deployment procedure is quite like iOS; both share several tools for development and builds.

Like iOS, a Bundle ID needs to be registered for macOS. After this, a new app needs to be registered on the app store. The procedure for this is like the iOS one, albeit with the macOS platform selected:

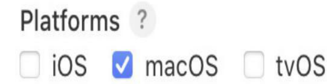


***Figure 19.12**: Selecting platform for deployment for Apple*

The icons and general app data lie in the same places: **`ios/Runner/Assets.xcassets`** and **`Info.plist`**, respectively.

To create the build for macOS, you can run the following:

Code 19.18:

```
flutter build macos
```

Once this is done, the archive created needs to be opened:

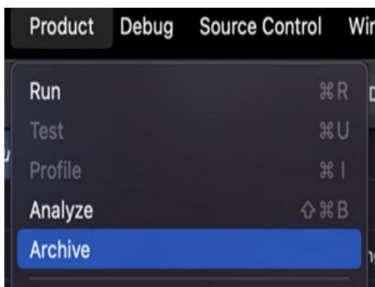


***Figure 19.13**: Creating the archive for the macOS application*

This can now be uploaded by clicking on **Distribute** app.

Deploying to Linux

Deploying to Linux

Linux has been a community heavily invested in Flutter development, including popular versions of the OS adopting it for various purposes. This section focuses on creating a Flutter build and uploading it to the Snap Store, the closest thing to the App / Play Store for Linux.

Initially, **`snapcraft`** and **`Multipass`** must be installed to create the build.

Snapcraft installation:

Code 19.19:

```
sudo snap install snapcraft -classic
```

Multipass installation:

Code 19.20:

```
sudo snap install multipass -classic
```

Next, add a snap directory inside the project with a **`snapcraft.yaml`** file. Here's an example from the Flutter documentation:

Code 19.21:

```
name: super-cool-app
version: 0.1.0
summary: Super Cool App
description: Super Cool App that does everything!

Confinement: strict
base: core18
grade: stable

slots:
  dbus-super-cool-app: # adjust accordingly to your app name
    interface: dbus
    bus: session
    name: org.bar.super_cool_app # adjust accordingly to your app name
and

apps:
  super-cool-app:
    command: super_cool_app
```

```
    command: super_cool_app
    extensions: [flutter-master] # Where "master" defines which Flutter channel to use for the build
    plugs:
    - network
    slots:
      - dbus-super-cool-app
parts:
```



```
  super-cool-app:
    source: .
    plugin: flutter
    flutter-target: lib/main.dart # The main entry-point file of the application
```

App name and icons can be specified via a **.desktop** file at **snap/gui/your-app.desktop**:

Code 19.22:

```
[Desktop Entry]
Name=Super Cool App
Comment=Super Cool App that does everything
Exec=super-cool-app
Icon=${SNAP}/meta/gui/super-cool-app.png # replace name to your app name
Terminal=false
Type=Application
Categories=Education; #adjust accordingly your snap category
```

To build the **snapcraft**, we can run the **snapcraft** command:

Code 19.23:

```
snapcraft
```

After this, you can upload it to the store via the following:

Code 19.24:

```
snapcraft upload –release=<channel> <file>.snap
```

This deploys a full Linux application to the Snap store.

Deploying to Windows

To deploy an app to Windows, you first need to join the Microsoft Partner Network that designates you as a developer.

The most basic way to build a Windows app is as follows:

Code 19.25:

```
flutter build windows
```


This builds a **.exe** file that can be run directly. This cannot be uploaded to the Microsoft store; the most popular format for this is MSIX.

The easiest way of doing this is the **msix** plugin on **pub.dev**. Let's add this to **dev_dependencies**:

Code 19.26:

```
dev_dependencies:
  msix: ^2.6.5
```

This allows adding configuration to the usual **pubspec.yaml** file:

Code 19.27:

```
msix_config:
  display_name: Flutter App
  publisher_display_name: Company Name
  identity_name: company.suite.flutterapp
  msix_version: 1.0.0.0
  logo_path: C:\path\to\logo.png
  capabilities: internetClient, location, microphone, webcam
```

The build can be created with the following command:

Code 19.28:

```
flutter pub run msix:create
```

After signing, this build can be uploaded to the store.

Conclusion

Deploying apps is the final step in the app development cycle, and probably also

the most enjoyable one since the work you've put into development materializes into a full-blown product. Knowing all the aspects of deployment well can help you get new releases out quickly and without hassle, it can help app maintenance be a breeze.

After knowing all these Flutter development aspects, from basic UI building to networking and databases, navigation and state management, testing, and finally deployment, you're on your way to become a full-time Flutter developer!

Questions

1. Why does every platform have a different way of deployment for an app?
2. What are the respective application stores for each platform?
3. Where can you define version numbers for a Flutter app?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Index

Symbols

?? operator 55, 56
??= operator 57, 58
?. operator 56, 57
~/ operator 60

A

Ahead-Of-Time (AOT) 86
Android Runtime (ART) 3
Android Studio
  reference link 94
  setting up 94-97
Android Studio extensions 101
AngularDart 4
AnimatedBuilder 248, 249
AnimatedCrossFade 257, 258
AnimatedOpacity 255
AnimatedPositioned 256, 257
animated_text_kit
  about 315
  fade 316
  rotate 315
  scale 316
  typewriter 317
animation
  about 234-241
  building blocks 235
  creating, from scratch 241-247
AnimationController 237-239
API calls
  UI build, problem from data 192
AppBar
  about 164, 165
  CenterTitle 166
  color 165
  leading and actions 165, 166

size 165
title 165
AppDelegate 121
application
versioning 328
App state 129
asynchronous programming
need for 64
async structure 65
auto_size_text 320, 321
await structure 65
Awesome Flutter Snippets 99

B

Backend-as-a-Service (BaaS) 335
bloc package 230-232
Boolean 15, 16
BottomNavigationBar 168
bottom property
about 168
body 169, 170
BottomNavigationBar 168
buttons
about 149
ElevatedButton 150
OutlinedButton 150, 151
properties 151
TextButton 149

C

cached_network_image 317, 318
cascade notation 58, 59
chewie package 318, 319
code generation 191
column 153-156
composition 125
conditionals
about 28
else block 29
if statement 29
switch statement 31, 32
ternary operator 30, 31
connectivity_plus plugin
about 311
cellular network, checking 312
connection status, checking 311
connection status, listening 312
wifi network, checking 312
constructors
in Dart 48
Container widget
about 159
alignment and padding 160
decoration 161, 162
size and color, adding 159
Counter app 174-176
cross-platform
versus hybrid 77, 78
CRUD operation
about 205
create 205
delete 206
read 206
update 206
Cupertino design 135-137

D

dark mode
adding 216, 217
Dart
about 3, 4

constructors 48
function types, defining 46
history 2
learning 8, 9
working, in Flutter 5, 6
Dart 1.0 version 7
Dart 2.0 version 7
Dart 3.0 version 8
Dart classes 48
Dart function
structure 42, 43
Dart language
Dart 1.0 version 7
Dart 2.0 version 7
Dart 2.12 version 7, 8
Dart 3.0 version 8
evoluting 6
Dart program
creating 6
data models
about 185
code generation 191
creating 189
QuickType 189, 190
data types
about 12, 198
app feed 199, 200
settings/preferences 199
dio
about 304
features 305, 306
double 14
do while loop 37

E

Element 140, 141, 265
ElevatedButton 150
else block 29
Ephemeral state 128
extension methods 47

F

fat arrow (=>) operator 59
fetch function
modifying 188, 189
file_picker
about 308
file types, picking 309
multiple files, picking 309
single file, picking 308
finders
about 289
find.ancestor() 292, 293
find.byKey() 291
find.byType() 289, 290
find.descendant() 292
find.text() 290, 291
flame 321-323
Flutter
about 82
documenting 86, 87
features 86
installing 90
installing, for Linux 93, 94
installing, for macOS 92, 93
installing, for Windows 90-92
layers 132, 133
overview 83
tests 116

trees 263, 264
used, for solving React Native issues 84-86
Flutter animation framework 235
Flutter application
deploying, to Apple App Store 331-334
deploying, to Google Play Store 328-331
deploying, to Linux 338-340
deploying, to macOS 337, 338
deploying, to Web 334, 335
deploying, to Windows 340, 341
uploading, to Firebase 335-337
Flutter approach 263
Flutter as UI toolkit 262, 263
flutter_bloc package 230-232
Flutter extensions
about 98
Android Studio extensions 101
Awesome Flutter Snippets 99
Flutter Pub Version Checker 101, 102
Flutter Snippets 102, 103
Pubspec Assist 100
Rainbow Brackets 100-104
VS Code extensions 99
Flutter layouts
approaching 144
Flutter project
about 106, 107, 118
Android manifest 118, 119
AppDelegate 121
build.gradle file 119, 120
info.plist 121
Flutter Pub Version Checker 101, 102
Flutter Snippets 102, 103
forEach() function 17
Foreign Function Interface (FFI) 7
for in loop 36
for loop 35
framework 74, 75
function
about 42, 65, 66
as first-class objects 45, 46
delayed futures 68
FutureBuilder 68
handling 66-68
function types
defining, in Dart 46

G
golden testing 280, 300, 301
google_maps_flutter 314, 315

H
Hive
about 206
adding 207
data storage 207, 208
objects, storing 208, 209
HTTP packages 182-184
hybrid
versus cross-platform 77, 78
hybrid frameworks 78
HyperText Transfer Protocol (HTTP)
about 182
methods 182

I

icon
  about 156, 157
  customizing 157
IDEs for Flutter development 94
if statement 29
image_picker
  about 309
  geolocator 310
  image, capturing 310
  last known location, obtaining 310
  location, listening 311
  location, obtaining 310
  multiple image, picking 309
  single image, picking 309
  video, capturing 310
implicit animations 252-254
implicitly animated widgets
  about 254
  AnimatedCrossFade 257, 258
  AnimatedOpacity 255
  AnimatedPositioned 256, 257
inheritance
  about 50
  basic 50
InheritedWidget 224
integer 13, 14
integration testing 280, 298, 299
interface 50, 51
internet connection 180
isolate 70, 71
iterable specific loop 38

J

JavaScript Object Notation (JSON) 185
Just-In-Time (JIT) 86

L

late variables 23
LICENSE 118
linear interpolation (Lerp) 236, 237
lists
  about 16
  forEach() function 17
  map() function 17, 18
ListView widget
  about 171, 172
  scroll direction 172, 173
  scroll physics 173
logic files 108
loop
  about 33, 34
  do while loop 37
  for in loop 36
  for loop 35, 36
  iterable specific loop 38, 39
  overview 39
  while loop 34

M

map() function 17, 18
maps 19
Material design 134, 135
maxLines property 148
mixins 50-52
mobile application 76, 77
multi-platform app 200, 201

N

named constructors 49
named parameters 44
Native Development Kit (NDK) 6
navigation 217, 218
Navigator methods
  about 218, 219
  pop page 219
  pop until 220
  push page 219
Netscape 75
network data
  UI, building 193-196
nullable variable
  converting, to non-nullable variable 23, 24
null-aware cascade operator 61, 62
null-aware index operator 61
null safety 21-23
numbers
  about 12
  double 14
  integer 13, 14
  numerical values, defining 13

O

Opacity Widget 267-270
optional parameters
  about 43
  default values, passing 45
  named parameters 44
  positional parameters 43, 44
OutlinedButton 150, 151
overflow property 148

P

package
  drawbacks 324
  importing 117, 118
padding 157-159
performance differences 131, 132
pop page 219
pop until 220
positional parameters 43, 44
private variables 21
properties, button
  button shape 152
  color 151
Provider 225-227
pub.dev 111
public variables 21
Pubspec Assist 100
pubspec.yaml file 109

Q

QuickType 189, 190

R

Rainbow Brackets 100-104
React Native
  about 79, 80
  working 81
README.md 118
RenderObjects
  about 138-140, 264
  types 264
RenderObjects classes
  RenderBox 264
  RenderProxyBox 265

RenderShiftedBox 265
RenderObjectWidget
about 266
LeafRenderObjectWidget 266
MultiChildRenderObjectWidget 267
SingleChildRenderObjectWidget 267
Riverpod 227-229
row 153-156

S

Scaffold widget
about 166, 167
AppBar, defining 167
FloatingActionButton (FAB) 167, 168
sensors_plus
about 312
accelerometer events, listening 312
gyroscope events, listening 313
magnetometer events, listening 313
user accelerometer events, listening 313
sets 18
setState() 126, 127
setState() function
problems 221-223
setUp() function
exploring 281
SharedPreferences
about 201
adding to 201
create 202
delete 203
read 202, 203
update 203
using 202
single-subscription stream 70
spread operator 60
sqflite 203
sqflite plugin
adding 204
CRUD operation 205
database, creating 204, 205
Stack
about 162
implementing 162, 163
positioned 163, 164
state 127-129
stateful widget
about 130, 131
rules 129
versus stateless widget 127
stateless widget
about 129, 130
rules 129
versus stateful widget 127
state management 221
state, types
App state 129
Ephemeral state 128
stream
about 69, 70
broadcast 70
single-subscription stream 70
strings 14, 15
switch statement 31, 32

T

tearDown() function
  exploring 281
ternary operator 30, 31, 54, 55
testing types
  about 280
  golden testing 280
  integration testing 280
  unit testing 280
  widget testing 280
tests
  setting up 281
test timeouts
  adding 284, 285
test variants
  exploring 282-284
TextButton 149
TextStyle
  about 145
  customization 146, 147
  font family, modifying 145, 146
  font size, modifying 145, 146
  font weight, modifying 145, 146
  text color 146
Text widget
  about 145, 270-276
  maxLines property 148
  overflow property 148
  TextStyle 145
themes
  adding 213-216
  creating 213-216
theming
  adding 212, 213
The Movies DataBase (TMDB) 181
  about 180
  reference link 180
trees
  types 133
Tween 235, 236
TweenAnimationBuilder 249- 252
type inference 20, 21

U

unit testing 280, 285, 286
url_launcher package
  about 306
  file_picker 308
  mail 307
  phone 307
  sms 308
  web link 307
user interface (UI)
  about 108
  assets, adding 115, 116
  building, from network data 193-196
  build problem, from data in API calls 192
  dependencies, adding 111
  dependency overrides 114
  developer dependencies 115
  Flutter package repository 111, 112
  lib folder 108
  metadata project, setting 110
  package, fetching 113
  package, versus plugin 113

publish_to command 116
pubspec.yaml file 109
versioning and environment 110
versioning dependencies 113, 114
YAML file 109

V

virtual machine (VM) 5
VS Code
reference link 97
setting up 97, 98
VS Code extensions 99

W

while loop 34
widget 124, 267
WidgetsApp 137
Widgets layer 138
WidgetTester
about 294
environment interaction 297, 298
pump() 294-296
pumpAndSettle() 297
pumpWidget() 294
widget testing
about 280, 287
creating 288, 289
finders 289
widget types
column widget 125
text widget 125

Y

Yet Another Markup Language (YAML) 109

Building Cross-Platform Apps with Flutter and Dart

DESCRIPTION

Flutter and Dart have emerged as a powerful duo that empowers developers to create stunning and feature-rich apps for Android, iOS, and web platforms from a single codebase. By leveraging Flutter's rich set of customizable widgets and Dart's reactive programming model, you can create visually appealing and interactive user interfaces that feel native on both iOS and Android devices.

This book adopts a hands-on approach to help you progress from fundamental to advanced concepts in Flutter development, establishing a solid foundation along the way. It will teach you how to create elegant user interfaces, utilize Flutter's Widget library, and incorporate captivating animations for enhanced user experience. It will also guide you through building apps that work seamlessly on all supported Flutter platforms, saving you time and effort. Additionally, you'll explore state management techniques for efficient app state handling and scalable applications. Following that, the book explores the process of connecting REST APIs and seamlessly integrating Firebase into your Flutter applications. It also includes testing and debugging techniques to ensure code quality and reliability. Lastly, it will guide publishing and distributing your app, covering code signing, release management, and app distribution to app stores.

By the end of this book, you will have the confidence and expertise to develop cross-platform apps efficiently.

KEY FEATURES

- Design visually striking UI with engaging animations using the Dynamic UI Capabilities of Flutter.
- Understand why Dart is an ideal choice for cross-platform app development.
- Learn how to secure, test, deploy and publish your Flutter apps.

WHAT YOU WILL LEARN

- Get familiar with different features of the Dart programming language.
- Learn how to leverage the vast collection of pre-built widgets provided by Flutter.
- Get tips to enhance the security of your Flutter app.
- Learn how to implement basic and advanced animations in Flutter.
- Explore various state management techniques in Flutter.

WHO THIS BOOK IS FOR

This book is for beginner and experienced developers who aspire to master Flutter and advance their cross-platform app development skills. It is also for mobile app developers who wish to harness Flutter's capabilities for creating apps across multiple platforms. Additionally, web developers interested in transitioning into mobile app development using the Flutter framework will find valuable insights.

BPB PUBLICATIONS

www.bpbonline.com