

Flutter —for— **Jobseekers**

Learn Flutter and take your cross-platform app development skills to the next level



Hans Kokx





Flutter for Jobseekers

Learn Flutter and take your cross-platform app development skills to the next level



Hans Kokx



Flutter for Jobseekers

Learn Flutter and take your cross-platform app development skills to the next level

Hans Kokx



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 978-93-55512-611

www.bpbonline.com

Dedicated to

*My beloved daughter,
Devon “Derren” Dea*

About the Author

Hans Kokx is a seasoned expert in the field of mobile application development, computer networking, and computer systems security. Driven by a passion for exploring the intersection of app development and computer security, he has become a trusted advisor in the industry.

Mr. Kokx holds degrees in Computer Systems Security and Computer Networking from Washtenaw Community College, where his research focused on the interconnection of computer systems and the security of the data transmitted between them. His work as a network security engineer paved the way for him in software development.

Throughout his career, Mr. Kokx has worked with companies and consumers alike to discover how people use (and often misuse) software. Driven by a passion for better experiences for users, he transitioned his career into software quality assurance, where he learned the economics of building software. His expertise in building high-quality applications that align business objectives with the ever-evolving digital landscape is informed by his time understanding users and helping to deliver bug-free features.

Mr. Kokx is the founder and an active member of the online Flutter community on the Matrix chat protocol, where he contributes to the growth, development, and understanding of Flutter developers' knowledge of industry standards and best practices. He also volunteers his time as a mentor for aspiring developers, guiding them through the nuances of building applications and troubleshooting any issues which may arise.

In his spare time, Mr. Kokx enjoys traveling and immersing himself in different cultures, always seeking inspiration for new dishes to cook at

home. He is also an avid photographer, with some of his works being published in calendars, newspapers, and across the world.

About the Reviewer

Monty Rasmussen has worked in a number of software industries, including education, process simulation, web design, real-time utility monitoring, medical software, and even gaming. He's a former officer of the Google Developer Group Salt Lake, but left to become a Google Developers Expert (GDE) for Angular, Dart, and Flutter. He has tech articles published on SitePoint and Dart Academy (<https://dart.academy>).

When Google announced Dart in 2011, it was love at first sight. Monty followed its development all the way to its 1.0 release on November 14, 2013 (his birthday, coincidentally), and he's been optimistic about the web's prospects as a serious application platform ever since. Now, he loves using frameworks that allow him to publish to multiple platforms from a single code base, like Flutter.

Acknowledgements

When I embarked on the journey of writing this book, I had no concept of the endless supply of support I would be shown. I would like to extend a huge shout-out to the baristas and patrons at Black Diesel, the coffee shop where much of this book was written, as without the supportive environment they fostered, I would have had a much more difficult time writing.

To my coworkers and colleagues who cheered me on, I am immensely grateful. Ahmet Akıl, Nate Noye, Adam Kunesh, Chris Nedlik, Austin Bell – thank you for pushing me ever onwards and upwards. I would also like to thank Monty Rasmussen, Randal Schwartz, Hank Grabowski, Sandro Lovnički, and Zachary Smith for their brilliant insights and informative conversations. I could not have completed this book without the exceptional support of Haley Dean, Ryan Francis, and Turner Kallen.

A very special thanks goes to Katherine Wiykovics for helping me “beta-test” the book, helping to ensure that there was a method to my madness. Her questions and insights were instrumental in helping me to fill in any missing blanks I would have otherwise glossed over.

I would also like to thank those at BPB for their patience and support as I wrote, rewrote, and edited each chapter. Without them, I wouldn’t have had the opportunity of a lifetime to bring this book to life.

And finally, I want to thank each and every reader for their support in picking up this book. Without you, there simply wouldn’t be a point to all of this.

Preface

In today's rapidly evolving job market, the ability to adapt, learn, and master new technologies is crucial for jobseekers aiming to stand out from the crowd. Among the myriad technologies that have emerged in recent years, one has gained remarkable popularity and revolutionized the way we build mobile applications - Flutter.

Welcome to "Flutter for Jobseekers," a comprehensive guide designed to equip you with the skills and knowledge needed to leverage Flutter in your job search journey. Whether you are a seasoned professional exploring new avenues or a fresh graduate embarking on your career, this book is tailored to empower you with the tools necessary to succeed in the job market using Flutter.

Flutter, developed by Google, has emerged as a powerful and versatile framework for building beautiful, fast, and cross-platform applications. Its unique approach of using a single codebase for multiple platforms has made it a favorite among developers and organizations alike. With Flutter, you can create stunning user interfaces, implement complex functionality, and deploy your apps to both Android and iOS platforms seamlessly.

In "Flutter for Jobseekers," we recognize the importance of mastering this game-changing technology as part of your job search strategy. We will take you on a journey through the core concepts of Flutter, starting from the fundamentals and gradually building up to advanced topics. You'll learn how to set up your development environment, understand Flutter's widget system, handle user input, navigate between screens, and utilize various plugins and packages to enhance your app's capabilities.

Furthermore, this book goes beyond just technical aspects. We understand that finding a job involves more than writing code. Throughout these pages,

we will also provide valuable insights into how Flutter fits into the job market, the demand for Flutter developers, and the specific skills and experiences that employers seek. We will guide you on how to showcase your Flutter expertise effectively in interviews, build a portfolio that impresses recruiters, and navigate the job application process successfully.

Whether you are aspiring to be a Flutter developer, looking to switch careers, or simply curious about this exciting technology, “Flutter for Jobseekers” will serve as your reliable companion. We have structured this book with clarity and conciseness, offering step-by-step explanations, practical examples, and real-world scenarios that will accelerate your learning and give you the confidence to apply your Flutter skills in real job situations.

Remember, jobseekers who adapt to new technologies and embrace innovation are more likely to excel in their careers. Flutter has rapidly gained momentum in the app development industry, and by mastering it, you position yourself as a highly desirable candidate in the job market.

Let this book be your guide to mastering Flutter for jobseekers. Embrace the opportunities it presents, invest in your skills, and unlock a world of possibilities in your professional journey. Good luck!

Chapter 1: Introduction to Flutter – Learn about the history of application development, with a focus on mobile applications. Find out what Flutter is, including who developed it and why, and the history of the development of Flutter.

Chapter 2: Market Opportunities for Flutter Developers – Dive into the explosive growth of Flutter’s marketplace dominance and see how it compares to its competitors. Then, learn who is using Flutter and what they’re using it for.

Chapter 3: Installing Flutter and Configuring Your IDE – Enjoy a comprehensive, step-by-step guide to setting up a development environment on your Mac or Windows computer. This chapter will introduce you to the tools you will be using, including where to download them, how to install them, and how to configure them for building applications with Flutter.

Chapter 4: Introduction to Widgets – In the world of Flutter, we like to joke that everything is a widget! Here, you’ll learn exactly what a widget is

and learn the difference between the different types of widgets. You'll also learn how Flutter takes the widgets you write and draws them to the screen.

Chapter 5: Handling User Input – An application wouldn't be an application if it didn't accept some sort of user input. Learn all about buttons, text input, checkboxes, and much more. Discover how to validate input and even build a simple signup form.

Chapter 6: Using 3rd Party Libraries and External Assets – In this chapter, you'll be introduced to Flutter's package management system, Pub. You'll learn how to include images and other assets in your application, and how to benefit from other developers' work by leveraging third-party packages to do amazing things.

Chapter 7: Working with APIs and Asynchronous Operations – Learn the basic concepts of asynchronous operations, such as reading and writing files to disk. Send data to APIs and learn how to work with the data they respond with, then discover data streams and the widgets we can use to handle the UI-portion of these operations.

Chapter 8: Navigation and Routing – Build a vocabulary to discuss the concepts of moving from screen to screen within an application. Then, explore a multitude of methods to accomplish it – from the most basic of basic approaches, to the most advanced methods used by some of the biggest applications around.

Chapter 9: State Management and the BLoC – Learn the difference between ephemeral state and persistent application state. Then, learn about some of the widgets and packages you can use to manage state of your application.

Chapter 10: Reactivity and Platform-Specific Considerations – Not all devices are made equal – and your apps will need to conform to each of them. Learn how to build responsive layouts, explore the power of dart:io to work with files and the filesystem, give your app an icon and splash screen, then discover how to make your app match the look and feel of some of the major operating systems today.

Chapter 11: Debugging, Troubleshooting, and Performance Considerations – Nobody gets it right the first time. That's why this chapter focuses on learning about the tools available to you for debugging

when things go wrong. Learn the basic concepts of debugging and troubleshooting, as well as how to use your development tools to find and fix problems.

Chapter 12: Creating Your First Application – It's time to build your first application! Learn about the application you'll be building, discover resources for bringing it to life, and learn where to put the files. You'll also learn about code generation in this chapter.

Chapter 13: Finding Flutter Jobs – Learn how to find a job with your newly discovered superpower. Then, all your questions about finding a job, getting hired, and building your resume will be answered by industry professionals.

Chapter 14: Preparing for and Succeeding in the Job Interview – Every step of the interview process is broken down for you in this chapter. Learn exactly what to expect from each stage, and how to negotiate your compensation, and then we answer nearly 150 interview questions for you to study from!

Chapter 15: Your Road Ahead – The final chapter of this book focuses on the period immediately following accepting your job offer, all the way through your first days. There's advice about working in software development teams, as well as additional skills you'll need to seek out on your own to further your career.

Code Bundle and Coloured Images

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

<https://rebrand.ly/26zpdB1>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Flutter-for-Jobseekers>. 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. Introduction to Flutter

Introduction

Structure

Objectives

An Abridged History of Apps

The History of Multi-Platform App Development

A New Multi-Platform App Development Solution

The Rapid Development of Flutter

Why Developers Are Flocking to Flutter

Looking Forward

Conclusion

References

2. Market Opportunities for Flutter Developers

Introduction

Structure

Objectives

The Growth of Flutter in the Marketplace

Who is Using Flutter, and What are They Using it for?

Case Study: Alibaba

Case Study: BMW

Case Study: ByteDance

[Case Study: eBay](#)

[Case Study: Toyota](#)

[Case Study: Google](#)

[Conclusion](#)

[References](#)

3. Installing Flutter and Configuring Your IDE

[Introduction](#)

[Structure](#)

[Objectives](#)

[Tools](#)

[Getting Started on Windows](#)

[Getting Started on macOS](#)

[Additional Information for Devices with Apple Silicon \(M1, M2, and so on\)](#)

[Installing and Configuring Visual Studio Code](#)

[Building and Running Your First Flutter Application](#)

[Conclusion](#)

[Further Reading](#)

4. Introduction to Widgets

[Introduction](#)

[Structure](#)

[Objectives](#)

[Briefly Exploring the Default Flutter Application](#)

[What is a Widget and How is it Drawn to the Screen?](#)

[Stateless Widgets](#)

[Creating a Custom Stateless Widget](#)

[Stateful Widgets](#)

[Inherited Widgets](#)

[Introduction to BuildContext](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

[Further Reading](#)

[5. Handling User Input](#)

[Introduction](#)

[Structure](#)

[Objectives](#)

[Buttons](#)

[Capturing Text Input](#)

[Pickers, Selectors, and Dropdowns](#)

[Radios, Checkboxes, Switches, and Segmented Controls](#)

[Sliders](#)

[Forms and FormFields](#)

[Building a Simple Signup Form](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

[Further Reading](#)

[6. Using 3rd Party Libraries and External Assets](#)

[Introduction](#)

[Structure](#)

[Objectives](#)

[Introduction to pubspec.yaml](#)

[Adding Images and Other Assets to Your Application](#)

[Introduction to the Pub Package Management System](#)

[Finding Useful Packages on Pub](#)

[Using a Package to Play Audio](#)

[Popular Pub packages](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

[Command Reference](#)

[Further Reading](#)

7. Working with APIs and Asynchronous Operations

[Introduction](#)

[Structure](#)

[Objectives](#)

[Welcome to the Future](#)

[Requesting Data From an API](#)

[Classifying API Responses](#)

[Sending Data to an API](#)

[Flutter's Asynchronous Widgets](#)

[Sinks and Streams](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

[Further Reading](#)

8. Navigation and Routing

[Introduction](#)

[Structure](#)

[Objectives](#)

General Discussion of Navigation and Routing Concepts

Introduction to Navigator 1.0

Routing Using Named Routes

Passing Arguments into a Route

Returning Data from a Screen

Animating Widget from One Route to Another

Introduction to Navigator 2.0 — the Router API

Using Packages to Help with Routing

Conclusion

Questions

Key Terms

Further Reading

9. State Management and the BLoC

Introduction

Structure

Objectives

Ephemeral Versus App State

Some Prep Work Before We Begin

A Brief Aside to Explain Some New Concepts

Using Inherited Widgets to Manage App State

Using Provider to Manage App State

Using BLoC's Cubit to Manage App State

Using BLoC to Manage App State

Conclusion

Questions

Key Terms

Further Reading

10. Reactivity and Platform-Specific Considerations

[Introduction](#)

[Structure](#)

[Objectives](#)

[Building Responsive Layouts](#)

[Benefits and Limitations of dart:io](#)

[App Icons and Splash Screens](#)

[Material, Cupertino, Yaru, and Fluent UI \(oh my!\)](#)

[Platform-Specific Navigation Paradigms](#)

[Using Packages to Build Cross-Platform UIs](#)

[Conclusion](#)

[Questions](#)

[Further Reading](#)

11. Debugging, Troubleshooting, and Performance Considerations

[Introduction](#)

[Structure](#)

[Objectives](#)

[Debugging and Troubleshooting Concepts](#)

[Logging Output to the Debug Console](#)

[Setting Breakpoints and Exploring Runtime Variables](#)

[Handling Exceptions](#)

[Introduction to Flutter's DevTools](#)

[Defining Performance Issues](#)

[Exploring Profile Mode](#)

[When a ListView is not a ListView](#)

[Optimizing Widgets](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

12. Creating Your First Application

[Introduction](#)

[Structure](#)

[Objectives](#)

[About the Application](#)

[Understanding the Technical Requirements](#)

[Building the Skeleton of Your Application](#)

[The WeatherIcon Class](#)

[Code Generation](#)

[Building the API Layer](#)

[Finishing the application](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

[Command Reference](#)

[Further Reading](#)

[References](#)

13. Finding Flutter Jobs

[Introduction](#)

[Structure](#)

[Objectives](#)

[How to Hack the System and Make the Jobs Come to You](#)

[Question and Answer Time with Top Flutter Recruiters](#)

[Conclusion](#)

14. Preparing for and Succeeding in the Job Interview

Introduction

Structure

Objectives

The Interview Process, Start to Finish

What to Expect During a Technical Interview

The Culture Fit Interview: Evaluating a Company and Being Evaluated

Negotiating Compensation

Answering Some Interview Questions

Conclusion

15. Your Road Ahead

Introduction

Structure

Objective

How to Prepare for Your New Job

Tips for Collaboration with Other Developers

Additional Skills You'll Need to Seek Out

Suggestions for Furthering Your Career

Conclusion

Flutter Community Links

Index

Chapter 1

Introduction to Flutter

Introduction

Flutter is a portable UI toolkit created by Google and released as open-source to the community in 2015. Since its initial unveiling under the working name *Sky* by Google developer *Eric Seidel* at the 2015 Dart Developer Summit, Flutter has seen extraordinary growth — both in the toolkit itself and within the developer community.

The transformative nature of application development with Flutter has been the biggest catalyst for its explosive growth since its inception. With features such as stateful hot reload, a robust set of default UI elements, developer-friendly tooling, and the ability to run on a large variety of platforms natively with minimal code modification, it is no surprise that developers and companies alike have fallen in love with Flutter.

Structure

In this chapter, we will discuss the following topics:

- An abridged history of apps
- The history of multi-platform app development
- A new multi-platform app development solution
- The rapid development of Flutter

- Why developers are flocking to Flutter
- Looking forward

Objectives

This book aims to familiarize readers with the evolution of mobile applications, highlighting the industry's journey towards user-friendly frameworks like Flutter. It explores Flutter's significance and diverse applications, revealing surprising use cases across industries. Aspiring Flutter developers will gain valuable insights into career opportunities and be inspired to join the enthusiastic and supportive Flutter community.

An Abridged History of Apps

In March of 1996, a then-little-known company named *Palm* released their first personal digital assistant: the Pilot 1000. There were several failed attempts at a pocketable, personal, app-centric digital device prior to the Palm Pilot, but it was truly the Pilot which ignited what would eventually dominate the global smartphone market.

By 2007, with Apple's release of the first iPhone, followed by the first release of Android in 2008, smartphones were beginning to take shape. The hardware was moving toward multi-touch paradigms (thanks to *Steve Jobs'* forward-thinking approach). The days of a dedicated, physical keyboard, made popular by the Blackberry, were numbered.

With both iOS (then iPhone OS) and Android came modern apps, which Palm and, later, Blackberry helped lay the groundwork for. Apple's and Google's approaches to application development differed, however one common thread remained consistent: users' insatiable demand for apps.

As Apple and Google continue to battle for market share, apps are key in the fight to win over users. As of 2021, there were a reported 2.2 million apps in Apple's App Store and around 3.5 million in Google's Play Store (Ceci, L., 2021). An entire industry has been built around delivering apps of all types to users, regardless of their platform of choice. This has led to massive duplication of effort, as applications once written solely for iOS are then ported to Android, and vice versa.

On iOS, Swift superseded Objective-C as the language of choice in 2014. Likewise, in 2019, Google announced Java (superseded by Kotlin) as the less-preferred language for Android app development in 2019. Regardless of the platform or language, one thing was clear: targeting both iOS and Android would require your app to be written once for Android and then again for iOS, leaving you with two separate codebases to maintain.

The History of Multi-Platform App Development

Due to the cost and hassle associated with maintaining separate iOS and Android codebases, many solutions were developed with varying amount of success to bridge the divide and unify multi-platform app development into a single codebase with a single development team.

Arguably, one of the earliest such attempts was by *Appcelerator, Inc.*, with their release of the *Titanium SDK* in December 2008 as an open-source framework for creating multi-platform applications using JavaScript. Similarly, *Nitobi* released *PhoneGap* in 2009, which focused on multi-platform app development using CSS3, HTML5, and JavaScript. Nitobi was later acquired by Adobe Systems in 2011, and PhoneGap was renamed first to *Apache Callback*, then finally to *Apache Cordova*.

In the years following the release of JavaScript-based multi-platform app frameworks, other companies tossed their hat into the ring with a variety of solutions. The most notable contenders of which were Xamarin and Facebook.

Xamarin, a subsidiary of Microsoft, released *Xamarin.Android* and *Xamarin.iOS* (formerly *Mono for Android* and *MonoTouch*, respectively) in 2011. By 2020, *Xamarin* had been merged into Microsoft's .NET framework as *.NET Multi-platform App UI (.NET MAUI)*. .NET MAUI applications are built in C# and able to target Android, iOS, and Windows.

Finally, Facebook (now also known as **Meta**) released *React Native* in 2015. *React Native* was derived from the *React* JavaScript library with an intent to target multi-platform devices rather than simply the Web. As of 2023, *React Native* remains in a pre-1.0 release state, and looks to continue that trend for a number of years to come.

It is easy to see that most of these multi-platform solutions employ the use of JavaScript as the underlying basis for creating apps, with the notable exception being *.NET MAUI*. JavaScript has a long history of developer support from the early days of the Web so JavaScript developers are plentiful and well-verses in creating Web apps. With the advent of Google's V8 JavaScript engine, JavaScript has seen steady increases in speed and performance. Yet, the fact remains that JavaScript applications are still interpreted and rendered by a JavaScript rendering engine, resulting in an additional level of abstraction and a potential bottleneck in creating smooth, performant applications.

A New Multi-Platform App Development Solution

Recognizing the need for a better solution, engineers at Google set to work on a new way of building multi-platform applications. Many frameworks and languages were evaluated during the early development phases of Flutter but eventually Dart was chosen over all others. Most important to the development team were four key pillars: increase developer productivity, allow for an object-oriented paradigm, be able to handle short-lived allocations quickly and efficiently in memory, and deliver extremely high-performance results. Many of the languages and frameworks evaluated delivered in some—but not all—of these categories. In the end, only Dart was able to deliver on every one of the four pillars that the team envisioned for Flutter. With Dart, the team had its first logo (*figure 1.1*):



Figure 1.1: The Dart logo
(Source: dart.dev)

Once Dart was chosen as the language on which Flutter would be built, the team agreed upon the remaining goals that would drive their decisions throughout development. These goals would be:

- Applications built in Flutter should be highly performant, with a target frame rate of 120 Hz.

- Applications built in Flutter should be platform agnostic; able to run on Android, iOS, and more.
- Applications should be fully privileged and have full access to the underlying operating system.
- Developers should have a fast development cycle, being able to edit and refresh the screen without the need to recompile and redeploy. No longer will it take upwards of 7 minutes to recompile changes to test code: stateful hot reload would allow for a sub-half-second test cycle.
- Flutter should be designed with continuous deployment in mind. Everything is always up-to-date and everywhere in the system recognizes that it is always online.
- Flutter should allow for a rich and flexible layout and painting of beautiful user interface elements.
- All the beautiful text that developers are used to creating on the web should be supported — right to left, up and down, ligatures, and so on.
- Flutter should be open, flexible, and extensible.

With these guiding principles laid out, the team set to work. On April 30, 2015, the team unveiled their creation at the Dart Developer Summit. Their framework was called: *Sky* (Seidel, 2015). Between Dart Developer Conference 2015 and Dart Developer Conference 2016, however, the team knew they needed a new name. They settled on *Flutter* and gave themselves a logo (*figure 1.2*):



Figure 1.2: The Flutter logo
(Source: flutter.dev)

The Rapid Development of Flutter

From its initial unveiling in 2015, Flutter has enjoyed a flurry of massive updates. The first alpha (v0.0.6) was released to the public in May 2017,

followed by the first beta less than a year later at Mobile World Congress 2018 (FlutterDev, 2018). There were several beta releases and release previews throughout 2018, but it was on December 4, 2018 when Flutter 1.0 was finally released as stable.

Included in the 1.0 release was a preview of *Hummingbird*, which would lay the groundwork for bringing Flutter to the Web. This represented a generational leap forward for the team as they worked toward their dream of being able to deploy Flutter applications on any screen. Five months would pass before the technical preview of **Hummingbird** was made available (*figure 1.3*):



Figure 1.3: The Hummingbird logo
(Source: youtube.com)

In the May 2019 update, which brought Flutter up to version 1.5, the *Hummingbird* moniker was dropped in lieu of Web being treated as a first-class target. In September 2019, Flutter 1.9 was released, with Flutter Web being integrated into the main repository.

In March 2021, Flutter 2.0 (*figure 1.4*) saw the first stable release of Flutter Web. Flutter 2.0 brought far more than stable Web support, however. Early access to desktop application support came in with this massive update, as well. For the first time, developers had the power to write code once and deploy it to Android, iOS, macOS, Windows, Linux, and the Web. The team's tireless efforts were beginning to pay off.



Figure 1.4: The Flutter 2.0 announcement logo
(Source: developers.googleblog.com)

Flutter 2.0 also signaled a massive paradigm shift in the framework. More robust navigation was introduced with the advent of Navigator 2.0 (officially named the Router API), and sound null safety was enabled by default for all Flutter 2.0 projects.

In May 2022, Flutter 3.0 was announced (*figure 1.5*). This version of Flutter brought with it stable support for Flutter on the desktop in macOS, Linux, and Windows. All three platforms included support for international text input and accessibility services, such as screen readers. macOS support received extra attention in the form of native, cascading menus and universal binaries.



Figure 1.5: The Flutter 3 announcement logo
(Source: medium.com)

Not to be left out of the Flutter 3.0 release party, mobile was front and center. New to Flutter 3 was support for foldable devices, variable refresh rate displays, Google's Material 3 design language support, and a preview of the new rendering engine, Impeller, for iOS.

By August 2022, at the Flutter Vikings conference in Oslo, Norway, Flutter 3.3.0 was announced. Further work was put into Impeller, which remained early access. This was also the time at which Eric Seidel, one of the co-founders of Flutter, parted ways with Google.

Then, in January 2023, Flutter 3.7 dropped at the Flutter Forward event in Nairobi, Kenya (*figure 1.6*). A keen-eyed observer will note that Flutter jumped from 3.3 to 3.7. This is indicative of just how much work went into the Flutter 3.7 release. Many of the base widgets were updated with enhanced Material 3 support, including adding new widgets for cascading menus on systems other than macOS. Impeller took several steps forward, with a mostly complete iOS implementation. The team would continue working on Impeller for Android and desktop for future releases. Speaking of iOS, the command to build a package for iOS now validates the application for any configuration changes which might be required by the App Store prior to release, making it quicker and easier to get through the app submission process. DevTools, Flutter's built-in developer tools, saw

significant work, with a redesign of the section of the tools used to analyze the memory usage of your application.



Figure 1.6: The Flutter Forward event announcement image
(Source: flutter.dev)

Additional improvements in Flutter 3.7 included custom context menus, smoother scrolling, a tool to assist in the internalization of your application, improvements to how text is selected by users, adding a text magnifier for fine-grained text selections, improvements to how background tasks are run, and much, much more.

Without a doubt, the biggest update to Flutter to-date has been Flutter 3.10, which was released on May 10, 2023, at Google I/O. This groundbreaking release brought with it Dart 3, which introduced many new features such as records, patterns, and class modifiers. Flutter itself saw a broad adoption of the Material 3 design spec throughout its myriad built-in widgets. It also managed to achieve Flutter Framework compliance with Supply Chain Levels for Software Artifacts (SLSA) at Level 1, acknowledging such security features as a scripted build process, multi-party approval with audit logging, and provenance (Chisholm, 2023). Additional improvements to Flutter included reduced bundle size for most compiled applications, faster loading times on the Web, element embedding allowing developers to embed Flutter into existing Web apps, support for fragment shaders, and many other new features and improvements.

Why Developers Are Flocking to Flutter

Understanding what makes the development experience with Flutter so desirable first requires us to understand the development experience that led up to Flutter's inception. Since Flutter targets so many platforms, we will have to discuss each of these somewhat independently.

The technology stack required to build a simple *Hello World* application (typically the first application a developer new to any language will write, as it is very simple in practice while showcasing the basic functionality of the programming language) on Android spans a significant number of pieces of software and programming languages. Android applications leverage Gradle, an automation tool, to build the application itself. Historically, applications are built in Android Studio: a very capable and powerful **integrated development environment (IDE)** with many features that might be confusing to novices and seasoned developers alike. Android resource files are laid out using the XML language, whereas the bulk of the code itself is written in either Java or Kotlin. Multiple Gradle scripts bind together the build process. All of this, and we have not even written a single line of code!

Creating a new Android application using Android Studio kicks off a flurry of processes, resulting in a minutes-long wait before one is even able to explore their code. Each change made requires a recompile and rebuild of the current activity, resulting in a slower development cycle. In contrast, creating a new Flutter application is as simple as running `flutter create` and opening the resulting folder in your IDE. Once the code is running (by issuing a `flutter run` command), changes are displayed instantly as you save your work due to Flutter's stateful hot reload functionality. The time between saving your work and seeing results on the screen is generally somewhere between 100 and 200ms with Flutter, all while maintaining the state of your application.

In addition to being able to see code changes nearly instantaneously, Flutter offers many tools to assist in a variety of ways. One of these tools, `dart fix`, which was introduced with Dart 2.12, scans your project for known issues and fixes your code *automatically*. This is an enormous time-saver when updating to newer versions of Flutter, as Widgets that may have been renamed or modified will be automatically migrated to the new version.

Likewise, `flutter analyze` will scan for known issues within your code and provide valuable feedback on what issues were found, where in the code they were found, and often a solution to the issue which was found. To suggest that the tooling is one of the reasons why developers love Flutter would be an understatement.

Developing a beautiful, platform-native UI using the Flutter framework is not only easy but fast. Flutter offers multiple robust Widget sets to create interfaces that adhere to *Google's Material Design* as well as *Apple's Cupertino Human Interface Guidelines* design language right out of the box. Furthermore, there are packages to create interfaces in the *Ubuntu's Yaru* style and *Microsoft's Fluent Design*. Despite the extraordinary differences in these different design languages, the process of laying out an interface with either of them is largely identical. There is no need to learn two frameworks to accomplish the same task because Flutter does all the heavy lifting.

In fact, with Flutter, it is possible to create an application that will *automatically* switch between the Material and Cupertino design languages depending on the platform on which the code is running. Furthermore, additional design languages can be easily added into the mix to create a truly native-looking and feeling application for every platform without the need to maintain separate codebases.

Looking Forward

With hindsight, it seems almost inevitable that a toolkit as robust, powerful, and flexible as Flutter would be created. From the earliest days of the smartphone market, developers have been tasked with meeting consumers where they are at. With the advent of Web 2.0, the Internet itself grew from a simple information portal to something that each person could contribute to and interact with. Where a Web browser was no longer sufficient, applications grew to fill the gaps. Certainly, some of these applications could have (and at one point *were*) capable of being replaced by their legacy Web brethren, but a greater need arose: market share.

As phones grew to give more and more screen real estate, a battle for supremacy over those precious pixels was fought. Why should your app exist as a bookmark in a Web browser when you could derive more usage

by being an icon that exists as a persistent reminder of the value you offer? Then, once you are an icon on the screen, do you not want to be an icon on *every* screen? Now you need to make a choice: leverage Web technologies or have separate teams to build native applications. Either way, there are trade-offs. That is: until Flutter came along. Now, you can have your native apps without the trade-offs.

Conclusion

These days, we take for granted the ubiquitous availability of highly polished and purpose-built mobile applications for any task we can dream up. It is easy to lose sight of just how far the industry has come, and how much trial and error it took to get us to an easy-to-use and easy-to-develop-with toolkit like Flutter. It is with great enthusiasm that you are welcomed into this community of Flutter developers!

This book will prepare you for a career in building applications. But first, we will investigate the market as it exists today for Flutter developers and explore where Flutter is being used in the wild. Where — and by whom — Flutter is being used may just surprise you. Without further ado, welcome to the world of *Flutter for Jobseekers!*

References

- Ceci, L. (2021, September 10). *Number of apps available in leading app stores as of 1st quarter 2021*. Retrieved from Statista: <https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/>
- Chisholm, K. (2023, May 10). *What's new in Flutter 3.10*. Retrieved from Medium: <https://medium.com/flutter/whats-new-in-flutter-3-10-b21db2c38c73>
- FlutterDev. (2018, Feburary 27). *FlutterDev on Twitter*. Retrieved from Twitter: <https://twitter.com/FlutterDev/status/968486429754933248>
- Seidel, E. (2015, April 30). *Sky: An Experiment Writing Dart for Mobile (Dart Developer Summit 2015)*. Retrieved from YouTube: <https://www.youtube.com/watch?v=PnIWl33YMwA>

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

Market Opportunities for

Flutter Developers

Introduction

In the few short years of Flutter's existence, it has seen a monumental amount of growth and adoption by all sectors of business. Flutter's high performance and low barrier to entry have led developers and businesses worldwide to embrace the framework. From banking to automotive, independent developers to large corporations, Flutter is becoming ubiquitous with the pace of adoption only accelerating.

Let us investigate, then, who is developing Flutter applications, in what industries, and for what purposes. Let us also examine how the growth of Flutter compares to the competition, and even to native app development. Flutter is being used in some innovative and exciting ways and we are going to talk about it starting right now!

Structure

In this chapter, we will discuss the following topics:

- The growth of Flutter in the marketplace
- Who is using Flutter, and what are they using it for?

Objectives

In this chapter, we will explore the exponential growth of Flutter in the marketplace, analyzing the reasons behind its increasing popularity among developers and businesses. Additionally, we aim to identify the wide range of Flutter users and their specific applications, uncovering the diverse use cases that have contributed to its widespread adoption. Ultimately, this investigation will provide valuable insights into Flutter's significant impact on the app development industry and beyond.

The Growth of Flutter in the Marketplace

Since the first stable release of Flutter in December 2018, it has experienced steady growth in popularity and adoption worldwide. To attempt to put this in context, we will need to compare the popularity of writing native applications for both iOS and Android, as well as comparing Flutter's top competitor: React Native. Quantifying the popularity of a language, framework, or toolkit is no simple task. However, there are a couple of ways we can attempt to break this down.

First, we will look at the Google Trends data. Google allows for some interesting data to be pulled from their collected search results. For example, we can compare the popularity of search terms in any given period. That is the first way we are going to explore the growth of Flutter when compared to the competition: how popular was the search term *Flutter* in comparison to its rivals on Google? Please note that the period for which the data was pulled is between December 2016 and September 2021. Here is the data for *Flutter vs Swift* (*figure 2.1*):

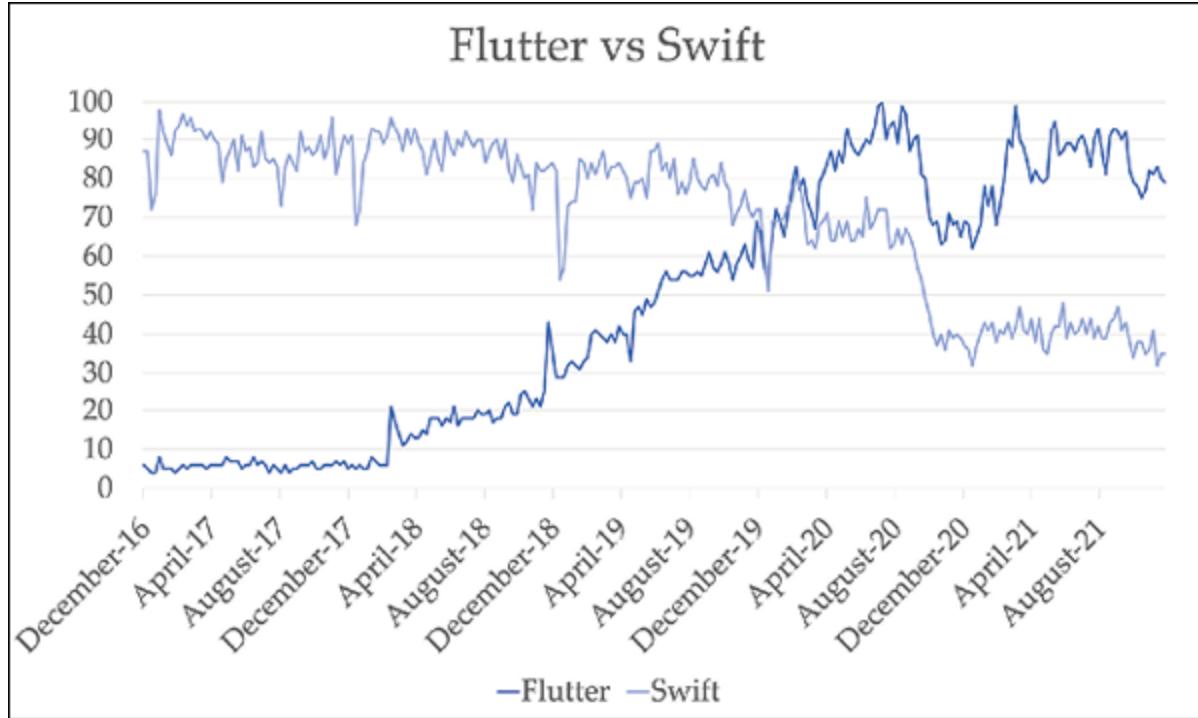


Figure 2.1: Popularity of “Flutter” vs “Swift” searches on Google, worldwide.
(Source: Google Trends)

As expected, Flutter has grown at a steady rate since its first stable release. Swift searches remained relatively stable right until something interesting happened in December 2019. This was when Flutter 2.0 released as stable, signaling to developers and companies that Flutter was here to stay. There was another dip in August 2020, which you will see repeated in *all* data sources. This is due to the global lockdowns during the COVID-19 pandemic. Interestingly, Swift, the programming language for developing native iOS applications, never quite recovers from this dip in popularity, whereas Flutter sees an immediate jump back up to its former popularity, where it has continued to hover. Next, let us look at Kotlin, the primary programming language for developing Android applications natively (*figure 2.2*):

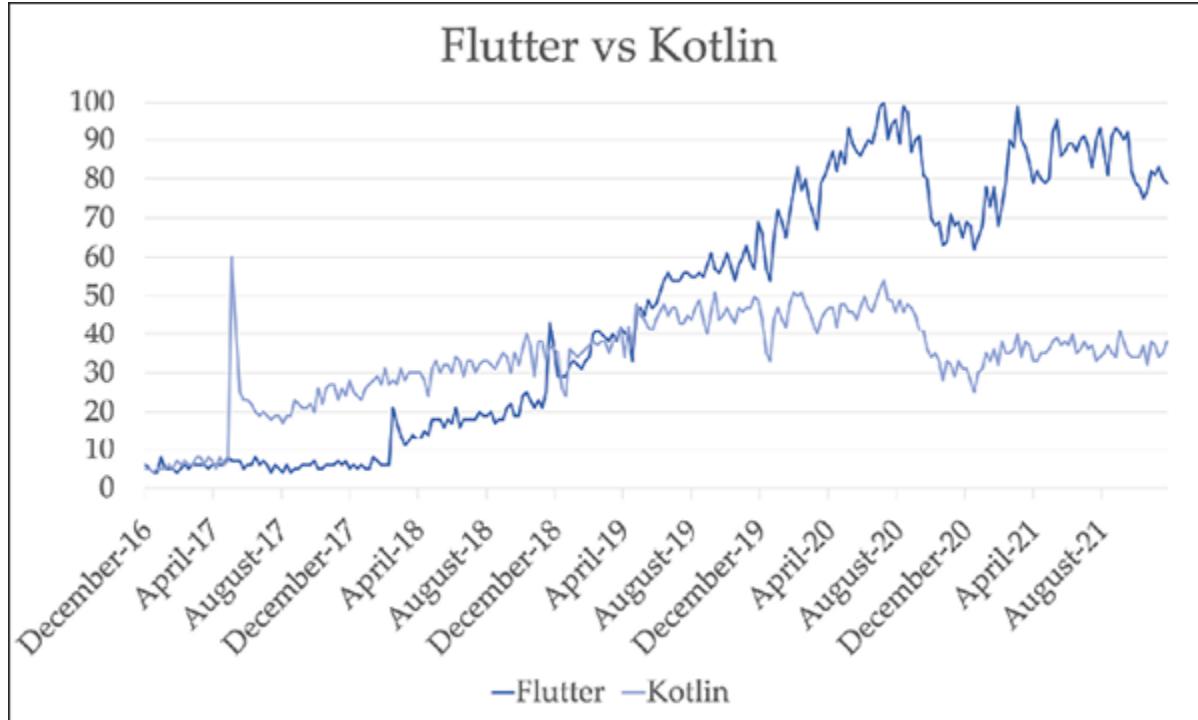


Figure 2.2: Popularity of “Flutter” vs “Kotlin” searches on Google, worldwide.
(Source: Google Trends)

We see some of the same trends with Kotlin that we do with Swift: a dip during the lockdowns and its popularity never quite recovering. However, the data uncovers something else. During Google’s I/O conference in 2017, it was announced that there would now be official support for the Kotlin programming language when writing Android apps, which explains the large spike in popularity we see in the graph data. However, two years later, at I/O 2019, when Google announced that Android development would skew toward a Kotlin-first approach, the data shows that there were no more and no fewer searches for Kotlin. Interestingly, we can derive from this data that either all (or most) developers who were writing native Android apps were already using Kotlin, or some other market interruption was already taking place. Next, take look at how React Native stacks up (*figure 2.3*):

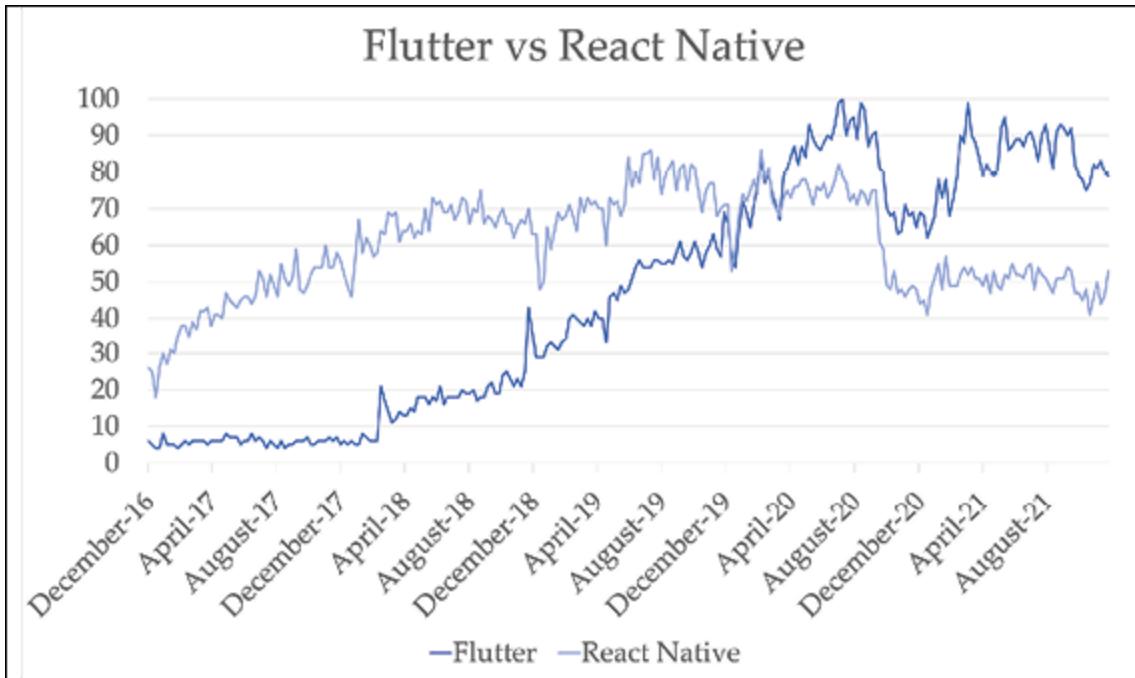


Figure 2.3: Popularity of “Flutter” vs “React Native” searches on Google, worldwide.
(Source: Google Trends)

As with both Swift and Kotlin, we see some of the same trends emerge. Of course, there’s the tell-tale pandemic slump in late 2020. There is also the lack of recovery following that slump. What is clear from the start is that React Native enjoyed a much stronger overall growth when compared to both Kotlin and Swift. React Native’s relatively late introduction to the game, when compared to Swift, would account for its seemingly lower popularity early on, as developers were already working with Swift as React Native matured. Kotlin, coming even later to the game, never quite saw the same popularity as React Native. However, Flutter overtakes React Native quite quickly. Again, we see the trends leaning toward Flutter after Flutter’s 2.0 release. Finally, we can draw additional context by comparing all these data points at once (*figure 2.4*):

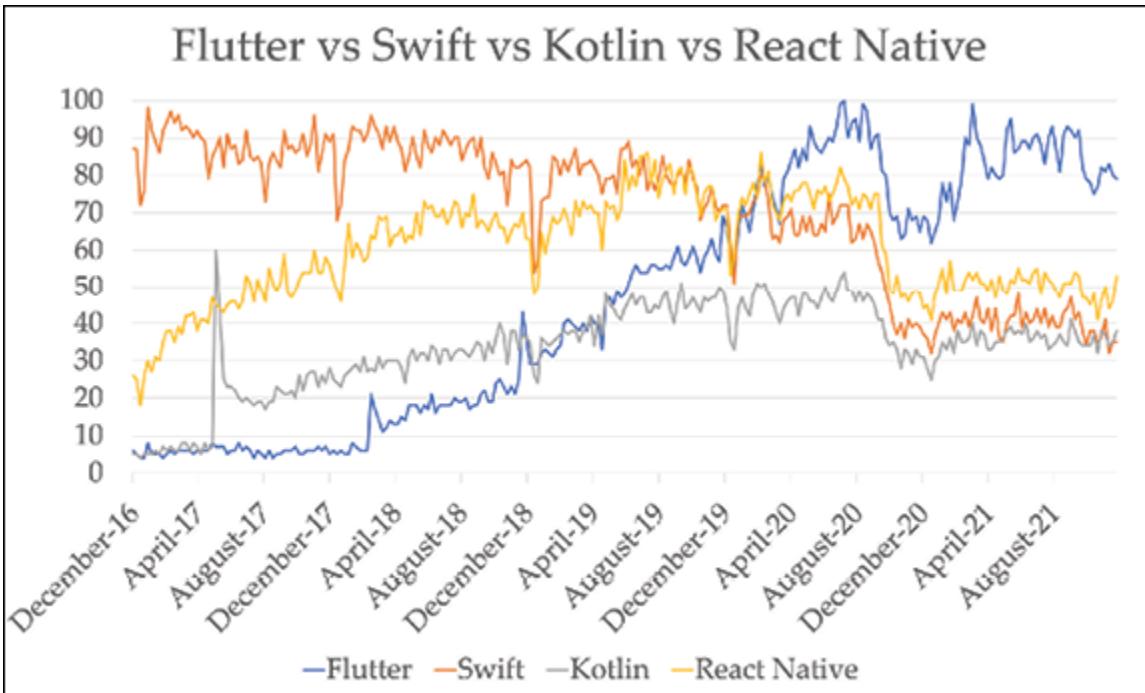


Figure 2.4: Popularity of “Flutter” vs the competition searches on Google, worldwide.
(Source: Google Trends)

What we can see is that Flutter completely overtook the competition early in 2020. This almost certainly coincides with the stable release of Flutter for Web in March. For the first time in history, there was now a toolkit that would allow developers to create beautiful, performant applications for *any* platform with a single codebase—and the market took notice. The demand for Flutter developers during this time skyrocketed. Even those who spoke in hushed tones and suggested that they knew Flutter were being approached for work.

The rise in popularity of Flutter has been unprecedented, and it shows no signs of slowing down any time soon. With the flexibility that Flutter offers, along with the prospect of a single codebase and single development team, coupled with the prospect of a single technology for any purpose, it is no wonder companies are scrambling to find Flutter developers.

Now that we can quantify the popularity of Flutter versus the competition, we are left with another question.

Who is Using Flutter, and What are They Using it for?

As we have discussed, Flutter is extremely versatile. It can target iOS, Android, macOS, Windows, Linux, and the Web, all with a single codebase. Additionally, it can be embedded in existing applications and can run on any platform with the use of a custom embedder.

There are many big names using Flutter, and many more are being added to that list on a regular basis. It should come as no surprise that Google is using Flutter for several of their big applications. Also, on the list of big players, we can find the likes of Alibaba, BMW, ByteDance, eBay, Toyota, Google, and many more. Let us take a closer look at each of these companies to see what they are building.

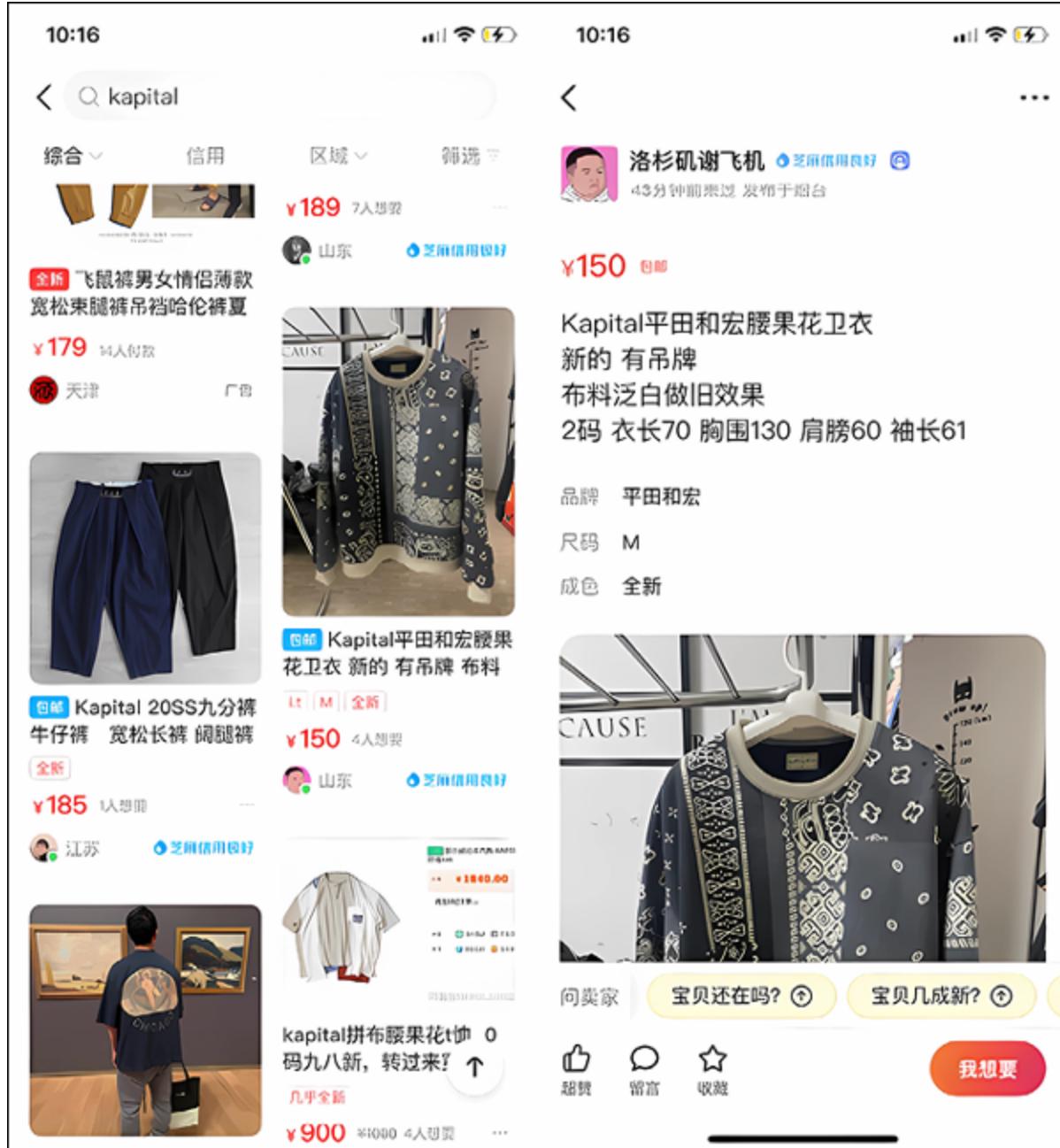
Case Study: Alibaba

Alibaba may not be a household name in America *yet*, but that does not mean they are not massive. The Wallstreet Journal helps to summarize Alibaba in one sentence: *It is a marketplace, a search engine, and a bank, all in one (Wallstreet Journal, 2014)*. Suffice it to say the Alibaba Group is *massive* in China.

One of Alibaba's biggest apps is *Xianyu*. Xianyu is a second-hand marketplace, much like eBay is in the West. The Xianyu development team faced the same problem that plagues development teams across the world: the need to deliver their application to both Android and iOS users as quickly as possible. Like other development teams, their efforts were hampered by maintaining two codebases. They wanted something that they could use to deliver fast and beautiful applications without being slowed down by maintaining separate codebases. Of course, they turned to Flutter.

The Xianyu team faced another challenge, however. They already had an application. Rebuilding the entire application from scratch in Flutter would have gone against their stated goal of delivering new features quickly. Luckily, Flutter was still able to accommodate them. Applications that use Flutter *can* be written entirely in Flutter, however they do not *have* to be. It is possible to embed Flutter widgets into existing applications, allowing a team to transition from a native-only codebase to a mixed codebase with Flutter, and eventually replace all native code with Flutter code on their own time (*figure 2.5*).

Without being required to rewrite their entire application from scratch while still having the power and flexibility of incorporating Flutter widgets, the team was able to start shipping new features in half the time. That is the power of Flutter:



*Figure 2.5: Screenshots of pages written in Flutter within the Xianyu application.
(Source: (Kermittfx, 2021))*

Case Study: BMW

As BMW's mobile app offerings grew across all their brands, they faced a common problem: the features and design languages present in each of the applications, including the same application on multiple platforms, began to diverge and have more and more discrepancies. The fragmentation broke down the cohesiveness of the experience for users, which was unacceptable to the team. Like many teams before them, they were faced with the question of how to manage the ballooning costs of juggling multiple native development teams, slow feature development, and high costs.

BMW spent a long time evaluating multi-platform solutions, and at first, they were skeptical that Flutter was mature enough to handle their needs as this evaluation period was during the latter months of 2019, still relatively early in Flutter's releases. They built several proof-of-concept applications using each of Flutter and its competitors. In the end, it was Flutter that the team decided on when they chose to forego Web-based content in lieu of a better user experience.

Once the decision was made to go with Flutter, BMW began to rewrite all its applications from scratch. They operate in more than 45 countries, all of which have different rules, regulations, and requirements. Their suite of applications would need to conform to all these regulations for each country they deploy in. To accomplish this, the team established a new internal platform, which they dubbed the *Mobile 2.0 Platform*.

The new app platform is built on three pillars: user friendliness, safety, and reliability. It provides a consistently designed set of functions spanning all brands based on feedback and our customers' usage behavior.

—Dr. Nicolai Kraemer,
Vice President Offboard Platform BMW Group

The new Mobile 2.0 Platform allowed them to automate their application builds. Each application would need a new build for the market it serves and the operating system it would run on. Those builds would need to run automated tests and eventually be deployed. In essence, this complexity led the team to 96 different variants for every single application they developed. They have been able to build, test, and deploy all of them with Flutter. In the end, they were able to solve all the problems that plagued them by unifying their codebase with Flutter, allowing them to deliver features more quickly with less effort and at a reduced expense.

Case Study: ByteDance

ByteDance may not be a household name but its biggest products sure are. ByteDance's most popular offering, TikTok (known in China as Dǒuyīn—albeit with a totally separate and separated content offering), is one of more than 70 applications in the company's portfolio. With more than 200 developers working with the toolkit, ByteDance is no stranger to Flutter development.

Their affinity for Flutter has benefitted the community at large, too. ByteDance has made major contributions to Flutter by submitting dozens of pull requests (*Zakhour, 2021*). These contributions include crucial debugging tools, such as the frames and timeline events charts, both used to help profile a Flutter application and identify potential slowdowns and resource usage. They have also helped drive the core performance of Flutter by helping to optimize the framework.

ByteDance tends to embrace new technologies with the attitude that every mature technology was once new. As they have so eloquently put it, *there are indeed many people in the industry who prefer mature technology, but it takes time for every technology to mature, and there will always be people like us who love to stay on the cutting edge* (*Zakhour, 2021*).

Although ByteDance has found that rewriting their established applications from scratch in Flutter has been inefficient, they have noted that new projects benefit from the speed at which they can develop using Flutter. They have stated that there has been an average 33% increase in productivity on Flutter projects versus writing native applications (*Flutter, 2021*).

Xigua Video is ByteDance's video-sharing platform, offering both longform and short-form videos, and it even has its own film and television content. With more than 131 million monthly active users, it is a shining example of one of their products which was written with Flutter. Xigua Video is available on iOS, Android, and the Web, all from a unified codebase, thanks to the power and flexibility of Flutter.

Case Study: eBay

As the first-ever online auction site, eBay is a company that needs no introduction. What some may not know is that eBay sells more than just trinkets. Their business extends into the automotive space, as well. eBay recognized that its main application was not the right place to direct users for their automotive sales, so in 2018 the eBay Motors team was tasked with creating a mobile application within the year for both iOS and Android.

Recognizing the monumental task ahead of them, the small team knew they would need to leverage a cross-platform framework to accomplish their goal. Within a couple of short months, the team realized their choice to leverage Flutter was an extraordinarily good idea. They decided to evangelize Flutter within the company, organizing several workshops to bring the rest of the engineering team up to speed.

Flutter has not only met our expectations—it has dramatically exceeded them.

—Corey Sprague,
Senior iOS Engineer, eBay

The larger engineering team at eBay was as impressed as the eBay Motors team with Flutter. With the groundwork having already been laid by the Motors team, the larger engineering team found it very easy to share code for the UI, the business logic, and all other parts of the application. According to the eBay Motors team, the amount of code shared is evident in the amount of Dart code in their repository (*Sprague & McKenzie, 2020*):

- 98.3% Dart code (approximately 220k lines of Dart)
- 1.1% Scripts, CI, and various Automation Tools for our development lifecycle
- 0.6% split across Kotlin, Java, Swift and Objective-C

Migrating to Flutter allowed all of eBay to coalesce into one team from many, giving everybody on the team equal footing and providing a more consistent experience for both users and engineers (*figure 2.6*):

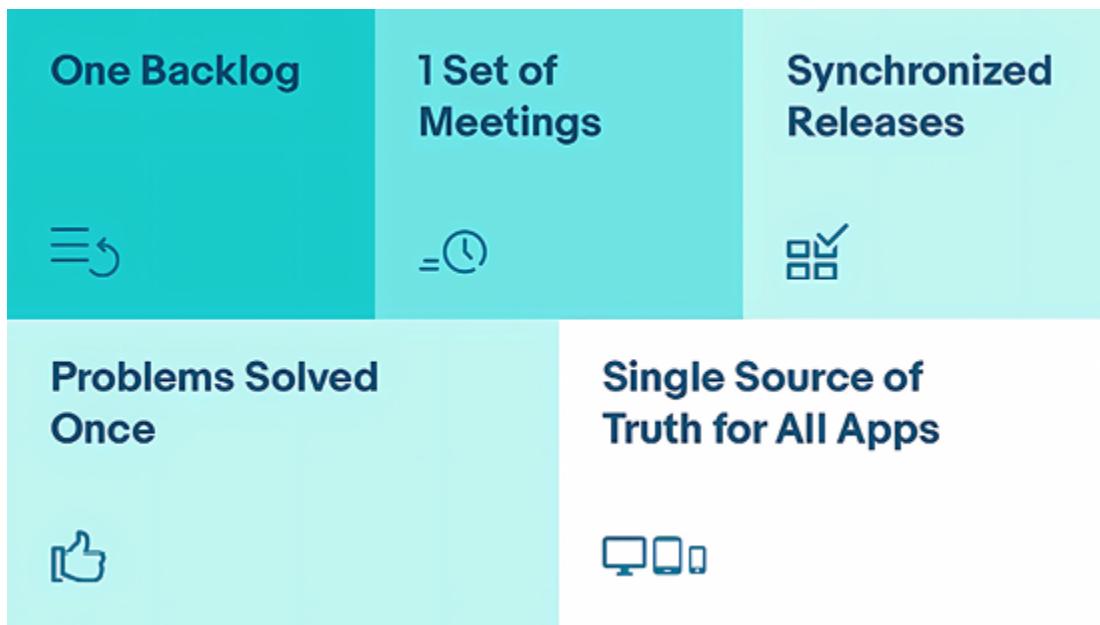


Figure 2.6: eBay has found using Flutter reduces the amount of work for engineers.
(Source: tech.ebay.com)

The benefits eBay found by moving their development to Flutter are not unique to eBay but do illustrate why Flutter is so important. Rather than build the same UI elements, business logic, and delivery pipelines for separate platforms, Flutter enables teams to share resources like never before. This dramatically reduces the complexity of problems that face engineering teams, allowing them to solve a problem just once instead of once for each platform.

Case Study: Toyota

Producing about 10 million vehicles per year, Toyota is one of the world's largest automotive manufacturers. As those vehicles become more and more technologically advanced, so too do the instrument panels and infotainment centers. Anyone who has driven or ridden in a car with an in-dash display might have felt disappointed by the lackluster user experience. Often clunky, slow to navigate within, and ugly, these interfaces have long been a pain point for many drivers and passengers alike.

Toyota is no stranger to these types of in-dash experiences. Having historically built their on-screen infotainment systems in-house, Toyota marveled at the speed and flexibility of Flutter. When they discovered Flutter was able to target embedded devices, they reached out to the Flutter

team to establish a partnership. Through this partnership, Toyota began work on its next-generation of in-dash experiences.

Daniel Hall, Toyota's Chief Engineer, recognized that non-smartphone touch interfaces are often frustratingly non-responsive. He and his team sought to remedy this by leveraging Flutter's first-class, smartphone-tier touch mechanics in their new (as of 2022) in-car experiences (*figure 2.7*):



Figure 2.7: Toyota's new in-dash experience.

Source: Carscoops (Carscoops, 2021)

Another way Toyota finds Flutter to be particularly well suited for development is by the breadth of platforms that can be targeted using a single codebase. Developing and testing code directly on an instrument panel or infotainment system is cumbersome and comes with significant challenges, such as cost and a sacrifice in developer mobility. (Try lugging an instrument panel with you to a coffee shop!) By testing the code as native desktop applications, the team can develop more quickly and efficiently while allowing freedoms that are traditionally not afforded to in-car software engineers. Furthermore, touch interactions are easily testable by deploying to iOS and Android devices. They are even able to leverage the power of Flutter Web to communicate more easily with designers and iterate over designs more quickly. The only reason all of this is possible is due to the flexibility of Flutter.

Case Study: Google

Finally, we come to Google itself. It comes as no surprise that Google would use the toolkit that they have spent so long building within their own suite of applications and projects. Their first application to be rewritten from the ground up in Flutter was Google Pay, their flagship mobile payments application. Immediately, users were treated to a faster, higher quality application when compared to the aging offering which it replaced. Their shift to Flutter also streamlined their codebase, eliminating over 500k lines of code! Since then, Google has moved many of its applications over to Flutter (*figure 2.8*):

Flutter-powered Google apps shipping now include:



Figure 2.8: A selection of Google apps written in Flutter.

Source: Google (Google, 2021)

Google also developed Stadia, their cloud streaming service for games, with Flutter. The Stadia app ran on both iOS and Android, sharing a single codebase. Because Stadia had its own custom game controller, the team needed to build a custom Bluetooth plugin using platform-native code. Once this plugin was embedded in their Flutter application, the process of communicating with the controller became seamless and transparent. The entire codebase weighed in at around half a million lines: far less than separate native applications would have required. Furthermore, the time they saved by using Flutter gave the team the opportunity to test and tweak the application for better performance and reliability before launch.

Bugs that would be a near-constant drag on productivity with two or more native clients, especially in a complicated flow, are practically a non-issue when using Flutter.

–Nick Sparks, Software Engineer, Stadia

As Google continues to develop Flutter, it is inevitable that more and more of its products and services will be written using the toolkit. Google has been working on a brand-new operating system called Fuchsia for several years, which already powers the Nest Hub, and uses Flutter as its primary app development language. Any applications written today in Flutter will run on future Fuchsia devices, which is certainly something to consider, as there is potential for Fuchsia to supplant Android in the mobile space. It is hard to say for sure, but one could envision a future in which Fuchsia is the next big thing.

Conclusion

As Flutter continues to mature, developers and companies alike have taken notice. Some of the biggest names in tech embraced Flutter, recognizing it for its game-changing ability to speed up development time, lower development costs, allow for feature parity across platforms, and unify development teams on a single codebase.

From iOS to Android, the desktop, Web, Fuchsia, and even embedded in the dashboards of Toyota vehicles, Flutter keeps popping up in some of the most exciting places. The tireless efforts of companies like ByteDance who contribute freely to Flutter, will certainly help to attract new developers and companies to the ecosystem!

Finally, before we move on to the upcoming chapter where we will get our developer environment set up and ready for us to start coding, I want to introduce somebody very special. Say hello to Dash, Flutter's very own hummingbird mascot (*figure 2.9*)!



Figure 2.9: Flutter's mascot, Dash.
(Source: docs.flutter.dev/dash)

References

- Carscoops. (2021, September 19). *We Check Out The All-New Infotainment System In The 2022 Toyota Tundra.* Retrieved from YouTube: <https://www.youtube.com/watch?v=pFNYtLLhjrk>
- Flutter. (2021, December 31). *ByteDance.* Retrieved from Flutter Showcase: <https://flutter.dev/showcase/bytedance>
- Google. (2021, March 3). *Announcing Flutter 2.* Retrieved from Google Developers Blog: <https://developers.googleblog.com/2021/03/announcing-flutter-2.html>
- Kermittfx. (2021, April 19). *[GUIDE] How to use Xianyu, Taobao's second-hand market!* Retrieved from Reddit: https://www.reddit.com/r/QualityReps/comments/mtznu0/guide_how_to_use_xianyu_taobaos_secondhand_market/
- Sprague, C., & McKenzie, L. (2020, September 8). *eBay Motors: Accelerating With Flutter.* Retrieved from eBay: <https://tech.ebayinc.com/product/ebay-motors-accelerating-with-fluttertm/>

- Wallstreet Journal. (2014). *What Is Alibaba?* Retrieved from WSJ: <https://graphics.wsj.com/alibaba/>
- Zakhour, S. (2021, June 29). *Google I/O spotlight: Flutter in action at ByteDance.* Retrieved from Medium: <https://medium.com/flutter/google-i-o-spotlight-flutter-in-action-at-bytedance-c22f4b6dc9ef>

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

Installing Flutter and Configuring Your IDE

Introduction

Congratulations on making it this far! By now, you should have a firm understanding of what Flutter is and what kind of problems it seeks to solve. Now, it is time to get our development environment set up and ready to write some code! Do not worry if this sounds daunting. This chapter will guide you through the process and try to answer any questions which may arise along the way. By the end of this chapter, you will have a development environment set up to write your very first app. As a bonus, you will even compile the default sample app, which ships with Flutter!

Structure

In this chapter, we will discuss the following topics:

- Getting started on Windows
- Getting started on macOS
- Installing and configuring Visual Studio Code
- Building and running your first Flutter application
- Briefly exploring the default Flutter application

Objectives

After completing this chapter, you will have Flutter installed on your computer and have configured your development environment to write apps using Flutter. You will learn how to create a new Flutter project, then run your new Flutter project in an Android Emulator on your computer.

Tools

The following tools will be used throughout this chapter:

- Git
- Windows PowerShell (Windows only)
- Terminal (macOS only)
- Visual Studio Code
- Visual Studio (Windows only)
- Xcode (macOS only)
- Android Studio

Getting Started on Windows

To develop Flutter applications on Windows, we are going to need to install several tools and utilities. We will tackle this one by one, and when we are done you should have everything you need to get started writing apps. The tools we will be installing are PowerShell, Git, Visual Studio, Android Studio, and Flutter. This guide will also assume you are running Windows 10 or Windows 11.

Windows Developer Mode needs to be enabled before we can begin developing applications. To do this, you will first need to go to your system settings. Whether you are using Windows 10 or Windows 11, the fastest way is to open the **Start** menu and type **settings**. This will open the Windows settings application. From there, you can either search for **developer mode** or navigate to **Update & Security** (on Windows 10) or **Privacy & security** (on Windows 11). On Windows 10, you will need to go to the **For developers** tab, and then on both versions of Windows, you will need to toggle the switch for Developer Mode to on. Windows will likely ask you to confirm this change by presenting you with a security control dialog.

Confirming the change will enable developer mode, and then you can close the Settings application.

PowerShell is a command line that we will be using throughout this book. Windows 10 and 11 ships with older versions of PowerShell, which we will need to upgrade to the latest version.

To get started, head over to <https://aka.ms/powershell-release?tag=stable> and scroll down to the assets section. Look for PowerShell-x.y.z-win-x64.msi, where x.y.z is the latest version available. Download the file and run through the installation process. The default install options are sufficient for us, and the installation is very straightforward.

To verify that PowerShell is installed, click your **Start** menu and type **PowerShell**. There may be several versions installed, such as **Windows PowerShell**, **Windows PowerShell ISE**, and **PowerShell v (x64)** where v is the major version of PowerShell matching the file you just installed. At the time of writing, the latest major version of PowerShell is version 7. If your search looks like the following screenshot(s) (*figure 3.1*), you have successfully installed PowerShell.

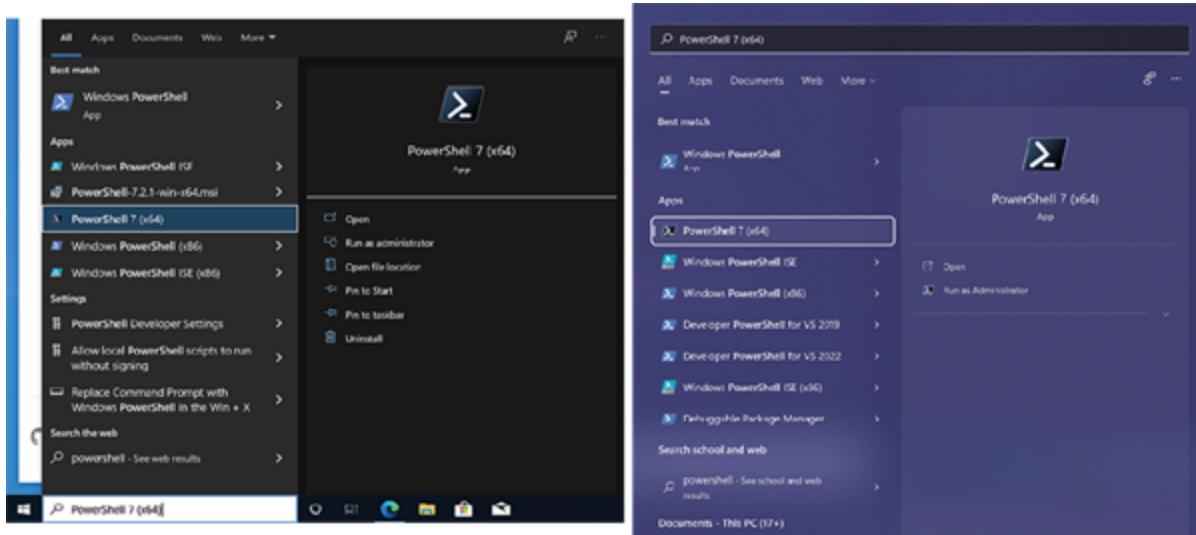


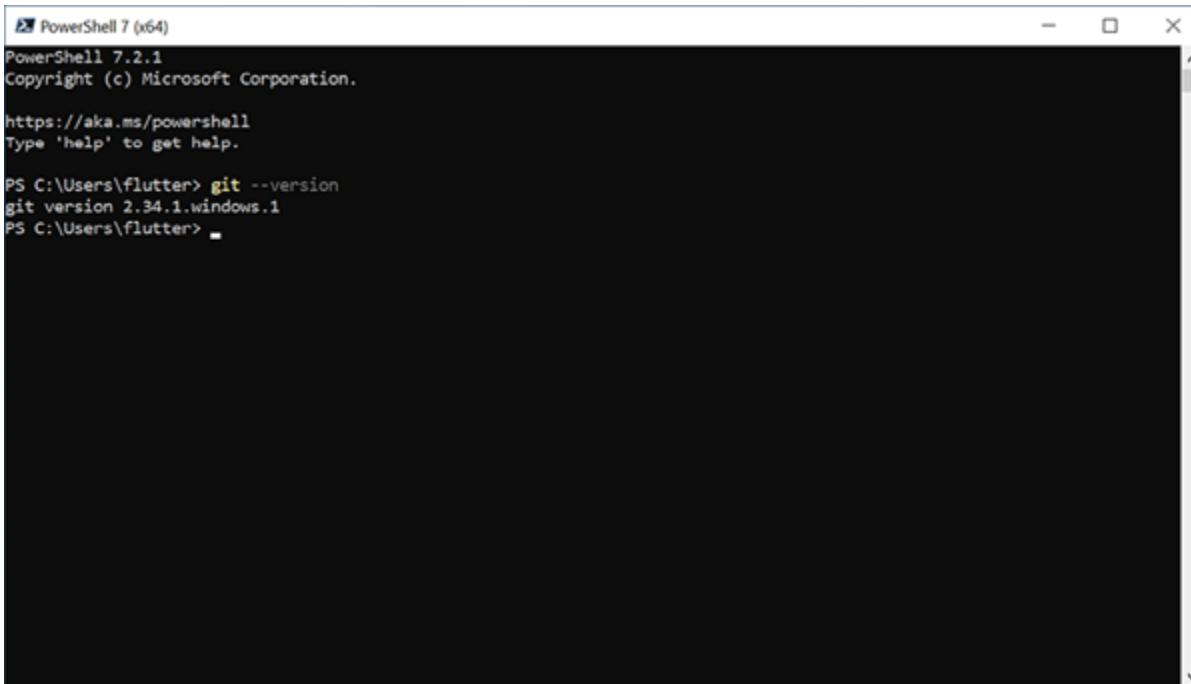
Figure 3.1: PowerShell in the Windows 10 (left) and Windows 11 (right) Start Menu

Git is a source control manager used to manage source code. It helps us to maintain a history of changes that we can refer to in case things go awry or we need more information about what changes were made, by whom, and for what reason. Furthermore, it is a crucial utility for working on a

software development team, as it enables multiple developers to collaborate on a single codebase.

To get started with Git, head over to <https://git-scm.com/download/win> and click the link at the top of the page that says, **Click here to download**. Once downloaded, run through the installer. For the most part, the default install options are fine. However, it is recommended to change the option for the default branch name when given the option. Click the **Override the default branch name for new repositories** option when presented. Accept the new default value of `main` and click next. The remaining options can all be left as their defaults.

Once you are finished installing Git, you can test to ensure it is installed properly by running PowerShell and typing `git --version`. If there are no errors, it will print the version of Git that you just installed and exit. If successful, it should look something like the following (*figure 3.2*):

A screenshot of a Windows PowerShell window titled "PowerShell 7 (x64)". The window shows the following text:

```
PowerShell 7.2.1
Copyright (c) Microsoft Corporation.

https://aka.ms/powershell
Type 'help' to get help.

PS C:\Users\flutter> git --version
git version 2.34.1.windows.1
PS C:\Users\flutter> -
```

The window has a standard Windows title bar with minimize, maximize, and close buttons. The main area is a dark gray terminal window displaying the command and its output.

Figure 3.2: Successfully verifying that Git has been installed on Windows 10

Microsoft Visual Studio is an **integrated development environment (IDE)** used by many developers to create all sorts of different types of software. While we will not be using it to develop our Flutter applications, we will be installing it to gain access to the tools it provides to us, including

its compiler. A compiler is a piece of software that converts our source code into machine code.

To start, head over to <https://visualstudio.microsoft.com/> and look for the **Download Visual Studio** button. As you hover over it, a dropdown will appear. Select the **Community edition**. Once downloaded, run the installer which will install and launch the Visual Studio Installer.

The Visual Studio Installer will prompt you for a workload to install alongside the IDE itself. The workload you will need to select is the **Desktop development with C++ workload** (*figure 3.3*):

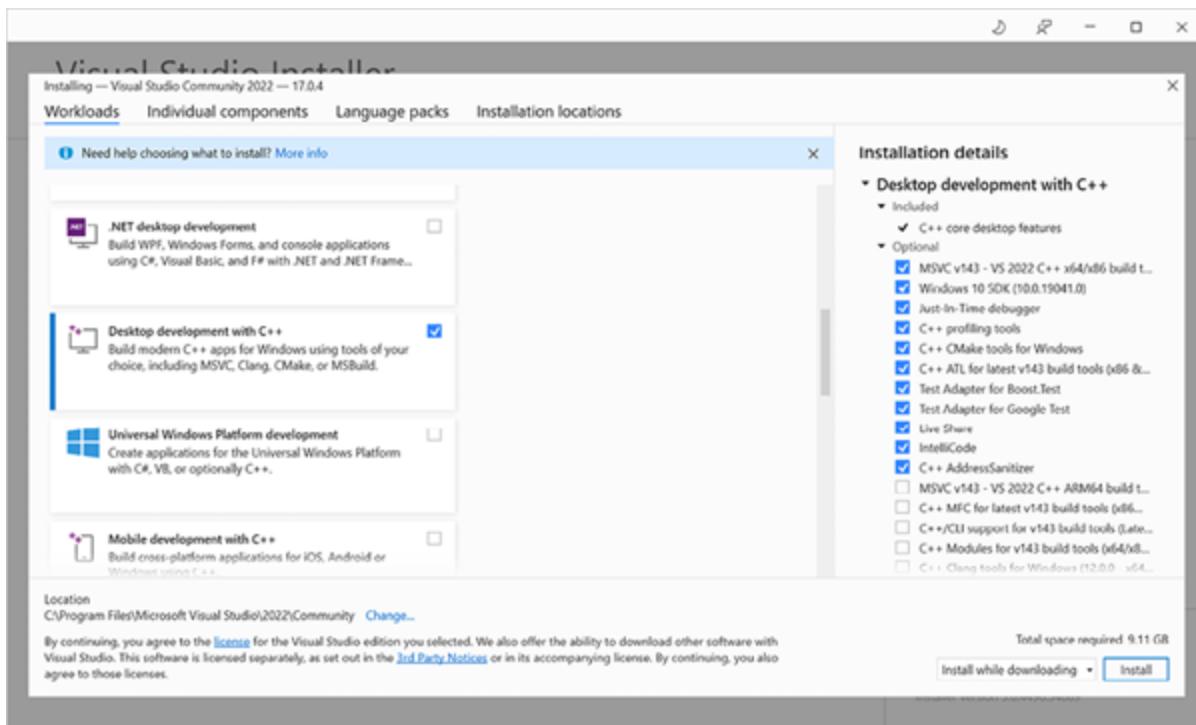


Figure 3.3: Selecting the correct workload during the Visual Studio install

There is an option to **start after installation** while Visual Studio is installed. Since we will not be using Visual Studio itself, there is no need to run it. Feel free to uncheck this box.

Android Studio is another IDE that is used to write Android applications. We will not be using it as our development environment, but we do want some of the tools that it provides, including the Android SDK, command line tools, and the **Android Virtual Device (AVD)**.

First, head over to <https://developer.android.com/studio> and download the latest version of Android Studio. By default, the installer will have the option to install the **Android Virtual Device (AVD)** selected. Since we will be using the AVD extensively as we develop and test our applications, ensure that the option to install the Android Virtual Device remains selected. Please note that the Android Virtual Device requires a processor which supports virtualization. If you find yourself unable to install or use the AVD, it is likely that your processor does not support virtualization. In that instance, you can skip the instructions throughout this book that pertain to the AVD directly and instead run your application using the Windows desktop mode.

Once Android Studio is finished installing, be sure to launch it. The first prompt we are presented with will ask us if we want to import settings. Since we do not have any to import, we will select **Do not import settings** and click **OK**. Next, we are greeted with the welcome screen. There will probably be an update to install, which a notification at the bottom of the window will inform you of. Install the update, and when prompted, restart Android Studio to apply the changes.

Once Android Studio is finished updating and has been restarted, you will be back at the welcome screen. Click **Next** to move to the install type screen. Choose the standard option and click **Next**. You will be offered the option to choose between a light and dark theme, so choose one according to your preferences. The next screen will ask you to verify your settings, which should all be fine. Click **Finish**. At this point, several components will be downloaded to complete the process.

Once all the components have been downloaded, you will be at a new welcome screen. You will need to create a new project to get to the main application window where we will install and configure the rest of the tools we need. The project you create does not matter, so do not stress too much over it. Just choose **New** and click through the default options.

Android Studio will create and open the new project, then download several components in the background. There will be a status bar at the bottom of the window which will update you on its progress, as well as a notice at the top that says **Gradle project sync in progress**. Wait for a while as this

process finishes. Once everything has finished, you should see a window that looks like the following (*figure 3.4*):

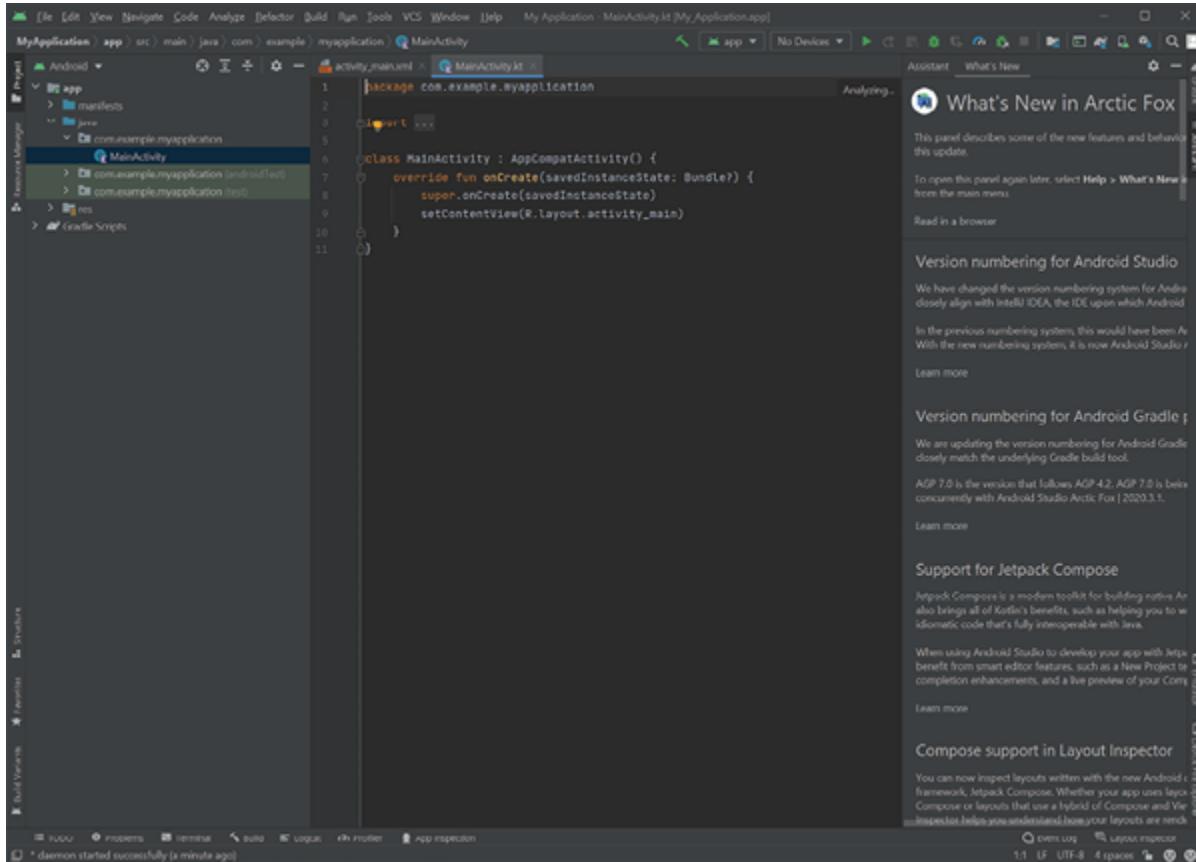


Figure 3.4: The Android Studio main window

Next, look for the **SDK Manager** icon (⬇️) in the upper right-hand corner. Clicking this will open the settings window to the Android SDK section. On the **SDK Platforms** tab, the latest version of Android's SDK will already be installed. This is entirely sufficient for our purposes; however, if you wish to target an older or newer version of Android, this is where you would come to download the SDK.

Now, move over to the **SDK Tools** tab (*figure 3.5*). Here, we want to make sure that the check boxes for the **Android SDK Build-Tools**, the **NDK (side by side)**, the **Android SDK command-line tools**, **CMake**, **Android Emulator**, and **Android SDK Platform-Tools**. If you have an Intel processor, you will also want to install the **Intel x86 Emulator Accelerator (HAXM Installer)**, as this will improve the speed and performance of your Android Emulator. Likewise, if you have an AMD processor, you will want

to install the **Android Emulator Hypervisor Driver for AMD Processors**. If your processor does not have support for virtualization, expect this to fail. Click **OK**, read and accept the license agreements, and allow the process to complete.

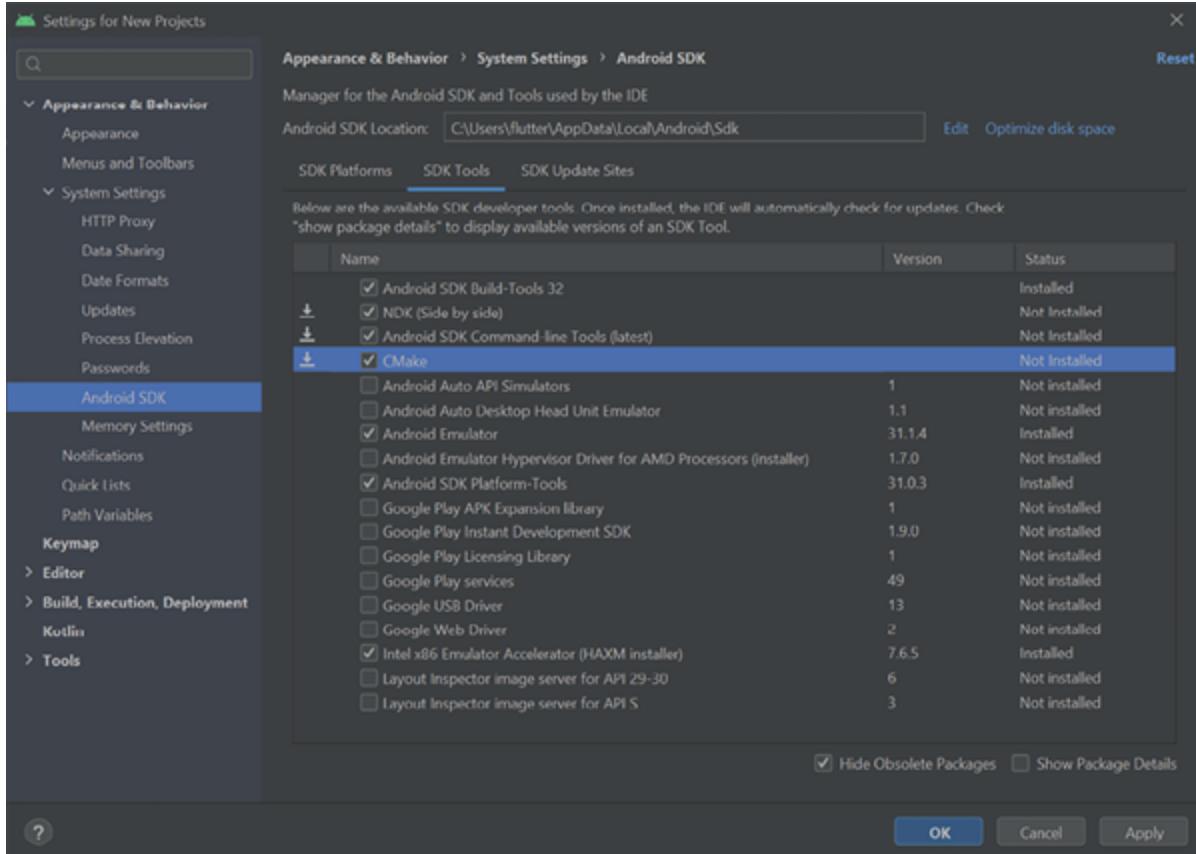


Figure 3.5: Choosing the necessary components to install in Android Studio

Next, we need to configure Windows so that it knows where to find the tools we just installed. Open your **Start** menu and type **Edit the system environment variables** then hit *Enter*. This will open the **System Properties** window where you will find a button near the bottom that says, **Environment Variables....** Tapping this button will open the **Environment Variables** window (*figure 3.6*):

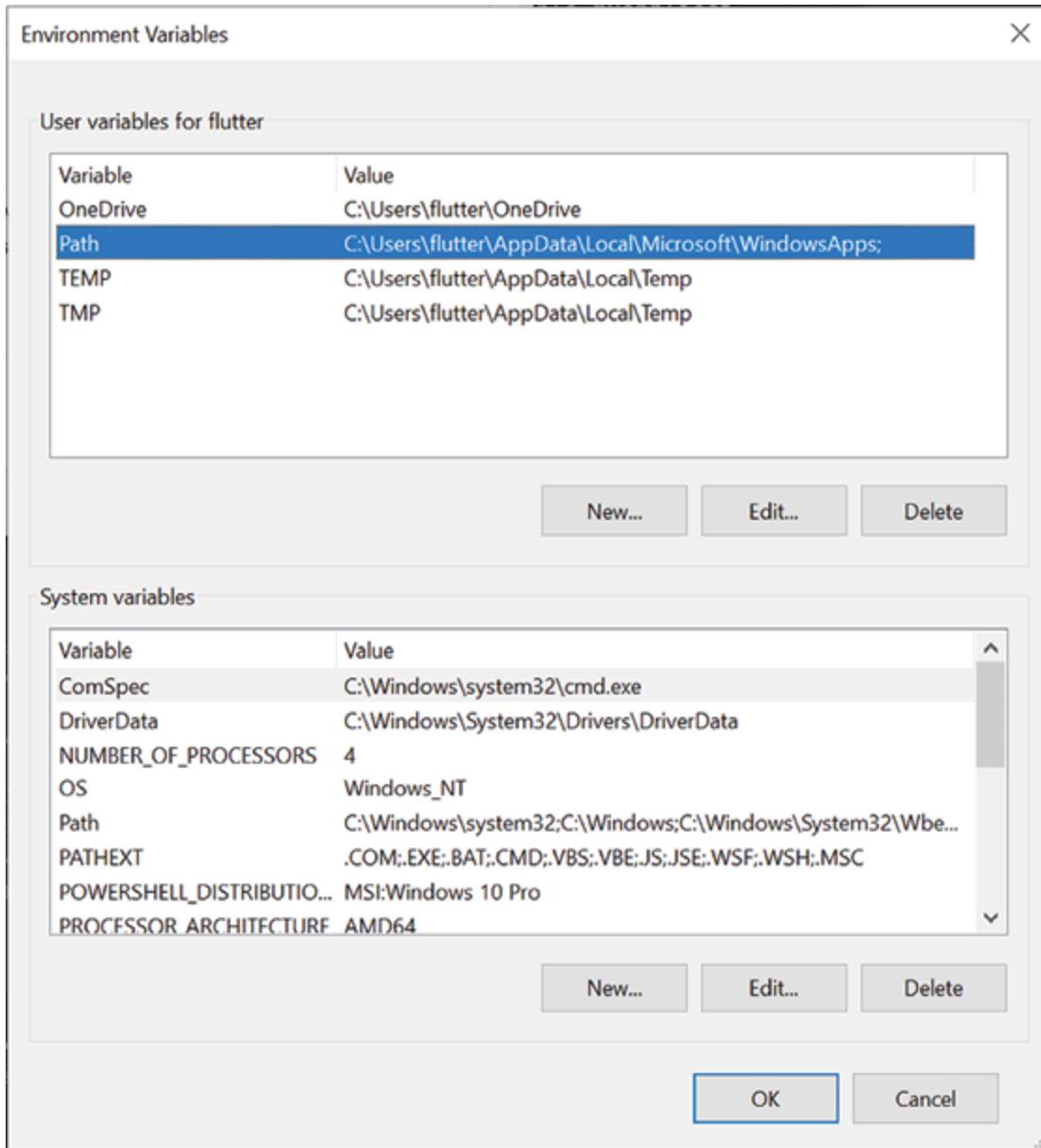


Figure 3.6: The Environment Variables window

In the **Environment Variables** window, look at the upper half where the user variables are. Click on the **Path** variable (highlighted in the preceding screenshot), and then press the **Edit...** button. A new window will appear with a list of filesystem locations in it. We need to add the Android SDK platform tools to this list, so click **New**.

The path we need to add can be found back in Android Studio in the SDK Manager. At the top of the SDK Manager window, you will find **Android SDK Location** with a filesystem path next to it. Copy this path and paste it

into the environment variables window. At the end of the path, add \platform-tools. The full path should resemble the following: C:\Users\<user>\AppData\Local\Android\Sdk\platform-tools

If you have done this correctly, you should see the path in the list of environment variables, such as in the following screenshot (*figure 3.7*):

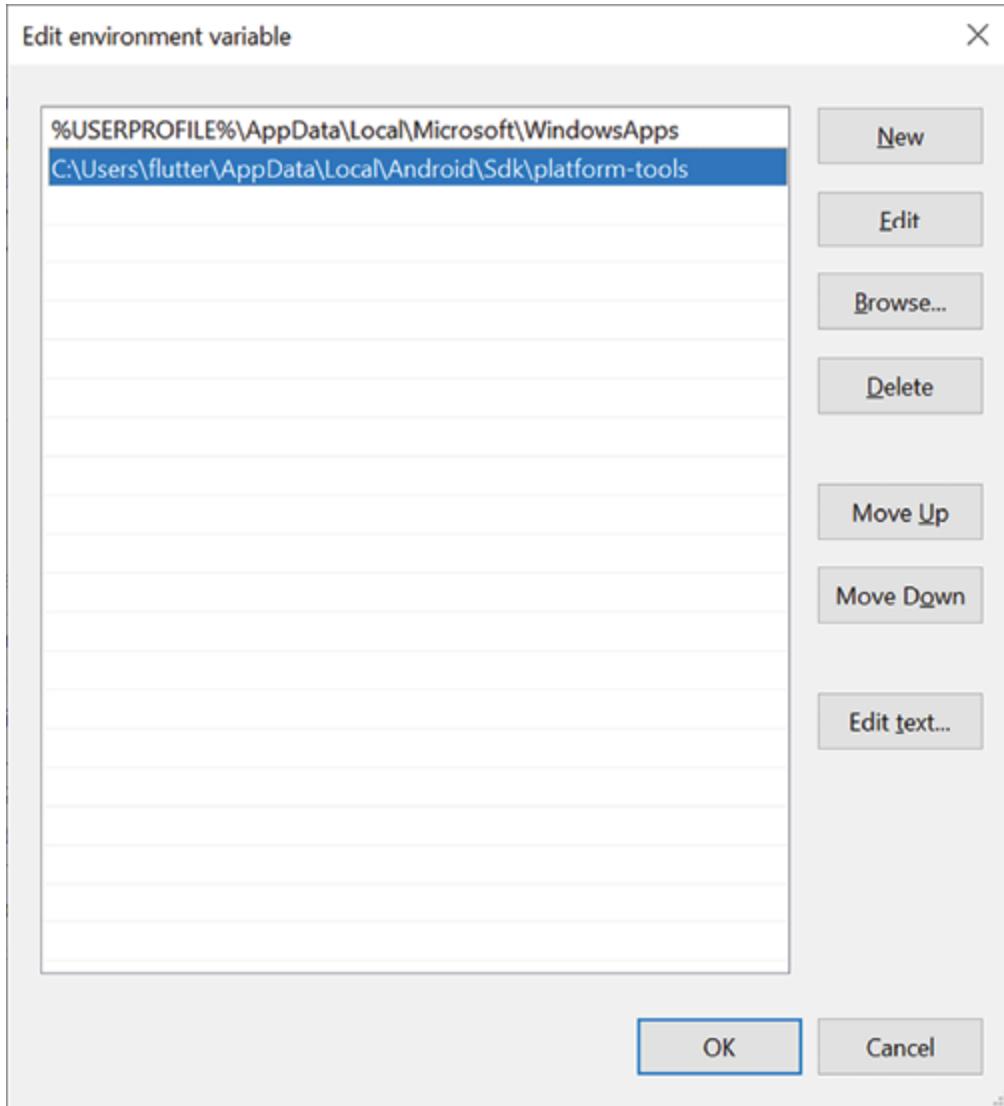


Figure 3.7: The Android SDK platform tools have been added to the Path environment variable

If everything looks good, click **OK** to close the edit environment variables window, then click **OK** again to close the environment variables window. Finally, click **OK** to close the **System Properties** window.

Back in Android Studio, we are going to create an Android Virtual Device (AVD), which we will be using to test and debug our code. Look for the

AVD Manager icon () next to where you found the **SDK Manager** icon (), in the upper right-hand corner. At this point, you will be presented with a window that has a **Create Virtual Device** button on it. Click the button to begin.

The first thing we need to do is choose what device to emulate. Select **Pixel 4** from the list of devices then click **Next**. Next, you will be prompted to choose which version of Android you want to run on this device. Click the **Download** link next to the release name in the list of releases for the latest stable release at the top of the list. When the download is complete, click the **Finish** button, then click **Next**.

On the next screen, click the **Show advanced settings** button and scroll down to find the **Internal Storage** setting. The default 2,048 MB is too small and will cause issues for us down the road, so we are going to need to increase it. Select **GB** from the dropdown list, then set the value to **64**. Finally, click **Finish**.

We are now done setting up and configuring Android Studio and our Android Virtual Device. You will then be returned to the AVD device list, where your new Pixel 4 will be listed. We are done with Android Studio at this point, so close this window and exit Android Studio.

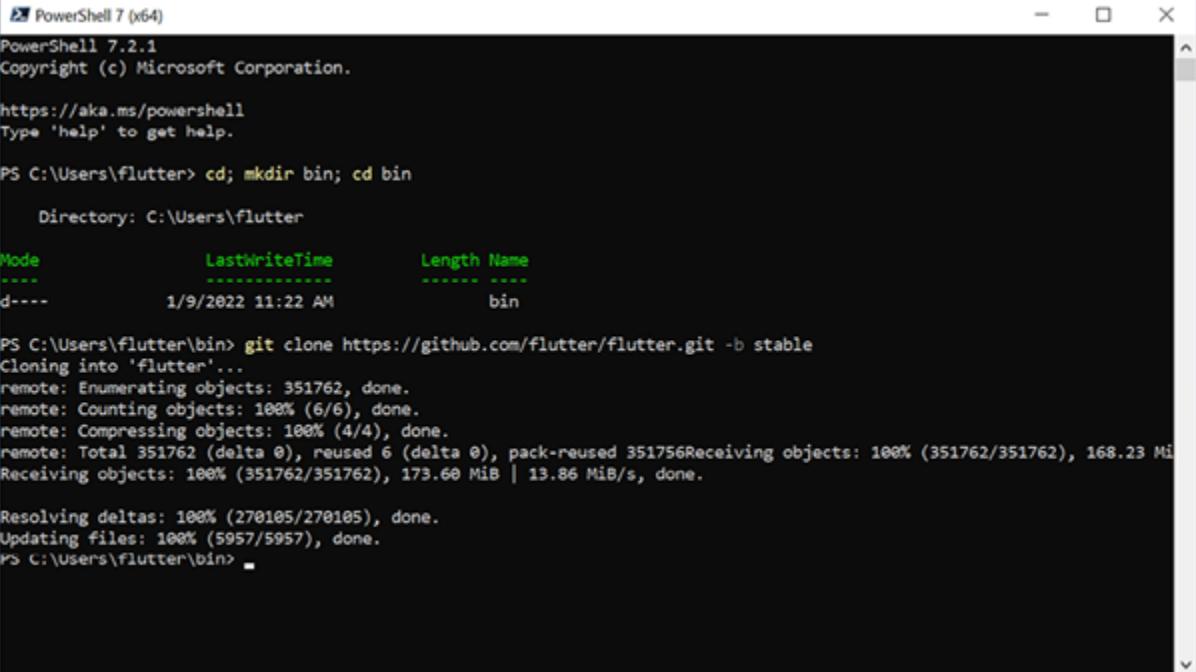
Now that we have all our tools installed, it is time to move on to installing Flutter itself. To install Flutter, we will be using PowerShell. If you still have PowerShell open, you will need to close and re-open it so that our updated **Path** variables are populated.

In PowerShell, we will create a directory where we will install Flutter to. Run the following command to create this directory: `cd; mkdir bin; cd bin`. This will create a directory, `bin`, in your home folder, then navigate to that directory.

Next, we are going to use Git to download the latest version of Flutter. While still in the `bin` directory in PowerShell, issue the following command:

```
git clone https://github.com/flutter/flutter.git -b stable
```

If you have done this correctly, you should see output resembling the following (*figure 3.8*):



A screenshot of a PowerShell window titled "PowerShell 7 (x64)". The window shows the command-line interface for cloning the Flutter repository. The user navigates to the "bin" directory within their Flutter workspace, lists the contents of that directory, and then runs a "git clone" command to download the Flutter source code from GitHub. The output of the "git clone" command shows the progress of the download, including object enumeration, counting, compressing, and receiving objects, along with the final size and speed information.

```
PowerShell 7.2.1
Copyright (c) Microsoft Corporation.

https://aka.ms/powershell
Type 'help' to get help.

PS C:\Users\flutter> cd; mkdir bin; cd bin

    Directory: C:\Users\flutter

Mode                LastWriteTime     Length Name
----                <-----          <----- 
d----       1/9/2022 11:22 AM           bin

PS C:\Users\flutter\bin> git clone https://github.com/flutter/flutter.git -b stable
Cloning into 'flutter'...
remote: Enumerating objects: 351762, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 351762 (delta 0), reused 6 (delta 0), pack-reused 351756Receiving objects: 100% (351762/351762), 168.23 MiB
Receiving objects: 100% (351762/351762), 173.60 MiB | 13.86 MiB/s, done.

Resolving deltas: 100% (270105/270105), done.
Updating files: 100% (5957/5957), done.
PS C:\Users\flutter\bin>
```

Figure 3.8: Successfully cloning the Flutter repository using PowerShell

Cloning the Flutter repository into the `bin` directory will pull down the latest versions of both Flutter and Dart for us. However, before we can use it, we will need to add it to our `Path`. So, just like before, head back to the **Environment Variables**, edit the `Path`, and add the following location to the list: `C:\Users\<user>\bin\flutter\bin`. Be sure to replace `<user>` with your actual username. The easiest way to find this is by referring to the PowerShell prompt. If you have done this correctly, you will see something resembling the following (*figure 3.9*):

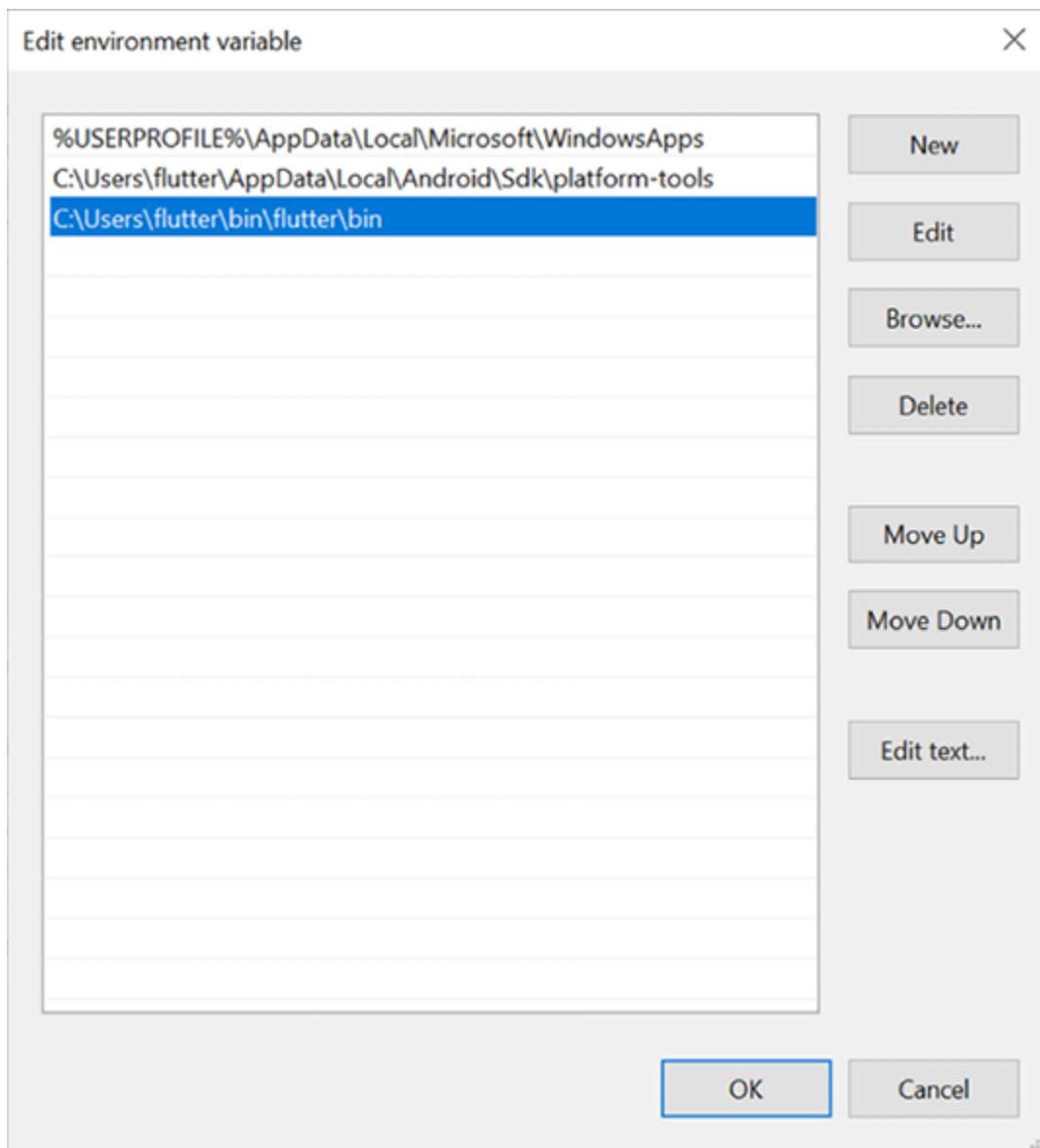
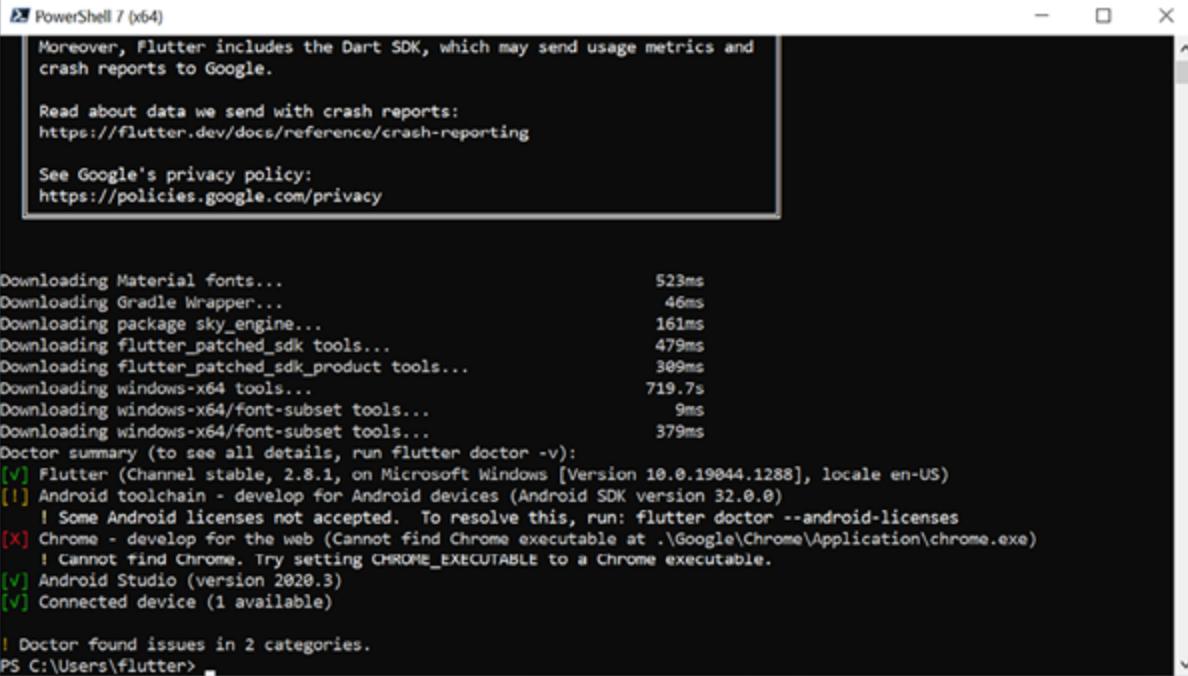


Figure 3.9: Adding Flutter to the Path environment variable

Once you have added Flutter to your **Path** variable, you will need to restart PowerShell to read in the new variables. At this point, you should now be able to run the Flutter command line tools!

The first command we will issue to Flutter is `flutter doctor`. This will download and install any missing components, then run through a series of checks to verify that Flutter and all the necessary components required to build and deploy applications are installed correctly. The first time you run `flutter doctor` will take a while, but any subsequent times you run it will go much faster.

Once `flutter doctor` has finished, a summary will be reported with any issues (and often their solutions) highlighted, similar to the following screenshot, but running a much newer version of Flutter (*figure 3.10*):



The screenshot shows a PowerShell window titled "PowerShell 7 (x64)". The output of the `flutter doctor` command is displayed. At the top, there are three informational messages about data usage and privacy. Below this, the command outputs tool download times and a doctor summary. The doctor summary includes categories for Flutter, Android toolchain, Chrome, and Android Studio, each with a status indicator (green checkmark for Flutter, yellow warning for Android toolchain, red X for Chrome, and green checkmark for Android Studio). A note at the bottom states that the doctor found issues in two categories.

```
Moreover, Flutter includes the Dart SDK, which may send usage metrics and crash reports to Google.

Read about data we send with crash reports:
https://flutter.dev/docs/reference/crash-reporting

See Google's privacy policy:
https://policies.google.com/privacy

Downloading Material fonts...                                523ms
Downloading Gradle Wrapper...                            46ms
Downloading package sky_engine...                         161ms
Downloading flutter_patched_sdk tools...                479ms
Downloading flutter_patched_sdk_product tools...        309ms
Downloading windows-x64 tools...                          719.7s
Downloading windows-x64/font-subset tools...             9ms
Downloading windows-x64/font-subset tools...             379ms
Doctor summary (to see all details, run flutter doctor -v):
[!] Flutter (Channel stable, 2.8.1, on Microsoft Windows [Version 10.0.19044.1288], locale en-US)
[!] Android toolchain - develop for Android devices (Android SDK version 32.0.0)
    ! Some Android licenses not accepted. To resolve this, run: flutter doctor --android-licenses
[X] Chrome - develop for the web (Cannot find Chrome executable at ./Google/Chrome/Application/chrome.exe)
    ! Cannot find Chrome. Try setting CHROME_EXECUTABLE to a Chrome executable.
[!] Android Studio (version 2020.3)
[!] Connected device (1 available)

! Doctor found issues in 2 categories.
PS C:\Users\flutter>
```

Figure 3.10: Flutter doctor has finished running for the first time

The first issue we see is a message about the licenses for the Android toolchain not having been accepted. Thankfully, it gives us the command to run to fix this: `flutter doctor --android-licenses`. Be sure to read the licenses carefully and agree to each of them. Once completed, running `flutter doctor` again should show a green checkmark next to each of the different categories. If you do not have Google Chrome installed, there will be a red X next to the Chrome category; however, this will not impact our ability to test our code on the AVD.

If all has gone well, you now have a fully functioning Flutter installation with all the necessary tools to build and run applications! Later in this chapter, we will install Visual Studio Code, which is the IDE we will be using to write our Flutter code.

Additional reading: If you get stuck anywhere along the way, the Flutter website has always-up-to-date instructions, which may include steps that were not required at the time this book was written. To view those steps,

refer to *Installing Flutter on Windows* (<https://docs.flutter.dev/get-started/install/windows>).

Getting Started on macOS

To write Flutter applications on macOS, we are going to need a couple of tools, which we will look at one-by-one. The tools we are going to install include Xcode and Android Studio, in addition to Flutter itself.

Xcode is Apple's all-in-one utility for developing applications that run on its line of products, from the iPhone and Watch to applications running on macOS and the iPad. Conveniently, it also includes the command line utilities we will need to build Flutter applications.

To install Xcode, you can either download it directly from Apple's developer website (<https://developer.apple.com/xcode/>) or from the App Store by searching for Xcode or by using the following link: <https://apps.apple.com/us/app/xcode/id497799835>. The download will take a while, depending on the speed of your internet connection, as Xcode is quite large.

Once XCode is installed, you will need to launch **Terminal.app**. Use Spotlight (⌘-Space) to search for terminal or find it in /Applications/Utilities. Once the terminal has loaded, you will need to run the following commands one at a time:

```
sudo xcode-select --switch /Applications/Xcode.app/Contents/Developer
```

```
sudo xcodebuild -runFirstLaunch
```

```
sudo xcodebuild -license
```

This will configure your system to use Xcode and allow you to read and agree to the license agreement, all without the need to load Xcode itself. At this point, we are done with Xcode for now, so let us move on.

Android Studio is Google's **integrated development environment (IDE)** used to create Android applications. We will not be using Android Studio to build our Flutter applications (although, that is certainly a possibility) and instead will be using it to install the tools, which will enable us to write, build, and test our applications using whatever IDE we want.

To install Android Studio, you will first need to navigate to <https://developer.android.com/studio> and download the latest version. After agreeing to the licensing agreement, you will be prompted to choose which version of Android Studio is right for you, depending on your Mac's processor. To determine which one to download, go to the Apple menu and select **About This Mac**. The processor line will help you to determine if you have an Intel processor or one of Apple's custom M chips. Continue to download the version of Android Studio that is appropriate for your system. Once the download is complete, mount the disk image, drag the application to the **Applications** folder, and launch Android Studio.

You will likely be prompted by the initial screen to update the plugins via a message at the bottom of the window. Go ahead and do that, allowing Android Studio to update its components and restart itself to complete the process. If you are prompted to import settings from a previous install, choose not to import anything. Eventually, you will find yourself at a **Welcome to Android Studio** window.

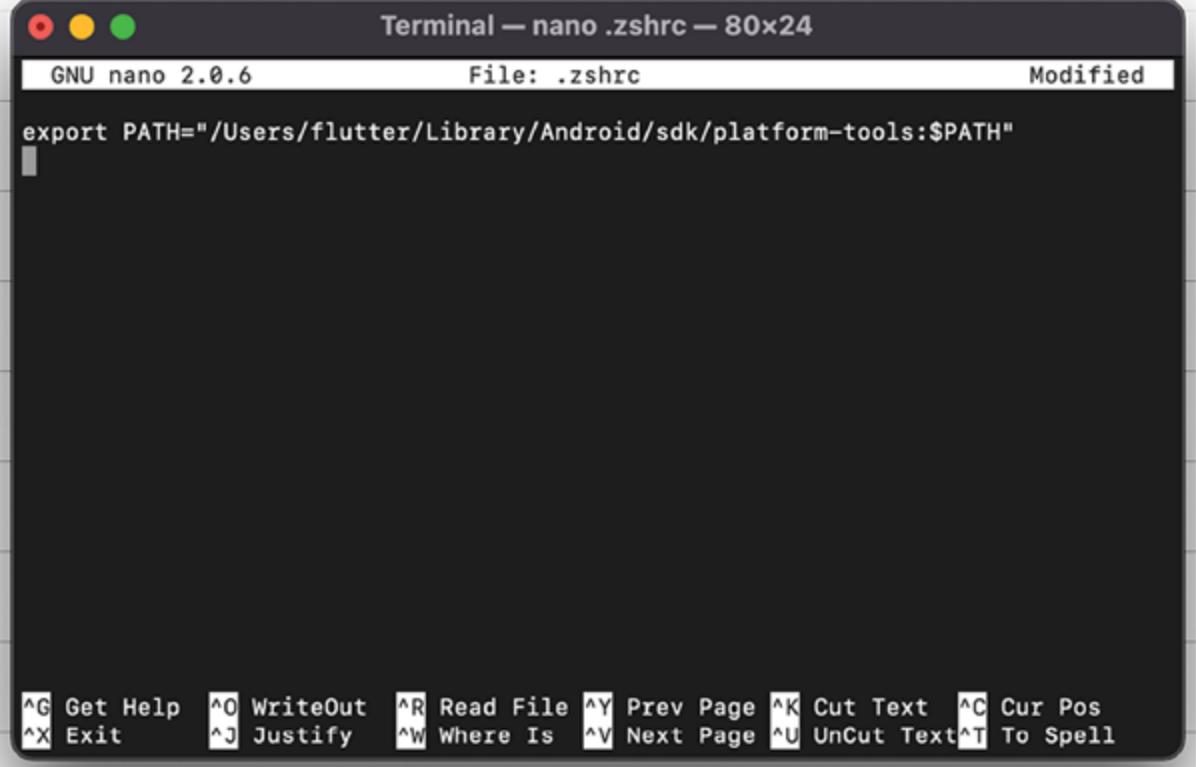
On the welcome window, look for the **More Actions** link and from the dropdown which appears, choose **SDK Manager**. Here, we will need to first make a note of the Android SDK Location at the top of the window. Momentarily, we will be using this to configure our system. For now, however, choose the **SDK Tools** tab. Check the boxes for **NDK (Side by side)**, **Android SDK Command-line Tools (latest)**, **CMake**, and if you are using a Mac with an Intel processor, also choose the **Intel x86 Emulator Accelerator (HAXM installer)**. Click **Apply**, accept the license agreements, and allow these components to download.

Meanwhile, head back into the terminal. We are going to take that Android SDK Location path that you took note of earlier and tell our system to use it to find the tools, which Flutter will need to work. So, in the terminal, type the following:

```
nano ~/.zshrc
```

This will open a file within a command-line-based text editor (*figure 3.11*). If the file is empty, simply add the following line using the Android SDK Location as your guide:

```
export PATH="/Users/<user>/Library/Android/sdk/platform-tools:$PATH"
```



The screenshot shows a terminal window titled "Terminal — nano .zshrc — 80x24". The window title bar includes the application icon, the title, and the dimensions. The status bar at the top right shows "GNU nano 2.0.6", "File: .zshrc", and "Modified". The main text area contains the command: `export PATH="/Users/flutter/Library/Android/sdk/platform-tools:$PATH"`. At the bottom of the window, there is a menu of keyboard shortcuts:

<code>^G</code>	Get Help	<code>^O</code>	WriteOut	<code>^R</code>	Read File	<code>^Y</code>	Prev Page	<code>^K</code>	Cut Text	<code>^C</code>	Cur Pos
<code>^X</code>	Exit	<code>^J</code>	Justify	<code>^W</code>	Where Is	<code>^V</code>	Next Page	<code>^U</code>	UnCut Text	<code>^T</code>	To Spell

Figure 3.11: Adding the Android SDK platform-tools to the PATH variable

Note that you will be replacing <user> with your username so that it matches the Android SDK Location and that you are *adding* `platform-tools` to the end of the path. Also, note that this is case-sensitive.

If there is already something in this file, find the line which starts with `export PATH=` and add the Android SDK Location (including the `/platform-tools` part) to the end of the line, just before the `:$PATH`. Each directory in this `PATH` variable will be separated by a colon (:), so double-check that you have done it correctly.

Once you have edited the file and updated the `PATH` variable, press `^O` (Control-O) to write the file to the disk, pressing *Return* when it prompts for the filename. Press `^X` (Control-X) to exit the editor. Quit and restart your terminal to read in the new variables.

You can verify that the changes have taken effect by typing `which adb`, which should return the path to the `adb` utility. If there is an error, chances are your

terminal is not configured to use the `zsh` shell. To fix this, type `chsh -s /bin/zsh` and restart the terminal.

Back in Android Studio, close the **SDK Manager** window to return to the welcome screen. Under the **More Actions** menu, locate the **AVD Manager** (*figure 3.12*). This will pop-up a window that will allow us to create an **Android Virtual Device (AVD)**. Click the button to create a device.

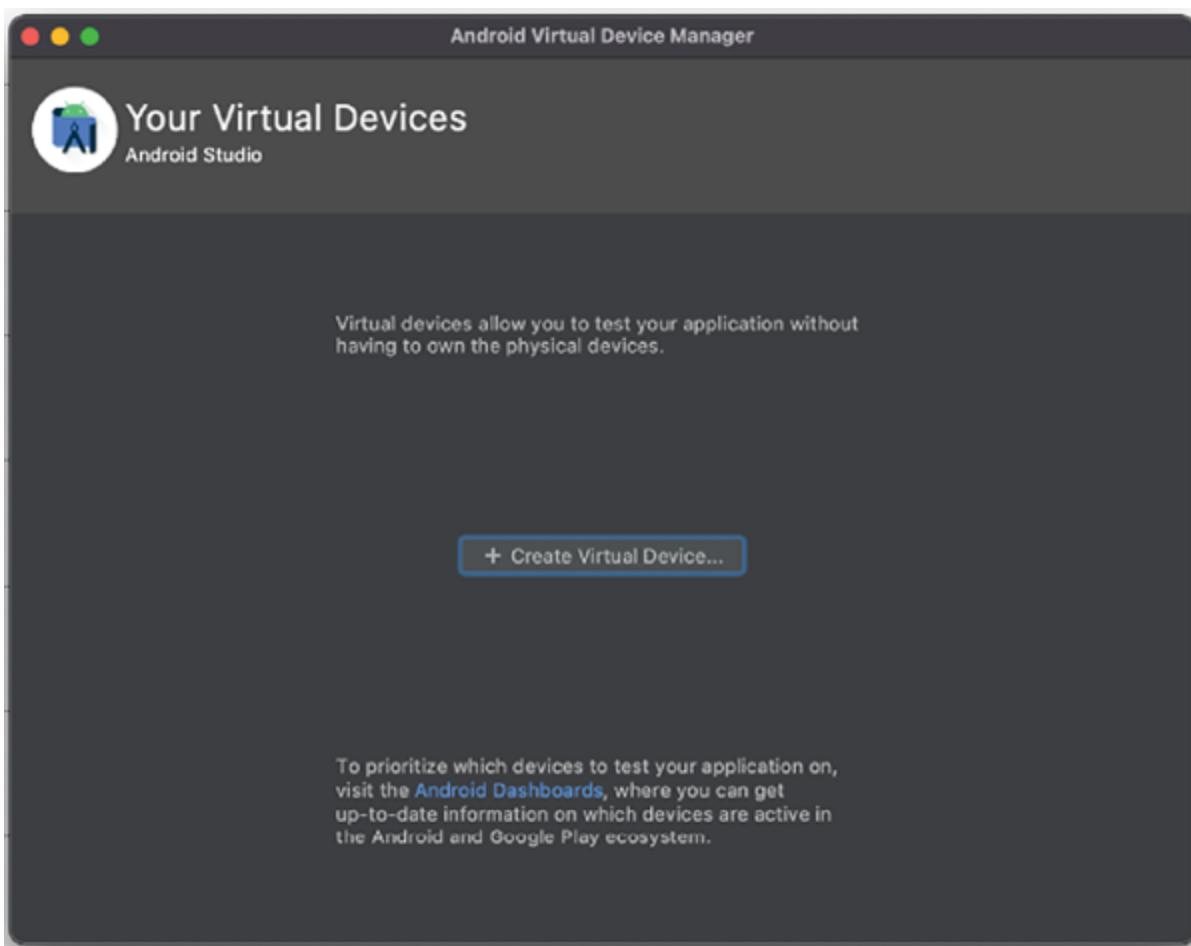


Figure 3.12: The AVD Manager

The device we are going to emulate is Google's Pixel 4 phone. Select it from the list and click **Next**. We will then be prompted to choose a system image. Click the **Download** link next to release with the highest API level. This will download an Android system image with the latest version of Android for us to use on our AVD. When it is done downloading, click the **Finish** button, then click **Next**.

By default, the AVD will be created with only 2 GB storage, which will cause us problems as we develop and test our applications. To remedy this, click the **Show Advanced Settings** button and scroll down to find the **Internal Storage** setting. Change the dropdown from **MB** to **GB**, then set the value to **16**. That will give us 16 GB of internal storage to work with, which should be more than sufficient (*figure 3.13*):

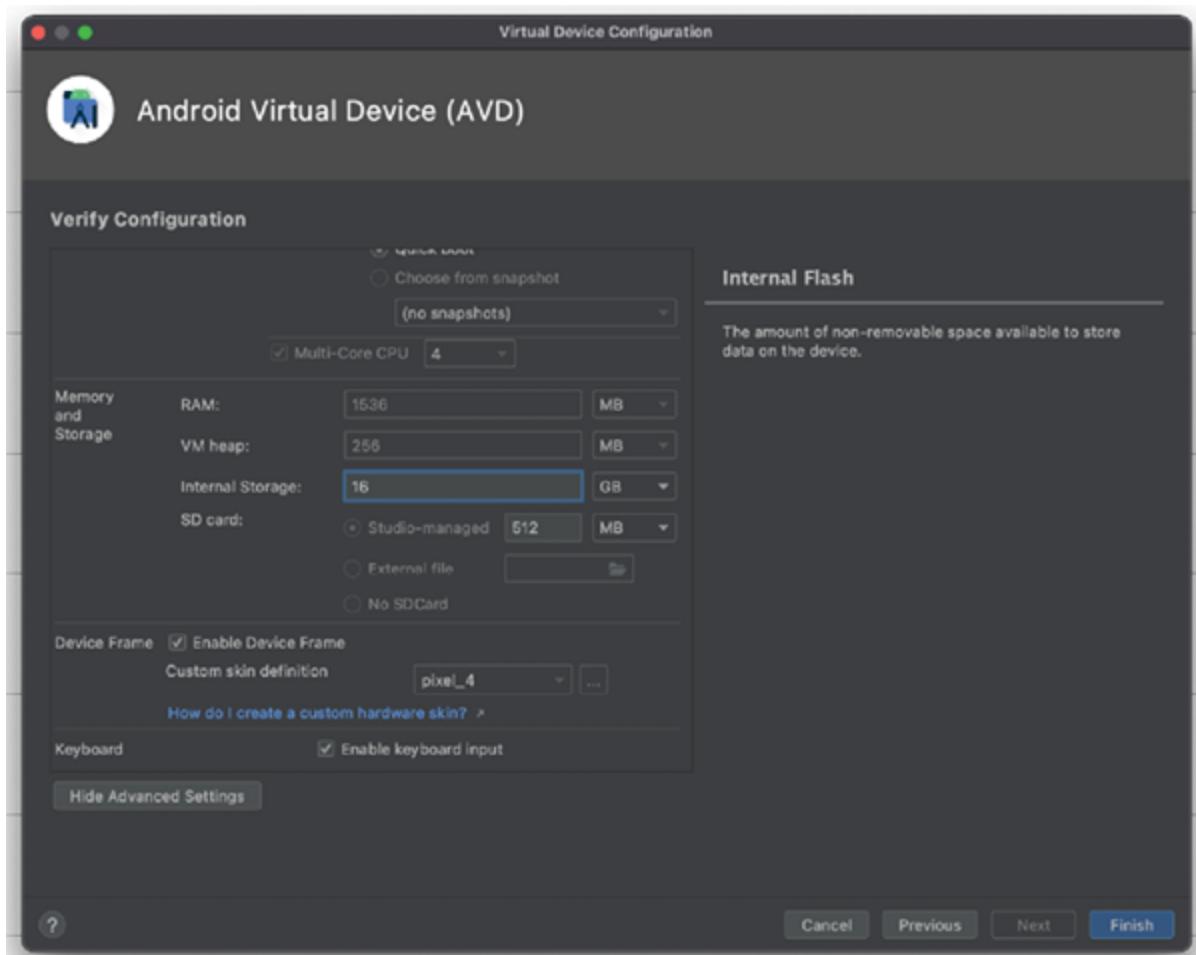


Figure 3.13: Adjusting the internal storage size of your new AVD

Click on **Finish** to create your AVD, then close the AVD Manager window and exit Android Studio.

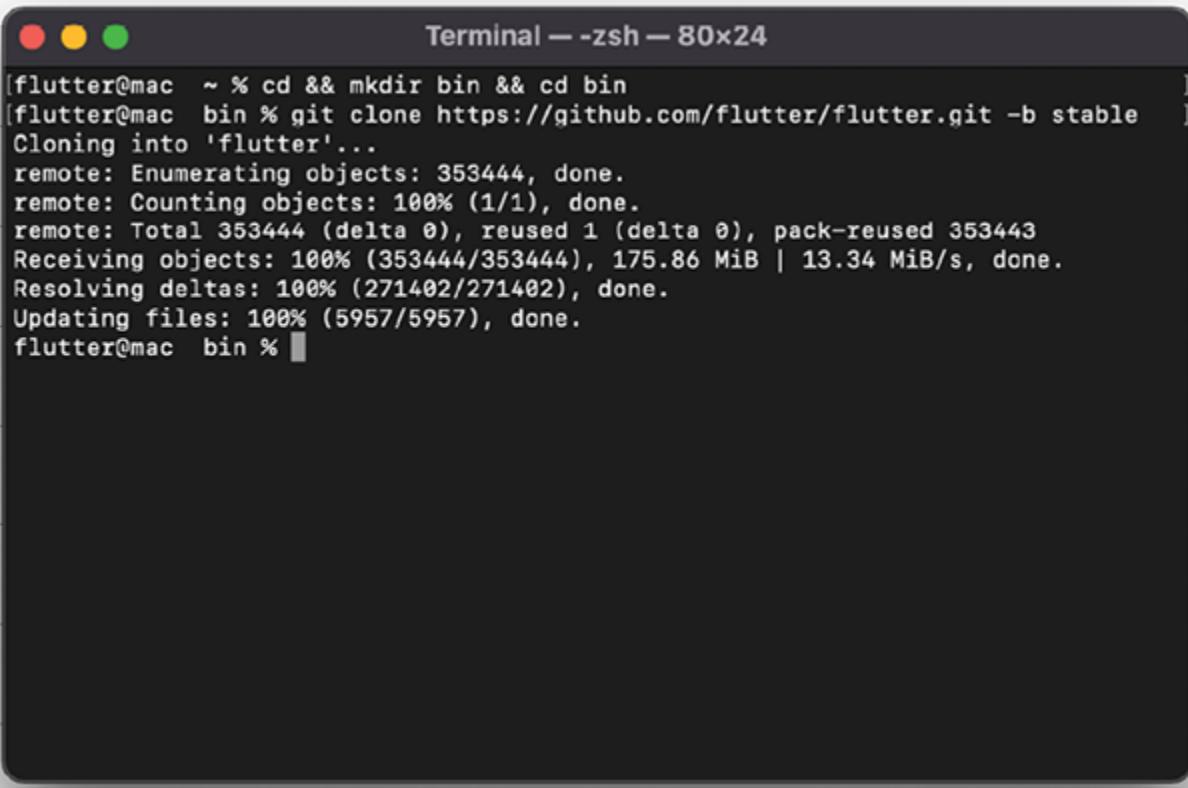
It is time to install Flutter. We will do this in the terminal, too. First, we need a place to install Flutter to, so we are going to create a directory where we will keep it. Run the following command to do that:

```
cd && mkdir bin && cd bin
```

Next, we will pull down the latest version of Flutter using Git with the following command:

```
git clone https://github.com/flutter/flutter.git -b stable
```

This will install the latest version of Flutter to `/Users/<user>/bin/flutter` (*figure 3.14*):



The screenshot shows a macOS Terminal window titled "Terminal — zsh — 80x24". The window contains the following text output from a git clone command:

```
flutter@mac ~ % cd && mkdir bin && cd bin
[flutter@mac bin % git clone https://github.com/flutter/flutter.git -b stable
Cloning into 'flutter'...
remote: Enumerating objects: 353444, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 353444 (delta 0), reused 1 (delta 0), pack-reused 353443
Receiving objects: 100% (353444/353444), 175.86 MiB | 13.34 MiB/s, done.
Resolving deltas: 100% (271402/271402), done.
Updating files: 100% (5957/5957), done.
flutter@mac bin %
```

Figure 3.14: Installing Flutter via the command line

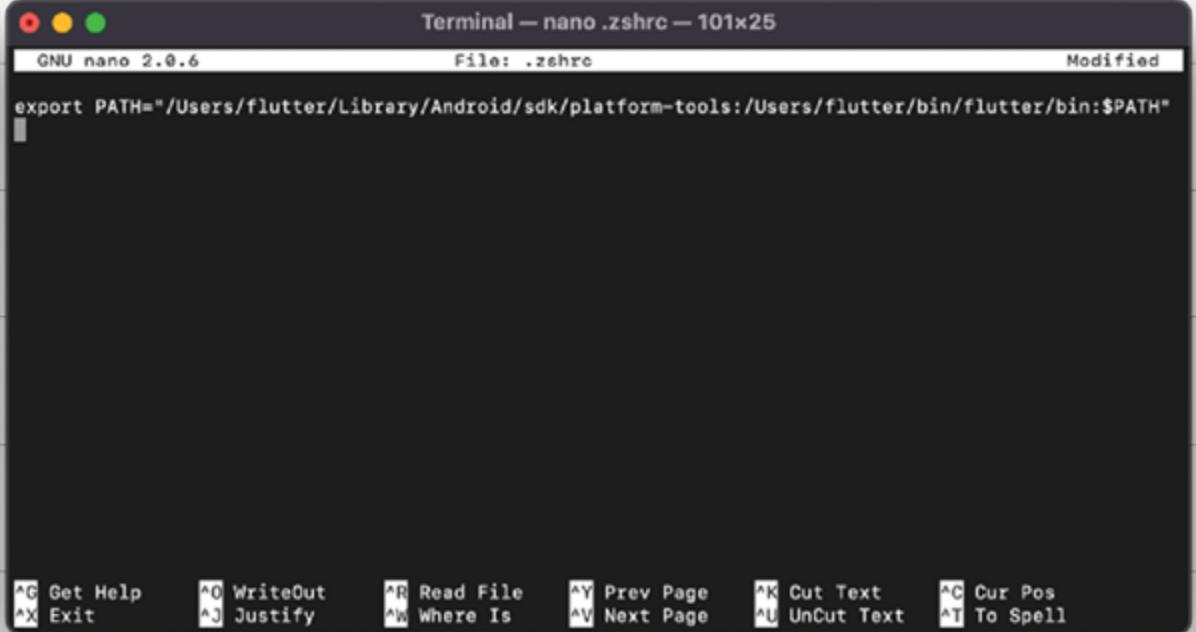
We are going to need to edit our `PATH` variable again so that we can use it, though. Just like before, open up `.zshrc` in `nano`:

```
nano ~/.zshrc
```

This time, you are going to add `/Users/<user>/bin/flutter/bin` to the `PATH`, replacing `<user>` with your username. If you are uncertain what your username is, you can run the `whoami` command. Make sure that you insert a colon (`:`) between the different paths in the `PATH` variable. Save the file and quit `nano` when you are finished. Your `PATH` should look something like the following in your `.zshrc` file (*figure 3.15*):

```
export PATH="...]/platform-tools:/Users/<user>/bin/flutter/bin:$PATH"
```

Like before, you will need to restart your terminal for these changes to take effect. You can test to make sure that Flutter is found in your `PATH` by running `which flutter`. If the command returns the path to Flutter, you are good to go.



```
Terminal — nano .zshrc — 101x25
GNU nano 2.0.6          File: .zshrc          Modified
export PATH="/Users/flutter/Library/Android/sdk/platform-tools:/Users/flutter/bin/flutter/bin:$PATH"

```

^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos
^X Exit ^J Justify ^W Where Is ^V Next Page ^U UnCut Text ^T To Spell

Figure 3.15: Adding Flutter to the PATH variable

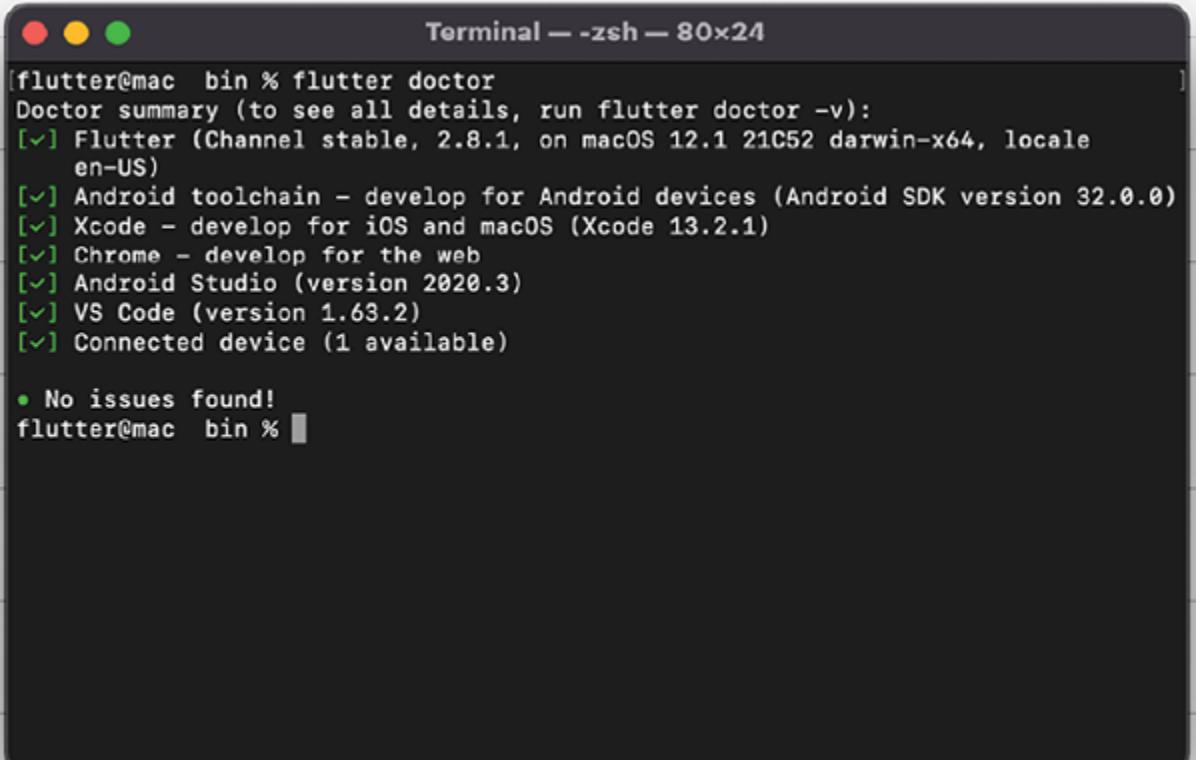
The next step is to run the `flutter doctor` command. This will download and install the necessary components for Flutter. It will also alert you to any issues which could prevent Flutter from working properly. At this point, you are likely to see a couple of different issues. First, you should be prompted to accept the Android licenses. To do this, run the following command:

```
flutter doctor --android-licenses
```

Next, it is likely that you will run into an issue with Xcode in that CocoaPods is not installed. We can fix this by running the following command:

```
sudo gem install cocoapods
```

If all goes well, running `flutter doctor` should not recommend any additional steps (*figure 3.16*). However, if you are using a Mac with Apple Silicon (M1, M2, and so on), you will need to perform a few additional steps.



```
[flutter@mac bin % flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 2.8.1, on macOS 12.1 21C52 darwin-x64, locale
en-US)
[✓] Android toolchain - develop for Android devices (Android SDK version 32.0.0)
[✓] Xcode - develop for iOS and macOS (Xcode 13.2.1)
[✓] Chrome - develop for the web
[✓] Android Studio (version 2020.3)
[✓] VS Code (version 1.63.2)
[✓] Connected device (1 available)

• No issues found!
flutter@mac bin % ]
```

Figure 3.16: Flutter doctor completes without errors

[Additional Information for Devices with Apple Silicon \(M1, M2, and so on\)](#)

If you are on macOS and using one of Apple's newer devices which has an M1, M2, or later chip, you are bound to run into some issues getting Flutter working. To remedy this, we need to install the x86 versions of CocoaPods and `ffi`. First, you will need to install Rosetta 2 using the following command:

```
sudo softwareupdate --install-rosetta --agree-to-license
```

Next, you will need to remove the existing CocoaPods and `ffi`, and replace them with their x86 counterparts using the following commands:

```
sudo gem uninstall ffi
```

```
sudo gem uninstall cocoapods
```

```
sudo arch -x86_64 gem install cocoapods
```

```
sudo arch -x86_64 gem install ffi
```

Any time you need to use CocoaPods and you find that something is not working quite right, try prepending the command with `arch -x86_64` and running it again, such as with the following example:

```
arch -x86_64 flutter run
```

Installing and Configuring Visual Studio Code

We are going to be using Visual Studio Code to develop our Flutter applications for the duration of this book. Whether you are on macOS or Windows, you can find the latest version on their website: <https://code.visualstudio.com/>. Download and install the latest version.

Once you have it installed, run the program. Visual Studio Code does not, by default, have any idea what Flutter or Dart is, which certainly makes developing applications difficult. Luckily, Visual Studio Code is highly extensible, with a thriving extensions marketplace. Extensions to support Flutter and Dart have been created for Visual Studio Code, and we will need to install them. To accomplish this, look for the **Extensions** icon on the left-hand side of the window, or press *Ctrl + Shift + x* on Windows or *⌘ + Shift + x* on macOS (*figure 3.17*):

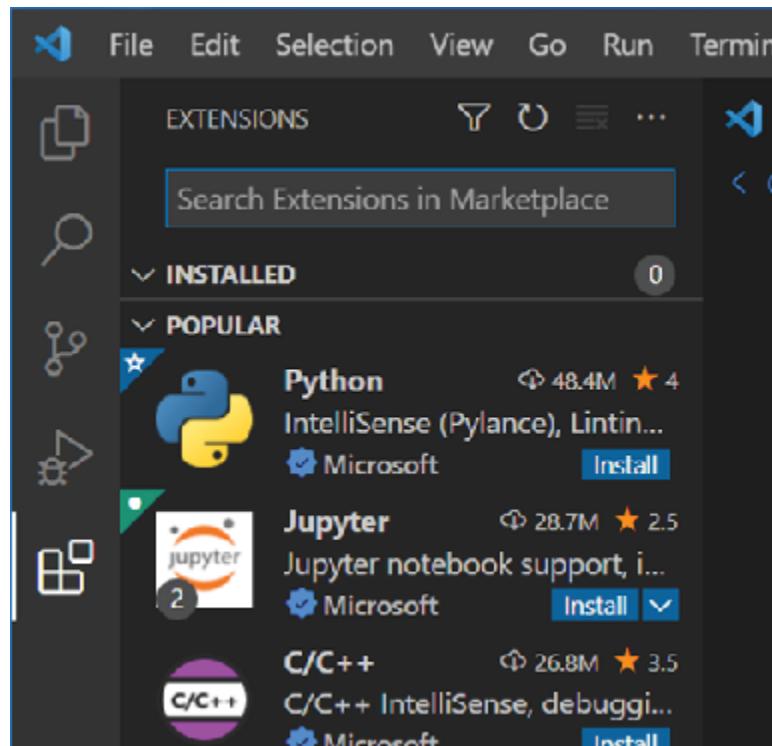


Figure 3.17: The extensions view in Visual Studio Code

Search the extensions marketplace for Flutter (the author is *Dart Code*) and press the blue **Install** button. This will install both the Flutter and Dart extensions, which you can verify by searching for `@installed`. There are many great extensions in the extension marketplace, so feel free to install any other extensions you believe might be useful while you are in there!

Congratulations! You now have a development environment set up for Flutter development! Now, let us create a new Flutter project and get it running in our AVD.

Building and Running Your First Flutter Application

Visual Studio Code has a built-in command line that we can use to create our Flutter project. To invoke it, press `Ctrl + `` or find the **Terminal** option in the **View** menu. First, we will want to create a directory in which we will keep all our projects. Run `cd; mkdir development; cd development` to create a development directory and navigate into it.

Next, we will create our Flutter application. Run `flutter create my_application` and let Flutter do its magic. At this point, we are finished with the terminal. Feel free to close it by clicking the **X** button on the top-right corner of the terminal. Now, we need to open our project in Visual Studio Code. On the left-hand side of the Visual Studio Code window, locate the top-most icon to navigate to the **Explorer** view. You will be prompted with two options: **Open Folder** and **Clone Repository**. Click the **Open Folder** button, navigate to your `home` folder, double-click the `development` folder, double-click again on the `my_application` folder, then press the **Select Folder** button.

Visual Studio Code will prompt you with a message regarding trust. This is a safety mechanism intended to prevent malicious code repositories from tampering with your system. Since we know where our code came from and we trust that it is safe, we will press the **Yes, I trust the authors** button.

A small dialog will appear at the bottom of the window, asking us if we want to use the recommended settings for Dart, which we do. Click the button to accept the changes to your configuration. Your workspace should now look like the following (*figure 3.18*):

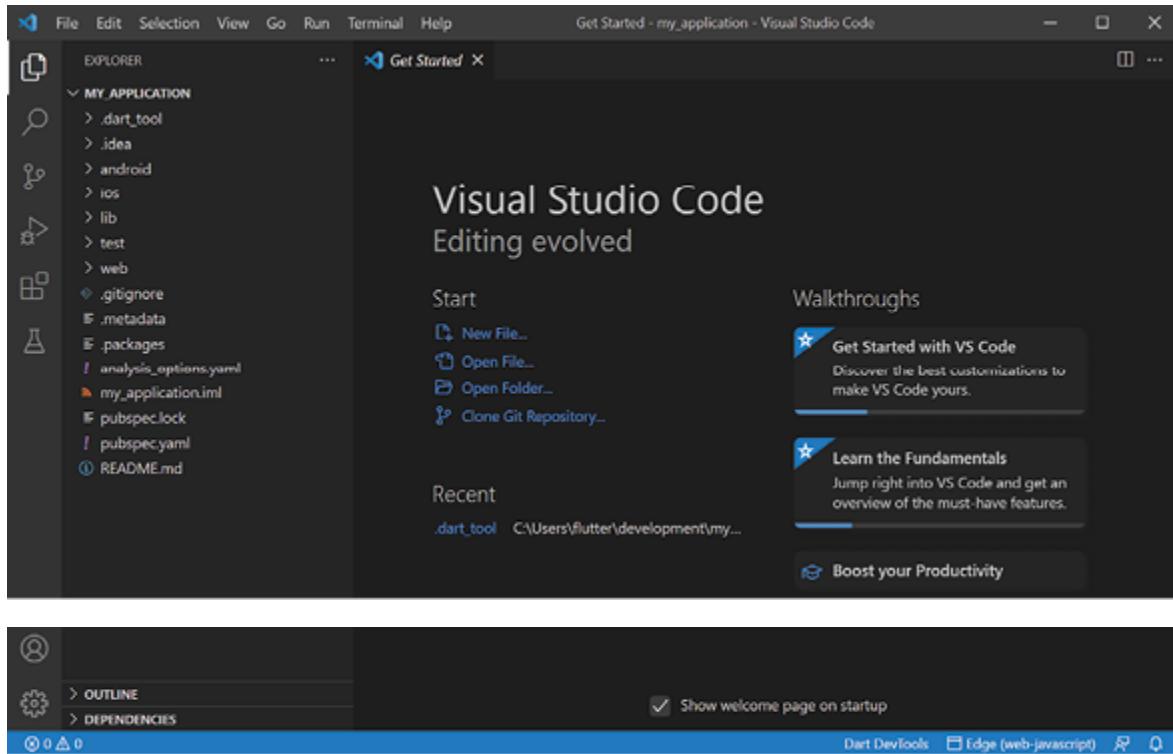


Figure 3.18: Visual Studio Code with a Flutter project open

Let us talk about some of the key files and directories in this list view, to understand what we will be working with:

- The `android`, `ios`, and `web` directories contain platform-specific configuration which is used when deploying your Flutter project to that platform.
- The `test` directory contains all the tests that you will be writing for your code and is consumed by the `flutter test` command.
- The `lib` directory contains all the source code for our application and is where we will be spending a lot of time throughout the rest of this book.
- `pubspec.yaml` is where we will add information about the application name, description, and version number. It is also where we will add any dependencies that our application relies upon, such as images, fonts, and packages.

To run our code, we will need to open the AVD we created earlier. There are a couple of ways to do this. You can either press `Ctrl + Shift + p` (on Windows) or `⌘ + Shift + p` (on macOS) to invoke the command palette and

search for **Flutter: Select Device** or you can look at the bottom-left of the window, to the right of the Flutter version number. Depending on your system, you may see something which says, **No device**, or perhaps you will see something like **Edge (web-javascript)** as highlighted in the following screenshot (*figure 3.19*):



Figure 3.19: The device selection option in Visual Studio Code

Either option will open a selector at the top-middle of the window. The option we are looking for is **Start Pixel 4 API x**, where x is the version of the API used with your chosen version of Android that is installed on the device. Selecting this option will launch the AVD and automatically connect to it.

Wait for your AVD to finish booting into Android and allow it to complete any initial setup. Once it is ready, navigate back to Visual Studio Code and find the **Run and Debug** icon on the left-hand side of the window, which looks like a play button with a little bug on it. On this screen, below the **Run and Debug** button, click the link to **Create a launch.json file**, choosing **Dart & Flutter** when prompted. This will automatically configure Visual Studio Code with the necessary settings to run our application and will automatically open the configuration file in our editor.

The final step is to press the green **Play** button to compile and run our application. The first time you compile your application will take the longest, as Flutter will need to download any necessary components and compile your entire application. When this process is complete, Flutter will automatically install the application on your AVD, launch it, and connect to

it via the debugger. If all goes well, you have just run your first Flutter application (*figure 3.20*)!

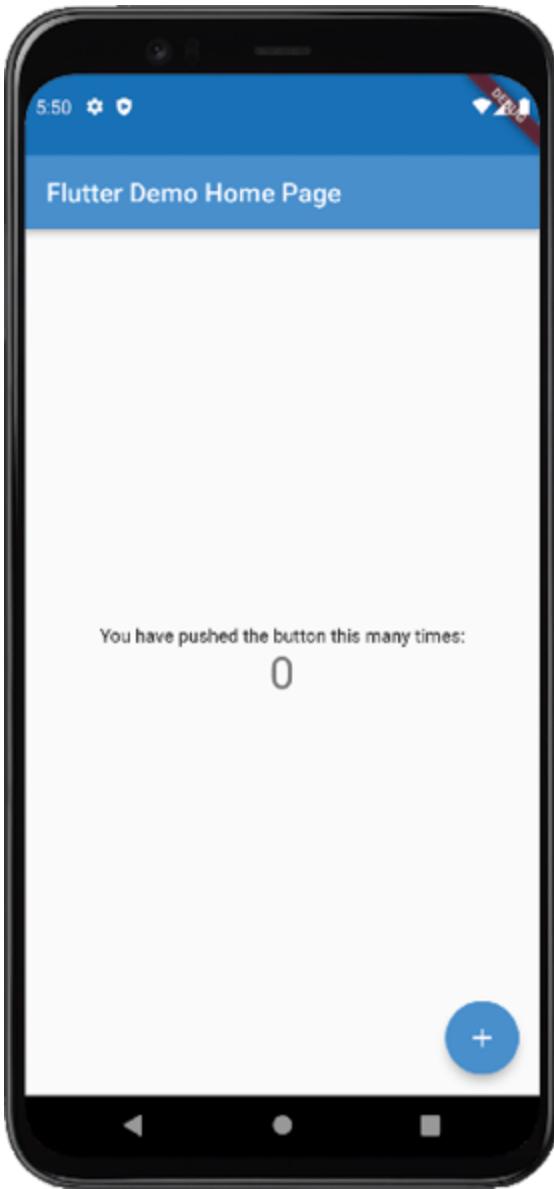


Figure 3.20: The Flutter demo running on an Android Virtual Device

Conclusion

There is a fair amount of setup work involved in getting Flutter running. Luckily, we only need to do it once. The feeling of watching your first Flutter application come to life in front of you can be magical, too.

In the upcoming chapter, we are going to look at the code for the demo application to get an idea of what is going on. We are also going to talk

about widgets, more generally speaking. (Like, what even *is* a widget, anyway?) Plus, we are going to make some widgets of our own. You are going to love this, so turn the page and get started!

Further Reading

- Although it is outside the scope of this book, PowerShell is a very powerful command line. For more information on using PowerShell, refer to the PowerShell Documentation: <https://aka.ms/powershell>
- Git is an extremely powerful piece of software which is used extensively in the software development industry. Throughout your journey, you will undoubtedly have questions about it. A good place to start reading is GitHub's introduction to using Git: <https://docs.github.com/en/get-started/using-git/about-git>
- Developing with Flutter on Apple Silicon from Flutter's GitHub Wiki: <https://github.com/flutter/flutter/wiki/Developing-with-Flutter-on-Apple-Silicon>

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

Introduction to Widgets

Introduction

Widgets are the building blocks of every Flutter application and, as a developer, you will spend a great deal of time composing and implementing within every Flutter application you work on.

In this chapter, we will discuss the different types of widgets and explore some of the most used ones that you will run across, look at our sample application and explore each of the widgets it is composed of, and briefly discuss the architecture of Flutter to understand the widget lifecycle.

Structure

In this chapter, we will discuss the following topics:

- Briefly exploring the default Flutter application
- What is a widget and how is it drawn to the screen?
- Stateless widgets
- Stateful widgets
- Inherited widgets
- Introduction to BuildContext
- Exploring and modifying the Flutter demo

Objectives

After completing this chapter, you will be able to describe how widgets are converted from code to elements painted on the screen. You will also be able to explain the difference between stateless, stateful, and inherited widgets. You will learn how to implement and then customize several common widgets, and then you will learn how to create your own widgets from scratch.

Briefly Exploring the Default Flutter Application

Take a moment to play around with the application you have just gotten running from the last chapter. It is a very simple application, with only a single button. Tapping the button increases the counter at the center of the screen. Let us look at the code to see what is going on. Do not worry if none of it makes sense – that is why you are reading this book!

Back in Visual Studio Code, you should see a list of files on the left side of the window. If not, look for the vertical bar of icons and click the one on the top which looks like a couple of stacked pages. Right now, we can ignore everything except a folder named `lib`, which we can expand to find the star of the show: `main.dart`. Double click `main.dart` to open it in the editor.

The Flutter team has done a great job of documenting most of the code, so it is worth reading over their comments. (Comments in Dart are typically written as a line of text preceded by `//`). There are a few things here that we are going to look at that are not covered by the comments. First, you will notice these lines at the top:

1. `void main() {`
2. `runApp(const MyApp());`
3. `}`

Although seemingly unassuming, these are the most important lines of any Flutter program. Without them, your application would not work. The first line tells Dart where to start executing your code. This is called the *entry point*. Dart always knows to start a program from the `main()` function.

Anything that your program does that is not inside of the `main()` function will have been called by something else that *was* in the `main()` function.

The next line has the `runApp()` function. This function tells Flutter where to start running its code. Flutter is written in the Dart programming language, which can be used to create far more than just Flutter applications. Sometimes, your application will execute Dart code before the Flutter code is ever invoked. The `runApp()` function has a single parameter: the *Widget* which will be the root of your application. We will get into what widgets are shortly, but there's one last thing you will want to know.

If you scroll down in the file a bit, you will find a line that says `return Scaffold()`. Just below that, you will find a line that says `body: Center()`. As we move into this chapter, you will undoubtedly want to follow along. You can do so by selecting the `Center` widget and deleting it. Then, you will put the examples where the `Center` widget once was. Deleting the widget will require you to find the corresponding closing parenthesis later in the file, and then delete everything in between. The closing parenthesis is helpfully labeled by Visual Studio Code, so it should be easy to find. Select `Center()` and everything up to the closing parenthesis, then delete it. However, you should leave the `body:` at the start. What we're looking for is as follows:

1. `return Scaffold(`
2. `appBar: AppBar(`
3. `// Here we take the value from the MyHomePage object that
 // was created`
4. `// by the App.build method, and use it to set our appbar title.`
5. `title: Text(widget.title),`
6. `),`
7. `body:`
8. `floatingActionButton: FloatingActionButton(`

Any time you want to try out an example in this chapter, you will put the code just to the right of the `body:` section. If that feels overwhelming right now, do not worry. You are not expected to understand anything, yet! You

soon will, though. Just as soon as we talk about what the heck a widget even is.

What is a Widget and How is it Drawn to the Screen?

Broadly speaking, a widget could be anything; it could be a box, an ink pen, a shoelace. What these three items have in common is that they are all small, simple, and purpose-built. They each do one thing, and they do it well.

Widgets in Flutter are also small, simple, and purpose-built. You will find widgets such as `TextArea`, `Container`, and `Column`. Each of these widgets has one job: provide a user with a place to enter text, hold another widget, or create a vertical stack of widgets, respectively.

Flutter uses an *aggressive composition* design paradigm. In this paradigm, simplicity is key. Rather than directly styling a widget (adding padding, shifting its position, rotating it, and so on) directly in the properties of that widget, a set of multiple widgets would build up a *widget tree* out of individual, purpose-built widgets.

Even some of the most complex widgets, such as the `Material` widget, are built using other widgets. (In the case of `Material`, an `AnimatedDefaultTextStyle`, `NotificationListener`, and `AnimatedPhysicalModel` are used.) So, understanding widgets is vital for understanding Flutter. To fully understand widgets, we need to talk about how widgets are rendered to the display.

Each widget has a `build()` method. When Flutter wants to render a widget, `build()` is called, the state of the application is evaluated, and a widget is returned. The widget which is returned could be a simple widget with no children, or it could have one or more children which each have its own `build()` method and potential child(ren). This process happens continually, always evaluating the current state of the application and returning new widgets where necessary. In almost every case, the returned widget(s) will be composed of other, more primitive widgets.

Collectively, these widgets are assembled into a *widget tree*, which Flutter uses to build its *element tree* and eventually its *render tree*. We are going to

explore the element and render trees later in this chapter, so for now let us look at the widget tree using the following example:

```
1. Container(  
2.   color: Colors.lightBlue,  
3.   child: Row(  
4.     children: const [  
5.       Image(  
6.         image: NetworkImage(  
7.           'https://flutter.github.io/assets-for-api-  
     docs/assets/widgets/owl.jpg'),  
8.         ),  
9.       Text('Flutter is fun!'),  
10.    ],  
11.  ),  
12. )
```

This code will create a `Container` widget, colored light blue, with a row of widgets contained within. The `Row` consists of two widgets: an `Image` widget and a `Text` widget. Once inserted into our application, this is what we will see (*figure 4.1*):

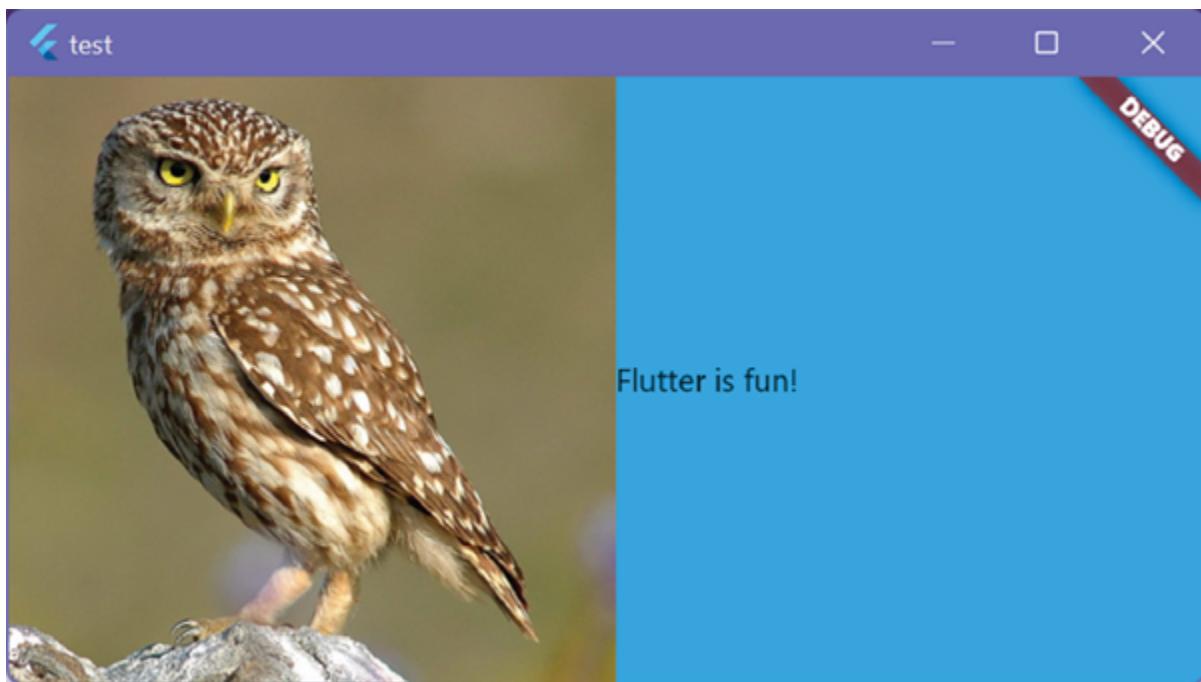


Figure 4.1: The results of our test code, running as a Windows application.

On the surface, the widget tree we have created appears quite simple (*figure 4.2*):

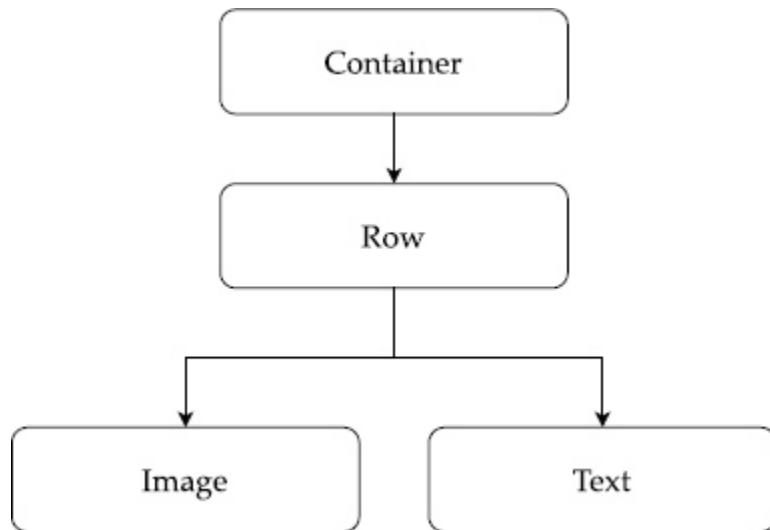


Figure 4.2: The widget tree from our example code

In practice, however, the widget tree Flutter created is somewhat more complex (*figure 4.3*):

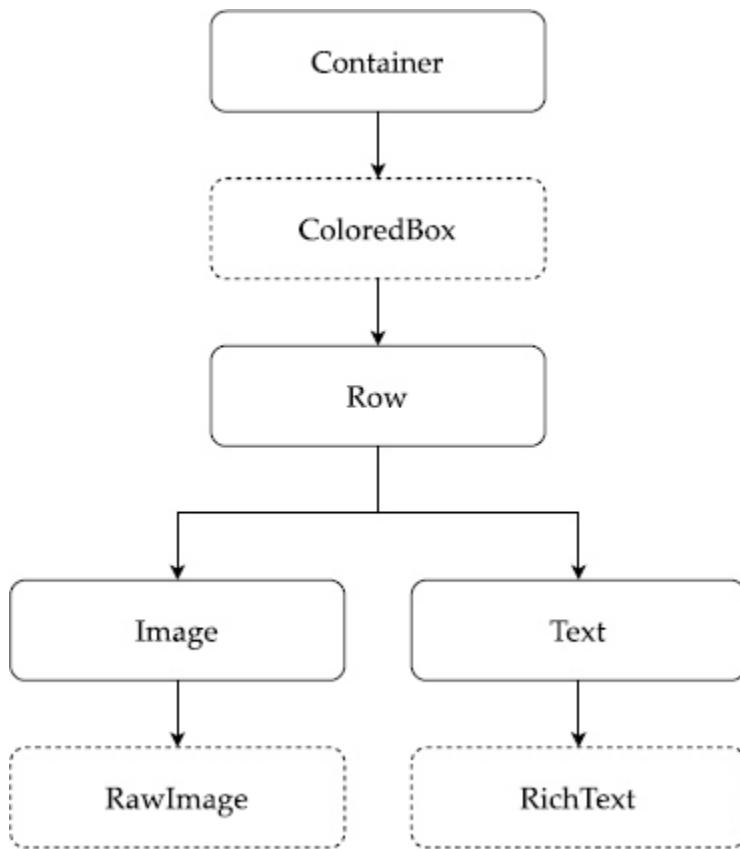


Figure 4.3: The actual widget tree Flutter builds from our example.

When building and debugging your Flutter applications, the widgets Flutter used to compose the layout you have created may not be immediately apparent. Through tools such as the Flutter inspector, which is one of the components of the Dart DevTools, you can inspect any widget that has been rendered to see which widgets have been added to the widget tree, as well as look at the widget tree itself. We will discuss the DevTools more in *Chapter 11: Debugging, Troubleshooting, and Performance Considerations*.

Each widget is passed down a set of constraints from its parent, while simultaneously passing its size back up the tree. This concept of constraints and sizes is key to composing a UI in Flutter.

Constraints determine how much space is *available* for a widget to occupy, whereas the size of a widget tells its parent how much space it *actually* takes up. If a widget's size exceeds its given constraints, we will get an error thrown on the screen when using debug mode. (This error is not

visible in release mode; however, the issue is still present.) We can explore this behavior with the following code (*figure 4.4*):

```
1. Container(  
2.   width: 100,  
3.   color: Colors.blue,  
4.   child: Row(  
5.     children: const [  
6.       Text(  
7.         'This text will overflow its constraints.',  
8.         softWrap: false,  
9.         overflow: TextOverflow.visible,  
10.      ),  
11.    ],  
12.  ),  
13. )
```

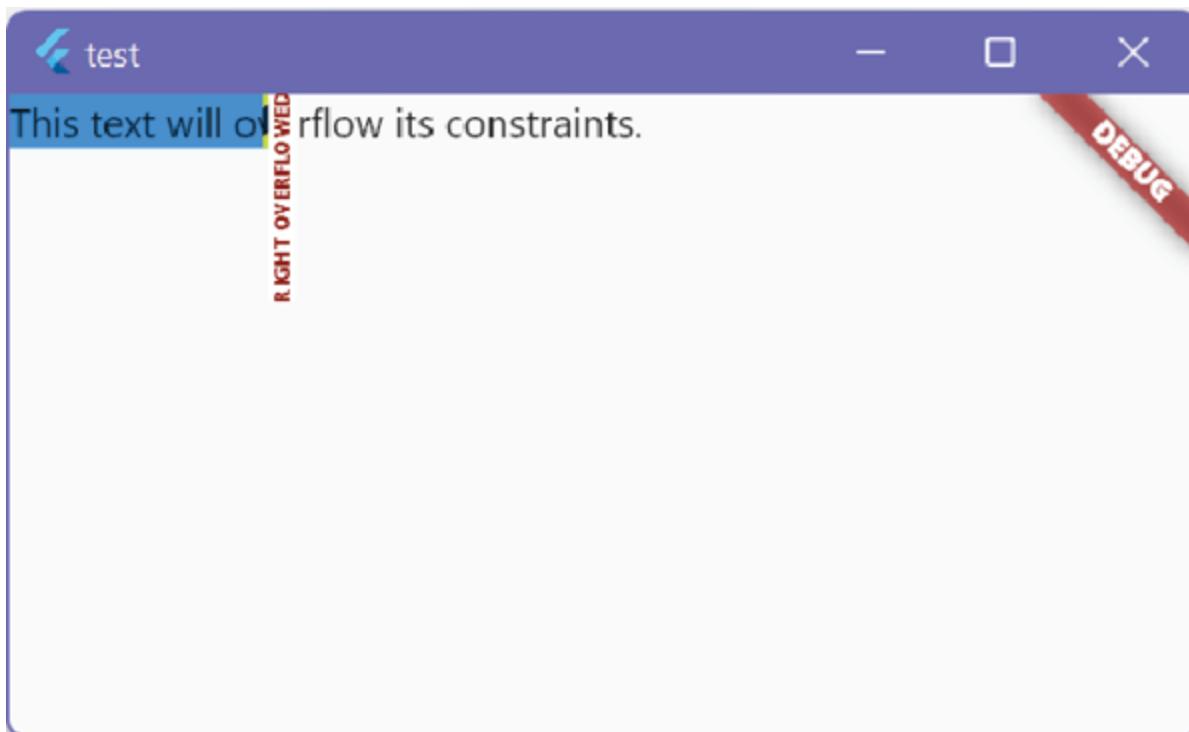


Figure 4.4: The error is thrown when a widget's size exceeds its given constraints.

The **Row** widget is very fussy about its constraints not being violated. By constraining a **Row** within a **Container**, we can limit the constraints of the **Row**. (Without doing this, the **Row** will expand to the size of the canvas, or whatever other parent's constraints have been passed down.) Once we have constrained the **Row**, we add a single child element, a **Text** widget, and provide arguments which will ensure the text does not wrap or clip if it runs over its given constraints. In practice, this is widely considered to be *bad code*. However, it does perfectly illustrate the sort of error you will receive when a widget's size exceeds its constraints.

By leveraging constraints, a UI can shift and adapt to its available screen size. In this way, your UI gains the ability to leverage a responsive design, allowing for different layouts in different scenarios, such as mobile versus a computer screen.

Now that we understand what a widget is and how it's rendered to the screen, let us talk about the three different types of widgets, starting with stateless widgets.

Stateless Widgets

Stateless widgets are the simplest of the three types of widgets, both in how they are created and used, and in their functionality. They are mostly used to describe the user interface. Importantly, stateless widgets are widgets that do not require a mutable state. However, they can have constructor arguments.

A constructor parameter allows you to pass a variable into the widget, called an argument, and is used during the **build()** method to customize the look, feel, behavior, or other aspects of the widget as part of the widget's properties. The **build()** method of a stateless widget is typically only called when the widget is first inserted into the widget tree, when the widget's parent changes, or when one of its dependent inherited widgets changes.

StatelessWidget is its own *abstract* class, meaning it is used to define the core behavior of all classes which are based on it, but it cannot itself be instantiated. Let us look at some of the more common stateless widgets which are part of the default Material package and their use cases, and then we will delve into creating our own.

Container:

One of the most common of all the stateless widgets is the `Container` widget. It is used as a layout widget, which can have a single child widget. `Container` might sound like a boring widget, but it is one of the most versatile out there with over a dozen properties available! The best way to describe `Container` is with an example. In our example, we are going to take things to the extreme by setting many, but not all, of the available properties. Look at our example `Container` (*figure 4.5*):

```
1. Container(  
2.   width: 100,  
3.   height: 100,  
4.   margin: const EdgeInsets.only(  
5.     top: 20,  
6.     left: 100,  
7.   ),  
8.   decoration: BoxDecoration(  
9.     shape: BoxShape.circle,  
10.    color: Colors.grey[50],  
11.  ),  
12.  child: const FlutterLogo(),  
13.  padding: const EdgeInsets.all(30.0),  
14.  foregroundDecoration: BoxDecoration(  
15.    border: Border.all(  
16.      width: 5,  
17.      color: Colors.grey,  
18.      style: BorderStyle.solid,  
19.    ),  
20.    gradient: const RadialGradient(  
21.      colors: [Colors.red, Colors.purple],  
22.      center: Alignment.topCenter,  
23.      radius: Radius.circular(100),  
24.    ),  
25.  ),  
26.  transform: Matrix4.translationValues(0, 0, 100),  
27.  alignment: Alignment.bottomCenter,  
28.  child: Text("Hello World!",  
29.    style: TextStyle(fontSize: 20, color: Colors.white),  
30.  ),  
31.);
```

```
21.         colors: [
22.             Colors.white,
23.             Colors.white,
24.             Colors.black,
25.         ],
26.     ),
27.     shape: BoxShape.circle,
28. ),
29. )
```

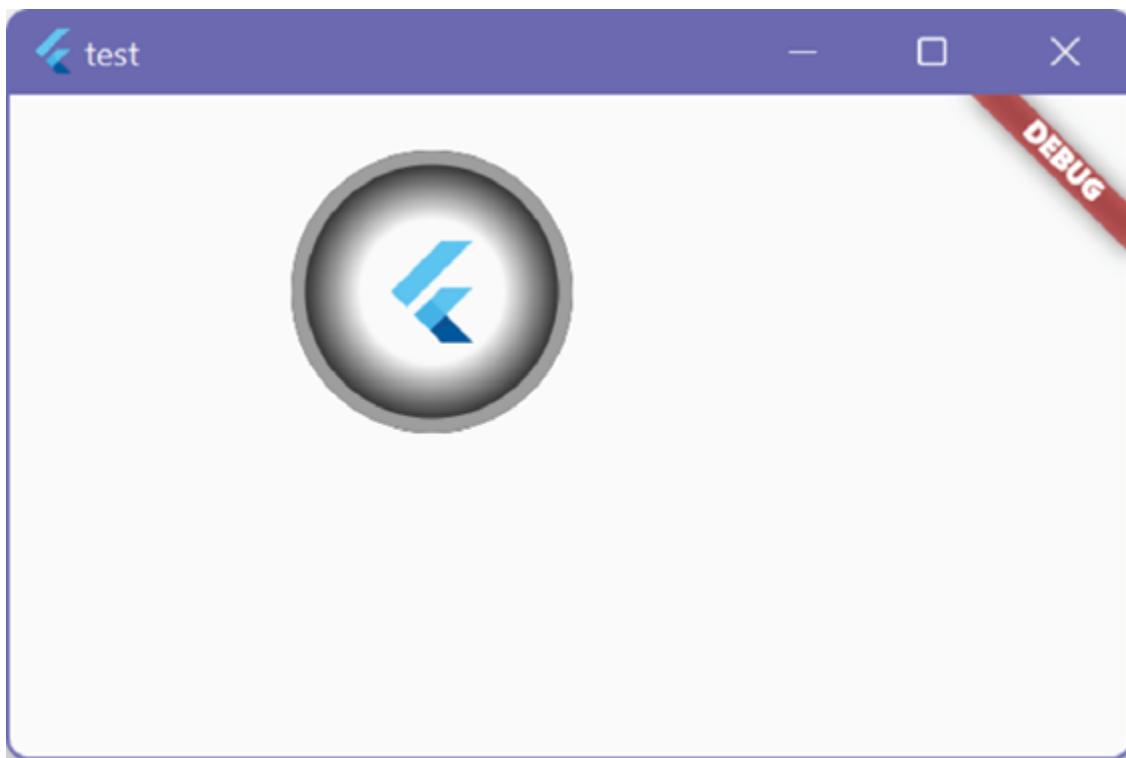


Figure 4.5: The resulting widget which is rendered from our example.

You might be asking yourself where you can find all the available properties for `Container`, or any other widget. There are several ways to do this. First, you can hover over the widget within Visual Studio Code and a box will pop up which describes the available properties. Another way you can find the properties for a widget is by referencing the online documentation, which can be found at

<https://api.flutter.dev/flutter/widgets/widgets-library.html> and contains a list of all available widgets and their properties. The third way to look up the properties is to look at the source code of the widget in question. By holding *Ctrl*/*⌘* and clicking on a widget, you will jump straight to the source code of the widget.

Column and Row:

Both the **Column** and **Row** widgets function similarly, taking in a list of widgets and displaying them sequentially, with **Column** ordering its children vertically and **Row** ordering them horizontally. **Columns** and **Rows** are well suited for grouping items together. For example, you might use a **Row** to place an icon next to a text label, as in this example (*figure 4.6*):

```
1. Row(  
2.   children: const [  
3.     Icon(Icons.account_circle),  
4.     Text('My Account'),  
5.   ],  
6. )
```

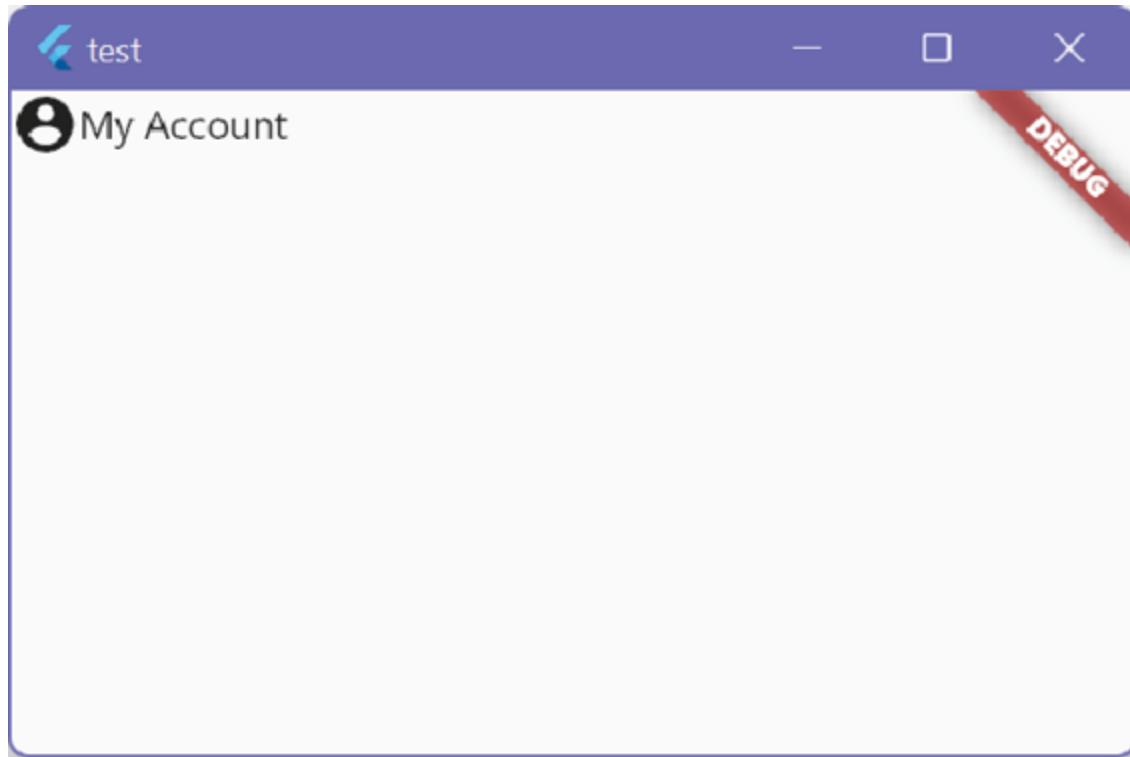


Figure 4.6: Using a Row to align two widgets.

By swapping out the `Row` for a `Column`, you would find that the `Icon` would be placed above the `Text`. By leveraging the `mainAxisAlignment` and `crossAxisAlignment` properties, we can further adjust the alignment of the child widgets. In the following example, the `mainAxisAlignment` property has been modified so that the elements will have equal space between them. We have also added another `Text` widget to further illustrate this behavior (*figure 4.7*).

1. `Column(`
2. `mainAxisAlignment: MainAxisAlignment.spaceBetween,`
3. `children: const [`
4. `Icon(Icons.account_circle),`
5. `Text('My Account'),`
6. `Text('Another widget')`
7. `],`
8. `)`

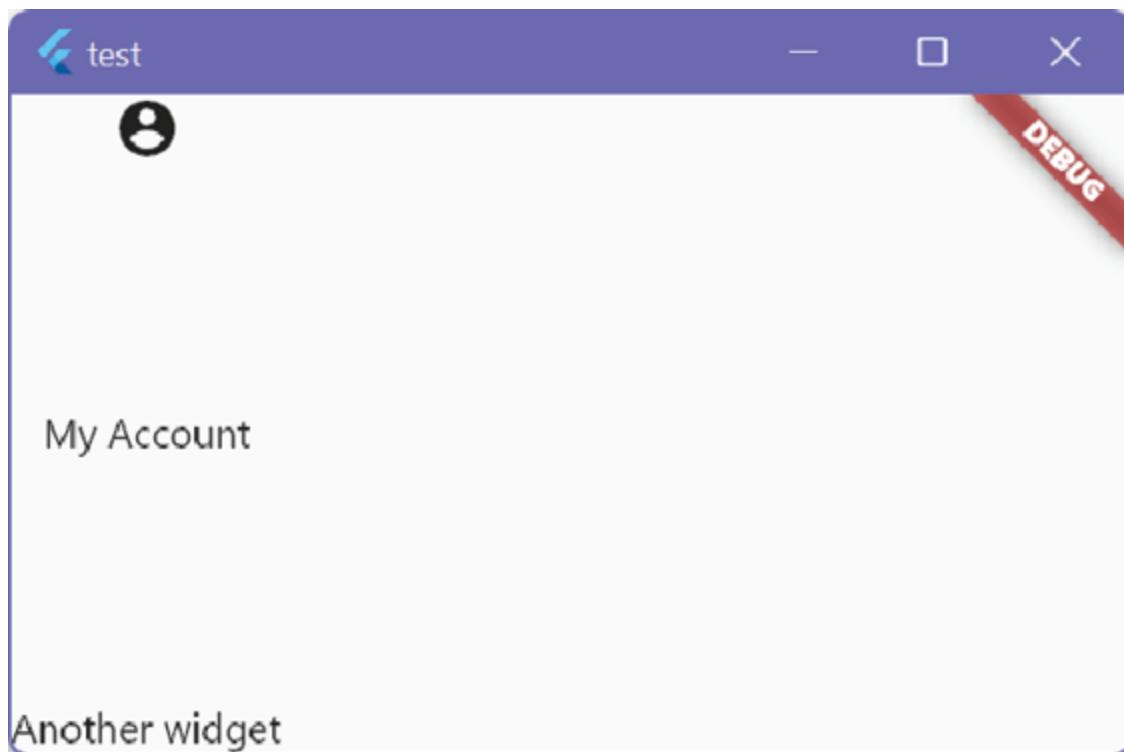


Figure 4.7: Changing our Row to a Column, adding another child, and adjusting the alignment.

Note that, by default, the `Column` widget centers its children horizontally. This behavior can be changed by using the `crossAxisAlignment` property. Both `Column` and `Row` are indispensable when composing a user interface in Flutter.

Text:

We have already seen the `Text` widget several times before in our examples, so let us take a minute to look at it more deeply. Displaying text is a surprisingly complex and nuanced task, especially when one considers the massive number of scripts (of which Latin is but one), typefaces, and styling options available. All this is before we even consider what happens to the text when it overflows its constraints! Fortunately, Flutter's `Text` widget has us covered.

Displaying some text in the default style is as easy as passing a string of text into your `Text` widget as a *positional* parameter, meaning there is no need to specify a parameter name (see *figure 4.8*).

1. `Text("Flutter is fun!")`

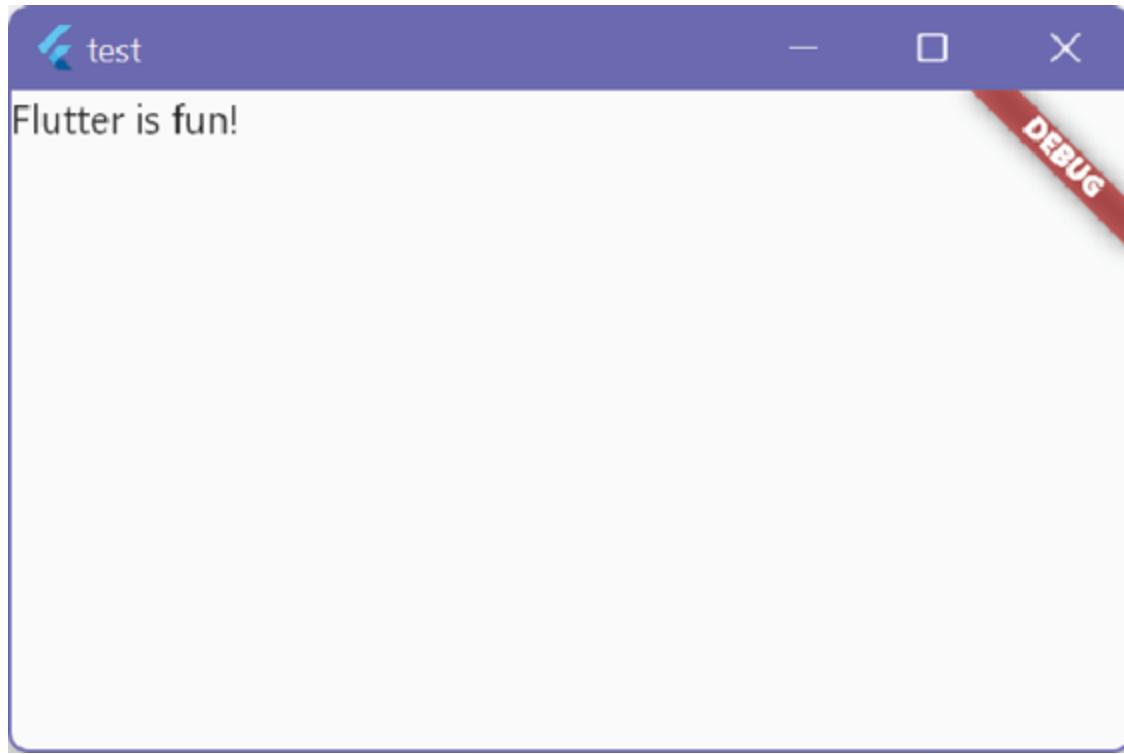


Figure 4.8: Displaying text using the `DefaultTextStyle`.

The Flutter framework defines its default text style in the `DefaultTextStyle` widget, which is quite appropriately named. This is then made available to widgets through the app's `Theme`. If we want to override the default text style, we can do so by specifying a style using the `style` parameter, where we can supply our own `TextStyle`, which will override the `DefaultTextStyle` for any of the arguments we supply (see *figure 4.9*).

```
1. Text(  
2.   "Flutter is fun!",  
3.   style: TextStyle(  
4.     backgroundColor: Colors.grey[300],  
5.     fontFamily: 'Neonderthaw',  
6.     fontSize: 36,  
7.     wordSpacing: 24,  
8.   ),  
9. )
```



Figure 4.9: Overriding the default text style.

What if we want to make our text larger and bold, center it in the parent container, and have a colorful border around the letters? With `Text`, that's a simple enough task (*figure 4.10*). In fact, we can even use a *cascade notation* to apply multiple properties.

1. `Text(`
2. `'Fancy text',`
3. `textAlign: TextAlign.center,`
4. `style: TextStyle(`
5. `fontSize: 72,`
6. `fontWeight: FontWeight.bold,`
7. `foreground: Paint()`
8. `..style = PaintingStyle.stroke`
9. `..strokeWidth = 3`
10. `..shader = ui.Gradient.linear(`

```
11.         const Offset(0, 20),  
12.             const Offset(300, 20),  
13.             <Color>[  
14.                 Colors.blue,  
15.                 Colors.red,  
16.             ],  
17.         ),  
18.     ),  
19. )
```



Figure 4.10: Some truly fancy text.

Cascade operators let us perform multiple actions on the same object in a shorthand form. Let us compare non-cascade notation to the same code using cascade notation:

1. import 'dart:ui' as ui;
- 2.

```
3. // Non-cascade notation
4. final Paint paint = Paint();
5. paint.style = PaintingStyle.stroke;
6. paint.strokeWidth = 3;
7. paint.shader = ui.Gradient.linear(
8.   const Offset(0, 20),
9.   const Offset(300, 20),
10.  <Color>[
11.    Colors.blue,
12.    Colors.red,
13.  ],
14. );
15.
16. // Cascade notation
17. Paint()
18.   ..style = PaintingStyle.stroke
19.   ..strokeWidth = 3
20.   ..shader = ui.Gradient.linear(
21.     const Offset(0, 20),
22.     const Offset(300, 20),
23.     <Color>[
24.       Colors.blue,
25.       Colors.red,
26.     ],
27.   );
```

Without cascade notation, we would need to create a new variable to assign the base object to before setting each of the values. Cascade notation gives

us cleaner, more concise code by allowing us to assign properties to an object as we create it.

What happens if we want to change font styles in the middle of a sentence, though? For example, what if we wanted to add some *emphasis* to a particular word in our text? This requires a somewhat different strategy, because we will need to tell Flutter exactly what text we want to have styled and what style to use. We do this by using the `Text.rich` constructor, that wants a `TextSpan` as its positional argument. Confused? Let us look at an example (*figure 4.11*).

```
1. Text.rich(  
2.   TextSpan(  
3.     text: 'Flutter',  
4.     children: <TextSpan>[  
5.       TextSpan(  
6.         text: ' is ',  
7.         style: TextStyle(  
8.           fontStyle: FontStyle.italic,  
9.         ),  
10.      ),  
11.      TextSpan(  
12.        text: 'fun!',  
13.        style: TextStyle(  
14.          fontWeight: FontWeight.bold,  
15.        ),  
16.      ),  
17.    ],  
18.  ),  
19. )
```

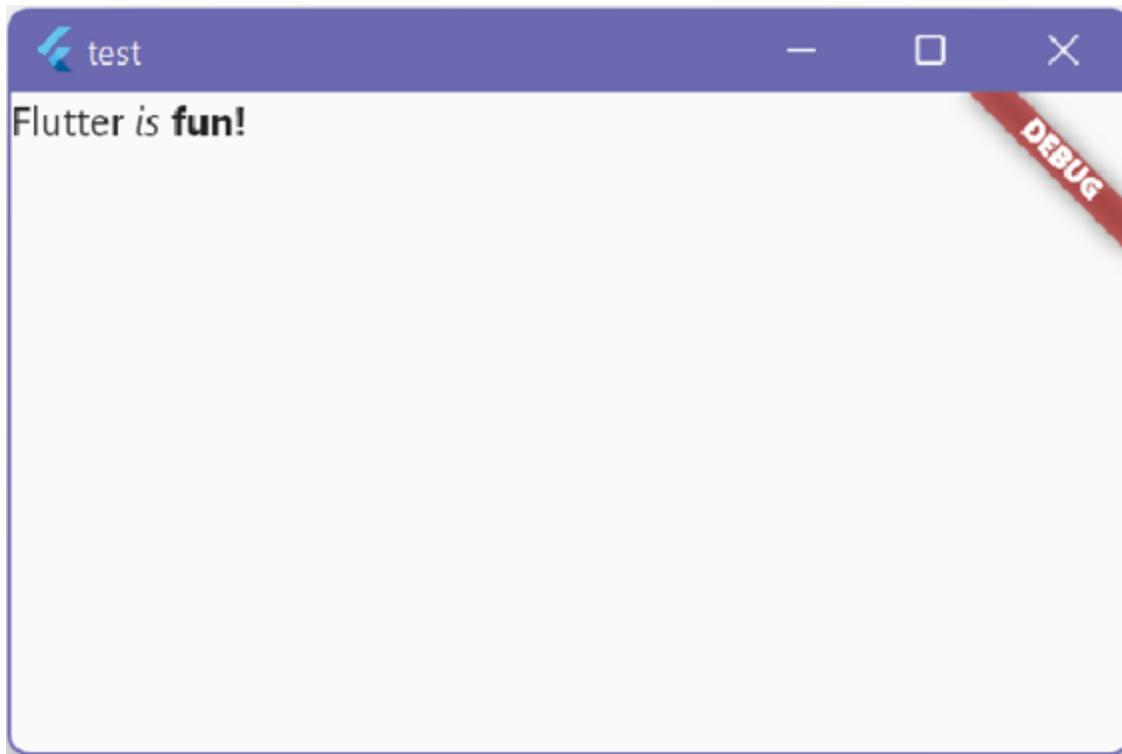


Figure 4.11: Styling in-line text using multiple styles.

Before we move on to the last stateless widget that we are going to look at right now, let us have a look at how text behaves when its constraints change. In the preceding examples, there was plenty of room for our text, but if we limit the constraints of the `Text` widget, Flutter will have to do *something* with the text as it runs up against those constraints. By default, Flutter will wrap the text to the next line, although this behavior can be prevented by setting the `softWrap` parameter to `false` (*figure 4.12*).

1. Container(
2. color: Colors.grey[300],
3. width: 105,
4. height: 105,
5. child: const Text(
6. "You text will automatically wrap to the next line, but what happens if it overflows its parent?",
7.),

8.)

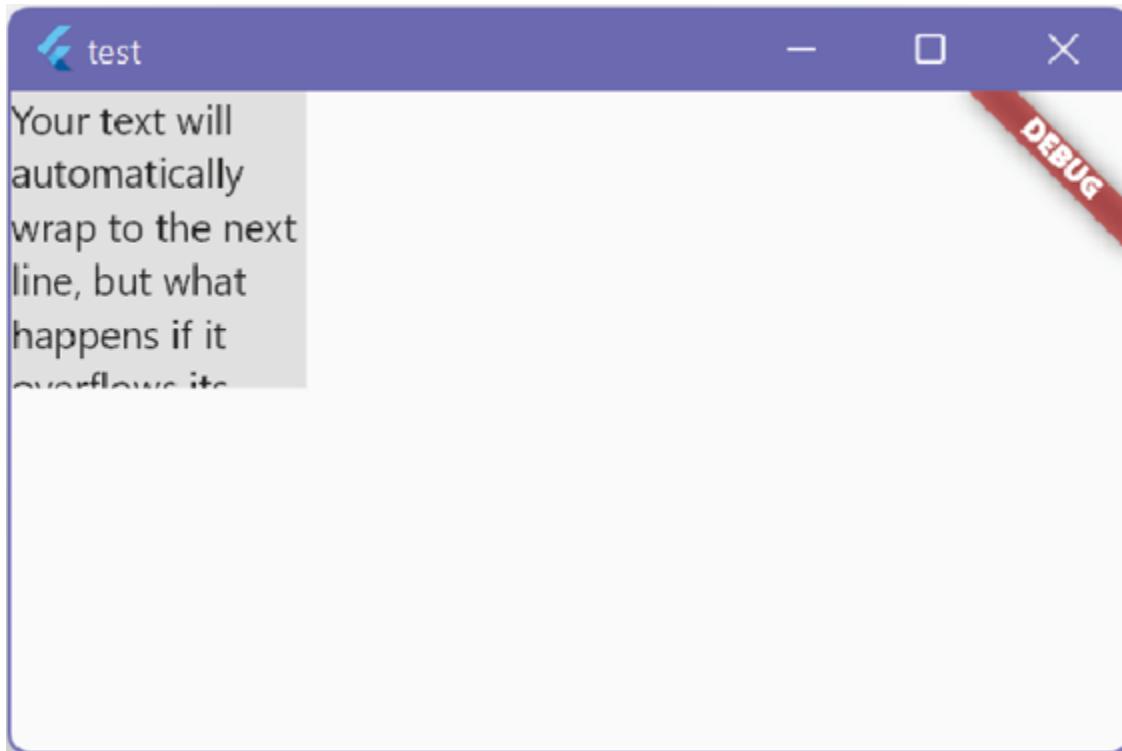


Figure 4.12: Flutter will wrap text to the next line for us by default.

A keen-eyed observer will note, however, that our text gets cut off when it no longer has room to wrap! We could fade out the text as it gets closer to the edges of its parent (*figure 4.13*):

1. Container(
2. color: Colors.grey[300],
3. width: 105,
4. height: 105,
5. child: const Text(
6. "You text will automatically wrap to the next line, but what happens if it overflows its parent?",
7. overflow: TextOverflow.fade,
8.),
9.)

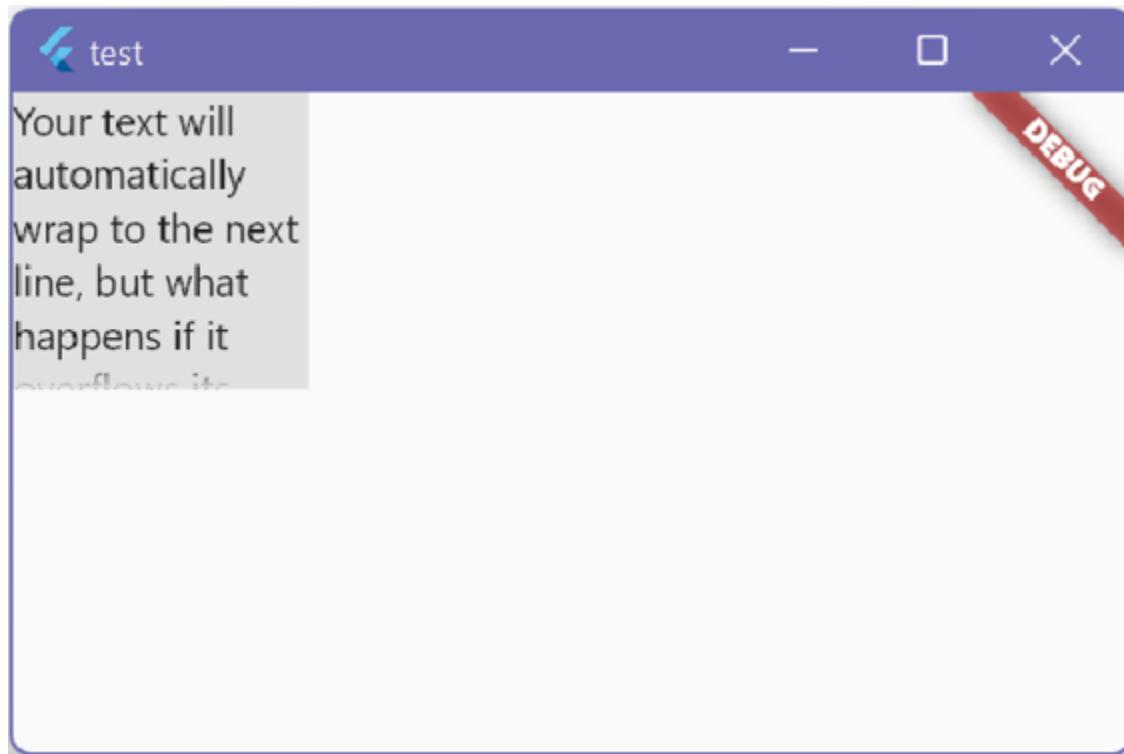


Figure 4.13: Fading out text which would otherwise be cut off.

Alternatively, we could simply truncate the text with an ellipsis, preventing it from wrapping to the next line altogether (*figure 4.14*). We do this by specifying an `overflow` property with the `TextOverflow.ellipsis` argument.

1. Container(
2. color: Colors.grey[300],
3. width: 105,
4. height: 105,
5. child: const Text(
6. "You text will automatically wrap to the next line, but what happens if it overflows its parent?",
7. overflow: TextOverflow.ellipsis,
8.),
9.)

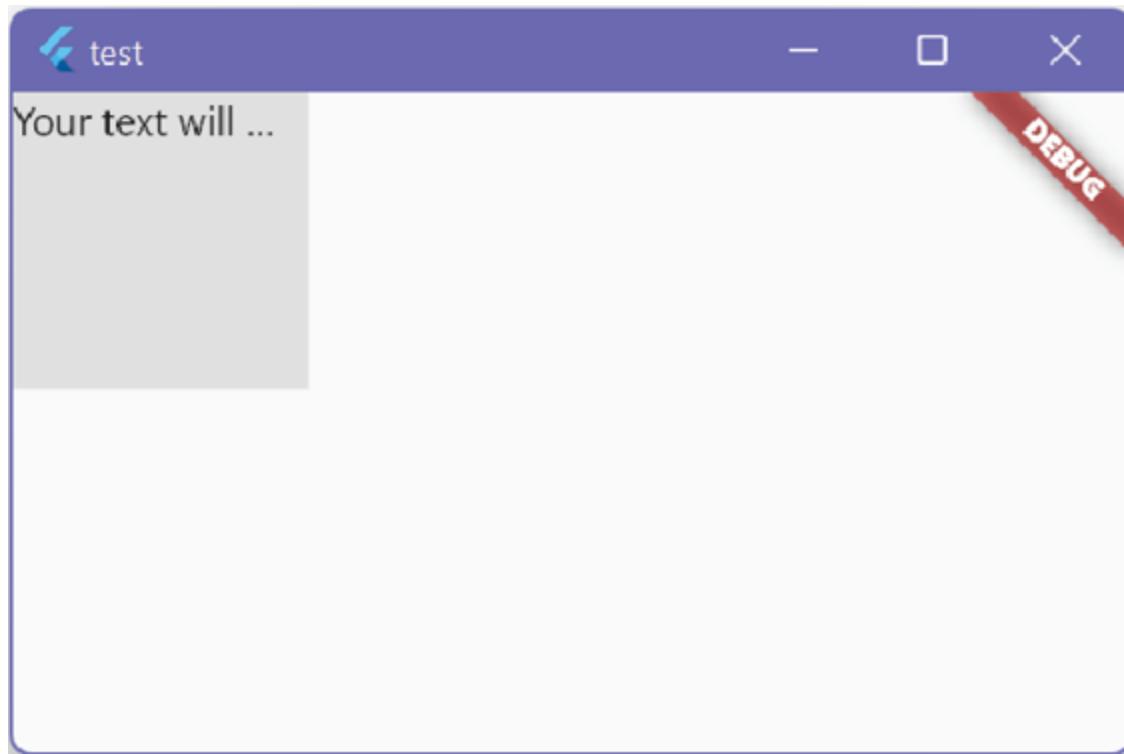


Figure 4.14: Truncating the overflowing text with an ellipsis.

As you can see, there is a lot of power and flexibility within the `Text` widget. There are even more options, which we haven't discussed here, that can modify your text to display right-to-left, change the relative character widths and heights, and much more.

Center:

If you ask a Web developer to center an element on the page both horizontally and vertically, they may sigh before writing several lines of code, using one of many different methods. Internally, they will be glad that they now have modern tools to accomplish that task, as a few short years ago it was a much more difficult proposition.

Accomplishing the same task in Flutter is as simple as using the `Center` widget (see *figure 4.15*).

1. `Center(`
2. `child: Text("Flutter is fun!"),`
3. `)`

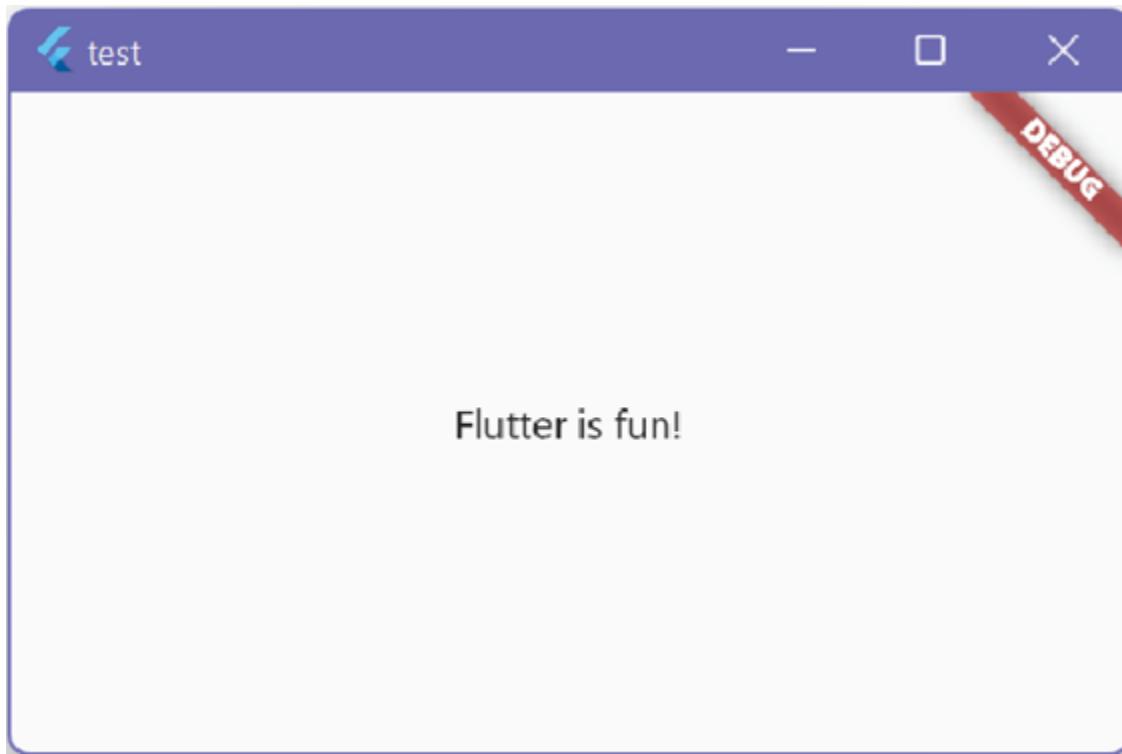


Figure 4.15: Centering an element in Flutter is easy with the Center widget.

Creating a Custom Stateless Widget

Sometimes you will find yourself reusing a group of nested widgets. If this happens, you might be tempted to copy and paste the whole tree of widgets, but what happens when you need to make a change to one of the elements in that tree? It would not make sense to go back through your code and find every place you have copied that code to so you can update the same element, in the same way, multiple times.

In cases such as this, you can create a new widget that returns the entire widget tree! When you update your widget, the changes will take effect wherever you're using it. It also helps to keep your code clean and tidy, which you will thank yourself for.

To create a stateless widget, start with this boilerplate:

1. class MyWidget extends StatelessWidget {
2. const MyWidget({super.key});
3. }

```
4.    @override
5.    Widget build(BuildContext context) {
6.        return Container();
7.    }
8. }
```

We use `class MyWidget extends StatelessWidget` to tell Flutter that we are creating a new `class` of `Widget` called `MyWidget` which is based on the class `StatelessWidget`. Let us examine what is happening here, line by line.

On the first line, we are defining our widget as `MyWidget` and telling Flutter to use `StatelessWidget` as the basis for doing so. Line two is our *constructor*. The `const` keyword tells Flutter that instances of the widget may have constant values when the code has been compiled, provided that all the properties are marked as `final`. On that same line, you will see `MyWidget({super.key})`, but what does this mean? Come to think of it, what does `super.key` mean?

To answer these questions, we must dive a bit more deeply. First, we have `MyWidget({super.key})`. You will notice this includes the name of our widget. That makes sense since we are defining a constructor *for* that widget. But what is `key`, why is it `super`, and why is it in curly braces?

In Flutter, a `Key` is used to control which widgets the framework matches up with the corresponding state object for that widget when a rebuild is triggered. In practice, this allows Flutter to preserve its state (which we will talk about more in the next section) when widgets move around in the widget tree. For the most part, you will not need to manually specify a `Key`.

Next, let us answer the question of why there are curly braces. Think back to when we were exploring the `Text` widget. When we wanted to print text to the screen, we first invoked the widget, and then set several of the widget's properties. The first was the text we wanted to display, followed by everything else. But did you notice that we did not have to specify `Text(text: "Our text")` when we wanted to just print some text? Instead, we just wrote `Text("Our text")`. In the first example, which is the *wrong* way to create a `Text` widget, we are specifying the text that we want the widget to display as a

named parameter, rather than as a *positional parameter*, as in the second (and correct) example.

This is where the curly braces come into play. Any variable we specify in our constructor that is inside the curly braces will be a *named* parameter. Any parameters specified outside of the curly braces will be a *positional* parameter. It is also important to note that positional parameters are required, by default.

Here is an example of a widget with a positional parameter:

```
1. class MyCustomTextWidget extends StatelessWidget {  
2.   final String text;  
3.   const MyCustomTextWidget(this.text, {super.key});  
4.  
5.   @override  
6.   Widget build(BuildContext context) {  
7.     return Text(text);  
8.   }  
9. }
```

Note that for us to access `text` within our `build()` method, we need to also specify `final String text;`. The same will be true for any parameter, regardless of whether it is a positional or named parameter.

Finally, we will answer the question of why the `Key` parameter is `super`. Starting in Dart 2.12 and Flutter 2, *sound null safety* was enabled by default. In computer science, *null* means “nothing.” So, if we simply define a variable and do not set it to a value, the value would be *null*. This fact alone has caused a great number of headaches for developers and bugs and crashes for users. So, how do we prevent those headaches and crashes? Easy! We make sure that our variables are not *null unless we say they can be!* To accomplish this, variables can be marked *nullable* by adding a question mark after the *type*. For example, a `final String myString` is not nullable, but a `final String? myString` *is*. The question mark tells Flutter (or, more specifically, Dart) that the *type* of the variable named `myString` is `String`,

but we will also allow for its type to be `null`. So, `myString` can either be a `String` object or `null`.

When creating a class in Dart, we can *inherit* a property from the immediate parent object. We do this by using the `super` keyword. In earlier versions of Dart and Flutter, the syntax was slightly different, and by examining it we can start to see where the current implementation came from. In previous versions, the code would have been written as `const MyCustomTextWidget(this.text, {Key? key}) : super(key: key);` which makes it clearer that the `key` property is a nullable, named parameter. By using `super(key: key)`, the `key` property will be inherited from the parent object if it is not specified. This means we can optionally specify a `Key` when creating an instance of our widget. (A `Key` controls how one widget replaces another widget in the widget tree. Two widgets with the same key must have the same `runtimeType`, so do not go replacing a stateless widget with a stateful widget, unless you want to give Flutter a headache!) In modern Dart and Flutter, the constructor has been shortened to allow for the parameter to be specified as `super.key`, but the result is the same as before.

Lines 5 and 6 of our custom widget go together. Line 5 says that we want to override the default `build()` function of the `StatelessWidget` on which we are basing our custom widget off. Line 6 starts with `Widget`, which is the `build()` function's return type. That is, when the `build()` function runs, we expect that a `Widget` will be returned after all is said and done. The Flutter framework itself will call the `build()` method automatically, so there is no need to invoke it manually. The parameter which is passed to `build()` is a positional parameter of type `BuildContext` and is named `context`. We will talk more about what `BuildContext` is later in this chapter. For now, just know that it is passed into our `build()` method.

Finally, on Line 7, we compose our new widget and `return` it. The `build()` method will always have a `return` statement, followed by what value it is returning (and that value will always return a `Widget`.) What happens within that `return` statement is up to you! That is where you will specify the nested widgets that you would like to return to save yourself a whole lot of time and energy, ensure your code is concise and readable, and make testing your application so much easier.

Stateful Widgets

An application built entirely with stateless widgets would be very boring, indeed. Sometimes, we want to add interactivity to our app. Whether this is in the form of a button to push, some text that updates according to another action (such as with the counter in the Flutter demo application, from the previous chapter), or any other type of interactivity, stateful widgets are where we need to turn to.

Stateful widgets are widgets which can create and maintain a custom `State` object. In contrast to a stateless widget, stateful widgets can keep changes made to them when a rebuild is triggered, via the `State` object. Let us look at the `MyHomePage` widget from the demo application, so we can start to understand how all the pieces fit together.

```
1. class MyHomePage extends StatefulWidget {  
2.   const MyHomePage({super.key, required this.title});  
3.  
4.   final String title;  
5.  
6.   @override  
7.   State<MyHomePage> createState() => _MyHomePageState();  
8. }  
9.  
10. class _MyHomePageState extends State<MyHomePage> {  
11.   int _counter = 0;  
12.  
13.   void _incrementCounter() {  
14.     setState(() {  
15.       _counter++;  
16.     });  
17.   }  
18.
```

```
19. @override
20. Widget build(BuildContext context) {
21.   return Scaffold(
22.     appBar: AppBar(
23.       title: Text(widget.title),
24.     ),
25.     body: Center(
26.       child: Column(
27.         mainAxisAlignment: MainAxisAlignment.center,
28.         children: <Widget>[
29.           const Text(
30.             'You have pushed the button this many times:',
31.           ),
32.           Text(
33.             '$_counter',
34.             style: Theme.of(context).textTheme.headline4,
35.           ),
36.           ],
37.         ),
38.       ),
39.     floatingActionButton: FloatingActionButton(
40.       onPressed: _incrementCounter,
41.       tooltip: 'Increment',
42.       child: const Icon(Icons.add),
43.     ),
44.   );
45. }
```

46. }

The first thing you will notice is that a stateful widget has two parts: the widget itself (`MyHomePage`) and a widget which holds its state (`_MyHomePageState`). Let us look at the first widget, `MyHomePage`, to start. We can see that there is a constructor which has a required `title` named parameter. Below that, we see that the title is being defined as a `String` type. This should all make sense to you by now.

Where the stateful widget differs from the stateless widget is that there is no longer a `build()` method. Instead, we override the `createState()` method. This is where we tell the stateful widget what `State` object to use.

Notice how the `_MyHomePageState` class starts with an underscore. The underscore indicates that the variable or method is scoped locally to the library in which it can be found. A library can be a single file or multiple files connected with the `part` and `part of` keywords. This is used to tell Flutter that we're defining a *private* object. A private object cannot be directly invoked from outside its context. You will see this inside the class, too, with the `_counter` variable and the `_incrementCounter()` method. Going back to our `MyHomePage` widget, what we find is that the `State` object which is being created is called `_MyHomePageState`, which is a private class. Now we know that this `State` object should not be referenced anywhere *except* from the widget which invokes it.

It is within `_MyHomePageState` that the widget's `build()` method is contained. Any variables that we want to be preserved between rebuilds are defined outside of the `build()` method and in this example, that includes the `_counter` variable. When this variable is defined initially, it is set to `0`. Whenever the `_incrementCounter()` method is called, the state of the widget is updated to add `1` to the old value, eventually resulting in a rebuild being triggered.

This brings us to the next concept of stateful widgets: setting the state. When a stateful widget is first added to the widget tree, the `initState()` method is called. The `_MyHomePageState` widget does not have an `initState()` method defined, though, so what gives? Well, remember when we said we must *override* the `build()` method? It is the same concept with the `initState()` method (and the other state management methods) except that this time, it is optional. If we want to override the default behavior, we need to add a method, like this:

```
1. @override  
2. void initState() {  
3.   super.initState();  
4. }
```

The `super.initState();` line is important because it will run the `initState()` method of the widget that you have based your custom widget on. Anything you want to add to the initial state of the widget should be done *after* this line.

So, what *would* you want to add to an `initState()`? Generally, this is where you would start to invoke some sort of logic. Perhaps you have a widget which will display the results of a `Future`. Consider the following code:

```
1. Future<bool> _futureFalse() async {  
2.   return Future.delayed(const Duration(seconds: 1), () {  
3.     return false;  
4.   });  
5. }
```

This method will wait one second before returning `false`. If you wanted to wait for this future to complete before building your widget for the first time, you could invoke the method from within the `initState()` and assign it to a variable. Here is how we would do that:

```
1. class ExampleWidget extends StatefulWidget {  
2.   const ExampleWidget({super.key});  
3.  
4.   @override  
5.   State<ExampleWidget> createState() => _ExampleWidgetState();  
6. }  
7.  
8. class _ExampleWidgetState extends State<ExampleWidget> {  
9.   bool alwaysFalse = true;
```

```
10.  
11. Future<bool> _futureFalse() async {  
12.     return Future.delayed(const Duration(seconds: 1), () {  
13.         return false;  
14.     });  
15. }  
16.  
17. void _resolveFutures() async {  
18.     alwaysFalse = await _futureFalse();  
19.     setState(() {});  
20. }  
21.  
22. @override  
23. void initState() {  
24.     super.initState();  
25.     _resolveFutures();  
26. }  
27.  
28. @override  
29. Widget build(BuildContext context) {  
30.     return Text('$alwaysFalse');  
31. }  
32. }
```

When the widget is created and the `build()` method runs, the text displayed will initially be `true` as we set the `alwaysFalse` variable to `true` when we defined it on Line 9. Since `initState()` is called before `build()`, the `_resolveFutures()` method is already running and has already invoked the `_futureFalse()` method by the

time the `build()` process even starts. However, it takes a single second for that process to complete, so after that second the text is updated within the widget to display `false` when a rebuild is triggered.

Notice that we had to invoke the `setState()` method (Line 19) to update the state of the widget and trigger the rebuild. If we had not done that, the variable would have been updated but no rebuild would have happened, so the text on the screen would not have changed.

The lifecycle of a stateful widget includes the following steps:

- `createState()`
- `mounted == true`
- `initState()`
- `didChangeDependencies()`
- `build()`
- `didUpdateWidget()`
- `setState()`
- `deactivate()`
- `dispose()`
- `mounted == false`

We have seen several of these steps so far in our example. `createState()` is called from our stateful widget, whereas `initState()` is called from the `State` widget. In between, we have `mounted == true`. This condition is set once a widget has had a `BuildContext` assigned to it.

After the state has been initialized, the `didChangeDependencies()` method is called. This will be called both right after the state has been initialized and whenever data that the widget depends on changes.

Next up is our trusty `build()` method, which we are well acquainted with, followed by the `didUpdateWidget()` method. This is called when the widget's configuration changes, which is often something the parent widget is responsible for, triggering a rebuild for our stateful widget. The usage of this method is a bit more nuanced but can be thought of as a *replacement* for the `initState()` method, but only in cases where the widget associated with the widget's state needs to be rebuilt.

Next, we have `setState()`, which we are already familiar with, followed by `deactivate()`. This is rarely used, but since a widget *can* be removed and reinserted into the widget tree at a different location, there exists a method to call in that event.

Finally, we have the `dispose()` method followed by `mounted == false`. When a `State` object is removed from the widget tree, the `dispose()` method is called. This is where you will want to call the methods to stop any animations, unsubscribe from data streams, and do a bit of housekeeping to clean up after yourself. Once a widget has been disposed of properly, the `mounted` variable is set, internally, to `false`.

Understanding the lifecycle of a stateful widget will be invaluable to you as you create more and more complex widgets. Knowing when `initState()` is called versus `build()` will save you many headaches in the future.

Some common examples of stateful widgets include things like buttons, radio buttons, checkboxes, sliders, switches, and anything else that can be interacted with to perform some sort of action.

Inherited Widgets

Inherited widgets are the third and final type of widget which is available to us in the Flutter toolkit. These widgets are often the least understood, and thus the most neglected by most new Flutter developers. We are going to demystify them and explore their genius and beauty. So, what is an inherited widget?

Let us say you are building an application, and you want to pass some data down the widget tree. Maybe this is some text, named `myText`, and you want to display it in an ancestor widget (see *figure 4.16*).

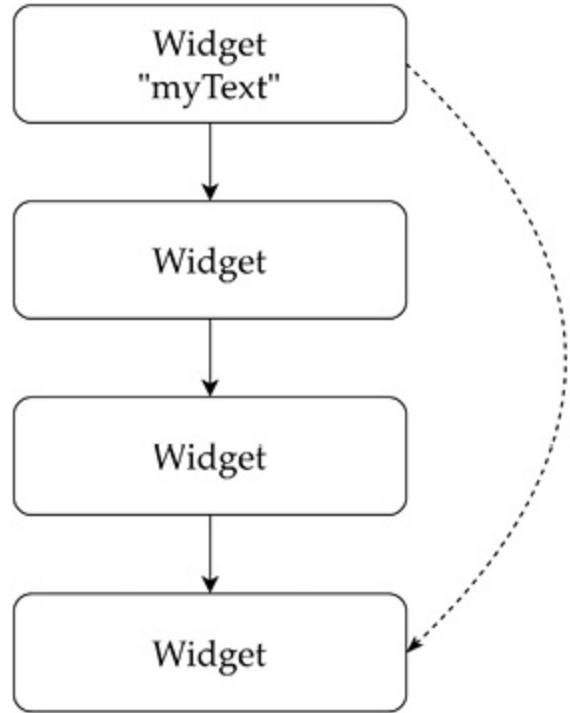


Figure 4.16: How do we pass down data to ancestors in the widget tree?

We *could* modify each of the widgets in the tree to accept a parameter and use that to pass down the data manually, but this is cumbersome and prone to all sorts of issues. Luckily, the Flutter team has us covered with inherited widgets.

To get started, we need to create a widget that extends `InheritedWidget`.

```

1. class MyInheritedWidget extends InheritedWidget {
2.
3.     @override
4.     const MyInheritedWidget({
5.         super.key,
6.         required super.child,
7.     });
8.
9.     @override
10.    bool updateShouldNotify(MyInheritedWidget oldWidget) {

```

```
11.     return true;
12. }
13.
14. static MyInheritedWidget? of(BuildContext context) {
15.     return
16.         context.dependOnInheritedWidgetOfExactType<MyInherited
17.             Widget>();
```

This boilerplate code includes everything we need to get started, but on its own it is not very helpful. Let us deconstruct what is going on, then figure out how to make it useful to us.

Each inherited widget is required to have a `child` parameter. Since we are extending the base class, we will need to override the constructor, as we have done before (Lines 3-7).

Next, we are overriding the `updateShouldNotify` method of the `InherhitedWidget` (Lines 9-12). This will ensure that if any values are updated inside our inherited widget, all its children will be aware that those values have changed.

Before we look at Lines 14-16, let us look at how to use our new widget. To relate it back to our earlier example, we will also modify it to add a `text` parameter.

```
1. class MyApp extends StatelessWidget {
2.     const MyApp({super.key});
3.
4.     @override
5.     Widget build(BuildContext context) {
6.         return const MaterialApp(
7.             home: Scaffold(
```

```
8.         body: MyInheritedWidget(
9.             text: "Hello, Flutter!",
10.            child: MyWidget(),
11.        ),
12.    ),
13. );
14. }
15. }
16.
17. class MyInheritedWidget extends InheritedWidget {
18.   final String text;
19.
20.   @override
21.   const MyInheritedWidget({
22.     super.key,
23.     required this.text,
24.     required super.child,
25.   });
26.
27.   @override
28.   bool updateShouldNotify(MyInheritedWidget oldWidget) {
29.     return oldWidget.text != text;
30.   }
31.
32.   static MyInheritedWidget? of(BuildContext context) {
33.     return
context.dependOnInheritedWidgetOfExactType<MyInherited
```

```
        Widget>());  
34.    }  
35. }  
36.  
37. class MyWidget extends StatelessWidget {  
38.   const MyWidget({super.key});  
39.  
40.   @override  
41.   Widget build(BuildContext context) {  
42.     final inheritedWidget =  
43.       context.dependOnInheritedWidgetOfExactType<MyInhe  
ritedWidget>();  
44.     final String text = inheritedWidget!.text;  
45.  
46.     return Text(text);  
47.   }  
48. }
```

You will notice that we added the parameter on Line 18 and made it a required parameter on Line 23. Then, on Line 9, we passed some text into the inherited widget. To print the text on line 46, we first need to locate our inherited widget (Lines 42 and 43), and then we get the value of `text` on Line 44.

It is Line 43 that we want to focus on, though. That is an awfully cumbersome amount of code to type just to refer to our inherited widget! Would it not be great if we could just call it by name? It turns out, we can! That is where Lines 32-34 come into play (or Lines 14-16 in our old example). This specifies an `of()` method for our inherited widget, which simply returns the long, cumbersome line. It only requires passing in a

`BuildContext` to function, too. Now when we want to search for our inherited widget, we can do it by name!

```
1. class MyWidget extends StatelessWidget {  
2.   const MyWidget({super.key});  
3.  
4.   @override  
5.   Widget build(BuildContext context) {  
6.     final inheritedWidget = MyInheritedWidget.of(context);  
7.     final String text = inheritedWidget!.text;  
8.  
9.     return Text(text);  
10. }  
11. }
```

It is important to note that all parameters in an inherited widget are *final*, meaning they cannot be reassigned once the widget has been built. That does not mean, however, that the values cannot be changed internally to the inherited widget. So, when would this be useful? Well, perhaps you want to attach some sort of *service object* such as a database handler or a Web proxy to access an API. Then, you could set it up once and invoke it from anywhere in the widget tree!

Flutter uses inherited widgets in several ways, right out of the box. Three of the biggest ways it is used are with `Theme`, `MediaQuery`, and `Navigator`. We are going to use all three of these inherited widgets throughout this book, so keep an eye out for them. For now, you can think of the `Theme` as an inherited widget that contains all sorts of styling properties that can be used to change the look and feel of widgets throughout the application. The `MaterialApp` and `CupertinoApp` widgets both accept a theme parameter, which they use to cascade down the styling properties. Meanwhile, `Navigator` is used to route the user to different parts of the application. Finally, the `MediaQuery` widget is one of the most useful tools available for building layouts. It contains tons of information about sizing and constraints, as well

as *many* more pieces of information that are useful when building flexible and responsive layouts. For example, we could use it to create a `SizedBox` whose width is precisely half its parent:

```
1. SizedBox(  
2.   width: MediaQuery.of(context).size.width / 2,  
3. )
```

Inherited widgets are some of the most useful widgets available in the whole toolkit due to their unique ability to allow us to access their data via the `BuildContext` of any other widget. However, it is not fair to keep bringing up `BuildContext` without explaining what it is, so let us investigate that now.

Introduction to BuildContext

In Flutter, we often like to think of everything as a widget. While this is not strictly true, it is also not far from the truth. What is more, each widget ends up in the widget tree, as we have discussed earlier in this chapter. How does any widget know what its place is in the widget tree? How does it know who its parent is, or what its constraints are? This is what `BuildContext` is for. Simply put, `BuildContext` contains information about where a widget is in the widget tree. But how does it do this?

We learned earlier about the widget tree, but as we touched upon briefly earlier, that is not the only tree that Flutter creates. Let us go back to our example from the beginning of the chapter to figure this all out:

```
1. Container(  
2.   color: Colors.lightBlue,  
3.   child: Row(  
4.     children: const [  
5.       Image(  
6.         image: NetworkImage(  
7.           'https://flutter.github.io/assets-for-api-  
docs/assets/widgets/owl.jpg'),
```

```
8.          ),  
9.          Text('Flutter is fun!'),  
10.         ],  
11.        ),  
12.      )
```

This is the widget tree which adds some text next to a picture of an owl. The widget tree this creates will look like *figure 4.17*:

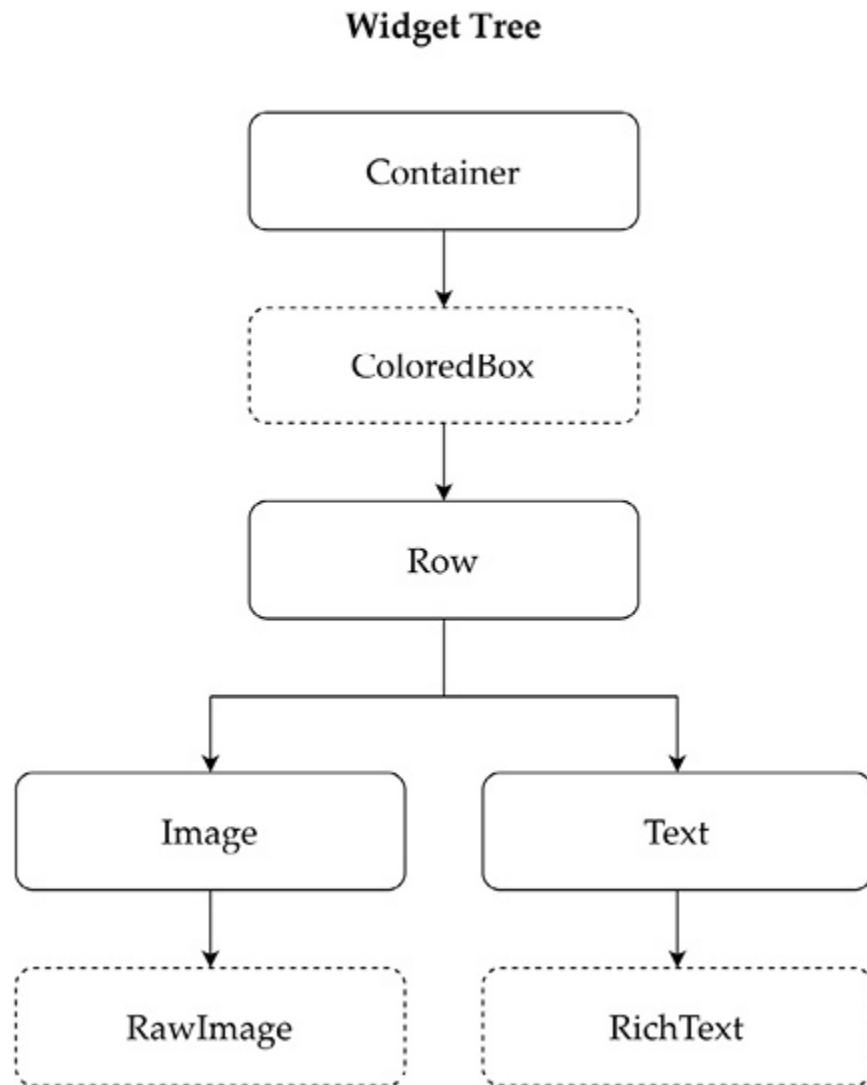


Figure 4.17: Our example widget tree.

Under the hood, Flutter uses these widgets as a blueprint to create an *element tree* (figure 4.18).

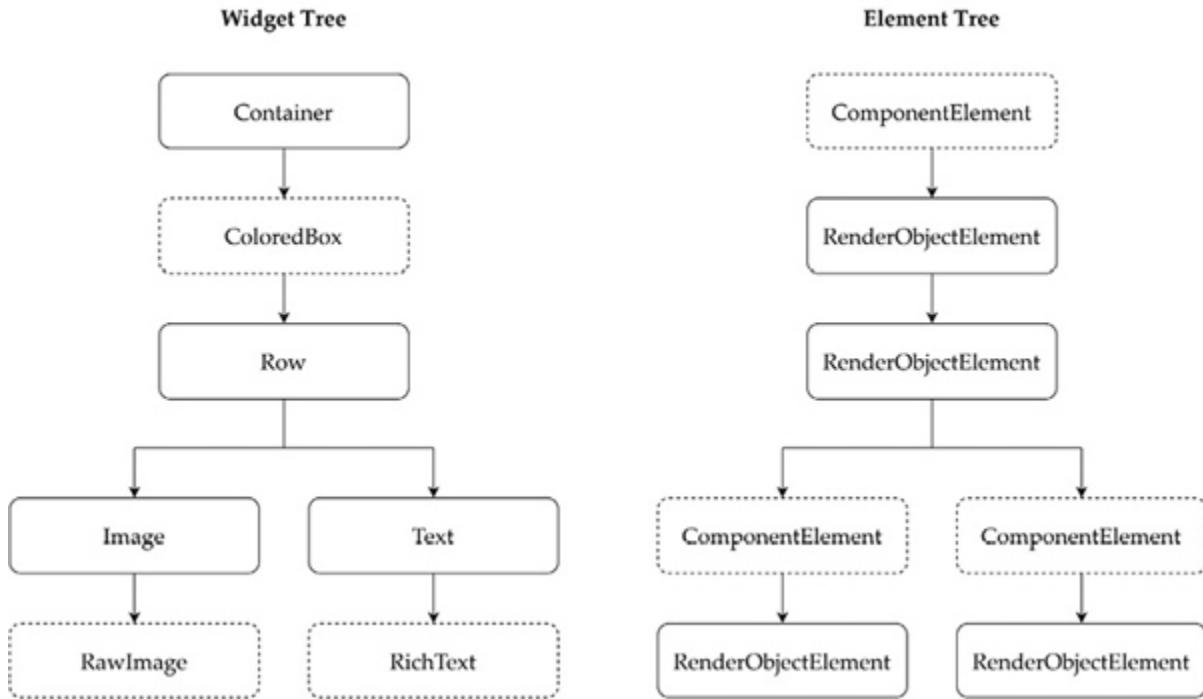


Figure 4.18: How the widget tree maps to the element tree.

For each widget created in the widget tree, Flutter creates a corresponding element in the element tree. These elements are eventually used to create a render tree, which is what Flutter uses to composite and draw the widgets to the screen using Skia or Impeller, which is slated to replace Skia, and may well have by the time this book makes it into readers' hands (see figure 4.19). Skia is an open source 2D graphics engine, developed by Google, released under a permissive BSD Free Software License, and is the core technology which powers the graphics engines for such things as Google Chrome, Chrome OS, Android, Flutter, Fuchsia, Mozilla Firefox, Firefox OS, and more. Impeller is the new, bespoke rendering engine created by the Flutter team to replace the usage of Skia and is a 3D rendering engine.

Elements on the element tree have an associated state object if the element is a `StatefulWidget`. When Flutter detects that the state of an element has changed, it will mark the corresponding widget as needing to be updated, causing it to become unmounted from the widget tree and replaced by a new widget, which is associated with the same element.

In the case that a stateful widget's state is changed, the state of the element is updated. When that occurs, the old widget is unmounted, and a new widget is mounted in its place. The element, and thus its state, never gets removed from the element tree — only the widget is replaced.

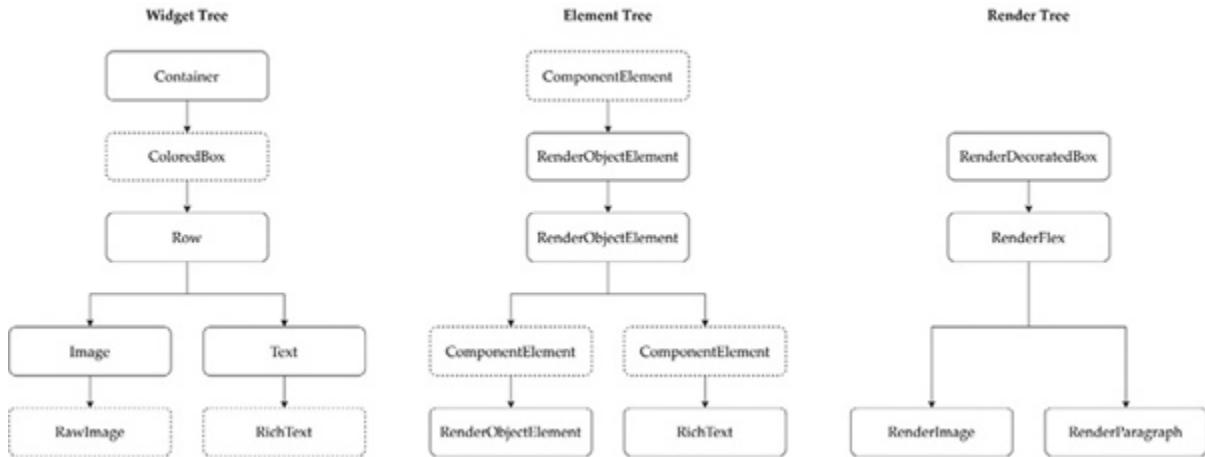


Figure 4.19: How all three trees relate.

This is where **BuildContext** comes in. Every element in the element tree — stateful elements and stateless elements — are extended from the base **Element** class. And what is an **Element**?

1. abstract class **Element** extends **DiagnosticableTree** implements **BuildContext**

It is nothing more than an implementation of **BuildContext**. So, any time you refer to **BuildContext**, you are referencing an element in the element tree.

Conclusion

In this chapter, we covered the three different types of widgets, including the widget lifecycle and how they are drawn to the screen. We also discussed how Flutter uses the widgets to create a widget tree, and then uses the widget tree to create an element tree and a render tree.

In the upcoming chapter, we are going to learn all about buttons, text fields, and other ways of collecting input from users. Before we do that, take a minute to congratulate yourself! These first four chapters have been densely packed with some very technical details. You have done well to make it this

far, and if there is anything you need to review, the pages will always be here for you.

Questions

1. What design paradigm does Flutter use?
2. What method is called for each widget when Flutter wants to draw a widget to the screen?
3. What are three ways you can determine what properties are available for a widget?
4. What is the difference between a positional parameter and a named parameter?
5. When creating a widget, how do you specify that a parameter is either positional or named?
6. What does the question mark after a type confer?
7. When would you need to use a **Key**?
8. What is a private object and how is one defined?
9. What is the lifecycle of a stateful widget?
10. What is the difference between a stateless and a stateful widget?
11. What is the purpose of an inherited widget? What is an example of one?
12. What is **BuildContext**?

Key Terms

- **BuildContext:** Contains information about where a widget is in the widget tree. It is the base class of an **Element**, which each element in the element tree is based on.
- **Element tree:** The tree that Flutter creates using widgets in the widget tree as configuration. The element tree does not mutate, meaning objects are not added or removed once they have been created. This allows corresponding widgets on the widget tree to be replaced when their state changes, without the need to rebuild and redraw the entire screen.

- **Inherited widget:** A type of widget that allows ancestors in the widget tree to access its properties, without the need to pass arguments down the widget tree. Some common examples are **Theme**, **MediaQuery**, and **Navigator**.
- **Render tree:** The tree Flutter builds from the element tree. The render tree is used by Flutter to give instructions to Skia or Impeller, which then draws to the canvas.
- **State:** A state object is created in the element tree for each stateful widget in the widget tree. The state is maintained as the widget is removed and re-added to the widget tree whenever the state has been updated.
- **Stateful widget:** A type of widget that has an associated **State** object, allowing for the widget to be changed and updated after it has been rendered to the screen. Stateful widgets are often used to add interactivity (for example, with a **Button**, **Radio**, **Checkbox**, and so on) but may also include things which perform background operations (such as fetching an image from the internet and displaying it once it has loaded).
- **Stateless widget:** A type of widget commonly used for the purposes of laying out the user interface or adding other non-interactive elements. Some stateless widgets include **Row**, **Column**, **Container**, **Text**, and **Center**, although there are many, many more available to choose from.
- **Widget:** a broad term that encompasses many of the elements available to developers to compose a user interface, add interactivity, and maintain state using Flutter.
- **Widget tree:** The tree-like representation of all the widgets, which make up all or part of an application written in Flutter. It is used as a blueprint by Flutter to configure the corresponding elements in the element tree.

Further Reading

- Be sure to check out the Flutter team's own documentation about widgets: <https://docs.flutter.dev/development/ui/widgets-intro>

- To learn more about Flutter's architecture, including an in-depth discussion of how widgets are mapped to the element tree and how that's used to create a render tree, take a look at the architectural overview: <https://docs.flutter.dev/resources/architectural-overview>
- Learn more about Dart's cascade notation: <https://dart.dev/guides/language/language-tour#cascade-notation>
- Take the Dart language tour! Learn all about the Dart programming language: <https://dart.dev/guides/language/language-tour>

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

Handling User Input

Introduction

One would be hard pressed to find many applications which did not include some sort of user input. Whether it is a signup or login form, a dropdown list of options, a group of radio buttons, or some checkboxes, handling user input in some way is an extremely common task. In this chapter, we will explore some of the different input widgets, how to validate user input and provide errors or feedback to the user, and how to submit the data for processing.

Structure

In this chapter, we will discuss the following topics:

- Buttons
- Capturing text input
- Pickers, selectors, and dropdowns
- Radios, checkboxes, switches, and segmented controls
- Sliders
- Forms and FormFields
- Building a simple signup form

Objectives

After completing this chapter, you will be familiar with many of the most common widgets that can be used to capture input from users. Since the applications you will be building can cater to users of different devices with different paradigms, you will learn to spot some of these differences and discover methods of gathering input from users using an appropriate method for the platform they are on. You will be able to ask a user for their birthday by way of a date picker, allow them to toggle options on and off, and select items from a list. You will learn how to validate the input in a text field against one or more rules and how to group several inputs into a form to validate or submit them as a group.

Buttons

There are many different types of buttons in Flutter. Between both the **Material** and **Cupertino** packages, there are several styles from which we can choose. Additionally, we can create our own buttons using whatever widgets we want. Let us look at some of the buttons in their default configurations, then explore how to make our own button.

Buttons, regardless of style, all have one thing in common: each of them has an **onPressed** callback function. But what exactly *is* a callback function? For our purposes, a callback is a function that is passed into a widget as a parameter that can then be invoked from within the widget, but it will be run in the context of the widget's parent. If this does not make sense quite yet, do not worry. Let us look at some examples of buttons and how their callbacks work.

First, we will start by looking at **CupertinoButton**, the ubiquitous iOS-styled button available to us, thanks to the **Cupertino** package. Consider the following widgets (*figure 5.1*):

1. const CupertinoButton.filled(
2. onPressed: null,
3. child: Text('Disabled'),
4.),

```
5. CupertinoButton(  
6.   onPressed: () {},  
7.   child: const Text('Enabled'),  
8. ),  
9. CupertinoButton.filled(  
10.  onPressed: () {},  
11.  child: const Text('Enabled'),  
12. ),
```

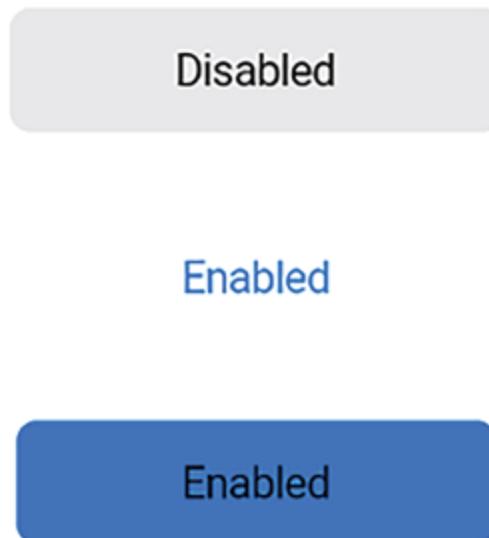


Figure 5.1: Various states of a CupertinoButton

There are two variants of the `CupertinoButton`: `CupertinoButton` and `CupertinoButton.filled`. The first button is the `filled` variety. As the name implies, this variant comes with a solid background. The disabled state of the button is grey, whereas the enabled state is blue, such as with the third button. The color and disabled color can be changed by passing a `Color` to the `color` and `disabledColor` properties, respectively.

We can see that there are two states: enabled and disabled, but how are these states controlled, since the `enabled` property is read-only? According to the documentation, the button is disabled by default. A button can be enabled by setting its `onPressed` property to a non-null value. So, by

providing a callback function, the button is enabled. Look at the code again and note that when the `onPressed` property is null, the button becomes a `const`.

Let us talk about this `onPressed` callback function again. The default value we have been using is a simple `0 {}`, which provides an empty function that performs no actions when it is invoked. Any code we want to run when the button is tapped will have to be inside the curly braces. If there was a callback value being passed to the function from the button, we would set that variable inside the parenthesis. However, the `CupertinoButton`, like other buttons, does not provide a value, so the parenthesis will be empty.

Try adding something between the curly braces, then tapping the button. Can you see your code being invoked?

```
1. () {  
2.   print("The button was pressed");  
3. }
```

This is the mechanism by which we will eventually be able to submit a form. For now, let us briefly look at some of the other buttons available to us, and then we will learn to create our own.

From the `Material` package, here is an `OutlinedButton` (*figure 5.2*):

```
1. OutlinedButton(  
2.   onPressed: () {},  
3.   child: const Text('OutlinedButton'),  
4. )
```

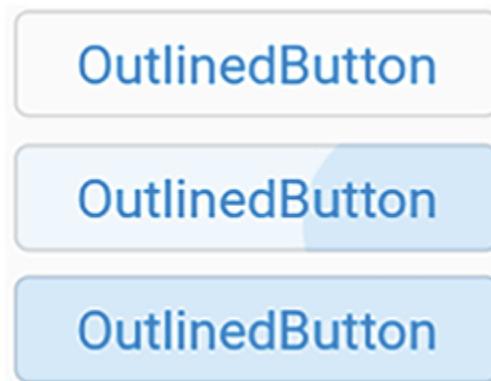


Figure 5.2: Visualizing the state changes of an OutlinedButton when tapping the lower right corner

There is even a variant which allows us to use an icon, which can be invoked with the `OutlinedButton.icon` constructor and then passing an icon widget to it. Note that the text is moved to a `label` parameter, rather than the `child` parameter (*figure 5.3*).

```
1. OutlinedButton.icon(  
2.   onPressed: () {},  
3.   icon: const FlutterLogo(),  
4.   label: const Text('OutlinedButton'),  
5. )
```



Figure 5.3: The icon variant of OutlinedButton

The `ElevatedButton` has a similar look and feel to the `CupertinoButton.filled` with a material design take, including a subtle drop shadow (*figure 5.4*).

```
1. ElevatedButton(  
2.   onPressed: () {},  
3.   child: const Text('ElevatedButton'),  
4. ),
```

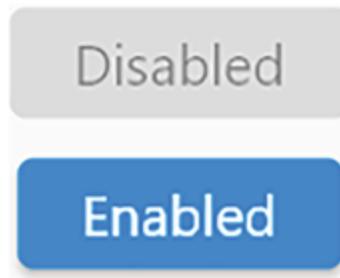


Figure 5.4: An ElevatedButton

If you are using the `Material` package and need a button that is a bit simpler, try the `TextButton` (*figure 5.5*).

```
1. TextButton(  
2.   onPressed: () {},  
3.   child: const Text('TextButton'),  
4. ),
```

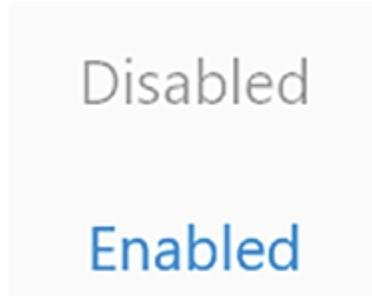


Figure 5.5: A simple TextButton

Sometimes it doesn't make sense for your button to have text. In those instances, look toward the **IconButton** for a solution (*figure 5.6*).

```
1. IconButton(  
2.   icon: const Icon(Icons.flutter_dash),  
3.   onPressed: () {},  
4. ),
```

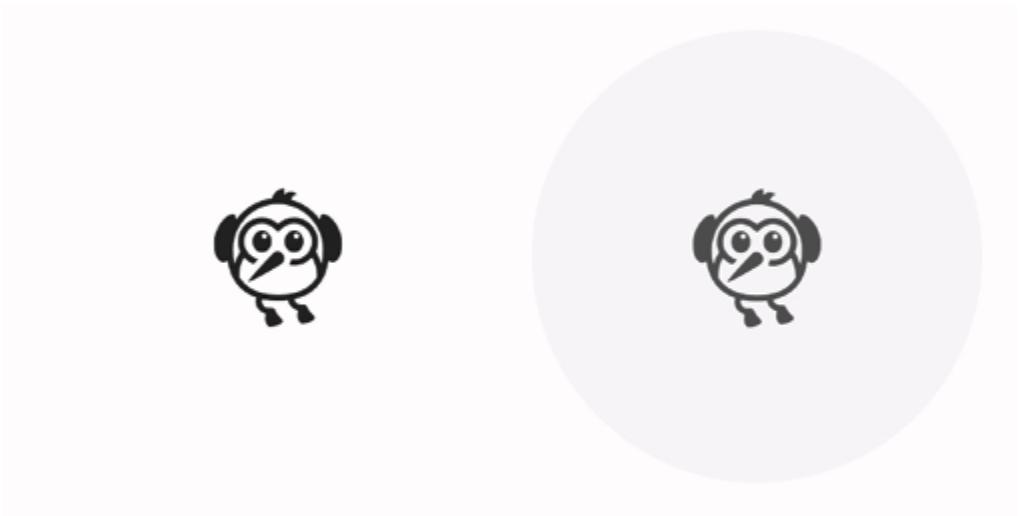


Figure 5.6: The default state (left) and hovered state (right) of an IconButton with Dash

Whichever button you choose, be sure to explore the button's relevant style option(s) to customize it to your liking. Let us say that none of these buttons give you quite what you are looking for, though. You have evaluated all of them and your app just needs something a bit more customized.

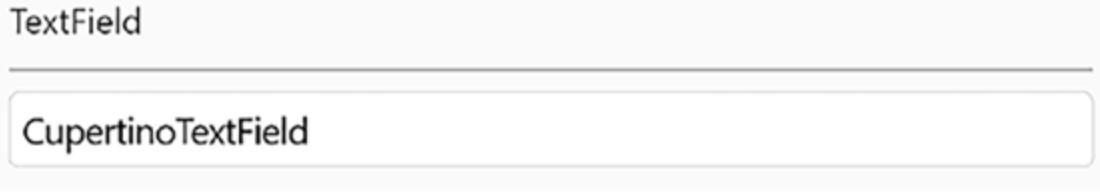
Thankfully, Flutter gives us the flexibility we need to easily accomplish this task. By leveraging the `GestureDetector`, we can turn *any* widget into a button. What is more, `GestureDetector` provides a nearly limitless array of possibilities when it comes to ways you can capture taps, swipes, and other gestures. For devices that support it, `GestureDetector` even has support for force presses.

In fact, `GestureDetector` is *so* powerful, it can be used for *far* more than just custom buttons. Use it to drag items around the screen, pan and zoom, capture taps, double taps, triple taps, and more with ease. Here is an example where we turn a humble `Container` into a button that only works when you double tap it:

```
1. GestureDetector(  
2.   onDoubleTap: () {},  
3.   child: Container(  
4.     height: 100,  
5.     width: 100,  
6.     color: Theme.of(context).primaryColor,  
7.   ),  
8. ),
```

Capturing Text Input

When it comes to capturing text input from users, we have a few options available to us. The `Cupertino` package provides the `CupertinoTextField` widget, whereas the `Material` package gives us the `TextField` (*figure 5.7*).



TextField

CupertinoTextField

Figure 5.7: *TextField* (top) and *CupertinoTextField* (bottom) with text entered in each

There are a seemingly infinite number of options available to customize the look and feel of either input widget, with `CupertinoTextField` accepting a `BoxDecoration` object in its `decoration` property and `TextField` accepting an `InputDecoration` object in its. Both styles can have `prefix` and `suffix` widgets to display a widget on either the left or ride side of the input. This could be useful for specifying a currency symbol, a password visibility toggle, and the like. Here, we have added both a `prefix` and `suffix` to either field (*figure 5.8*).

```
1. class TextFieldExample extends StatelessWidget {  
2.   const TextFieldExample({super.key});  
3.   @override  
4.   Widget build(BuildContext context) {  
5.     const Icon _prefixIcon = Icon(Icons.currency_bitcoin);  
6.     const Icon _suffixIcon = Icon(Icons.lock);  
7.     return Column(  
8.       children: const [  
9.         TextField(  
10.           decoration: InputDecoration(  
11.             icon: _prefixIcon,  
12.             suffixIcon: _suffixIcon,  
13.           ),  
14.         ],  
15.       );  
16.     }  
17.   }  
18. }  
19. 
```

```

14. ),
15.         SizedBox(height: 10),
16.         CupertinoTextField(
17.             prefix: _prefixIcon,
18.             suffix: _suffixIcon,
19.         ),
20.     ],
21. );
22. }
23. }

```



Figure 5.8: Adding a prefix and suffix icon to each text input style

There are plenty of other styling options available, so be sure to explore the various options when you decide to implement these widgets for yourself.

Almost more interesting than styling are the various functional options of these fields. Let us explore a few of these useful options, such as `keyboardType`, `textInputAction`, and `inputFormatters`.

The `keyboardType` property accepts a `TextInputType` object as its value. `TextInputType` has several options available to us to pick from:

1. <code><TextInputType>[text, multiline, number, phone, datetime, emailAddress, url, visiblePassword, name, streetAddress, none]</code>

Presenting an appropriate keyboard to users when they are trying to enter text helps to build trust and goodwill between you, as the developer, and

your users, who may have varying levels of expectations. Think of a time when you were presented with a text input that wanted a numerical value, yet you were presented with a full keyboard, or a time when you were asked to enter your email address, but the @ key was hidden behind a function modifier. These are modern annoyances we all deal with from time to time, but the truth is that the tools to provide a better user experience are available for us to take advantage of, and we should do so whenever possible.

While we are on the subject, let us talk about the `textInputAction`. This is the default action of the enter key on the keyboard on mobile devices. Certainly, you have noticed that this button has changed from time to time. Sometimes it will display an arrow, or a carriage return symbol. Other times, it may display a send icon or a search icon. The options available to Android differ slightly from those on iOS, so it's a good idea to read up on the documentation. Here's a list of the available options:

1. <TextInputAction>[continueAction, done, emergencyCall, go, join, newline, next, none, previous, route, search, send, unspecified]

Android does not have any concept of the `continueAction`, `emergencyCall`, `join`, or `route` options, so ensure that you take this into account when writing your code.

Each text input can optionally have an associated `TextEditingController` associated with it. Generally, you want to create a variable outside of the `build()` method where you assign a `TextEditingController` to its value and then pass that variable into the text input's `controller` parameter. This could look something like the following:

```
1. class TextInputDemo extends StatefulWidget {  
2.     const TextInputDemo({super.key});  
3.       
4.     @override  
5.     State<TextInputDemo> createState() => _TextInputDemoState();  
6. }
```

```
7.  
8. class _TextInputDemoState extends State<TextInputDemo> {  
9.     final      TextEditingController      _textController      =  
    TextEditingController();  
10.  
11.    @override  
12.    Widget build(BuildContext context) {  
13.        return TextField(  
14.            controller: _textController,  
15.        );  
16.    }  
17. }
```

Ok... you may be asking yourself, *but why?* An excellent question, indeed. Think of this: if you want to retrieve the text that the user has entered in the field, you need a `TextEditingController` to do so, as one example. You would do this by calling `_textController.value.text`, but it is not the only thing you can do with a `TextEditingController`. You could also use it to clear the field when a user taps a button. There are lots of possibilities available to you, so it is worth spending some time thinking about and exploring the `TextEditingController` to see what else it is capable of. Of course, be sure to dispose of it properly in your widget:

```
1. @override  
2. void dispose() {  
3.     _textController.dispose();  
4.     super.dispose();  
5. }
```

Finally, let us talk about the `inputFormatters` option. This parameter accepts a list of validation and formatting functions, which are applied from top to bottom each time the text in the field changes (for example, on each

keypress). We can use this to do things like limiting the characters which are allowed to be typed into a field or automatically apply formatting to the text, such as entering dashes into phone numbers automatically.

In this example, we only allow numbers to be typed into a `TextField`:

```
1. TextField(  
2.   inputFormatters: [  
3.     FilteringTextInputFormatter.allow(RegExp(r'[0-9]')),  
4.   ],  
5. )
```

Since the `inputFormatters` property accepts a list of input formatters, you can apply multiple formatters to input. Just keep in mind that the filters are applied from top to bottom.

Pickers, Selectors, and Dropdowns

There are more inputs available than just text input. If you want to ask a user for their birthday, text input may not be the best option. Likewise, asking someone to choose an item from a list and type it in would be far less preferable to asking them to pick an item from a dropdown. Let us explore some of the input options which cover these use cases.

Let us start by exploring the `Material` style date picker. Rather than being defined as a widget, the Material date picker is invoked using the function `showDatePicker`. Four parameter values are required for this function, but many other options are available. The required parameters are as follows:

- `context`
- `firstDate`
- `lastDate`
- `initialDate`

Aside from `context`, which should have an obvious purpose at this point, the remaining three values expect a `DateTime` object. You could simply use `DateTime.now()` to get a value, which is, well, *now*. There is also a `DateTime.parse()` method, where a `DateTime` object will be constructed from a

string of text, which represents a given moment in time. The documentation on the method itself provides example strings that it can parse, but one such example could be `DateTime.parse("2000-01-01")`. Several other options are available, as well. Say you want a point in time that is precisely one year ago from the current time. `DateTime.now().subtract(const Duration(days: 365))` would give you what you are looking for.

Let us put all this knowledge into practice by creating a button that displays a material date picker.

```
1. ElevatedButton(  
2.   child: const Text("Press me"),  
3.   onPressed: () async {  
4.     await showDatePicker(  
5.       context: context,  
6.       firstDate: DateTime.now().subtract(const Duration(days:  
365 * 50)),  
7.       lastDate: DateTime.now(),  
8.       initialDate: DateTime.parse("2000-01-01"),  
9.     );  
10.    },  
11.  )
```

This example will give us a date picker from where we can choose from any day in the past 50 years. Here is what that looks like (*figure 5.9*):

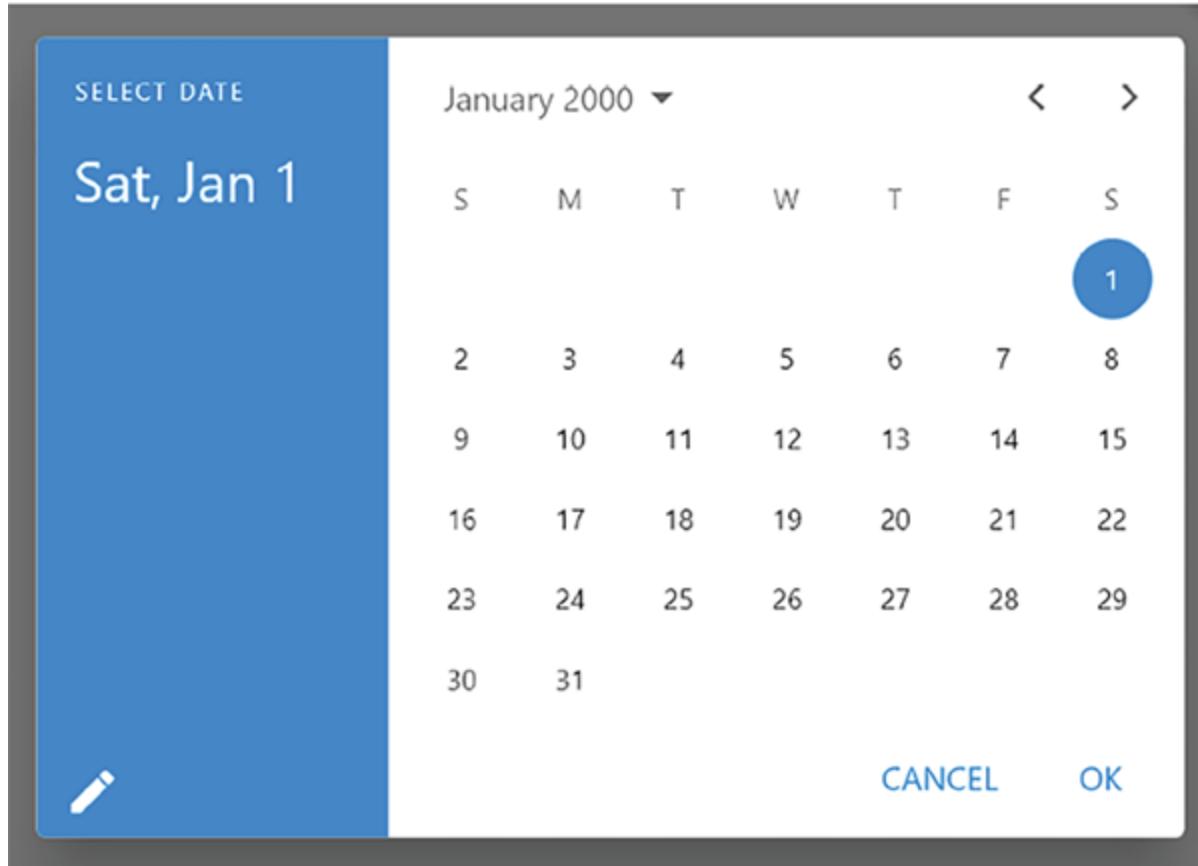


Figure 5.9: The Material date picker

There is even a companion method, `showTimePicker`, which will allow you to choose a time of day the same way you would choose a date. The only required parameters are `context` and `initialTime`, which takes a `TimeOfDay` object. Here is what it looks like in practice (*figure 5.10*):

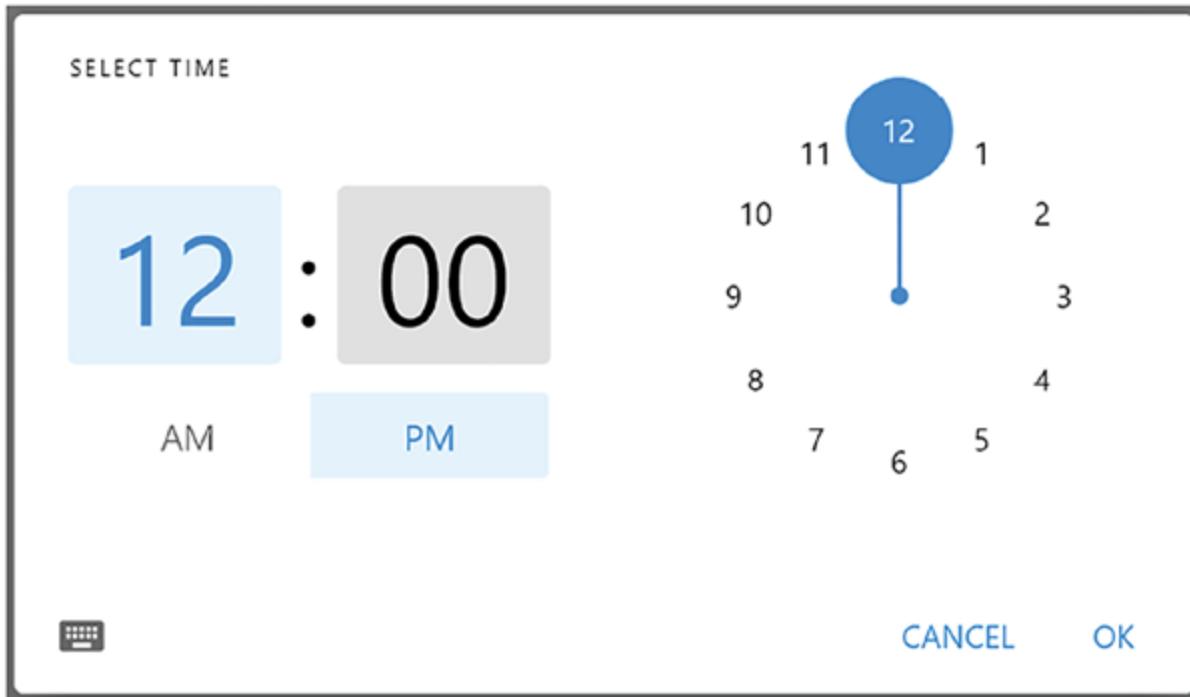


Figure 5.10: The Material time picker

The **Material** pickers are not the only game in town, though. How about the **Cupertino** style? Unlike the **Material** pickers, the **Cupertino** pickers are widgets. However, they take a bit more setup work. First, we will need a modal popup that we can pin to the bottom of the display. Inside our widget, we will define it something like this:

```
1. void _showDialog(BuildContext context, Widget child) {  
2.   showCupertinoModalPopup<void>(  
3.     context: context,  
4.     builder: (BuildContext context) => Container(  
5.       height: 216,  
6.       margin: EdgeInsets.only(  
7.         bottom: MediaQuery.of(context).viewInsets.bottom,  
8.       ),  
9.       color:  
  CupertinoColors.systemBackground.resolveFrom(context),
```

```
10.         child: SafeArea(  
11.             top: false,  
12.             child: child,  
13.         ),  
14.     ),  
15. );  
16. }
```

This will give us a bottom-aligned dialog in which we can display the picker. Next, we can invoke that dialog and add the `CupertinoDatePicker` as its child:

```
1. ElevatedButton(  
2.     child: const Text("Press me"),  
3.     onPressed: () => _showDialog(  
4.         context,  
5.         CupertinoDatePicker(  
6.             initialDateTime: DateTime.parse("2000-01-01"),  
7.             mode: CupertinoDatePickerMode.date,  
8.             onDateTimeChanged: (DateTime newDate) {  
9.                 print('$newDate');  
10.            },  
11.        ),  
12.    ),  
13. )
```

This will result in the following date picker being displayed (*figure 5.11*):

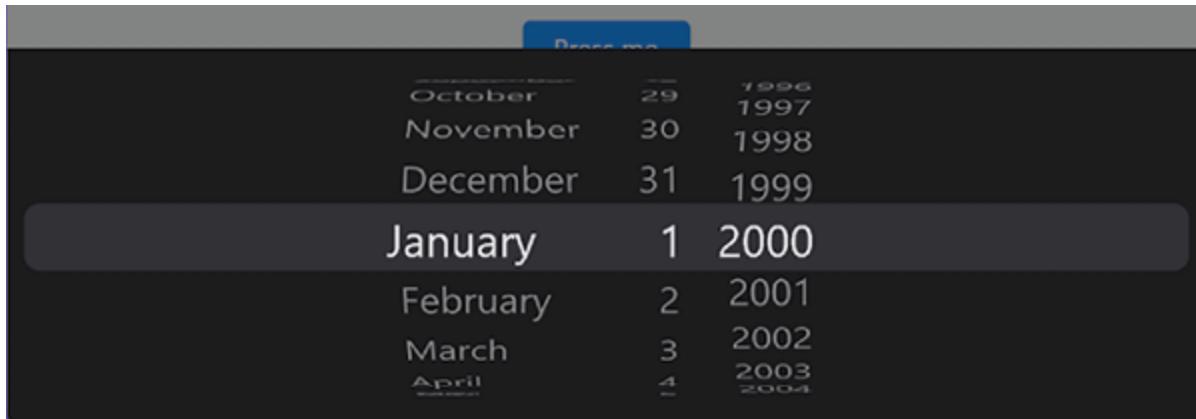


Figure 5.11: A Cupertino date picker with the dark theme applied

By swapping out the picker for a `CupertinoTimerPicker`, you could determine a duration of time for, say, a timer. Here is what it would look like to get an hours and minutes duration (*figure 5.12*):

1. `CupertinoTimerPicker()`
2. `mode: CupertinoTimerPickerMode.hm,`
3. `onTimerDurationChanged: (Duration duration) {`
4. `print('duration');`
5. `},`
6. `)`

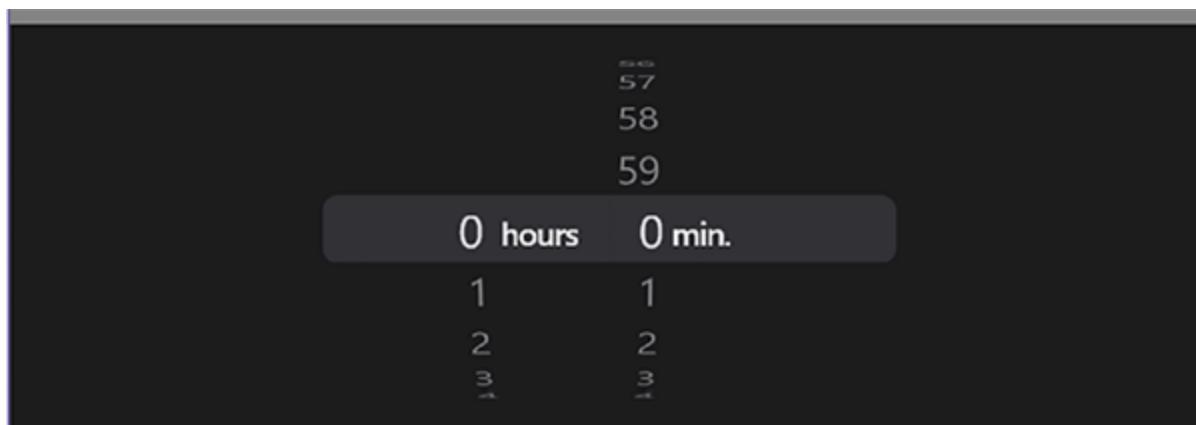


Figure 5.12: A Cupertino timer picker with the dark theme applied

Of course, the picker is far more versatile than just getting a date or a duration. In fact, it could be used for any arbitrary list of items. Let us look

at an example where you can choose a day of the week. First, we need to define those days. We will do that in a list:

```
1. static const List<String> daysOfWeek = [  
2.   "Monday",  
3.   "Tuesday",  
4.   "Wednesday",  
5.   "Thursday",  
6.   "Friday",  
7.   "Saturday",  
8.   "Sunday",  
9. ];
```

Next, we will use a **CupertinoPicker** to display each item in the list as an option. The **itemExtent** property tells the picker what *height* to give each item in the picker list.

```
1. CupertinoPicker(  
2.   itemExtent: 30,  
3.   useMagnifier: true,  
4.   children: daysOfWeek.map((day) => Text(day)).toList(),  
5.   onSelectedIndexChanged: (int selectedItem) {  
6.     print(daysOfWeek[selectedItem]);  
7.   },  
8. )
```

And just like that, we have taken an arbitrary list of items and created a picker from them (*figure 5.13*):

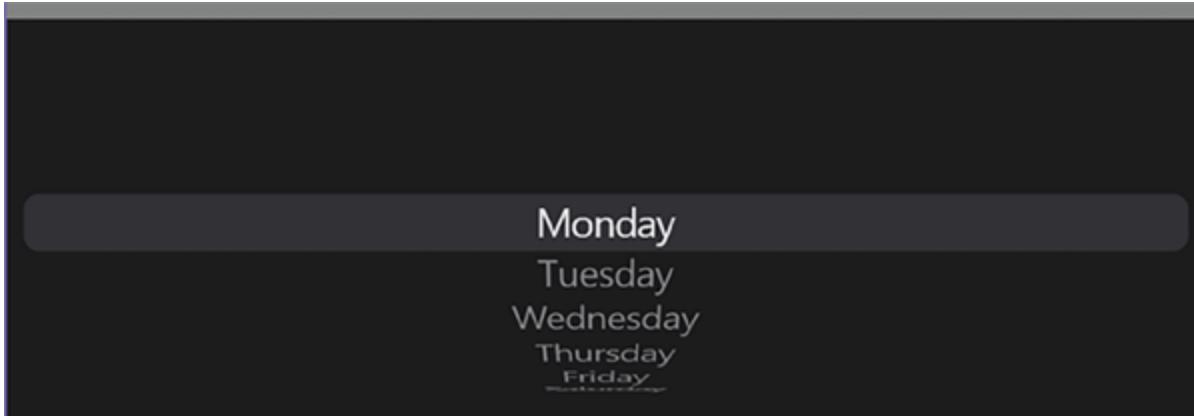


Figure 5.13: A Cupertino picker of an arbitrary list of items with the dark theme applied

Note that in this example we used a list of strings, but in practice you could use *any* type of data.

Finally, let us look at the `DropdownButton` widget, provided by the `Material` package. Like the generic picker from the `Cupertino` package, we can use this widget to allow users to pick from a list of items. Here, we have converted the `Cupertino` weekday picker into a material dropdown:

```
1. class WeekdayDropdown extends StatefulWidget {  
2.     static const List<String> daysOfWeek = [  
3.         "Monday",  
4.         "Tuesday",  
5.         "Wednesday",  
6.         "Thursday",  
7.         "Friday",  
8.         "Saturday",  
9.         "Sunday",  
10.    ];  
11.      
12.    const WeekdayDropdown({super.key});  
13.      
14. }
```

```
14.    @override
15.    State<WeekdayDropdown>          createState()      =>
16.    _WeekdayDropdownState();
17.
18.    class           _WeekdayDropdownState        extends
19.    State<WeekdayDropdown> {
20.
21.    @override
22.    Widget build(BuildContext context) {
23.        return DropdownButton<String>(
24.            value: dropdownValue,
25.            icon: const Icon(Icons.arrow_downward),
26.            onChanged: (String? newValue) {
27.                print(newValue);
28.            },
29.            items:
30.            WeekdayDropdown.daysOfWeek.map<DropdownMenuItem<String>>((String value) {
31.                return DropdownMenuItem<String>(
32.                    value: value,
33.                    child: Text(value),
34.                );
35.            }).toList(),
36.        );
37.    }
38.
```

```
36.    }
```

```
37. }
```

Here it is in action (*figure 5.14*):

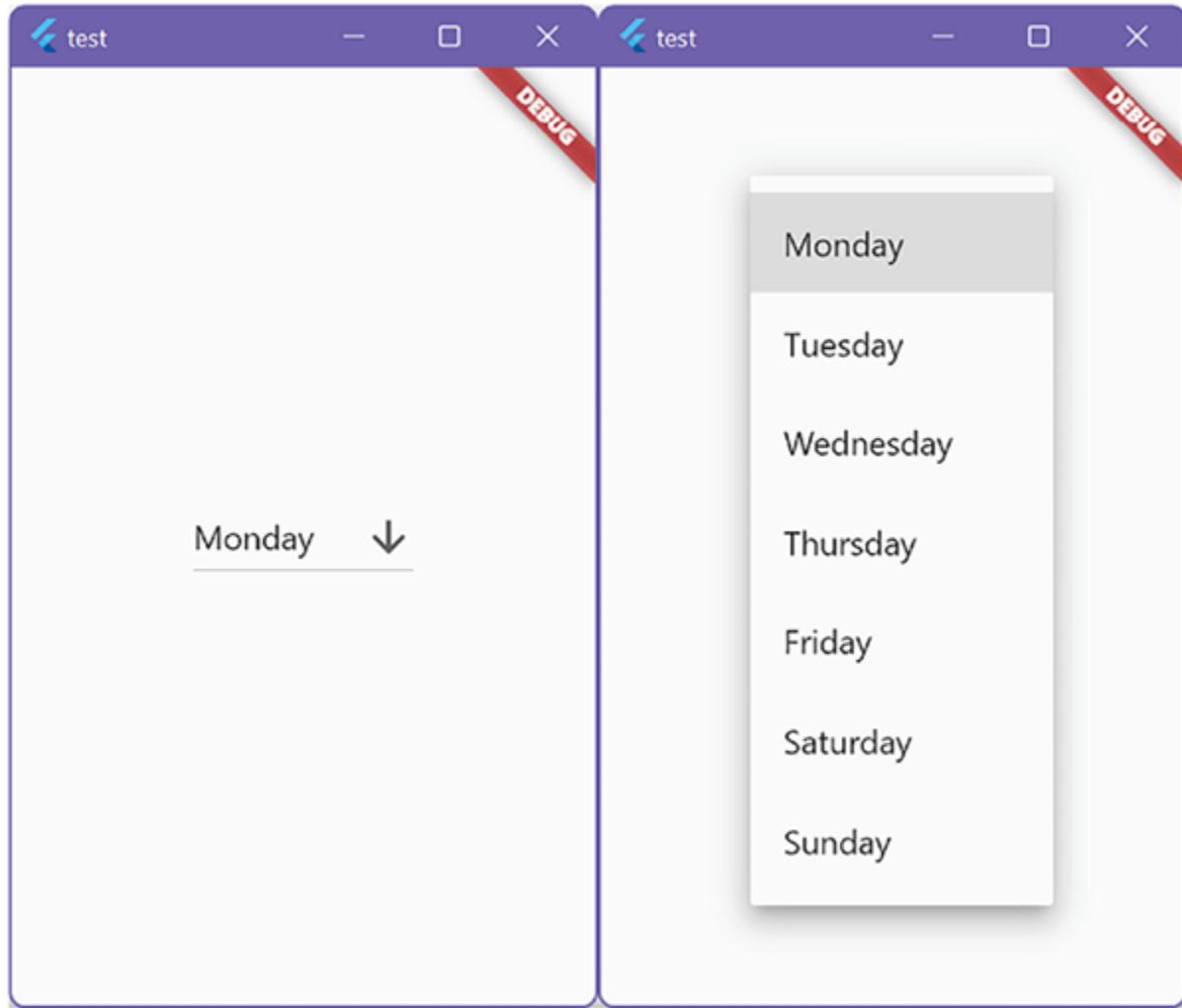


Figure 5.14: A Material dropdown

[Radios, Checkboxes, Switches, and Segmented Controls](#)

Choosing an item from a list is not always the appropriate method, either. Sometimes what we need is to choose multiple items from a small group of similar items or a single item from a small number of items, which would not make sense in a dropdown. That is where this class of inputs comes in handy. Let us delve in.

When it comes to choosing items from a small, mutually exclusive list, the humble radio needs no introduction. Let us look at it in action. One thing to note is that the radio needs to be in a stateful widget so the value can be updated when the user selects a new option. For brevity's sake, we will forego the stateful widget portion of the code, and instead focus on the state itself.

```
1. class _MaybeState extends State<Maybe> {  
2.   String _value = "Yep";  
3.   @override  
4.   Widget build(BuildContext context) {  
5.     return Column(  
6.       children: <Widget>[  
7.         ListTile(  
8.           title: const Text('Yep'),  
9.           leading: Radio<String>(  
10.             value: "Yep",  
11.             groupValue: _value,  
12.             onChanged: (String? value) {  
13.               setState(() {  
14.                 _value = value!;  
15.               });  
16.             },  
17.             ),  
18.             ),  
19.         ListTile(
```

```

20.           title: const Text('Nope'),
21.           leading: Radio<String>(
22.             value: "Nope",
23.             groupValue: _value,
24.             onChanged: (String? value) {
25.               setState(() {
26.                 _value = value!;
27.               });
28.             },
29.           ),
30.         ),
31.       ],
32.     );
33.   }
34. }

```

This will produce two radios which a user can choose from: *Yep* and *Nope*. Here is what it looks like in action (*figure 5.15*):

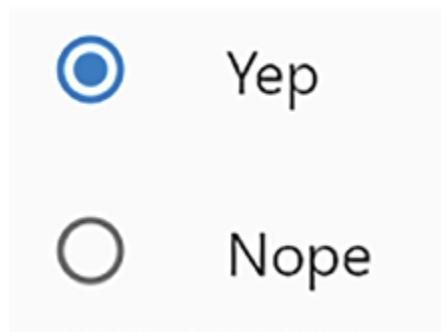


Figure 5.15: Material radio buttons

This might feel out of place for an iOS user, though. Here is the same example using the `CupertinoSegmentedControl`, which ought to feel more appropriate on iOS:

```
1. class _MaybeState extends State<Maybe> {
2.   String _value = "Yep";
3.   @override
4.   Widget build(BuildContext context) {
5.     return CupertinoSegmentedControl(
6.       groupValue: _value,
7.       onValueChanged: (String? value) {
8.         setState(() {
9.           _value = value!;
10.        });
11.      },
12.      children: const {
13.        'Yep': Padding(
14.          padding: EdgeInsets.symmetric(horizontal: 10),
15.          child: Text('Yep'),
16.        ),
17.        'Nope': Padding(
18.          padding: EdgeInsets.symmetric(horizontal: 10),
19.          child: Text('Nope'),
20.        ),
21.      },
22.    );
23.  }
24. }
```

And, of course, here it is in action (*figure 5.16*):



Figure 5.16: A Cupertino segmented control

There is even an iOS 13-style version, where the selector slides from option to option. This is, appropriately, a `CupertinoSlidingSegmentedControl`, and it can be used interchangeably with the `CupertinoSegmentedControl`, like the following (*figure 5.17*):

```
1. class _MaybeState extends State<Maybe> {  
2.   String _value = "Yep";  
3.   @override  
4.   Widget build(BuildContext context) {  
5.     return CupertinoSlidingSegmentedControl(  
6.       groupValue: _value,  
7.       onValueChanged: (String? value) {  
8.         setState(() {  
9.           _value = value!;  
10.        });  
11.      },  
12.      children: const {  
13.        'Yep': Padding(  
14.          padding: EdgeInsets.symmetric(horizontal: 10),  
15.          child: Text('Yep'),  
16.        ),
```

```

17.         ' Nope ': Padding(
18.             padding: EdgeInsets.symmetric(horizontal: 10),
19.             child: Text('Nope'),
20.         ),
21.     },
22. );
23. }
24. }
```



Figure 5.17: The Cupertino sliding segmented control

Finally, let us discuss checkboxes and selecting multiple items. The **Material** package gives us the **Checkbox** widget, which we are all familiar with. What you might not expect is that there are three possible states for a checkbox, including checked, unchecked, and null. The third of those is only available when the `tristate: true` property is set. Here is an example of the three different states (*figure 5.18*):

```

1. class _CheckboxExampleState extends State<CheckboxExample> {
2.     bool checkOneIsChecked = true;
3.     bool checkTwoIsChecked = false;
4.     bool? checkThreeIsChecked;
5.     @override
6.     Widget build(BuildContext context) {
7.         return Row(
8.             mainAxisAlignment: MainAxisAlignment.center,
```

```
9.     children: [
10.       Checkbox(
11.         value: checkOneIsChecked,
12.         onChanged: (bool? value) {
13.           setState(() {
14.             checkOneIsChecked = value!;
15.           });
16.         },
17.       ),
18.       Checkbox(
19.         value: checkTwoIsChecked,
20.         onChanged: (bool? value) {
21.           setState(() {
22.             checkTwoIsChecked = value!;
23.           });
24.         },
25.       ),
26.       Checkbox(
27.         tristate: true,
28.         value: checkThreeIsChecked,
29.         onChanged: (bool? value) {
30.           setState(() {
31.             checkThreeIsChecked = value;
32.           });
33.         },
34.       ),
35.     ],
36.   );
37. 
```

```
33.     },
34.   ),
35. ],
36. );
37. }
38. }
```

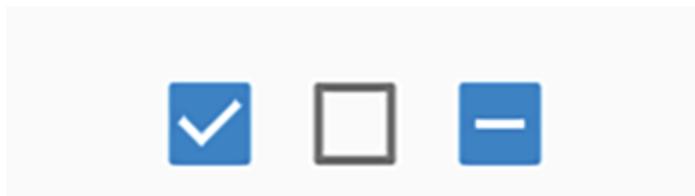


Figure 5.18: Checked, unchecked, and tri-state (null) checkboxes

Checkboxes might feel somewhat out of place for iOS users, so consider offering them the option of a `CupertinoSwitch`, instead (*figure 5.19*):

```
1. class _SwitchExampleState extends State<SwitchExample> {
2.   bool switchOne = true;
3.   bool switchTwo = false;
4.   @override
5.   Widget build(BuildContext context) {
6.     return Row(
7.       mainAxisAlignment: MainAxisAlignment.center,
8.       children: [
9.         CupertinoSwitch(
10.           value: switchOne,
11.           onChanged: (bool? value) {
12.             setState(() {
13.               switchOne = value!;
```

```
14.           });
15.           },
16.           ),
17.           CupertinoSwitch(
18.             value: switchTwo,
19.             onChanged: (bool? value) {
20.               setState(() {
21.                 switchTwo = value!;
22.               });
23.             },
24.             ),
25.             ],
26.           );
27.         }
28.     }
```



Figure 5.19: An example of the CupertinoSwitch

Of course, if you like the idea of a `CupertinoSwitch` and you find yourself wanting a `Material` variant, simply swap it out for the `Switch` widget to get that `Material` style (*figure 5.20*):



Figure 5.20: The Material style switch

Sliders

Occasionally, it might make sense to offer the user the option to pick a number from a range. A common way to do this is by allowing the user to drag a slider. The Material `Slider` widget has plenty of options available, such as setting the `min` and `max` values, adding stops by way of the `divisions` parameter, and even adding a `label` above the slider when it is being dragged (*figure 5.21*).

```
1. class _SliderExampleState extends State<SliderExample> {  
2.     double _currentSliderValue = 20;  
3.     @override  
4.     Widget build(BuildContext context) {  
5.         return Center(  
6.             child: Slider(  
7.                 value: _currentSliderValue,  
8.                 max: 100,  
9.                 divisions: 5,  
10.                label: _currentSliderValue.round().toString(),  
11.                onChanged: (double value) {  
12.                    setState(() {  
13.                        _currentSliderValue = value;  
14.                    });  
15.                },  
16.            ),  
17.        );  
18.    }
```

```
19. }
```



Figure 5.21: The default Material slider state (top) and the active state when dragging (bottom)

Although the **Cupertino** variant lacks the label, many of the options are the same (*figure 5.22*):

```
1. class _SliderExampleState extends State<SliderExample> {  
2.     double _currentSliderValue = 20;  
3.     @override  
4.     Widget build(BuildContext context) {  
5.         return Center(  
6.             child: CupertinoSlider(  
7.                 value: _currentSliderValue,  
8.                 max: 100,  
9.                 divisions: 5,  
10.                onChanged: (double value) {  
11.                    setState(() {  
12.                        _currentSliderValue = value;  
13.                    });  
14.                },  
15.            ),
```

```
16.      );
17.  }
18. }
```



Figure 5.22: The Cupertino slider

Forms and FormFields

If you would like to collect a bunch of input from a user all at once and submit it to a backend system for processing, it makes sense to group the inputs together in a **Form**. Wrapping input fields in a **Form** widget is not required, but doing so has some advantages. For example, it opens the possibility of resetting all inputs with a single action or validating all the text fields simultaneously, plus the state of the form can be both saved and reset.

The first thing you will need for your form is a **Key**. In this case, you will want to use a **GlobalKey<FormState>** for the form key. Assign this to a variable outside of the **build** method, then set the **Form**'s **key** property to the variable.

Now that you have got a **Form** with an associated key, you can check to see if all elements in the form are valid using **_formKey.currentState!.validate()** or reset the form back to its default state using **_formKey.currentState!.reset()**. Validating the **Form** will run the **validator** on each **FormField** which is a descendant of the **Form**.

This brings us to the **FormField** and validators. A **FormField**, when used inside a **Form**, makes it easier to do things like save the state of the **FormField**. When this happens, the **onSaved** callback is run on the field. If you have ever used an application that automatically saves your changes as soon as you are done editing a field (for example, typing your name into a field and the changes are saved automatically as soon as you tap out of the input), this is the same sort of mechanism.

The save, reset, and validation of a **FormField** is not bound explicitly to the **FormField** being a descendant of a **Form**. In fact, you could attach a **GlobalKey** to the **FormField** itself and perform these same actions. What makes the **Form** powerful is that only a single **GlobalKey** is needed to save, reset, and validate *all* the ancestor **FormField** widgets of that **Form**.

FormField, itself, is somewhat abstract since it is not bound to any specific type of input method. There are other widgets that leverage and extend the **FormField**, such as **TextFormField**, which you may recognize as being very similar to our earlier example of the **TextField**. In fact, these widgets are identical, save that the **TextFormField** is simply a **TextField** wrapped in a **FormField**. This opens for us the possibility of using the **onSaved** and **validator** methods, which were otherwise unavailable in the standard **TextField**. In addition, we are also able to set a default value inside the field, which we could not have done with a **TextField**.

Now that we understand what a **TextFormField** is let us look at what validation is and how to put it to good use. Broadly speaking, input validation is the process of analyzing the input provided by a user and comparing it to a set of rules which will determine if the input is valid.

Let us look at an example.

A user has been asked to enter their full legal name into a text field. An assumption you might make is that they will have both a given name and a surname. A validator on the input might check to make sure that at least two distinct words have been entered into the field. Here is what that could look like, in practice:

```
1. validator: (String? value) {  
2.   if (value.trim().split(' ').length < 2) {  
3.     return "Please enter your full, legal name.";  
4.   }  
5.   return null;  
6.  
7. }
```

We start by trimming any whitespace from the input because failing to do so would cause validation to pass as soon as the user enters some string and presses the spacebar. Next, we split the input into words, using a space as a delimiter. This takes each word that is separated by a space and adds it to a list. Then we check to see that there are at least two items in the list. If there are, validation was successful, so we return a null value. If validation fails, we return a message that describes the validation problem to the user.

Simply supplying a validator to a field is not enough to trigger validation of that field. To validate the field, you could call the validate method on the **GlobalKey** when the submit button is pressed, as an example. If you want to automatically validate a field while the user is inputting data, you can supply an **AutovalidateMode** object to the field via its **autovalidateMode** parameter. Whichever option you choose, just remember that validation does not happen automatically by default.

Building a Simple Signup Form

Let us use some of what we have learned in the last section to build a simple signup form. We want to capture their name, e-mail address, and password. We will make sure to have them enter their password twice, so we can confirm that they got it right. Plus, we will want to make sure their password contains at least one special character and is at least eight characters long. We also want to do some basic validation on the email address to make sure it is properly formatted as an e-mail address. Finally, we will have a signup button that is disabled until everything has been filled in and validates the form before submitting.

The first thing we are going to need is a stateful widget for our signup form. This will allow us to validate the input and display any errors without losing the user's input. Let us call our new widget **SignUpForm**.

We are going to need a **Form** that will group together all our inputs so we can validate them all at once. To do that, we will also need our form key. The **Form** requires a **child**, so let us give it a **Column** where we will add our inputs later.

```
1. class SignUpForm extends StatefulWidget {  
2.     const SignUpForm({super.key});
```

```
3. 
4. @override
5. State<SignUpForm> createState() => _SignUpFormState();
6. }
7. 
8. class _SignUpFormState extends State<SignUpForm> {
9.   final GlobalKey<FormState> _formKey = GlobalKey<FormState>();
10.
11. @override
12. Widget build(BuildContext context) {
13.   return Form(
14.     key: _formKey,
15.     child: Column(
16.       children: [],
17.     ),
18.   );
19. }
20. }
```

With that out of the way, we are ready to start adding our text inputs. Each will need its own `TextEditingController`, and we can label them with a `hintText` by using the `decoration` parameter. Create your `TextEditingController` variables, just like you did with the form key, for each field and attach them to the proper field using the `controller` parameter. The following is an example key:

```
1. final TextEditingController _nameController =  
  TextEditingController();
```

And here is an example of how to attach it to the input, then set the hint:

```
1. TextFormField(  
2.   controller: _nameController,  
3.   decoration: const InputDecoration(  
4.     hintText: 'Name',  
5.   ),  
6. ),
```

We will also want a button at the bottom of the list that we can use to submit the form later. Feel free to use any kind of button you want, although this example will use the `ElevatedButton` with some padding to separate it from the preceding inputs.

```
1. Padding(  
2.   padding: const EdgeInsets.only(top: 20.0),  
3.   child: ElevatedButton(  
4.     onPressed: () {},  
5.     child: const Text('Sign Up'),  
6.   ),  
7. ),
```

We now have a basic form (*figure 5.23*), but it is far from complete:

The image shows a simple sign-up form with four text input fields and a central 'Sign Up' button. The fields are labeled 'Name', 'Email Address', 'Password', and 'Password (confirmation)'. The 'Sign Up' button is a blue rectangle with white text.

Figure 5.23: Our basic form layout

Since we know the name field is required, let us add a validator to it. As a reminder, the `validator` is a function that will run whenever we validate the form. It has a `String?` input and a `String?` return value. We return `null` when the field is valid and a `String` which will be displayed as the error message when validation fails. Here is a very simple validator that checks to see if the user has entered any text in the field, then checks to see if they have entered their full name:

```
1. validator: (value) {  
2.     if (value == null || value.isEmpty) {  
3.         return 'Required';  
4.     }  
5.     if (value.trim().split(' ').length < 2) {  
6.         return "Please enter your full, legal name.";  
7.     }  
8.     return null;  
9. }
```

11. },

Of course, this code will never run unless we try to validate the form, so we have no way of testing that it works. Let us fix that now. We want to validate the form whenever we press the button, so let us add some code into the `onPressed` argument that will call the form validation function.

```
1. onPressed: () {  
2.   if (_formKey.currentState!.validate()) {  
3.     // Process the form data.  
4.   }  
5. },
```

Now you can test the validation of the name field by leaving it blank and pressing the button to see an error or by entering text and watching the error disappear when you push the button. You can also check to make sure the other validator is working by typing just your first name and pressing the button.

Validating email addresses is a far more difficult task than simply checking whether the input is empty. We must check for an @ symbol in the middle, with some sort of domain name on the right side. Plus, there are certain characters which are invalid in an e-mail address, like a semicolon. In short, e-mail address validation can become very complex, very quickly.

A regular expression (or *Regex*, for short) is a pattern used to match character combinations in strings of text. Regular expressions are far outside the scope of this book, so you are encouraged to do some research on your own to understand them. At first glance, they can be terrifying and unapproachable, but with some practice, you will be able to break them down into smaller components that make far more sense. The website <https://regextester.com/> has an excellent toolset for writing, testing, and understanding regular expressions. For our purposes, any necessary regular expressions will be provided here, but certainly take time to do research on the subject to set yourself apart. You'll be glad you did.

Using the power of regular expressions, we can create a pattern that ought to match *most* valid e-mail addresses:

```
1. RegExp emailValidator = RegExp(  
2.   r'^(([<>()[]\.,,:s@"]+([<>()[]\.,,:s@"]+)*|(.+))@(([0-9]  
  {1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}])|(([a-zA-Z-0-9]+.)+[a-zA-  
  Z]{2,}))$',  
3.   caseSensitive: false,  
4.   multiLine: false,  
5. );
```

Then, we just need to attach it to our e-mail field's validator function, like so:

```
1. validator: (value) {  
2.   if (value == null || value.isEmpty) {  
3.     return 'Required';  
4.   }  
5.   if (!emailValidator.hasMatch(value)) {  
6.     return 'Invalid email address';  
7.   }  
8.   return null;  
9. },
```

When it comes to e-mail address validation, specifically, we generally would not want to validate by using a regular expression because of how monstrously complex proper e-mail address validation is, instead opting for a package such as `email_validator` to do the work for us. We will talk more about packages in *Chapter 6: Using 3rd Party Libraries and External Files*. For the purposes of illustration, however, the regular expression is sufficient.

Next, let us tackle the password fields. We know that both fields need to be the same, so we only need to validate the actual input on one of them and make sure the second field matches it. There are a lot of valid special characters that we want to allow our users to use, so rather than listing all of them, let us use our old friend Regex to solve the problem.

If we consider *normal* characters to be all 26 characters of the Latin alphabet, along with the numbers 0 through 9, we can then consider *special* characters to be *anything* that is not a letter or a number. Luckily, a regular expression that matches all letters and numbers is very easy to write. Then, all we need to do is check that the value of the field *is not* matched by that regular expression, at which point we will know that we have a special character:

```
1. if (!value.contains(RegExp('[^A-Za-z0-9]'))) {  
2.     return 'Special character required';  
3. }
```

We can add this check inside our validator alongside any other checks we want to perform, such as making sure the value is not empty. We also want to check our password to make sure it is at least eight characters long. See if you can figure out the test for that and check your work with the following code:

```
1. if (value.length < 8) {  
2.     return "Password is too short";  
3. }
```

It seems like we have got all the validation done for the password complexity requirements, so let us add a validator to the password confirmation field. Like with the other fields, we want to ensure this field has been filled out, so check for that first. Then, we can use the **TextEditingController** that we assigned to the password field to get its current value and compare it to the input on this field.

```
1. if (value != _passwordController.text) {
```

```
2.     return "Passwords do not match";  
3. }
```

To finish these fields off, we should specify a couple of additional parameters to obscure the text, disable autocorrect, disable suggestions, and prevent the keyboard from learning about our password. The `obscureText`, `autocorrect`, `enableSuggestions`, and `enableIMEPersonalizedLearning` parameters, respectively, will allow you to accomplish this.

If you were so inclined, you could also add an icon to the field to toggle the password visibility. That would look something like the following:

```
1. TextFormField(  
2.   decoration: InputDecoration(  
3.     suffixIcon: IconButton(  
4.       onPressed: () {  
5.         setState(() {  
6.           _hidePassword = !_hidePassword;  
7.         });  
8.       },  
9.       icon: _hidePassword  
10.      ? const Icon(Icons.visibility)  
11.      : const Icon(Icons.visibility_off),  
12.    ),  
13.    ),  
14.    obscureText: _hidePassword,  
15.    autocorrect: false,  
16.    enableIMEPersonalizedLearning: false,  
17.    enableSuggestions: false,
```

18.),

Just make sure you have created a `_hidePassword` bool that you can reassign, then tapping the eye icon will toggle visibility of the field and change the icon.

To complete our signup form, we should do something with all the values we have collected once the form has been validated. Typically, you will pass these values to some sort of API or backend for processing, but we are just going to print them out to the debug console.

Find the `if` statement in your button's `onPressed` argument, where we call the `validate` method on the form. Any values in our form that we try to `print` from inside this statement will be guaranteed to be considered *valid*. Try printing the values from each of the fields, using the fields' controllers to get the value:

```
1. if (_formKey.currentState!.validate()) {  
2.   // Process the form data.  
3.   debugPrint('Name: ${_nameController.text}');  
4.   debugPrint('Email Address: ${_emailController.text}');  
5.   debugPrint('Password: ${_passwordController.text}');  
6. }
```

Refer to the accompanying repository for the complete example of what we just built.

Conclusion

Gathering information from users, whether that information is intent (such as by tapping a button) or direct input (such as typing their name into a field), is fundamental to creating any application. Validating that input is just as important, and you now have the tools to do just that.

We looked at how `Cupertino` and `Material` have different ways of accomplishing similar tasks, and you should now be thinking about building applications which feel at home to a user, regardless of what type of device they are using. Later in the book, we will talk more about this in depth.

In the upcoming chapter, we will learn how to find and use packages in our application that are provided by the community. We will learn how to add assets such as images to our application, then learn how to access them throughout the application. We have got a lot to cover, so take a break: stretch, grab a cup of tea, and when you are ready, let's move on.

Questions

1. What is the difference between **Material** and **Cupertino** styled widgets, and when would you use one over the other?
2. What is a callback function?
3. What is the best way to add an icon to an **OutlinedButton**?
4. How do you style a button, including changing its color?
5. How do you disable a button?
6. What widget can you use to convert an arbitrary widget, such as a **Container**, into a button?
7. What widget can be used to detect a swipe gesture?
8. How do you change the type of keyboard presented to a user on a mobile device when they tap a text field? For example, how would you present the user with a keyboard which is more appropriate for numerical input?
9. What is a **TextEditingController** used for?
10. In what order are input formatters applied?
11. How can you prevent users from typing numbers in a text field and instead only allowing letters?
12. You have a **DateTime** object with a given date. How do you determine what the date will be precisely one year in the future from that date?
13. What is the key difference between the material date picker and the Cupertino date picker?
14. What does the **map** method of a **List** do?
15. What are some considerations when you want to use a group of **Radio** buttons?
16. How many states does a **Checkbox** have?

17. How can you limit the minimum and maximum values of a **Slider**?
18. What is a **Form**, and what's the benefit of using one? What are some cases in which a **Form** might not be necessary?
19. What's the difference between a **TextField** and a **TextFormField**?
20. How do you validate an input?
21. What is a regular expression?

Key Terms

- **Callback function:** A function that is passed into a widget as a parameter that can then be invoked from within the widget, but it will be run in the context of the widget's parent.
- **Regular expressions (Regex):** A pattern used to match character combinations in strings of text.

Further Reading

- Learn more about Google's material design principles:
<https://material.io/>
- Read up on Apple's Human Interface Guidelines:
<https://developer.apple.com/design/human-interface-guidelines/>
- Learn regular expressions with simple, interactive exercises:
<https://regexone.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>



Chapter 6

Using 3rd Party Libraries and External Assets

Introduction

Writing software is hard. That is why, as software developers, we strive to solve a problem only once. For instance, imagine you develop a practical widget that proves valuable across your entire application. In this scenario, you have successfully addressed a problem once and can now leverage the code repeatedly within your application. Would it not be great if you could benefit from the collected knowledge of Flutter developers everywhere, using *their* widgets to solve the very problem you are currently facing?

In this chapter, we will look at the package ecosystem for Flutter and Dart, explore some popular packages, and learn how to include things such as images, sounds, fonts, and other resources in your application.

Structure

In this chapter, we will discuss the following topics:

- Introduction to pubspec.yaml
- Adding images and other assets to your application
- Introduction to the Pub package management system
- Finding useful packages on Pub

- Using a package to play audio
- Popular Pub packages

Objectives

Upon completion of this chapter, you will have learned how to include custom resources, such as images, fonts, and sounds in your application. You will also have a familiarity with the package management ecosystem for Flutter and Dart and learn how to evaluate the quality and usefulness of packages. You will also learn how to use a package to play an audio file that you have included in your application as a custom resource. Finally, you will learn about popular packages that are often used by the community.

Introduction to pubspec.yaml

Every Flutter application comes with a special file called `pubspec.yaml`. The file exists at the top-level directory of the project, alongside the `lib` directory. We often refer to this file as simply “pubspec,” which is how we will be referring to it from here on out. Pubspec contains a lot of metadata that Flutter and Dart need to know about your application and is written in the YAML language. What is important to know about YAML is that whitespace (spaces versus tabs) matters. You can learn more about YAML at <https://yaml.org/>.

Pubspec is used not only by Flutter but by Dart as well, although many of the sections (called *directives*) in the file are specific to Flutter. The directives that both Flutter and Dart use are `name`, `description`, `publish_to`, `version`, `environment` (and `sdk`), `dependencies`, and `dev_dependencies`. Directives that are specific to Flutter are generally either defined within a `flutter` directive or are named `flutter`.

Ok, so, with all these directives, what do each of them do, and why would you want to change anything there? Let us run down the list and break it down.

The first two directives in pubspec are `name` and `description`. The `name` directive is the name of your project, but with some specific requirements. This is not the label that will show up under the application icon, but rather the Dart *package* name. The requirements for the name are that it should be all lowercase with underscores separating the words. It cannot be one of the

reserved keywords (as they are described here: <https://dart.dev/guides/language/language-tour#keywords=>), should not include any non-Latin characters, any accented characters, and *may* include numbers (although it cannot *start* with a number).

The **description** is optional and should be relatively short. (Think the length of a Tweet.) This is mostly useful if you are building a package to share with others, rather than an application. If you are only building an application, you do not technically need to put anything here.

The **publish_to** directive should be set to '`none`' if you are working on an application, so that your application isn't accidentally published to Pub (which we will talk about later in this chapter).

The **version** directive is where you can increment the version number of your application each time you are ready to release a new version. Flutter is configured, by default, to use this value in the application packages which are created when building the application.

Next, under the **environment** directive, there is an **sdk** variable. This is used to define the minimum and maximum versions of the Dart programming language which the application can be compiled and run with. From time to time, Dart introduces new features (such as when *constructor tearoffs* were introduced in Dart 2.15.) If your application was created before Dart 2.15 was introduced and you wanted to leverage the new constructor tearoff feature in Dart, you would not be able to do so until you updated the minimum version in pubspec. Likewise, when the next major version of Dart is released, there are bound to be some massive, breaking changes that would require a substantial rewrite of your application. This directive allows you to specify a maximum version, so if Dart is upgraded to a new major version, it does not simply break everything.

The **dependencies** and **dev_dependencies** directives are where you will add any external packages that your project will use. We will discuss this throughout the rest of the chapter.

Finally, under the **flutter** directive, you can add any assets or external files to be included in your application. Let us explore this next with a real-world example: you are building an application based on designs presented to you by a product designer.

Adding Images and Other Assets to Your Application

The product designer has given you a design that includes some beautiful illustrations to spice up the application you will be building. Now, you need to find a way to include those illustrations in your application. So, what do you do?

It is customary to create an `assets` folder at the top level of the application where your external assets such as fonts, images, sounds, and anything else you may need to include in your application will live. Many developers choose to further organize their assets folder into several subdirectories, as well.

You take the illustrations your designer has provided to you and save them into a well-organized `assets` directory. Simply adding the files, however, does not allow Flutter to access them. We must go back to pubspec and add the directories there, too. Under the `flutter:` directive, we will add an `assets:` directive, followed by any subdirectories. Here is what that might look like, although it will differ based upon what directories you've created:

1. `flutter:`
2. `assets:`
3. `- images/`
4. *# If there are any subdirectories you'd like to include,*
5. *# be sure to add those to the list as well.*
6. `- images/another_subdirectory/`
7. `- fonts/`
8. `- sounds/`

Note that it is not necessary to specify each file explicitly. By specifying a directory where multiple files exist, Flutter is smart enough to know to include everything within the directory. Finally, we are ready to add the image to our application! The `Image` widget has an easy way to display images that you have included in your application as assets. It can be

accessed using the `Image.asset()` constructor, then passing in the path to the image as a positional parameter:

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

That is all there is to it! Now you can add all those beautiful illustrations that the designer gave you. A job well done!

Adding fonts can be done in a similar way, although there are a couple of extra options available to us. First, we specify the name of the font we will be using. This name is how we will refer to the font within the application and does not necessarily have to be the name of the font itself. Next, we specify each font file as an asset for the font family. We do this because some fonts are split into multiple files, where each file may define the font with a different font *weight*, or how thick the letters are. A complete example with multiple font files of different font weights and styles might look something like the following:

```
1. flutter:  
2.   fonts:  
3.     - family: Comic Sans  
4.       fonts:  
5.         - asset: assets/ComicSans-Regular.ttf  
6.         - asset: assets/ComicSans-Bold.ttf  
7.           weight: 700  
8.           - asset: assets/ComicSans-Italic.ttf  
9.             style: italic
```

However, if your font only has one file, it is possible to specify a single asset with no associated font weight or style:

```
1. flutter:  
2.   fonts:
```

3. - family: Comic Sans
4. fonts:
5. - asset: assets/ComicSans.ttf

Now that you have got your font set up in pubspec, you can reference it by the name you have given it in your application:

1. Text(
2. 'Comic Sans is arguably the greatest font face of all time.',
3. style: TextStyle(
4. fontFamily: 'Comic Sans'
5.),
6.),

When it comes to playing audio, things become a bit trickier. Since Flutter has no native way of playing audio, we will have to employ the use of a package. Let us first explore Flutter's package management system, and then we can look at a couple of useful packages, including one which will allow us to build a fully featured audio-playing application.

Introduction to the Pub Package Management System

From the very start, Flutter was designed to allow the ability to import other people's code and use it in your own project. This ability to reuse code written by others opens the door to building apps and experiences that would not otherwise be possible for a sole developer to accomplish on their own. There are thousands of packages — all open-source — that have been created by and shared with the Flutter community, for free. Developers who create a package and share it with the community submit their package to a website called Pub (<https://pub.dev/>), where anyone can go to discover and see the documentation for all manner of packages shared by and to the community.

Packages are added to your Flutter project by modifying pubspec in the `dependencies:` directive. The way you add a package is by specifying first the package name, followed by a colon, then the version number. A complete example would look like the following:

1. `dependencies:`
2. `flutter:`
3. `sdk: flutter`
4. `google_fonts: ^5.0.0`
5. `path: any`
6. `provider: 6.0.5`

Note that `flutter` is always included as a dependency for our Flutter projects. In this example, we have added the `google_fonts` package to our application, allowing us to use any of the fonts listed at <https://fonts.google.com/> in our application. Also, note the caret (^) in front of the version number. We use this to specify a version range, rather than a specific version. We also added the `path` and `provider` packages, with `path` being set to the version `any` and `provider` being set to the version `6.0.5`.

Much like an application, a package can have its own dependencies. These dependencies will also have version numbers specified in their pubspec. When adding a package with its own dependencies as a dependency to your application, the package's dependencies are also added to your application as transitive dependencies. If a package requires a particular version of a dependency, for example, version 2.4.1, and you have the same dependency in your application, but you have set the version to 2.4.0, there would be a version mismatch. Your application would require an older version, whereas the package would require a newer version.

This is where version ranges come into play. By specifying a caret before the version, Flutter knows that it's allowed to use a range of acceptable versions. In our example, specifying `^5.0.0` would allow any version from 5.0.0 to 6.0.0, satisfying the needs of both the application's dependency as well as the package's dependency. It is for this reason that it is

recommended to use version ranges when adding dependencies to your application.

However, it is also possible to use a specific version, such as when we specified a version for `provider`. In our example, only version 6.0.5 is allowed to be resolved, requiring other packages to conform to that decision. This is the most restrictive way to specify a version number, whereas the version of `path` we chose, `any`, is the least restrictive. If you specify `any` for the version, the latest possible version that works with the rest of your dependencies will be used. This can cause breaking changes if the package suddenly updates to a new, major version, so be careful when using this.

Once the dependency has been added to your pubspec, it will need to be downloaded and cached before you can start using it. Be sure to save the file, then use the command `flutter pub get` to download the package. Now that the package has been downloaded, it is ready for use in your project. The last step to using the package in your project is to import it into the file where you want to use it. Let us import the `google_fonts` package that we added as a dependency earlier. At the top of the file, you will include it like the following:

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

From there, you will be able to use the widget(s) provided by the package as though they were code you have written in the file, yourself. You are already used to doing this: we have been using the `Material` and `Cupertino` packages throughout this book!

Packages are updated all the time by their developers to meet their own needs and the needs of the community. It is a good idea to keep your packages up to date with the newest versions, but doing that by hand can be a long, frustrating process. Thankfully, we have another command for that.

To upgrade the packages in your project to the latest versions within the version ranges you specified in your pubspec, you can run the command `flutter pub upgrade`. If you would like to include newer versions of packages that are not within your specified version ranges as well, you can instead use the command `flutter pub upgrade --major-versions`.

Finally, there may come a time when you find yourself needing to troubleshoot why your application is not building and you suspect it might have something to do with the packages you have imported. You can use `flutter pub cache clean` to remove all packages which have been downloaded (this is generally followed by `flutter pub get` to re-download the packages necessary for your project), or by running `flutter pub cache repair` which will re-download all the packages for your project without cleaning the cache first. Keep in mind that running these commands will affect the packages for all projects, not just the one you are currently working on.

Finding Useful Packages on Pub

Pub has a huge selection of packages to choose from. Some packages are written for Dart (which can then be used in any Dart-based application, including Flutter applications), whereas others explicitly require the Flutter framework. Furthermore, packages are sometimes designed to work with specific platforms and may or may not have support for others. There may be a package that works on Android and iOS but isn't compatible with the Web or desktop operating systems, for example. With so many packages available, though, how do you decide which are worth using and which are better to pass on?

Choosing the right package starts with knowing what problem you are trying to solve. In fact, there is very little (if anything) that you cannot accomplish on your own by using a package from Pub. However, the aim is to save development time and rely on the experience of others to provide value. Not all packages on Pub provide widgets that will be drawn to the screen. In fact, many packages on Pub offer features that are more abstract, such as widgets that help with state management, interface with databases and other services, or make sharing content to other applications easier. Also, many, many other things that would just be time-consuming to write from scratch every time you need to do something that the Flutter framework does not immediately provide to you. It is important to pick a package that is suitable for the problem you are trying to solve, so ensure that you take the time to consider this carefully.

The next consideration for choosing a package is determining the platform(s) you will be targeting with your application. As previously mentioned, many packages are only available for *some* of the available

platforms that Flutter is capable of building for. Furthermore, some packages implement features that might not make sense for all platforms. If you are building an application that targets the Web, it does not make sense to use a package that implements the ability to report app crashes to Google's Firebase Crashlytics. (In fact, the package available on Pub to do this, `firebase_crashlytics`, doesn't have support for web, at all, nor does it have support for Windows, at least as of the time of writing.)

Finally, and arguably the most important consideration, is the *quality* of the package. There are three broad metrics by which you can judge the quality of a package: likes, popularity, and Pub Points. The first of these, likes, is a simple enumeration of how many developers have logged into Pub and pressed the thumbs-up button next to a package. Were you to do the same, you could find your list of liked packages under the `My pub.dev` menu at the top of the page, then by navigating to *likes*. Next is the popularity metric. Currently, this is a measure of the number of apps that depend on the given packages over the past 60 days, measured by download count (and adjusted for automatic downloads, such as with build tools), and displayed as a percent from 0% (the least used package) to 100% (the most used package.) This metric's calculation is bound to change once the team has completed work on measuring absolute usage counts, however that is not currently the case as of the time of writing.

The final metric by which a package's quality can be evaluated is by looking at its Pub Points. Alongside the popularity metric, this is arguably the most useful metric. Unlike the other metrics, this value is automatically calculated and cannot be directly influenced by the number of people liking or using the package. There are many checks which are done automatically on a package to determine its Pub Points by a tool called `pana`. Pana automatically evaluates the quality of the package by looking at a multitude of criteria within several categories, then scoring each category independently before producing a total number (which, at the time of writing, is out of 140 possible points). The criteria for how these points are awarded, at least at the time of writing, are as follows:

- Follow Dart file conventions (30 points total)
 - Provide a valid `pubspec.yaml` (10 points)
 - Provide a valid `README.md` (5 points)

- Provide a valid `CHANGELOG.md` (5 points)
- Use an OSI-approved license (10 points)
- Provide documentation (20 points)
 - Provide an example usage of the package (10 points)
 - 20% or more of the public API has `dartdoc` comments (10 points)
- Provide support for all 6 platforms (20 points)
- Pass static analysis (code has no errors, warnings, lints, or formatting issues) (30 points)
- Support up-to-date dependencies (20 points)
- Dart 3 and Flutter 3.10 compatibility (20 points)

Given the rigorous criteria by which Pub Points are calculated, it should come as no surprise that Pub Points are one of the easiest ways to determine the overall quality of any given package.

Finally, it is always an excellent idea to read the package description thoroughly. There have certainly been times when a package was developed to overcome a shortcoming in Dart or Flutter, then later that shortcoming was remedied, making the package obsolete. An excellent example of this is the `supercharged` package.

Using a Package to Play Audio

Now that we have a firm grasp of what a package is, where to find them, and how to evaluate them for quality before depending on them in our application, let us loop back to the earlier example of finding and using a package to play audio.

As we discussed earlier, Flutter does not (natively) support playing audio, so using a package is the easiest way to add this functionality. Any package we use will have platform-specific code for each supported platform which does the heavy lifting of playing audio. There will also be a widget (or perhaps several) which will enable us to choose what audio to play and control the playback.

By searching for `audio` on <https://pub.dev/>, we are presented with several options. One thing to note is that these results are certainly outdated from the time of writing to the time you are reading this. Many of these packages

may have been updated in the intervening time. Your results are almost certain to be different than those listed here, but generally speaking you will probably notice the packages `audio`, `dart:web_audio`, `dart:html`, `material`, `flutter_sound`, `flutter_widget_from_html`, and `just_audio`, which is what we will be focusing on for our example. Let us start by looking at the `audio` package (*figure 6.1*):

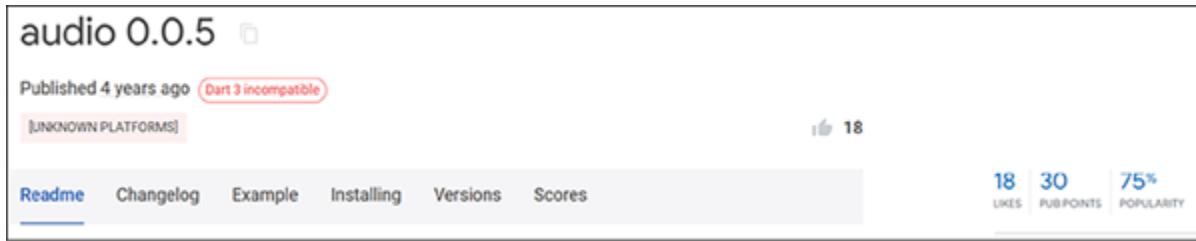


Figure 6.1: The pub.dev search result listing for the audio package

This package has very few likes, a very low number of Pub Points, and relatively weak popularity. Also note the last update was version 0.0.5, which was released four years prior. In addition, Pub cannot determine which platforms this package will run on. Finally, there's no Dart 3 compatibility. Given all these elements together, it's a safe bet that this package is not a good choice for our application.

Let us look at the next set of results; refer to *figure 6.2*:

dart:web_audio

High-fidelity audio programming in the browser.

v 3.0.2 • Dart SDK library [Core library](#) [Null safety](#)

API results: ► [dart-web_audio/dart-web_audio-library.html](#)

dart:html

HTML elements and other resources for web-based applications that need to interact with the browser and the DOM (Document Object Model).

v 3.0.2 • Dart SDK library [Core library](#) [Null safety](#)

API results: ► [dart-html/dart-html-library.html](#)

material

Flutter widgets implementing Material Design.

Flutter SDK library [Core library](#) [Null safety](#)

API results: ► [material/Material/Material.html](#)

flutter_sound

Europe Stand With Ukraine. Pray for Ukraine. A complete api for audio playback and recording. Audio player, audio recorder.

v 9.2.13 (12 months ago) [tau.canardoux.xyz](#) [MPL-2.0](#) [Dart 3 compatible](#)

[SDK](#) [FLUTTER](#) [PLATFORM](#) [ANDROID](#) [IOS](#) [WEB](#)

1155	130	99%
LIKES	PUB POINTS	POPULARITY

flutter_widget_from_html

Flutter package to render html as widgets that supports hyperlink, image, audio, video, iframe and many other tags.

v 0.10.1 (22 days ago) / 0.13.0-alpha.1 (6 days ago) [daohoangson.com](#) [MIT](#) [Dart 3 compatible](#)

[SDK](#) [FLUTTER](#) [PLATFORM](#) [WEB](#)

API result: [flutter_widget_from_html/flutter_widget_from_html-library.html](#)

Figure 6.2: More search results for the query “audio” on pub.dev

From these results, we can see that the top three are *core libraries*, meaning that they are built-in to the Flutter framework or Dart itself. Since we know that Flutter (and Dart) are incapable of playing audio, we can safely ignore them.

Next up is `flutter_sound`. This package has a huge number of likes, a maximum number of Pub Points, and an excellent popularity rating. Additionally, it supports the Web and mobile platforms and works with Dart 3. This would be an excellent choice, so let us add it to our list of potential solutions and keep looking.

The next result, `flutter_widget_from_html`, sounds like it will render HTML code as Flutter widgets, which does not match what we are looking for. Despite the rest of the stats looking good, it will not solve our problem.

There are several more search results on the page, and several pages of search results in total. However, the very next search result looks promising (*figure 6.3*):

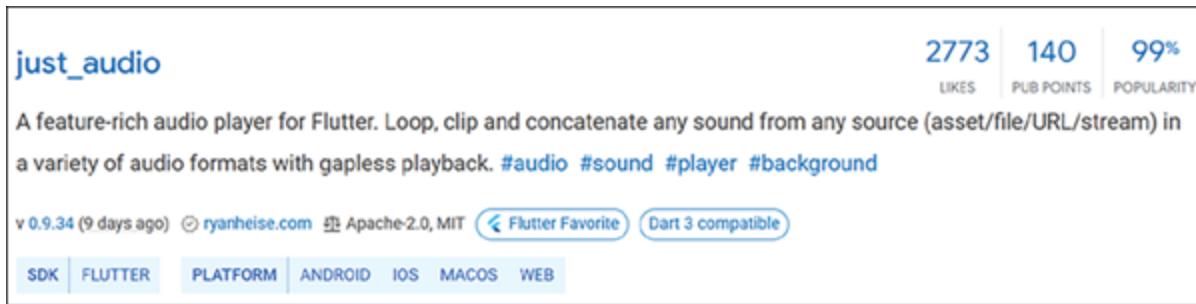


Figure 6.3: The pub.dev search result listing for the `just_audio` package

The `just_audio` package, like the `flutter_sound` package, has the maximum number of Pub Points, was updated within the past couple of days, supports Dart 3, and has an incredibly popular rating. Where it sets itself apart is the enormous number of likes and the *Flutter Favorite* badge. Good thing we did not stop looking when we found `flutter_sound`, because `just_audio` looks like it is an even better choice!

Now that we have found the package we want to use, let us explore it a bit further. To start, let us look at the package page (as shown in *figure 6.4*). You can find this on your own by navigating to https://pub.dev/packages/just_audio.

Here, we can find a slew of useful information about the package. At the top, we see the package name and current version. (There is even a handy copy icon next to the version, which will copy the line we need to paste into our pubspec!). It tells us the last time the version was updated, the developer, and all the useful bits about Dart 3 support, being a Flutter Favorite, the supported platforms, and so on:

pub.dev

just_audio 0.9.34

Published 9 days ago • @ [ryanheise.com](#) Dart 3 compatible

SDK FLUTTER PLATFORM ANDROID IOS MACOS WEB 2.7K

Readme Changelog Example Installing Versions Scores 2773 LIKES 140 PUB POINTS 99% POPULARITY

just_audio

just_audio is a feature-rich audio player for Android, iOS, macOS, web, Linux and Windows.

[Platform Support](#) — [API Documentation](#) — [Tutorials](#) — [Background Audio](#) — [Community Support](#)

just_audio is a feature-rich audio player for Android, iOS, macOS, web, Linux and Windows.

[Platform Support](#) — [API Documentation](#) — [Tutorials](#) — [Background Audio](#) — [Community Support](#)

URLs
Assets
Files
Byte streams
Icy Metadata
Volume
Composite state
Loop modes
Request headers
Concatenating
Gapless transitions

Caching (EXPERIMENTAL)
Visualizer (EXPERIMENTAL)
Background
DASH/HLS
Radio/Livestreams
Time stretching
Buffer position
Shuffling
Clipping
Audio effects
Playlist editing

Topics
#audio #sound #player #background

Documentation
API reference

License
Apache-2.0, MIT ([LICENSE](#))

Dependencies
async, audio_session, crypto, flutter, just_audio_platform_interface, just_audio_web, meta, path, path_provider, rxdart, uuid

More
[Packages that depend on just_audio](#)

Figure 6.4: The package details page for the just_audio package on pub.dev

The tabbed view below defaults to the package's readme, which has even more useful information. If you scroll down, you will even see some examples detailing exactly how to use it in the application. The changelog tab will detail all the changes made in each version that has been released,

the example tab has a complete working example that you could copy/paste into your application, and the installing tab tells you how to add it as a dependency in your code. The versions tab is a useful way to download a copy of any given version (or see the documentation for said version). Finally, the scores tab breaks down the Pub Points calculation to see exactly what points were awarded for.

Other useful information on this page can be found on the right sidebar, where you can see links to the homepage, GitHub repository (in case you would like to browse the source code), and a place to see/report any issues with the package. There is also a complete API reference, which covers the entirety of the package and how to use every aspect of it (although, in the case of `just_audio`, the relevant information is on the readme tab, so the API reference simply displays this information.)

Normally, this is where we would go step-by-step over how to add this package to your application and start using it. This time, however, why not try following the instructions on the *installing* tab, then use the example to get it up and running on your own? After all, this is often the process you will need to follow when you are out there building apps. (Plus, you have got this!) Do not forget to run `flutter pub get` once you have added the dependency!

Popular Pub packages

There is no shortage of packages on Pub. Sometimes the most difficult part of choosing a package is even knowing what is even available and what other people are using to solve problems. If you have already visited `pub.dev`, you will no doubt have noticed the *most popular packages* and *top Flutter packages* sections on the home screen (*figure 6.5*). (There is also a *top Dart packages* section if you are looking for something specific to Dart.)

This is an excellent place to start browsing packages that other developers are commonly using. Sometimes inspiration for a new application or feature can come from just browsing these lists. You might even discover a solution to a problem you did not know you were having!

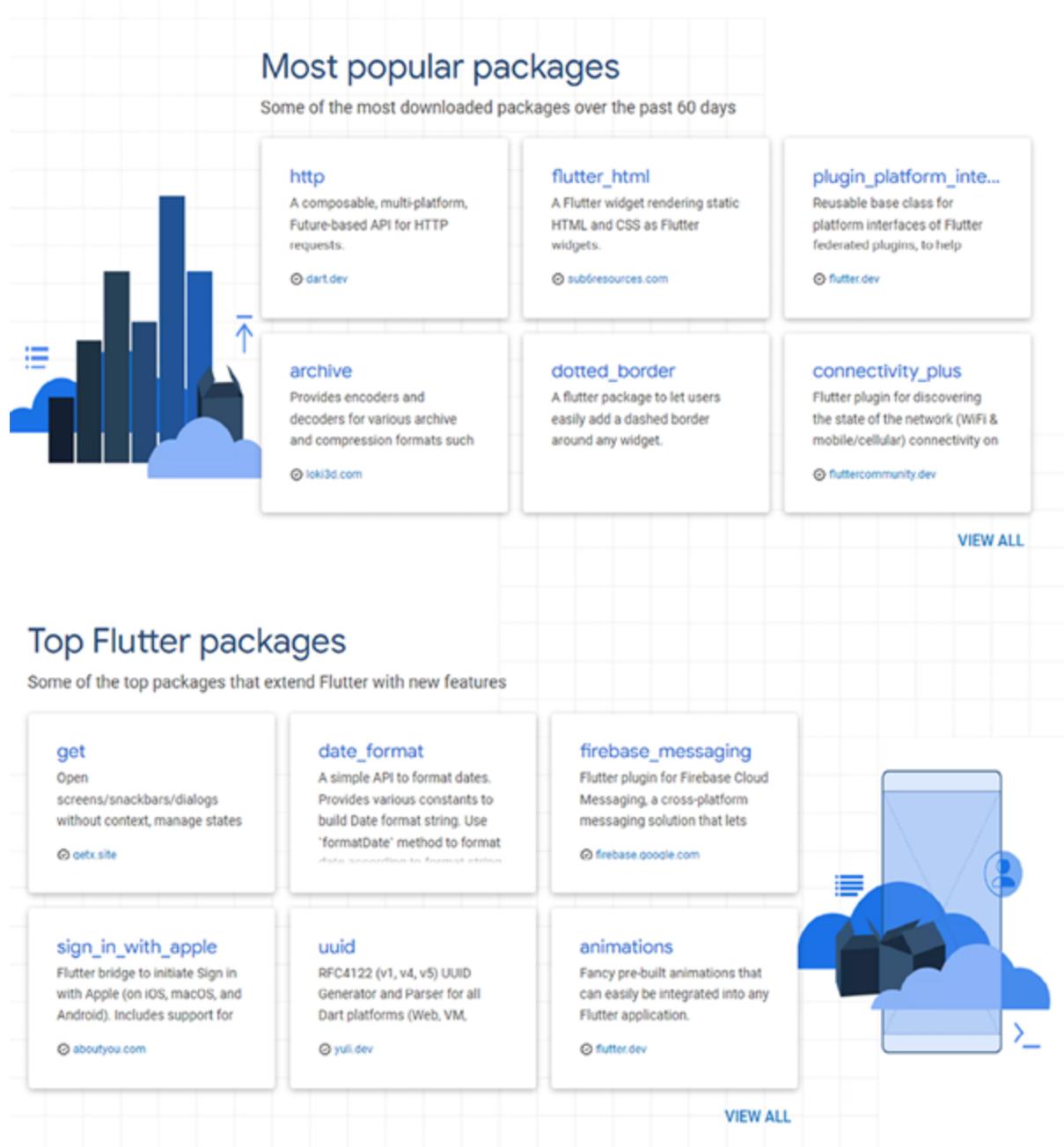


Figure 6.5: Popular packages on pub.dev

Let us look at a couple of the most popular packages to understand what problems other developers are trying to solve.

shared_preferences: Have you ever thought it might be useful to store some user data on the device? Maybe you are storing the user's preference for a light/dark mode. Perhaps you want to keep track of how many times they have opened the application. Maybe you want to store user-generated

content (such as a list of items) so that it is available next time the app is launched. If you have answered *yes* to any of these questions, look no further than `shared_preferences`!

With `shared_preferences`, you can store integers, Booleans, doubles, strings, and lists of strings to a user's device with ease. Then, you can easily recall those values later. It works on all platforms that Flutter can target but beware that the data is stored locally on the device — if you build an application that uses `shared_preferences`, do not expect the same data to be available on multiple devices that the user uses.

`url_launcher`: If you want to have a tappable email address which automatically launches the user's email client and pre-fills the to, subject, and body lines, look no further than `url_launcher`! Why stop at emails, though? You could launch a website, automatically enter a phone number into the user's default phone application so all they need to do is press call, initiate an SMS conversation, or even open a file. All of this is possible on every platform that Flutter targets.

`dio`: The default HTTP client that comes with Dart is usually good enough for most tasks. However, there are times when you need to do something that the built-in client just cannot handle. This is where `dio` comes to the rescue! Use `dio` to configure global settings for all HTTP connections made within your application, such as setting an authorization header. Want access to better debug logging to see what requests and responses are being made more easily? Try using an interceptor! (Write your own *or* find a package that does it for you!) If for whatever reason you find that Dart's HTTP client is not quite doing the trick, look at Dio. You will be glad you did.

`flutter_bloc`: Stateful widgets can only get you so far in life. The minute a stateful widget is disposed of the state is lost. That might be fine for some use cases, but other times you might need to save that state. You could be building a list/details application. When viewing the details, you have got the option to update the title. Save the changes by passing the updated details to a BLoC, and the list will rebuild automatically.

A lot of business logic can be moved outside the widget's build method by using a BLoC, as well. By leveraging the event-driven nature of the BLoC, you can send an event to the BLoC (sometimes with parameters), allow the

BLoC to do the heavy lifting, and then the BLoC will return a given state. That state can be used to build your widget in a different way. For example, sending an event of *load all the things* to the bloc would trigger the BLoC to reach out to an API and acquire some data. Then, the data would be processed. If successful, a state of *all good!* could be sent back; otherwise a state of *that did not quite work out* could be sent instead. When the widget is built, it could check for the success/failure states and either return the details of the response or an error message. And we are only scratching the surface of what `flutter_bloc` can do! In fact, we are going to discuss it in-depth in *Chapter 9: State Management and the BLoC*.

`isar`: If you have found yourself using `shared_preferences` and realizing the limitations of what you are able to store and how you can retrieve it, you need `isar`. This package is an easy-to-use NoSQL database for storing and retrieving any type of data imaginable. It is easy to use, fast, and has excellent documentation. The limitations of `shared_preferences` are a thing of the past with `isar`!

This is just a small sampling of some of the more popular packages which can be found on Pub. The Flutter team has a *Package of the Week* playlist on YouTube (see the *Further reading* section at the end of the chapter) in which they choose a package from Pub each week and give a brief overview of the problem that the package seeks to solve, then show an example of it in action. They are short videos, and you are highly encouraged to check out the playlist!

Conclusion

We have covered a lot of ground in this chapter. By now, you know what `pubspec.yaml` is and how to use it to add packages and resources to your application. You learned how to find and evaluate the quality of packages using Pub, the package manager for Flutter and Dart, and `pub.dev`, the website where all the available packages can be found. Then, we learned how to make our application play sounds. Finally, we explored a small selection of some popular packages that many developers come back to time and time again.

How are you feeling about all this new information? If you are overwhelmed, that is perfectly okay. Take a break, stretch, drink some

water, and come back to it when you are ready. Or are you feeling confident that you are really starting to get a good grasp on Flutter? That is great! We have almost all the tools we need to start building our first application. There are just a couple more concepts we need to learn before we can start digging into building something from scratch. Next up, we are going to explore the concept of asynchronous operations and use them to communicate with an API. So, whenever you are ready, let's `async await` some new concepts!

Questions

1. What is `pubspec.yaml` used for?
2. What language is `pubspec.yaml` written in?
3. What are the required directives for Flutter in `pubspec.yaml`, and what do they do?
4. How do you update the package versions in `pubspec.yaml` with a single command?
5. How do you add images, fonts, sounds, or other assets to your application?
6. What is Pub?
7. How does dependency version pinning work?
8. Where can one browse a list of packages that are available to add to a Flutter or Dart project?
9. What are Pub Points and how are they calculated?
10. How can one save a list of their favorite packages on Pub?
11. How can you play audio in your Flutter application?
12. Where can you find a list of popular Flutter and Dart packages?
13. What are some ways in which data can be persisted across app launches?
14. What YouTube playlist does the Flutter team use to showcase popular packages?

Key Terms

- **Asset**: Any file which is included with a project, such as an image or font.
- **Pana**: The tool used to automatically calculate the number of Pub Points a given package has earned.
- **Pub**: The package ecosystem for Flutter and Dart.
- **pubspec.yaml**: The configuration file for a Flutter or Dart project which includes useful items such as the package name and description, the version of Dart used by the project, and any dependencies the project may have.
- **Pub points**: A number generated by `pana`, which is used to loosely determine the quality of a given package.

Command Reference

- **flutter pub cache clean**: Removes all cached packages, globally.
- **flutter pub cache repair**: Re-downloads all cached packages, globally.
- **flutter pub get**: Downloads all of the packages specified in the `pubspec.yaml` file and caches them locally for use in your application.
- **flutter pub outdated (--major-versions)**: Lists out-of-date dependencies.
- **flutter pub upgrade**: Updates `pubspec.yaml` with the newest available versions of packages within their allowed version ranges.
- **flutter pub upgrade --major-versions**: Updates `pubspec.yaml` with the newest available versions of packages, even if those new versions have a version available outside of the allowed version ranges.

Further Reading

- Check out the Flutter team's *Flutter Package of the Week* playlist on YouTube: https://www.youtube.com/playlist?list=PLjxrf2q8roU1quF6ny8oFHJ2gBdrYN_AK
- Learn more about package versioning: <https://dart.dev/tools/pub/versioning>
- Learn more about how Pub Points are calculated: <https://pub.dev/help/scoring#pub-points>

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

Working with APIs and Asynchronous Operations

Introduction

These days, most applications communicate with services on the internet in some way or another. This generally happens by communicating with the service's API, or Application Programming Interface, via HTTP(s) communications. Since we have no control over how long that communication may take, we need to build our applications in such a way that the application is not blocked from further interaction and processing while we wait for a response. We do this by a method called asynchronous programming. In this chapter, we will delve into asynchronous programming and learn how to communicate with APIs.

Structure

In this chapter, we will discuss the following topics:

- Welcome to the Future
- Requesting data from an API
- Classifying API responses
- Sending data to an API
- Flutter's asynchronous widgets

- Sinks and streams

Objectives

After completing this chapter, you will have a fundamental understanding of concurrency in programming. You will know the difference between synchronous and asynchronous operations and how each style of code can impact the performance of your application. Furthermore, you will learn what APIs are and how to interact with them. Additionally, you will learn about data streams and how to use them in your code. Finally, you will use this knowledge to build a simple application that streams data from an API to build a list in Flutter, and then you will learn how to create new items on that list by sending data to the API and listening for a response.

Welcome to the Future

To understand asynchronous operations, the first concept we need to learn about is **Futures**. As the name suggests, a **Future** is something that occurs in the future. It is a *promise* that something will eventually happen.

Let us start by writing some synchronous code to see what that looks like and how it functions:

```
1. import 'dart:convert';
2. import 'dart:io';
3.
4. void main() {
5.   final String fileContents =
6.     _readFileSynchronously('my_file.json');
7.
8.   final Map<String, dynamic> jsonData =
9.     jsonDecode(fileContents);
10.
```

```
11. String _readFileSyncously(String filename) {  
12.     final File file = File(filename);  
13.     final String contents = file.readAsStringSync();  
14.     return contents.trim();  
15. }
```

This code will read the contents of a JSON file, decode it, and print out the first value. The `${jsonData.values.first}` is using what we call **String interpolation**. String interpolation will always return a string, even if the variable is null, in which case the string it returns is simply *null*. That is because the variable has the `.toString()` method called on it, which will usually return the string *null* when the value is, indeed, null. This can be used to get the contents of a variable, like we are doing here, but any code that's in between the braces will also be run. It is not entirely uncommon to run one or more methods within string interpolation, but do keep in mind that, if abused, this could lead to your code becoming less readable. Anyway, let us get back to analyzing this code.

The problem with this block of code is that the application stops processing *everything* until the file has been read in, in its entirety. If you have got a very large file, this would mean the application would simply stop responding until the entire file has been read into memory. Clearly, this is not ideal. So, let us take the same code and update it to read the file asynchronously:

```
1. import 'dart:convert';  
2. import 'dart:io';  
3.  
4. void main() async {  
5.     final String fileContents = await  
        _readFileAsynchronously('my_file.json');  
6.     final Map<String, dynamic> jsonData =  
        jsonDecode(fileContents);  
7.
```

```
8.     print('The first value is: ${jsonData.values.first}');
9. }
10.
11. Future<String> _readFileAsynchronously(String filename) async {
12.     final File file = File(filename);
13.     final String contents = await file.readAsString();
14.     return contents.trim();
15. }
```

Note the changes throughout the file. The `async` and `await` keywords have been added, the method called on the file has been updated to use the asynchronous version, and the return type of the method is now a `Future`. When you mark a method as `async`, it will immediately return a `Future` when invoked while using the `await` keyword. This suspends execution of the code within the function until the `Future` that is returned by `readAsString()` completes. Once `readAsString()` has finished its work, the function will resume processing until it encounters a `return` statement, which completes the function by returning a `Future<String>`, where the string is the value returned from the `readAsString()` method. This allows the application to continue processing even as the file is being read into memory.

Let us look at another example. Consider the following code:

```
1. String? hello = await Future<String?>.delayed(const Duration(seconds: 1)).then(() => "Hello");
```

In this example, we have foregone calling a function that is explicitly marked as `async`, but we are still waiting for a return value. Can you see how the variable `hello` will only be set to the string "`Hello`" after the `Future.delayed()` completes? We are calling a `Future` directly and using the `.then()` method to get the value. This code will be synchronous, meaning that the rest of the application will pause execution until it completes the `Future`. This contrasts with the way we called a function marked with the `async` keyword before. The `async/await` method *simulates* the `.then()` way of completing the `Future` but is not blocking.

Also in this example, the `Future` is obvious as a `Future` and it will only take a single second for that `Future` to complete. Not all futures will always appear to be of the type `Future` (although we now know better, since we realize that *all* future functions return a `Future` and can, once the future has been resolved, be cast into the eventual final type we are expecting), nor will most futures complete in such a specific amount of time. Think back to the first example, where we read in the contents of the file. The `file.readAsString()` method returned a `Future`, but it certainly did not explicitly say so, nor did we have any control over how long the process would take to complete.

Now that we have a basic understanding of asynchronous operations, futures, and the `async/await` keywords, let us look at the `http` package and learn how to use it to make an API call.

Requesting Data From an API

An API works a lot like ordering food from a restaurant. You give your order to a waiter, who disappears for a little while before bringing the food you ordered back to your table. You have no idea how long it will take to get that food, who cooked it or how it was prepared, or where any of the ingredients came from. APIs are, in this regard, very similar. You approach the API with a request, then wait for a response, and you have no idea what happened on the server between those two actions.

To communicate with the API, we will first need an HTTP client which will do the communication on our behalf. The `http` package will suffice, but there are other, more powerful HTTP clients (such as Dio) out there if you find yourself running into the limitations of `http`. The `http` package comes with a multitude of classes, and the two we are going to focus on right now are `Client` and `Response`.

`Client` is the HTTP client itself, used to communicate with APIs or make other network calls, such as downloading a file. There are a couple of different ways that people invoke a `Client` object. Some will import the `http` package using a namespace like `http`, which looks like the following:

```
1. import 'package:http/http.dart' as http;
```

Then, they use it to make requests with `http.get(...)`. This is a valid way of working with the package, but it does mean that every class that the `http`

package provides will need to be referenced by a preceding `http.`, such as `http.Response`. This can make the code a bit harder to work with in some instances. However, there are some advantages to using namespaces, as well:

1. **Avoid naming conflicts:** By using a namespace, you can avoid naming conflicts between different packages or libraries that you are importing. Since each package has its own namespace, you can use the same variable or class name from different packages without any conflicts.
2. **Better readability:** When you use a namespace, it makes your code more readable and organized. It allows you to group related classes, functions, and constants together under a common name, which makes it easier to understand the purpose of each import.
3. **Faster compilation:** When you use a namespace, it can improve the compilation time of your code. This is because the compiler can optimize the imports by only including the necessary code from each package, instead of importing the entire package.
4. **Easier code maintenance:** Using a namespace can make it easier to maintain your code in the long run. If you need to update a package, you can simply update the import statement without having to change all the references to that package throughout your codebase.
5. **Better code reusability:** When you use a namespace, it allows you to reuse the same code across different projects without any conflicts. Since each package has its own namespace, it can be easily included in different projects without any issues.

Another way is by simply importing `http`, then using `Client().get(...)`, where you instantiate the class and use it in the same line. No longer would you have to prefix each of the classes with the library name, however this can also become confusing. Plus, if you are authenticating to the API by using HTTP headers, you will have to add those headers each time you make a request, instead of having a single HTTP client which remembers those headers. Not to mention the fact that by instantiating a new `Client` for each call is horribly inefficient.

For these reasons, it is recommended to go with a hybrid solution of importing http normally, then creating a Client variable, which you can use to make your HTTP calls. Let us put this into action with a quick example:

```
1. import 'dart:convert';
2. import 'package:http/http.dart';
3.
4. Client http = Client();
5.
6. void main() async {
7.   final Response response = await fetchAlbum();
8.   if (response.statusCode == 200) {
9.     final data = jsonDecode(response.body);
10.    print('The album title is: ${data["title"]}');
11.  }
12. }
13.
14. Future<Response> fetchAlbum() {
15.   return
16.     http.get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'));
17. }
```

This code will fetch some data from an API, then print out a piece of it. There is a lot of room for improvement, though. For starters, it only checks for an HTTP status code of **200 (OK)** and has no error handling whatsoever. Instead of checking explicitly for a success state, we should instead check for the opposite:

```
1. void main() async {
2.   final Response response = await fetchAlbum();
```

```
3.     if (response.statusCode != 200) {
4.         print('There was an error fetching the album.');
5.         return;
6.     }
7.
8.     final data = jsonDecode(response.body);
9.     print('The album title is: ${data["title"]}');
10. }
11.
12. Client http = Client();
13.
14. Future<Response> fetchAlbum() {
15.     return
16.         http.get(Uri.parse('https://jsonplaceholder.typicode.com/albums/a'));
17. }
```

Note how the URI for the album to fetch has changed from a `1` to an `a` causing the API call to return a non-200 status code. We now check for that non-200 status code in the same way we were checking for a successful response before. If we receive any status code other than a 200, we print out our error message and then prevent the rest of the function from executing with the `return` statement. By doing it this way, we are using a guard clause style of programming. That is, we know what code we want to run if everything goes well, so let us just check for all the things which could go wrong *before* running the happy-path code. This will reduce the number of errors we have to check for later in our code.

Classifying API Responses

In our previous example, we referenced the data returned from the API when printing it out using `data["title"]`. This is not very robust, however. If the element lacks a title field, the code will break. Additionally, it does not

give us a nice Dart object to work with. It is time we create our own class to handle this, but first we need to understand the structure of the data we are working with. By printing out the `response.body`, we can see the JSON which is returned to us:

```
1. {userId: 1, id: 1, title: quidem molestiae enim}
```

JavaScript Object Notation (JSON) is a structured key-value way of representing data. Note that when we ask Dart to print out the JSON for us, it has stripped away the quotes which would normally be around any strings. A *raw* JSON response would look like the following:

```
1. {"userId": 1, "id": 1, title: "quidem molestiae enim"}
```

Understanding the difference in how the data can be displayed is vital. See how in both examples the integers are unquoted? They are a different data type (`int`) than the other value (a `String`). Trying to access the integer as a string will throw an error in your code, thus why it is important to know how the JSON data is structured.

Alright, now that we understand the structure of the data in the JSON response, we can create a class that we will eventually use to cast the response data into.

```
1. class Album {  
2.   int? userId;  
3.   int? id;  
4.   String? title;  
5.  
6.   Album({this.userId, this.id, this.title});  
7.  
8.   factory Album.fromJson(Map<String, dynamic> json) => Album(  
9.     userId: json["userId"],  
10.    id: json["id"],
```

```
11.     title: json["title"],  
12. );  
13. }
```

We have made all the different fields nullable, since we cannot guarantee the response of the API, and we have no idea if all the fields will be returned. We also created a **factory** that we can call when creating our new **Album** class. The **factory** will take in the JSON data (as a **Map<String, dynamic>**) and return the **Album** object. See how we were able to map the JSON values to the corresponding **Album** parameters?

Now that we have a class with parameters matching the JSON response, we can update our code to use it. First, let us fix the **fetchAlbum** method. Instead of returning a **Response**, we should return our **Album**. After all, it is not a **fetchResponse** method!

```
1. Future<Album?> fetchAlbum() async {  
2.   final Response response = await  
3.   http.get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'));  
4.  
5.   if (response.statusCode != 200) {  
6.     print('There was an error fetching the album.');//  
7.     return null;  
8.  
9.   final Map<String, dynamic> data = jsonDecode(response.body);  
10.  final Album album = Album.fromJson(data);  
11.  
12.  return album;  
13. }
```

This is already much better. We moved the error handling from the main method into the `fetchAlbum` method, too. It makes more sense for that logic to be here, rather than in the main method, because this is where the HTTP call is being made and where the logic deciding whether to cast the resulting JSON into an `Album` is stored. This results in a much cleaner `main` method, by comparison:

```
1. void main() async {  
2.   final Album? album = await fetchAlbum();  
3.   print('The album title is: ${album?.title ?? "Undefined"}');  
4. }
```

Also, take note of the line where we turn the data into an `Album` using our new `fromJson` method. If we wanted, we could check some parameters in the `Album` before returning it, such as making sure the title is not null. In this instance, we are just going to return whatever we end up with. Another way of handling this would be to specify defaults in the `Album`'s `fromJson` method. For example, you could use `json["title"] ?? "Undefined"` and mark the `title` as no longer nullable. That might not work in every situation, though, so instead we are simply checking for the title when we print it out and handling the error state there. There is one other method we can use to handle errors like this, which we will look at in the next section.

There is one more thing we can do to improve the `fetchAlbum` method, though. Since we know that the API references a given album by its integer ID, we can pass in that integer ID to the `fetchAlbum` method when we call it. This would result in a much more reusable piece of code, instead of the method only ever returning a single, given album. Here is what that might look like:

```
1. void main() async {  
2.   final Album? album = await fetchAlbum(id: 1);  
3.   print('The album title is: ${album?.title ?? "Undefined"}');  
4. }  
5.
```

```

6. Future<Album?> fetchAlbum({required int id}) async {
7.     final Response response = await
       http.get(Uri.parse('https://jsonplaceholder.typicode.com/albums/$i
d'));
8.
9.     if (response.statusCode != 200) {
10.         print('There was an error fetching the album.');
11.         return null;
12.     }
13.
14.     final Map<String, dynamic> data = jsonDecode(response.body);
15.     final Album album = Album.fromJson(data);
16.
17.     return album;
18. }
```

Now that we are passing in an ID to the method, we can leverage this flexibility to grab a whole bunch of albums:

```

1. void main() async {
2.     final List<int> albumsToFetch = [1, 2, 3];
3.     for (final int albumToFetch in albumsToFetch) {
4.         final Album? album = await fetchAlbum(id: albumToFetch);
5.         print("Album ${album?.id}'s title is: ${album?.title} ???
'Undefined'");
6.     }
7. }
```

Sending Data to an API

Now that we know how to receive data from an API, it is time to talk about sending data to an API. We will expand our previous example to include a `createAlbum` method that we can call. This method will tell the API to create an album with a given title:

```
1. Future<Album> createAlbum(String title) async {  
2.     final response = await http.post(  
3.         Uri.parse('https://jsonplaceholder.typicode.com/albums'),  
4.         headers: <String, String>{  
5.             'Content-Type': 'application/json; charset=UTF-8',  
6.         },  
7.         body: jsonEncode(<String, String>{  
8.             'title': title,  
9.         }),  
10.    );  
11.    if (response.statusCode != 201) {  
12.        throw Exception('Failed to create album.');  
13.    }  
14.    return Album.fromJson(jsonDecode(response.body));  
15.}  
16. }  
17. }
```

Here, we are using `http.post` rather than `http.get` to send data to the API. The structure is a bit different when we are doing a **POST** instead of a **GET**. For one, we need to specify the headers, so the API knows what type of data we are sending over. In this instance, we tell it that we are sending JSON data that has been encoded using UTF-8. Second, we must specify a body of the HTTP request that includes a JSON object with the data we are sending to the API.

The body starts out as a **Map**, and then we use `jsonEncode` from the `dart:convert` package to convert it to a JSON string. The `<String, String>` tells us what type of data the **Map** has for its keys and values. If you were sending a mix of strings, integers, lists, and so on, you would use `<String, dynamic>`, instead. Once the data has been converted to a JSON object, the last thing which differs is the URI to which we send the request.

If you recall, in the previous example, our HTTP **GET** specified an ID for the album. In this instance, since we are creating a new album, we have not been assigned an ID yet. For this reason, we omit the ID from the URI, and once the API request has been completed, the resulting data which is returned will have our brand-new ID.

Not all APIs are made equal, and it is important to note that. This method of creating and retrieving data is common, but far from the only way for APIs to be designed. It is important to check with the documentation of the API that you are working with to understand what URIs to use for the different **create**, **read**, **update**, and **delete** (CRUD) operations and what format the data should be sent in. To read the documentation of the example API we have been using, head over to <https://jsonplaceholder.typicode.com/>.

To run our code and create an album, it is as simple as the following:

```
1. void main() async {  
2.   final Album album = await createAlbum("My own album");  
3.   print("Album ${album.id}'s title is: ${album.title}");  
4. }
```

Look back at the `createAlbum` method for a minute. Remember how we handled error cases in the `fetchAlbum` method? The `createAlbum` method uses a different way of handling those error cases. This time, we are checking for the HTTP status code **201 (Created)** and in the instance that we do not receive a successful response, we **throw** an **Exception**, rather than making our return value nullable. This is an ideal way to handle errors, as we can use other built-in logic to catch those errors.

```
1. void main() async {  
2.   try {
```

```
3. final Album album = await createAlbum("My own album");
4. print("Album ${album.id}'s title is: ${album.title}");
5. } catch (e) {
6.   print("There was an exception: $e");
7. }
8. }
```

As you can see, sending data to an API is just as easy as retrieving data from the API. These examples are all in pure Dart code, however. Let us see how these examples might be used in the context of a Flutter application.

Flutter's Asynchronous Widgets

We need some basic boilerplate code to display a Flutter UI that we can work with; otherwise, we are just going to keep both the `fetchAlbum` and `createAlbum` methods, and the `Album` class:

```
1. void main() => runApp(MyApp());
2.
3. class MyApp extends StatelessWidget {
4.   @override
5.   Widget build(BuildContext context) {
6.     return MaterialApp(
7.       title: 'Flutter Demo',
8.       debugShowCheckedModeBanner: false,
9.       theme: ThemeData(
10.         primarySwatch: Colors.blue,
11.       ),
12.       home: Scaffold(
13.         appBar: AppBar(
```

```
14.           title: const Text('Albums'),  
15.           ),  
16.           body: AlbumList(),  
17.           ),  
18.       );  
19.   }  
20. }  
21.  
22. class AlbumList extends StatelessWidget {  
23.   @override  
24.   Widget build(BuildContext context) {  
25.     return const Text('TODO: Fetch and display a list of albums');  
26.   }  
27. }
```

With our boilerplate code in place, let us first look at how to fetch the albums and display them. For this, we are going to use a pair of widgets: `FutureBuilder` and `ListView`. We are also going to need to create a method that will let us fetch all the albums in a range of integers:

```
1. Future<List<Album>> fetchAlbums() async {  
2.   final List<int> albumsToFetch = List<int>.generate(20, (i) => i +  
1);  
3.   final List<Album> albums = [];  
4.  
5.   for (final int albumToFetch in albumsToFetch) {  
6.     final Album? album = await fetchAlbum(id: albumToFetch);  
7.     if (album != null) albums.add(album);  
8.   }
```

```
9.     return albums;  
10. }
```

You may notice some code in there that you do not yet recognize: the `List.generate` method. By giving it an integer (in this example, 20), it runs a method that many times and performs some action. The second parameter, (`i`), assigns the value of the current list index to the variable `i`, then returns a value of `i+1` and adds it to the list. This is repeated until the process has run for the number of iterations specified. The result is a list of integers ranging from 1 to 20. This is the list of albums that we will be fetching for our eventual list. (Extra credit: how could you expand this to get the next *page* of results, albums 21-40?)

Now that we have got a method that will fetch a given number of albums, we need a widget that will run that method for us. This is where `FutureBuilder` comes into play. `FutureBuilder` has two parameters: `future` and `builder`. The `future` parameter takes in the object that returns a future, which in our case will be the `fetchAlbums` method. The `builder` method will return a widget and has access to the `FutureBuilder`'s `AsyncSnapshot` data as well as the current `BuildContext`. The `AsyncSnapshot` has some very useful information that we will use to decide what widget to display to the user. Let us look at all this in action:

```
10. );
11. }
12. if (snapshot.connectionState == ConnectionState.done
&&
13.     snapshot.hasData) {
14.     return const Text("Done loading data");
15. }
16. return const Text("There was a problem loading the
album list.");
17. },
18. );
19. }
20. }
```

We are mostly interested in the `builder` method right now, as the rest is self-explanatory. Remember, the `builder` must *always* return a widget, which is why we have a guard at the bottom that returns an error message. The first check will return a `CircularProgressIndicator` while we are waiting for the `Future` to complete, whereas the second check will return the widget we want to display once the `Future` has successfully completed. If neither of those states can be satisfied (such as in the case where the connection times out, there is no data returned, and so on), an error message will be shown to the user.

At this point, we have a widget which is calling the API, retrieves the list of albums, and handles the waiting and error states. Next, we need to replace the placeholder text with a list of the album titles that were received. We are going to do this with the `ListView` widget.

```
1. if (snapshot.connectionState == ConnectionState.done &&
snapshot.hasData) {
2.     final List<Album> albums = snapshot.data;
3.     return ListView.builder(
```

```
4.     itemCount: albums.length,  
5.     itemBuilder: (BuildContext context, int index) => ListTile(  
6.       title: Text(albums[index].title ?? "Undefined"),  
7.     ),  
8.   );  
9. }
```

`ListView.builder` is a magical little widget. It builds a scrollable list and only builds the exact number of widgets that can be displayed on the screen. For this reason, it is highly performant with very long lists, especially in contrast to something like a `Column`, which builds all its children all the time, even when they are not on screen.

By specifying an `itemCount`, the `ListView.builder` knows precisely how many items it will have to build. It uses that information to iterate through the `itemBuilder` method, passing the current `index` of the item it is building as a parameter. We can use that `index` to grab the corresponding item from our list, which we do with `albums[index]` in this example. The result is that a scrollable, performant list of the album titles is generated automatically for us.

We chose to use a `ListTile` to display the title of the album, but any widget will do. Try swapping the `ListTile` for a `Card` widget for some added dimension to the resulting list. The title parameter will need to be swapped for a child parameter, and the content will benefit from some extra padding, like the following:

```
1. if (snapshot.connectionState == ConnectionState.done &&  
  snapshot.hasData) {  
2.   final List<Album> albums = snapshot.data;  
3.   return ListView.builder(  
4.     itemCount: albums.length,  
5.     itemBuilder: (BuildContext context, int index) => Card(  
6.       child: Padding(
```

```
7.         padding: const EdgeInsets.all(8),
8.         child: Text(albums[index].title ?? "Undefined"),
9.     ),
10.    ),
11. );
12. }
```

Now that we can display a list of albums, we should add a way for us to create an album. Let us add a `FloatingActionButton` to our `Scaffold`, so we can add an album from anywhere. However, to properly pass the `BuildContext` into the `onPressed` method, we will want to break out the `Scaffold` into its own widget, which we will call `Home`:

```
1. class MyApp extends StatelessWidget {
2.   const MyApp({super.key});
3.
4.   @override
5.   Widget build(BuildContext context) {
6.     return MaterialApp(
7.       title: 'Flutter Demo',
8.       debugShowCheckedModeBanner: false,
9.       theme: ThemeData(
10.         primarySwatch: Colors.blue,
11.       ),
12.       home: const Home(),
13.     );
14.   }
15. }
16.
```

```
17. class Home extends StatelessWidget {  
18.     const Home({super.key});  
19.  
20.     @override  
21.     Widget build(BuildContext context) {  
22.         return Scaffold(  
23.             appBar: AppBar(  
24.                 title: const Text('Albums'),  
25.             ),  
26.             floatingActionButton: FloatingActionButton(  
27.                 onPressed: () async {  
28.                     // TODO: Implement dialog to get an album title, then  
                     //       create the album  
29.                 },  
30.                 child: const Icon(Icons.add),  
31.             ),  
32.             body: const AlbumList(),  
33.         );  
34.     }  
35. }
```

Flutter offers a `showDialog` method for us to display a popup where we can collect information from the user. This is an asynchronous method that will display a `Dialog` over the UI and can return some sort of value, if desired. It takes in a `BuildContext` and uses that context in its `builder` method. The `builder` method returns the `Dialog` itself. We can leverage this system to create a popup that asks the user to enter the name of an album.

Let us start by building the helper method that will show the **Dialog** and return the value:

```
1. Future<String?> _userInputDialog(BuildContext context) async {  
2.     return await showDialog<String>(  
3.         context: context,  
4.         builder: (BuildContext context) {  
5.             return SimpleDialog(  
6.                 title: const Text('Create Album'),  
7.                 contentPadding: const EdgeInsets.all(24),  
8.                 children: [  
9.                     TextFormField(  
10.                         decoration: const InputDecoration(  
11.                             hintText: "Album name",  
12.                         ),  
13.                         onFieldSubmitted: (String? value) {  
14.                             Navigator.pop(context, value);  
15.                         },  
16.                         ),  
17.                     ],  
18.                 );  
19.             },  
20.         );  
21.     }
```

This simple **Dialog** will have a title and a text input. We are using a **SimpleDialog**, which is a subclass of the **Dialog**. It provides us with the

necessary options we need, but there are other types of **Dialogs** you could use, such as the **AlertDialog** for example. For the user to submit their input, they simply need to press the *Enter* key on the keyboard once they are finished. To cancel or dismiss the dialog, they can tap or click anywhere outside of the dialog.

With our helper function in place, we only need to hook it up to the **onPressed** method of our **FloatingActionButton**:

```
1. class Home extends StatelessWidget {  
2.   const Home({super.key});  
3.  
4.   @override  
5.   Widget build(BuildContext context) {  
6.     return Scaffold(  
7.       appBar: AppBar(  
8.         title: const Text('Albums'),  
9.       ),  
10.      floatingActionButton: FloatingActionButton(  
11.        onPressed: () async {  
12.          final String? title = await _userInputDialog(context);  
13.          if (title == null) {  
14.            throw Exception('Cancelled Album creation.');  
15.          }  
16.  
17.          if (title.isNotEmpty) {  
18.            final Album album = await createAlbum(title);  
19.            print("Created new album! ${album.title}");  
20.          }  
21.        },  
22.      );  
23.    }  
24.  }  
25.  
26. void _userInputDialog(BuildContext context) {  
27.   showDialog(context: context, builder: (context) {  
28.     return AlertDialog(  
29.       title: const Text('Create New Album'),  
30.       content: const Text('Please enter the title of the new album'),  
31.       actions: [  
32.         TextButton(onPressed: () {  
33.           Navigator.of(context).pop();  
34.         },  
35.         child: const Text('Cancel'),  
36.       ],  
37.     );  
38.   });  
39. }  
40.  
41. Future<Album> createAlbum(String title) {  
42.   // Implementation of creating an album goes here.  
43.   return Future.value(Album(title: title));  
44. }
```

```
20.          }
21.          },
22.          child: const Icon(Icons.add),
23.          ),
24.          body: const AlbumList(),
25.        );
26.      }
27. }
```

Since we made our `onPressed` method an `async`, we can `await` the return value of the `Dialog` as a nullable `String` (remember, the user can always cancel.) After a quick check to make sure the title is not empty, we can `await` our `createAlbum` method with the new title, returning a newly created `Album`. Finally, we print the album details out to prove that it was, indeed, created. The only problem is that the new album is not added to the list. For that, we are going to need to talk about streams.

Sinks and Streams

Sibling to the `async/await` operators is the `async*/yield` operators. Let us look at an analogy to understand the relationship between these concepts.

Let us suppose that you are thirsty and would like a glass of water. You pick up the glass, fill it with water, and once it is full you take a drink. You can do other things while you are waiting for the glass to fill, but you cannot take a drink until it is full. This is `async` (you can do other things while you wait for the glass to fill) and `await` (you must wait for the glass to fill completely before taking a drink).

Now, suppose you want to drink from the glass while it is being filled. Not only that, but you have got a button you can push that will add lemonade into the stream of water. Each time you press the button, a bit of lemonade is added to the stream. Refreshing! To take a drink while your glass is being filled, you grab a straw and start sucking on it while the water fills the glass. Then, you decide to push the button to add some lemonade to the mix. This is `async*` (the act of filling the glass does not block you from consuming the

beverage) and `yield` (pushing the button adds more things into the stream in a process called a sink, which is then emitted, or yielded, from the stream).

In our previous example, our `ListView.builder` used the data from the `FutureBuilder` to compile a list of `Albums`, which it then used to build the individual `Cards` displaying the album titles. We displayed a `CircularProgressIndicator` while the albums were being fetched from the API, which prevented us from showing any album until they were all loaded. Furthermore, once the `fetchAlbums` method has completed, the list of albums is finalized, meaning we cannot add anything new to it without fetching everything all over again. So much for adding our newly created album to the list.

Clearly, we are going to need a different approach to accomplish what we want here. Let us refactor our code to use sinks and streams. The first thing we will need to do is pull in the `dart:async` library to get access to the `StreamController` widget. This widget will allow us to control our stream. After we have imported the library, we can go ahead and create a new `StreamController`:

1. import 'dart:convert';
2. import 'dart:async';
3. import 'package:flutter/material.dart';
4. import 'package:http/http.dart';
- 5.
6. void main() => runApp(const MyApp());
7. Client http = Client();
8. final StreamController streamController = StreamController();
9. ...

We will use this `StreamController` to add our events to the steam's sink and to listen to whatever steams back out. Let us start by reworking the `fetchAlbums` method to use the new `StreamController`. Since we are going to be sending a sink event instead of returning a value, our return type becomes `void`. We still need the `async` keyword because we will be `awaiting` each album as we fetch it from the API, though. The other major change is that we no longer

keep track of all the `Albums` that come back from the API. Instead, we pass each of them off to the `StreamController` to deal with. This means we no longer need our `List<Album>` variable, and we can remove the `return` statement. In place of those, we will take the album we get back and add an event to the `StreamController`'s sink:

```
1. void fetchAlbums() async {  
2.   final List<int> albumsToFetch = List<int>.generate(10, (k) => k +  
1);  
3.  
4.   for (final int albumToFetch in albumsToFetch) {  
5.     final Album? album = await fetchAlbum(id: albumToFetch);  
6.     streamController.sink.add(album);  
7.   }  
8. }
```

Note that we could technically add the event directly to the `StreamController`, too. This way, however, we are being more verbose for the sake of education. If we were not concerned about adding new events to our sink later, we could forego the `StreamController` entirely. Instead, we would opt for a return value of `Stream<Album>` and use the `async*` keyword. Then, instead of adding an event to the `StreamController`'s sink, we would `yield` the album. That code would look like the following:

```
1. Stream<Album> fetchAlbums() async* {  
2.   final List<int> albumsToFetch = List<int>.generate(10, (k) => k +  
1);  
3.  
4.   for (final int albumToFetch in albumsToFetch) {  
5.     final Album? album = await fetchAlbum(id: albumToFetch);  
6.     yield album;  
7.   }  
8. }
```

```
8. }
```

At that point, we would not care about the `StreamController` at all, so we could simply subscribe to `fetchAlbums` as a stream. However, since we know that we eventually want to create a new album and add it to the stream, it is better to use the `StreamController` in this instance.

The next thing we need to do is update the `AlbumList` widget. Previously, we were using a `FutureBuilder` widget and calling the `fetchAlbums` method to get the data we needed to build the list. Since `fetchAlbums` no longer returns a list of albums, this will not work for us anymore. Instead, we will use the `StreamBuilder` widget to accomplish the task, and instead of a `future` parameter, we have a `stream` parameter. Where does the stream come from; you ask? The answer lies within our `StreamController`. By specifying `streamController.stream` as the stream to use, any data emitted from the `StreamController` will be used to build the list.

We are not quite done yet, though. The `ListView.builder` still wants a list of items to use to build itself. Before, that list was provided directly by the `fetchAlbums` method. Now, we no longer have a list — we just have a stream of items. We are going to have to build and maintain our own list. We will prefix the variable with an underscore, just to remind ourselves that we are only using it within the context of this single widget, and we should not be trying to access it from anywhere else. We will also want to place it outside our `build` method, so it does not get reinitialized to an empty list each time the widget rebuilds:

```
1. ...
2. final List<Album> _albums = [];
3. ...
4. @override
5. Widget build(BuildContext context) {
6. ...
}
```

We will also want to convert our `StatelessWidget` into a `StatefulWidget`, so we can start the stream when the widget is initialized. Finally, we want to change the way we handle the incoming data. Instead of assigning `snapshot.data` to

the `albums` variable, we want to add that `snapshot.data` (which is now a single `Album`) to our new list of albums, `_albums`. We can also remove our `CircularProgressIndicator` and clean up the logic which tests the connection state, since we no longer care what the connection state is and only if there is new data to add:

```
1. class AlbumList extends StatefulWidget {  
2.   const AlbumList({super.key});  
3.  
4.   @override  
5.   _AlbumListState createState() => _AlbumListState();  
6. }  
7.  
8. class _AlbumListState extends State<AlbumList> {  
9.   final List<Album> _albums = [];  
10.  
11.  @override  
12.  Widget build(BuildContext context) {  
13.    return StreamBuilder(  
14.      stream: streamController.stream,  
15.      builder: (BuildContext context, AsyncSnapshot snapshot) {  
16.        if (snapshot.hasData) {  
17.          _albums.add(snapshot.data);  
18.        }  
19.        return ListView.builder(  
20.          shrinkWrap: true,  
21.          itemCount: _albums.length,
```

```
22.         itemBuilder: (BuildContext context, int index) {
23.             return Card(
24.                 child: Padding(
25.                     padding: const EdgeInsets.all(8),
26.                     child: Text(_albums[index].title ?? "Undefined"),
27.                 ),
28.             );
29.         },
30.     );
31. },
32. );
33. }
34.
35. @override
36. void initState() {
37.     super.initState();
38.     fetchAlbums();
39. }
40. }
```

Now, if you run the code, you will see the individual albums stream into the list, one by one. There is just one last thing we must do: when we create a new album, we need to add it to the list. Fortunately, this is the exact same process that we use in the `fetchAlbums` method. In our `Home` widget, once we have created the new album, we simply add an event to the `StreamController`:

```
1. class Home extends StatelessWidget {
2.     const Home({super.key});
```

```
3.
4. @override
5. Widget build(BuildContext context) {
6.   return Scaffold(
7.     appBar: AppBar(
8.       title: const Text('Albums'),
9.     ),
10.    floatingActionButton: FloatingActionButton(
11.      onPressed: () async {
12.        final String? title = await _userInputDialog(context);
13.        if (title == null) {
14.          throw Exception('Cancelled Album creation.');
15.        }
16.
17.        if (title.isNotEmpty) {
18.          final Album album = await createAlbum(title);
19.          streamController.sink.add(album);
20.        }
21.      },
22.      child: const Icon(Icons.add),
23.    ),
24.    body: const AlbumList(),
25.  );
26.}
```

27. }

Now, whenever you create a new album, it will be added to the bottom of the list, automatically!

Conclusion

Congratulations on making it through this chapter. Some of the content we covered here was relatively complex. We learned all about concurrency, including the `async/await` keywords, `async*/yield` keywords, and using a `StreamController` to add an event to the sink and listen to the objects being emitted from the stream, along with their corresponding widgets. We also learned both how to receive data from and send data to an API. Then, we learned how to convert the raw data we received from the API into a Dart object that we care about. Armed with this knowledge, you will have all the tools you need to build dynamic applications that leverage the power of APIs — opening a whole new world of possibilities! What will you build first? Maybe a weather app?

In the upcoming chapter, we are going to explore just what exactly we were doing when we invoked `Navigator.pop()` in our `Dialog`. In fact, we will take an in-depth look at what exactly `Navigator` is, how it works, and what its shortcomings are. Then, we will look at the newer API that unlocks even more potential, and finally we will explore a package that helps make our life easier when we are working with navigation and routing. See you over there!

Questions

1. What is a `Future`?
2. What is the difference between synchronous code and asynchronous code?
3. What happens to the application when there is a long-lived, synchronous task being executed?
4. What are the `async` and `await` keywords used for?
5. What type of value does an asynchronous operation return?
6. What is an API?

7. What package(s) can be used to communicate with an API?
8. What sort of data is generally returned from an API?
9. What is an HTTP status code, and what are some common ones?
10. What is a guard clause?
11. What is JSON, and what are some considerations which are important to consider when working with JSON data?
12. When communicating with an API using an HTTP client, a “verb” is used to determine which kind of action the client should take. The verbs **POST** and **GET** are used for which types of operations? Extra credit: What about **PUT**, **PATCH**, and **DELETE**?
13. When sending a POST request to an API, which properties are required to be passed into the request?
14. What does CRUD mean?
15. What HTTP verbs relate to which CRUD operations?
16. How can you generate `List<int>` containing all the odd numbers from 27 to 99?
17. How can you tell if a `FutureBuilder` has completed its future?
18. Why would somebody choose to use a `ListView` over a `Column`? Why would somebody choose to use a `Column` over a `ListView`?
19. How do you display a dialog to a user?
20. Given a dialog with two options, yes and no, how do you return either `true` or `false` from the dialog?
21. What is a sink? What is a stream?
22. What are the `async*` and `yield` operators used for?
23. What is the `async*` equivalent to a `FutureBuilder`?
24. What library provides widgets which are used with sinks and streams?
25. How do you add an event to a sink?

Key Terms

- **Application Programming Interface (API)**: A connection between two pieces of software to exchange information.

- **async**: Dart’s `async` keyword is used to mark a method as having a return type of `Future<Object>`.
- **async***: Dart’s `async*` keyword is used to mark a method as having a return value of `Stream<Object>`.
- **Asynchronous operations**: Code that executes in parallel, neither blocking the other.
- **await**: Dart’s `await` command is used to wait for the execution of an asynchronous operation to complete its future. The `await` keyword must exist within an asynchronous method or widget which expects a future.
- **Future**: A promise given by an asynchronous operation to return a value when it has finished processing.
- **JSON**: JavaScript Object Notation. A key/value pair-based textual representation of data. JSON is commonly used when sending data to and receiving data from APIs.
- **String interpolation**: Code that is executed whenever a `String` is evaluated during a widget build or method call. A null string being interpolated will return the string “`null`” while methods being called will run at the time that the string is evaluated. Code that is interpolated is contained within `{}$` in a string.
- **Synchronous operations**: Code that executes one after the other. The next line cannot execute until the previous has finished, which is *blocking*.
- **yield**: Dart’s `yield` keyword is used inside an `async*` method to add an item to the stream.

Further Reading

- Learn more about JSON and serialization: <https://docs.flutter.dev/development/data-and-backend/json>
- HTTP status codes and their meanings: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>

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

Navigation and Routing

Introduction

Most modern applications have their content organized in multiple, distinct screens, which break up a workflow from being a single and monolithic behemoth into smaller and purpose-driven views. The concept of sending a user from one screen to another is routing, which we accomplish using the action of navigation. A user navigates to a new screen, where the application then routes them to it.

In this chapter, we are going to explore how navigation and routing work in Flutter. We will explore Navigator 1.0, the simplest and most approachable method of routing (albeit with its own drawbacks and limitations), then we will investigate the newer and more powerful Navigator 2.0. Finally, we will explore a package that seeks to make Navigator 2.0 as easy to use as possible.

Structure

In this chapter, we will discuss the following topics:

- A general discussion of navigation and routing concepts
- An introduction to Navigator 1.0
- Routing using named routes
- Passing arguments into a route

- Returning data from a screen
- Animating a widget from one route to another
- Introduction to Navigator 2.0 — the Router API
- Using packages to help with routing

Objectives

After finishing this chapter, you will be able to accurately describe the concepts behind navigation and routing. You will be able to use this knowledge to build a simple application that uses Flutter's most basic navigation system, then learn how to use the newer and more powerful system which was introduced later. Finally, you will learn how to simplify a more complex routing scenario by using a package from Pub.

General Discussion of Navigation and Routing Concepts

As we briefly touched on in the introduction, applications respond to a user's intent to navigate by routing them to the place they would like to go. If you have ever clicked a link on a website, you will know exactly what this process looks like: you click a link, the browser thinks for a minute, the URL changes, then your new page loads. The browser manages your *intent* to navigate to the new page in the way of updating the URL that is currently being displayed, then displaying the page.

In an application, the mechanics can differ, but the overall process is remarkably similar. As a user, you tap on a cog wheel icon with the *intent* to access settings and the application *routes* you to the settings screen, very much like a Web page would work. In fact, if you will recall, Flutter itself was born out of the Web-world, so many of the concepts and paradigms are still closely related to its history with the Web.

Where Flutter differs from the Web, however, is in how it manages the state objects of screens not currently in the foreground. On the Web, once you navigate away from a page, you lose the state of the elements on the page (unless the state has been written to a local cache, then injected back into the page when you navigate back to it, or the backend manages the state similarly). This is the reason why your Web browser will ask you if you are

quite certain that you would like to navigate away from a form once you have started to fill it out but have not yet submitted it: if you do, the state of the form will be lost, and you will have to start over if you navigate back to the page.

In contrast to the Web, Flutter can retain state for screens not currently being displayed to the user, but from which they had previously navigated. The best way to visualize this is to consider a deck of playing cards. When navigating around the Web, only a single playing card can be on the table at a time. When navigating within Flutter, however, cards are added and removed from the stack of cards on the table.

It is this concept of a navigation *stack* that will help us to make sense of how routing works in applications. When an application launches, the first card in the stack is a loading screen. Once the application has loaded, the loading screen card is removed from the stack and replaced with the main application card. When a user performs an action that routes them to a new screen, a card is placed on top of the main application card, thereby creating a stack of two cards. Since the main screen card has not been removed from the stack, its associated state is maintained, along with the state of the new card on the stack. When the user presses the back button, the new card is removed from the stack, returning us to the original card in the stack — the main screen — where it has maintained its own state while it was *under* the new screen.

This whole navigating and routing process of stacking cards happens inside the scope of a **Navigator**, which we will discuss in the next section. What is important to note is that an application can have *multiple* **Navigators**, thereby creating *nested* navigation stacks. Let us dive into Navigator 1.0 and learn how all this works.

Introduction to Navigator 1.0

Flutter has both imperative and declarative routing mechanisms, using **Navigator** and **Router**, respectively. Before we break down what that means, it is important to note that **Navigator** is sometimes referred to as *Navigator 1.0* whereas the Router API is sometimes referred to as *Navigator 2.0*. This does not mean that **Router** replaces **Navigator** — far from it (this may contrast with how one might typically consider version numbers to work, with

newer versions replacing older ones). In fact, **Router**, as we will discuss later, is built using **Navigator**. We need to understand how **Navigator** works before we can understand **Router**, so let us dig into the concept, starting with the difference between imperative and declarative code, as understanding this concept is vital to understanding how **Navigator** and **Router** work.

Code that is *imperative* has step-by-step instructions to complete the task at hand. *Declarative* code, on the other hand, simply provides a description of what you *want* but not necessarily how to go about getting it. To put it more simply in our context of routing and navigation, **Navigator** (the imperative system) is given a list of routes, then told to navigate to one of those routes, whereas **Router** requires a bit more boilerplate code to describe how to parse the intent of a route (by, for example, pulling out arguments passed along with the route) before performing the routing action. By using **Router**, we are simply describing where we want to navigate *to* and with what properties; then **Router** figures out how to make that happen for us. These are not mutually exclusive, however. In fact, **Router** (the declarative system) is built *using* **Navigator** (the imperative system).

Let us explore using **Navigator** to route to a new screen and back. The first thing we will need is two screens to route between. One of those screens can be the main screen of your application. The other screen should have its own **Scaffold** and body. Feel free to use this as a starting point:

```
1. class MainScreen extends StatelessWidget {  
2.   const MainScreen({super.key});  
3.  
4.   @override  
5.   Widget build(BuildContext context) {  
6.     return Scaffold(  
7.       appBar: AppBar(  
8.         title: const Text('This is the main screen of my  
application'),  
9.       ),
```

```
10.        body: Center(
11.            child: ElevatedButton(
12.                child: const Text('Navigate!'),
13.                onPressed: () {
14.                    // Navigate to new screen when tapped.
15.                },
16.            ),
17.        ),
18.    );
19. }
20. }
21.
22. class SecondScreen extends StatelessWidget {
23.     const SecondScreen({super.key});
24.
25.     @override
26.     Widget build(BuildContext context) {
27.         return Scaffold(
28.             appBar: AppBar(
29.                 title: const Text("I'm so glad you found your way over.
30.                     "),
31.                 body: Center(
32.                     child: ElevatedButton(
33.                         onPressed: () {
```

```
34.           // Navigate back to main screen when tapped.  
35.           },  
36.           child: const Text('See you later!'),  
37.           ),  
38.           ),  
39.           );  
40.       }  
41.   }
```

Now all that is left is to set up **Navigator** to route from the first screen to the second, then back. On the first screen, set the **onPressed** method to ask **Navigator** to push a new *card* onto the stack:

```
1. onPressed: () {  
2.   Navigator.push(  
3.     context,  
4.     MaterialPageRoute(builder: (context) => const  
5.     SecondScreen()),  
6.   );  
}
```

Tapping the button now will route you to the second screen. Getting back to the first screen is just as easy. On the second screen, set the **onPressed** method to ask **Navigator** to pop the current card off the stack:

```
1. onPressed: () {  
2.   Navigator.pop(context);  
3. }
```

And... that is it! You now know all there is to know about navigating to a new screen and back! Except... what if you want to navigate to the same

screen from multiple places in your application? Would that not mean writing a bunch of duplicate code? Or, what if you want to replace the current screen with the new one so a user cannot press the `back` button to get back to a screen they should not, like when the splash screen is finished, and the user is presented with the main screen? And what if you want to pass information from the first screen to the second? This method will not work at all since no data can be passed into the route. We have a bit more to investigate, so let us investigate each of these use cases one at a time.

Routing Using Named Routes

Let us suppose that your application has a settings screen, which you want to be easily accessible from any screen. On each new screen you create, you add a button to navigate to the settings screen, but the `onPressed` method always has a `Navigator.push` to a `MaterialPageRoute` with a `builder` method. That is a lot of code just to navigate to a new screen! Plus, it means that we must remember exactly what our settings screen's widget name is (and what happens if you have a similarly named widget somewhere else in the application? Oh boy!) Would it not be great if we could simply route to the new screen *by name*? Like, *hey, Navigator! Send me to the settings screen!* Fortunately, we can: by using named routes.

Named routes will look familiar to anyone who has used a Web browser before. Each route will look like a path in a URL. In fact, `Navigator` will treat these named routes *like* a URL path. The difference will be that there will not be any `www` or `.com` associated with the routes since that is all handled internally. Instead, we are worried about what would come *after* that. It is like going to `https://www.my-favorite-website.com/my-favorite-page` except that `Navigator` knows about the `https://www.my-favorite-website.com` part, so we do not have to specify it. That leaves us with just the `/my-favorite-page` to manage. If you went to the `main` website, there would not be anything after the slash, so the slash itself is the `root` of the route. This will make more sense as we dig into some examples.

To set up named routes, you will have to find your main `MaterialApp` and define a couple of properties. The first thing we will need to define is the `routes` parameter, which contains a `map` of route names to the screen we are navigating to:

```
1. MaterialApp(  
2.   routes: {  
3.     '/': (context) => const MainScreen(),  
4.     '/settings': (context) => const SettingsScreen(),  
5.   },  
6. )
```

The next thing we need to do is set an initial route, which will display the main content of the application when it is loaded. Before, we used to do this by specifying the `home` property. Now, we will use the `initialRoute` property — so go ahead and remove `home`!

```
1. MaterialApp(  
2.   initialRoute: '/',  
3.   routes: {  
4.     '/': (context) => const MainScreen(),  
5.     '/settings': (context) => const SettingsScreen(),  
6.   },  
7. )
```

At this point, we have everything we need to solve our first *two* problems: we can now navigate to a *named* route, and we can *replace* a route with a new one. Let us do that.

First, to navigate to a named route, we replace `Navigator.push` with `Navigator.pushNamed`, and instead of passing in a `MaterialPageRoute`, we pass in the name of the route:

```
1. onPressed: () {  
2.   Navigator.pushNamed('/settings');  
3. }
```

If you want to replace the current card in the stack with your new route, you use `Navigator.pushReplacementNamed`, instead:

```
1. onPressed: () {  
2.   Navigator.pushReplacementNamed('/settings');  
3. }
```

Of course, there is also a `Navigator.pushReplacement` for when you are not using named routes and navigating back whether using named routes or a `MaterialPageRoute` works just the same.

You also have the option of specifying the route name in the destination widget. Outside of the `build()` method, try adding something like the following:

```
1. static const routeName = '/settings';
```

Then you can use it in your routing table:

```
1. routes: {  
2.   '/': (context) => const Home(),  
3.   SettingsScreen.routeName: (context) => const SettingsScreen(),  
4. },
```

Using named routes is an excellent and effortless way to keep the routing in your app clean and simple. If you decide to write a new settings screen but you do not want to immediately remove the old one, it is much easier to update the named route than it is to track down all the different places where you are generating a `MaterialPageRoute`.

[Passing Arguments into a Route](#)

Sometimes it is necessary to pass some data back and forth when routing. For example, you may find yourself with a product page where the user can choose the quantity of an item to purchase, then immediately checkout their order. The checkout screen would need to know what the desired product is, the quantity, and the price. Passing in this data using arguments means you

can build a more generic checkout screen, capable of accepting *any* item(s), in any quantity, at any price.

There are a couple of separate ways we can do this, including defining the arguments and passing them into the widget, or by extracting the arguments using `onGenerateRoute`. Let us first look at the former method.

To begin, we need to define a class that contains the arguments that we will pass into our route. Let us build one for the shopping cart example, making sure that it is flexible enough to accept multiple items when we go to checkout:

```
1. class StoreItem {  
2.   final String name;  
3.   final double price;  
4.  
5.   const StoreItem({required this.name, required this.price});  
6. }  
7.  
8. class CheckoutScreenArguments {  
9.   final List<Map<StoreItem, int>> shoppingCart;  
10.  
11.  const CheckoutScreenArguments(this.shoppingCart);  
12. }
```

Here, we have created a `StoreItem` class, which will hold both the name of the item and its price. Next, we made a `ChecoutScreenArguments` class with one property (although we could have as many properties as we want.) This property will accept a `List` of `Maps` that map a `StoreItem` to the quantity of items in the cart. That might look like the following:

```
1. const StoreItem shirt = StoreItem(name: "Super Rad Shirt", price:  
12.99);  
2. const StoreItem shoes = StoreItem(name: "Comfy Shoes", price:  
34.99);
```

```
3. 
4. const CheckoutScreenArguments shoppingCart =
  CheckoutScreenArguments(
5.   [
6.     {shirt: 4},
7.     {shoes: 1},
8.   ],
9. );
```

Now that we have a proper screen arguments class, we need to make a couple of changes to the screen that takes the arguments. First, we will move the route name into the screen, itself (Line 2, in the following example), then we can extract the arguments using the `ModalRoute.of(context)` inherited widget (Lines 8 and 9):

```
1. class CheckoutScreen extends StatelessWidget {
2.   static const routeName = '/checkout';
3. 
4.   const CheckoutScreen({Key? key}) : super(key: key);
5. 
6.   @override
7.   Widget build(BuildContext context) {
8.     final args =
9.       ModalRoute.of(context)!.settings.arguments
10.      as CheckoutScreenArguments;
11.     return Scaffold(
12.       body: ListView.builder(
13.         itemCount: args.shoppingCart.length,
```

```
14.         itemBuilder: ((context, index) => ListTile(  
15.             title: Text(args.shoppingCart[index].keys.first.name),  
16.             subtitle: Text(  
17.                 '${args.shoppingCart[index].values.first} ×  
18.                 ${args.shoppingCart[index].keys.first.price}'),  
19.             ),  
20.         );  
21.     }  
22. }
```

Now that you have a variable holding the screen arguments, all that is left is to pass your shopping cart over to the checkout screen:

```
1. ElevatedButton(  
2.     child: const Text('Checkout'),  
3.     onPressed: () {  
4.         Navigator.pushNamed(  
5.             context,  
6.             CheckoutScreen.routeName,  
7.             arguments: shoppingCart,  
8.         );  
9.     },  
10. ),
```

That is all it takes to pass arguments into a route! There is another way we can do it, though. The second method uses the `onGenerateRoute` parameter in your `MaterialApp`. Using this method, you do not need to extract the arguments inside the widget using `ModalRoute.of(context)`:

```
1. MaterialApp(  
2.   onGenerateRoute: (settings) {  
3.     if (settings.name == MessageScreen.routeName) {  
4.       final args = settings.arguments as  
      MessageScreenArguments;  
5.  
6.     return MaterialPageRoute(  
7.       builder: (context) {  
8.         return MessageScreen(  
9.           title: args.title,  
10.          message: args.message,  
11.        );  
12.      },  
13.    );  
14.  }  
15.  
16.  if (settings.name == CheckoutScreen.routeName) {  
17.    final args = settings.arguments as  
      CheckoutScreenArguments;  
18.  
19.    return MaterialPageRoute(  
20.      builder: (context) {  
21.        return CheckoutScreen(  
22.          shoppingCart: args.shoppingCart,  
23.        );  
24.      }  
25.    );  
26.  }  
27.  
28.  return MaterialPageRoute(  
29.    builder: (context) {  
30.      return HomeScreen(  
31.        user: args.user,  
32.      );  
33.    }  
34.  );  
35.  
36.}
```

```
24.        },
25.    );
26.  }
27. 
28. assert(false, 'Need to implement ${settings.name}');
29. return null;
30. },
31. )
```

In this example, we are building a `MessageScreen` using a `MessageScreenArguments` class, as well as our trusty `CheckoutScreen` using a `CheckoutScreenArguments` class. While this reduces the complexity of the individual screens you are routing to, it *increases* the complexity of your `MaterialApp`. For this reason, it is common to break out the method into its own file, which would look something like in the following example:

```
1. MaterialApp(
2.   onGenerateRoute: routeHandler,
3. );
4. // The following method can be saved to a different file, then
5. // imported.
5. MaterialPageRoute? routeHandler(settings) {
6.   (settings) {
7.     if (settings.name == MessageScreen.routeName) {
8.       final args = settings.arguments as
9.       MessageScreenArguments;
10.      return MaterialPageRoute(
11.        builder: (context) {
```

```
12.         return MessageScreen(
13.             title: args.title,
14.             message: args.message,
15.         );
16.     },
17. );
18. }
19.
20. assert(false, 'Route handler not configured for ${settings.name}');
21. return null;
22. };
23. }
```

Ultimately, Flutter gives you the flexibility to build your application in a way that makes the most sense for whatever project you are working on. Using `onGenerateRoute` in this way may be overkill, especially if only one or two of your screens require arguments. That will be up to you to decide. Can you think of another thing that you can use `onGenerateRoute` for? (Here is a hint for one thing you could use it for: analytics.)

[Returning Data from a Screen](#)

Passing data to a new screen is extremely useful, but sometimes we want to pass data back to the screen we were just looking at. You are building an input that asks your user to choose between two options. The design of the option picker is such that it would be clearer to the user by being on its own screen, rather than inline on the form. The user taps on a widget which routes them to a new screen where they make their selection, and then they are routed back to the initial screen where their option is used to update the form. The question becomes: how do we accomplish passing the data back to the initial screen after they have made their selection?

The solution is much simpler than one might imagine. When routing to a new screen, you can simply `await` the result and pass the data back in the `Navigator.pop` method. Let us start by building a simple screen where we will both display the result of the selection and a button that we will use to route to the selection screen. We know this will need to be a `StatefulWidget` because we need to update the widgets based on the return value:

```
1. class _HomeState extends State<Home> {  
2.   String? _chosenOption;  
3.   @override  
4.   Widget build(BuildContext context) {  
5.     return Scaffold(  
6.       body: Column(  
7.         children: [  
8.           Text(_chosenOption ?? ''),  
9.           ElevatedButton(  
10.             child: const Text('Choose one'),  
11.             onPressed: () {  
12.               // TODO: Route to the option screen and get the  
13.               // value back  
13.             },  
14.             ),  
15.           ],  
16.         ),  
17.       );  
18.     }  
19. }
```

Since `_chosenOption` is nullable, the `Text` will display nothing, for now. Next, we need a screen to route to, which includes our options:

```
1. class ChooseOneScreen extends StatelessWidget {  
2.   const ChooseOneScreen({super.key});  
3.  
4.   @override  
5.   Widget build(BuildContext context) {  
6.     return Scaffold(  
7.       body: Column(  
8.         children: [  
9.           ElevatedButton(  
10.             child: const Text('Option One'),  
11.             onPressed: () {  
12.               // TODO: Return "Option One"  
13.             },  
14.           ),  
15.           ElevatedButton(  
16.             child: const Text('Option Two'),  
17.             onPressed: () {  
18.               // TODO: Return "Option Two"  
19.             },  
20.           ),  
21.         ],  
22.       ),  
23.     );  
}
```

```
24.    }
```

```
25. }
```

Now that we have all the boilerplate code in place, we need to write the `onPressed` methods. Let us look at the buttons on the `ChooseOneScreen` first. For those, we will need to pop the current route off the stack, then return the value to the initial screen. Previously we used `Navigator.pop(context)` to route back to the previous screen without returning a value. We will do the same thing here, the only difference will be that we are adding another argument to the method, being the value that we would like to return. Update each of the `onPressed` methods to pop the route with an appropriate return value. In this case, we know that we expect a String to be returned, so let us return exactly that. Here is an example for the first button:

```
1. onPressed: () {
```

```
2.   Navigator.pop(context, 'Option One');
```

```
3. },
```

Next, we need to route to the screen from the initial one and wait for the value to be returned. Back on the initial screen, let us start by creating an `onPressed` method that navigates to our new screen:

```
1. onPressed: () {
```

```
2.   Navigator.push(
```

```
3.     context,
```

```
4.     MaterialPageRoute(
```

```
5.       builder: (context) => const ChooseOneScreen(),
```

```
6.     ),
```

```
7.   );
```

```
8. },
```

This will let us route to the screen, but it lacks a way to receive data when we pop the route. As mentioned before, we want to `await` the result and the

first step of that is turning our `onPressed` method into an `async` method. Then, we can `await` the `Navigator` call:

```
1. onPressed: () async {  
2.     await Navigator.push(  
3.         context,  
4.         MaterialPageRoute(  
5.             builder: (context) => const ChooseOneScreen(),  
6.         ),  
7.     );  
8. },
```

Finally, by assigning the result of the `Navigator` call to a variable, we can successfully retrieve the data which was returned from the popped route:

```
1. onPressed: () async {  
2.     String? _chosenOption = await Navigator.push(  
3.         context,  
4.         MaterialPageRoute(  
5.             builder: (context) => const ChooseOneScreen(),  
6.         ),  
7.     );  
8.     print('The chosen option is $_chosenOption');  
9. },
```

If this is combined with a stateful widget and `setState`, you could capture the value and display it on the screen. Alternatively, you could capture the returned value and pass it into a method to parse and perform an action upon. Truly, the options are limitless.

[Animating Widget from One Route to Another](#)

There is an enormous difference between creating a fully functioning application and an application that is well-polished. The former assumes you have met the bare minimum in terms of the app meeting the functional goals, while the latter implies that you have taken the time to think about the look and feel of the application. A well-polished application does not just include a well-coordinated color scheme and widgets that look and feel like they belong together and support each other. A well-polished application uses *animations* to unify the experience for the user.

Animations are an integral design element that serves to smooth out state transitions and focus a user's attention on the right place at the right time. They can be subtle or grand but serve the same purpose. When you `push` or `pop` a route, Flutter creates a subtle animation to indicate that you have changed screens. Still, the content from one screen to the next, although it may be related, appears to disappear from one and re-appear on the other.

Let us build a couple of screens to play with:

```
1. class Home extends StatelessWidget {  
2.   const Home({super.key});  
3.  
4.   @override  
5.   Widget build(BuildContext context) {  
6.     return Scaffold(  
7.       body: Column(  
8.         children: [  
9.           const Icon(Icons.flutter_dash),  
10.          ElevatedButton(  
11.            child: const Text('Say "hi" to Dash!'),  
12.            onPressed: () {  
13.              Navigator.push(  
14.                context,
```

```
15.         MaterialPageRoute(          MaterialPageRoute(  
16.             builder: (context) => const HelloDash(),  
17.         ),  
18.     );  
19.     },  
20.   ),  
21.   ],  
22. ),  
23. );  
24. }  
25. }  
26.  
27. class HelloDash extends StatelessWidget {  
28.   const HelloDash({super.key});  
29.  
30.   @override  
31.   Widget build(BuildContext context) {  
32.     return Scaffold(  
33.       body: Center(  
34.         child: Column(  
35.           mainAxisSize: MainAxisSize.center,  
36.           children: [  
37.             const Icon(  
38.               Icons.flutter_dash,  
39.               color: Colors.blue,
```

```
40.           size: 128,  
41.           ),  
42.           const Text('Hi Dash!'),  
43.           ElevatedButton(  
44.             child: const Text('Back'),  
45.             onPressed: () {  
46.               Navigator.pop(context);  
47.             },  
48.           ),  
49.           ],  
50.         ),  
51.       ),  
52.     );  
53.   }  
54. }
```

In this example, we have an icon of Dash with a button that routes to a new screen. The new screen has a larger, more colorful version of the same icon. There is a subtle animation when the route transitions, but otherwise, Dash appears to teleport from one place to another. How do we know this is the *same* Dash that was on the previous screen?

Let us fix this by making Dash *fly!* Wrap the **Icon** widget on both screens with a **Hero** widget and give them both a **tag** property with the same value:

```
1. const Hero(  
2.   tag: 'dash',  
3.   child: ...,  
4. ),
```

Now, tapping the button makes Dash *fly* from one screen to the next! Choosing one or two elements to transition between routes as a **Hero** will help the user keep track of how elements on the two screens relate, adding polish to the finished product.

The **Hero** widget is one of *many* widgets which can be used to animate elements in Flutter. Although animating widgets in Flutter is not covered in this book, there are loads of excellent resources online which cover many aspects of animating widgets, including creating your own animations. Check the Further Reading section at the end of this chapter for a link to the documentation.

Introduction to Navigator 2.0 — the Router API

What Navigator 1.0 lacks in ability, it makes up for with simplicity. Unfortunately, the simplicity of Navigator 1.0 can be quite limiting at times. It is not hard to imagine a scenario wherein one would want to implement a custom screen transition animation, use deep linking (routing within the application based on a URI, such as `/contact/0` telling the application to show us the first contact), or perhaps needing to use *nested* routing. More elaborate applications are usually better served by Navigator 2.0.

When Navigator 2.0 was first introduced, it was named as such. This is still the name by which many people know it and refer to it as. However, the more appropriate name that has since been adopted by the Flutter team is the *Router API*.

The Router API is not a breaking change. Rather, it adds a new set of APIs for us to leverage and build more complex applications with far more freedom. With that freedom, however, comes a lot more complexity. Let us first start by looking at a diagram of how the Router API works (*figure 8.1*):

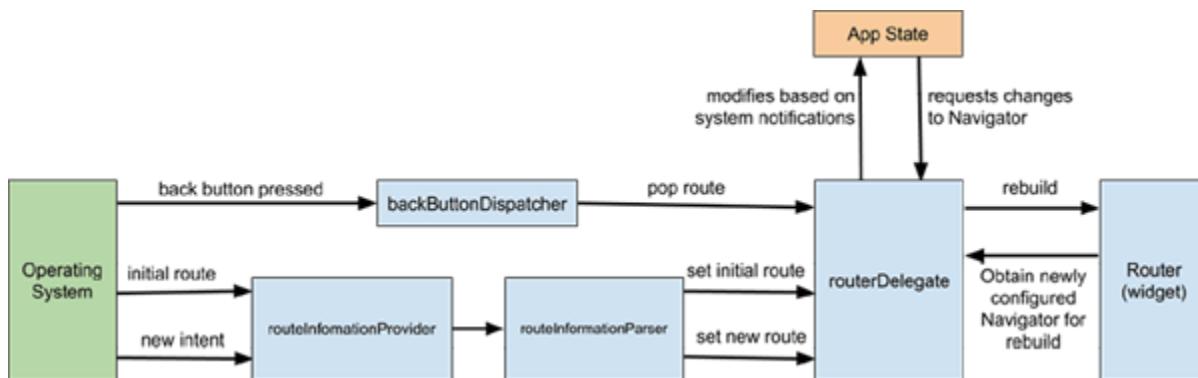


Figure 8.1: A flow-chart of the functionality of the Router API. Source:
<https://medium.com/flutter/learning-flutters-new-navigation-and-routing-system-7c9068155ade>

We are going to be implementing each of these blocks to create a complete Router API-enabled application. Using our earlier examples, we are going to create a simple application that lists several contacts and routes to a contact details screen when we tap on one. We will target Flutter Web, just so we can observe the URI changing (although in true Flutter form, the application works anywhere.) We will implement back button handling, so we can use a system back button (for example, Android's back button) and will customize the screen transition. We will even add a custom 404 page just in case the user tries to enter an invalid path.

To get started, we will need to set up a `Contact` class, build the base application screen, create a screen that lists the contacts, a contact details screen, and a 404 screen. Since we haven't set up any routing yet, we will have no way to navigate between these screens, but once we are done, it will be nice to have gotten the boilerplate code out of the way.

Without further ado, here is the boilerplate code for our application:

```
1. import 'package:flutter/material.dart';
2.
3. void main() {
4.   runApp(const ContactsApp());
5. }
6.
7. class ContactsApp extends StatefulWidget {
8.   const ContactsApp({super.key});
9.
10.  @override
11.  State<StatefulWidget> createState() => _ContactsAppState();
12. }
13.
```

```
14. class _ContactsAppState extends State<ContactsApp> {  
15.       
16.     @override  
17.     Widget build(BuildContext context) {  
18.         return MaterialApp(  
19.             title: 'Contacts App',  
20.             home: Container(),  
21.         );  
22.     }  
23. }  
24.  
25.   
26. class Contact {  
27.     final String name;  
28.     final String email;  
29.       
30.     const Contact({  
31.         required this.name,  
32.         required this.email,  
33.     });  
34. }  
35.   
36. class ContactsListScreen extends StatelessWidget {  
37.     final List<Contact> contacts;  
38.     final ValueChanged<Contact> onTapped;  
39. }
```

```
40. const ContactsListScreen({  
41.     super.key,  
42.     required this.contacts,  
43.     required this.onTapped,  
44. );  
45.  
46. @override  
47. Widget build(BuildContext context) {  
48.     return Scaffold(  
49.         body: ListView(  
50.             children: [  
51.                 for (final Contact contact in contacts)  
52.                     ListTile(  
53.                         title: Text(contact.name),  
54.                         subtitle: Text(contact.email),  
55.                         onTap: () => onTapped(contact),  
56.                     ),  
57.                 ],  
58.             ),  
59.         );  
60.     }  
61. }  
62.  
63. class ContactDetailsScreen extends StatelessWidget {  
64.     final Contact contact;
```

```
65.
66. const ContactDetailsScreen({
67.   super.key,
68.   required this.contact,
69. });
70.
71. @override
72. Widget build(BuildContext context) {
73.   return Scaffold(
74.     body: SafeArea(
75.       child: Padding(
76.         padding: const EdgeInsets.all(8.0),
77.         child: Column(
78.           crossAxisAlignment: CrossAxisAlignment.start,
79.           children: [
80.             Text(
81.               contact.name,
82.               style: Theme.of(context).textTheme.headline6,
83.             ),
84.             Text(
85.               contact.email,
86.               style: Theme.of(context).textTheme.subtitle1,
87.             ),
88.           ],
89.         ),
```

```
90.          ),
91.          ),
92.      );
93.  }
94. }
```

The only thing that might be new in this code is the `ValueChanged` widget. When combined with the `notifyListeners()` method, we can trigger widget rebuilds when values are changed.

Now that we have our boilerplate code in place, we can start thinking about the different pieces that make up the Router API. At a minimum, we will need a `RouterDelegate` and a `RouteInformationParser`, but what are these?

The `RouterDelegate` defines the behavior of how the application's `Router` learns about and responds to changes in the routing state. It receives the initial route intents and maintains the routing state, then uses this information to build the `Navigator` with the current list of `Pages` in the navigation stack.

This just leaves more questions, though. Like, what is a `Page` and what is a `Router`? A `Page` is used to set the `Navigator`'s history stack and a `Router` configures the list of `Pages` for the `Navigator`.

Still not clear? Let us try to break it down. An application will have a `Navigator` (managed by a `Router` widget), which is given a list of `Pages`. These `Page` objects return a `PageRouteBuilder` object from their `createRoute()` method. The `PageRouteBuilder` has both a `settings` parameter and a `pageBuilder` parameter. The `settings` parameter accepts a value of a nullable `RouteSettings` object and the `pageBuilder` returns the widget we will eventually show the user. The `RouteInformationParser` is used to parse the raw URI into an object that `RouterDelegate` wants to work with. It, along with the `RouterDelegate` are both passed into the `Router`. The `RouterDelegate` is the object that sets up the current configuration of the `Router`.

Yeah, the Router API is complex.

Let us just start building out these pieces and see how they work as we go. First, let us create our `RouterDelegate` and `RouteInformationParser` objects. We will

also need to create a `RoutePath`, which is a representation of the raw URI that the `RouterDelegate` will work with, so let us start there:

```
1. class ContactRoutePath {  
2.     final int? id;  
3.     final bool isUnknown;  
4.  
5.     ContactRoutePath.details(this.id) : isUnknown = false;  
6.  
7.     ContactRoutePath.home()  
8.         : id = null,  
9.             isUnknown = false;  
10.  
11.    ContactRoutePath.unknown()  
12.        : id = null,  
13.            isUnknown = true;  
14.  
15.    bool get isDetailsPage => id != null;  
16.  
17.    bool get isHomePage => id == null;  
18. }
```

This class takes in a nullable integer `id`. We will call this class using a constructor later, which could mean calling `ContactRoutePath.home()` or `ContactRoutePath.details(id)`. This will make more sense once we put it to use in our `RouterDelegate`, so let us build that next.

```
1. class ContactRouterDelegate extends  
   RouterDelegate<ContactRoutePath>
```

```
2.      with ChangeNotifier,
PopNavigatorRouterDelegateMixin<ContactRoutePath> {
3.      static const List<Contact> contacts = [
4.          Contact(name: 'Alice', email: 'alice@flutter.dev'),
5.          Contact(name: 'Bob', email: 'bob@flutter.dev'),
6.          Contact(name: 'Charlie', email: 'charlie@flutter.dev'),
7.      ];
8.
9.      @override
10.     final GlobalKey<NavigatorState> navigatorKey;
11.     bool show404 = false;
12.     Contact? _selectedContact;
13.
14.     ContactRouterDelegate() : navigatorKey =
GlobalKey<NavigatorState>();
15.
16.     @override
17.     ContactRoutePath get currentConfiguration {
18.         if (show404) {
19.             return ContactRoutePath.unknown();
20.         }
21.         return _selectedContact == null
22.             ? ContactRoutePath.home()
23.             :
24.             ContactRoutePath.details(contacts.indexOf(_selectedContact!));
25.     }
26. }
```

```
25. 
26. @override
27. Widget build(BuildContext context) {
28.   return Navigator(
29.     key: navigatorKey,
30.     pages: [
31.       MaterialPageRoute(
32.         key: const ValueKey('ContactsListPage'),
33.         child: ContactsListScreen(
34.           contacts: contacts,
35.           onTapped: _handleContactTapped,
36.         ),
37.       ),
38.       if (_selectedContact != null)
39.         ContactDetailsPage(contact: _selectedContact!),
40.       if (show404)
41.         const MaterialPage(
42.           key: ValueKey('UnknownPage'),
43.           child: UnknownScreen(),
44.         ),
45.     ],
46.     onPopPage: (route, result) {
47.       if (!route.didPop(result)) {
48.         return false;
```

```
49.     }
50.
51.     _selectedContact = null;
52.     show404 = false;
53.     notifyListeners();
54.
55.     return true;
56.   },
57. );
58. }
59.
60. @override
61. Future<void> setNewRoutePath(ContactRoutePath configuration)
  async {
62.   if (configuration.isUnknown) {
63.     _selectedContact = null;
64.     show404 = true;
65.     return;
66.   }
67.
68.   if (!configuration.isDetailsPage) {
69.     _selectedContact = null;
70.   }
71.   if (configuration.id == null) {
72.     _selectedContact = null;
```

```

73.         show404 = false;
74.         return;
75.     }
76.     if (configuration.id! < 0 || configuration.id! > contacts.length -
1) {
77.         show404 = true;
78.         return;
79.     }
80.
81.     _selectedContact = contacts[configuration.id!];
82.
83.     show404 = false;
84. }
85.
86. void _handleContactTapped(Contact contact) {
87.     _selectedContact = contact;
88.     notifyListeners();
89. }
90. }
```

This is where our **RoutePath** comes into play. (This is also where we specify our list of contacts — although, in a production application, these would be stored somewhere else. For the sake of simplicity, we are just going to specify a static list of contacts, though.)

The **RouterDelegate** is what returns the **Navigator** widget, which we discussed earlier. There are two flows that the **RouterDelegate** controls: from *platform to user* and from *user to platform*. When a URI (as **RouteInformation**) comes (through **RouteInformationProvider**) to our app and **RouteInformationParser** parses it into our **ContactRoutePath**, then the **setNewRoutePath()** is called where **RouterDelegate** configures the *routing state* (**show404**, **_selectedContact**, and so on)

to match the incoming route information and build the `Navigator`. Conversely, when we navigate within the app (for example, with `_handleContactTapped`), the routing state is first updated and `notifyListeners` is called. This will trigger both a rebuild of `Navigator` and cause the platform to fetch the `currentConfiguration` (for example, causing the browser to update the URI). Here, the `RouteInformationParser` is again used to restore (parse back) the current `ContactRoutePath` into a `RouteInformation` that the platform can understand.

Now, let us implement the `RouteInformationParser`, which is the object that converts the raw URI back and forth into something that the `RouterDelegate` understands:

```
1. class ContactRouteInformationParser
2.     extends RouteInformationParser<ContactRoutePath> {
3.     @override
4.     Future<ContactRoutePath> parseRouteInformation(
5.         RouteInformation routeInformation,
6.     ) async {
7.         final Uri uri = Uri.parse(routeInformation.location!);
8.         // Handle '/'
9.         if (uri.pathSegments.isEmpty) {
10.             return ContactRoutePath.home();
11.         }
12.
13.         // Handle '/contact/:id'
14.         if (uri.pathSegments.length == 2) {
15.             if (uri.pathSegments[0] != 'contact') return
16.             ContactRoutePath.unknown();
17.             String remaining = uri.pathSegments[1];
18.             int? id = int.tryParse(remaining);
```

```

18.         if (id == null) return ContactRoutePath.unknown();
19.         return ContactRoutePath.details(id);
20.     }
21.
22.     // Handle unknown routes
23.     return ContactRoutePath.unknown();
24.   }
25.
26.   @override
27.   RouteInformation? restoreRouteInformation(ContactRoutePath
28.   configuration) {
29.     if (configuration.isUnknown) {
30.       return const RouteInformation(location: '/404');
31.     }
32.     if (configuration.isHomePage) {
33.       return const RouteInformation(location: '/');
34.     }
35.     if (configuration.isDetailsPage) {
36.       return
37.         RouteInformation(location:
38.           '/contact/${configuration.id}');
39.     }

```

The **parseRouteInformation** method is what does the heavy lifting of pulling the parameters out of the URI, whereas the **restoreRouteInformation** method is what returns the **RouteInformation** that is used to set the new URI.

That is a *ton* of boilerplate code, but all of it is necessary! Let us update our application to use it:

```
1. class _ContactsAppState extends State<ContactsApp> {  
2.     final ContactRouterDelegate _routerDelegate =  
    ContactRouterDelegate();  
3.     final ContactRouteInformationParser _routeInformationParser =  
    ContactRouteInformationParser();  
5.  
6.     @override  
7.     Widget build(BuildContext context) {  
8.         return MaterialApp.router(  
9.             title: 'Contacts App',  
10.            routerDelegate: _routerDelegate,  
11.            routeInformationParser: _routeInformationParser,  
12.        );  
13.    }  
14. }
```

At this point, everything should work as expected — the list of contacts, tapping to view the contact details, the URI updating, the back button — everything. There is one more tweak we can make to this. Let us change the page transition:

```
1. class ContactDetailsPage extends Page {  
2.     final Contact contact;  
3.  
4.     ContactDetailsPage({  
5.         required this.contact,  
6.     }) : super(key: ValueKey(contact));
```

```
7. 
8.     @override
9.     Route createRoute(BuildContext context) {
10.        return PageRouteBuilder(
11.            settings: this,
12.            pageBuilder: (context, animation, animation2) {
13.                final tween = Tween(
14.                    begin: const Offset(1.0, 0.0),
15.                    end: Offset.zero,
16.                );
17.                final curveTween = CurveTween(
18.                    curve: Curves.easeInOut,
19.                );
20.                return SlideTransition(
21.                    position: animation.drive(curveTween).drive(tween),
22.                    child: ContactDetailsScreen(
23.                        key: ValueKey(contact),
24.                        contact: contact,
25.                    ),
26.                );
27.            },
28.        );
29.    }
30. }
```

Here, the `pageBuilder` ends up building a `SlideTransition` animation instead of the default animation. What other animations or transitions could you use instead of a `SlideTransition`?

The Router API is, as we have seen, significantly more complex than Navigator 1.0. Although we moved quickly through it and did not examine every line of code (which could have been its own book!), we do have a working example from which we can build upon later and study further.

Using Packages to Help with Routing

Expectedly, a lot of packages emerged that try to simplify the usage of the Router API. Let us see an implementation of the same app using one of them — Beamer, which was created by Sandro Lovnički ([@slovnicki](#)). As we will see, a lot of things we needed to implement by hand before are abstracted out for us in Beamer. In fact, all we need to define are the routes we want to handle and, optionally, how we want to handle them. Beamer also has an advanced API that we could use, but that is a topic for another day.

Here is the same application we just wrote using the Router API, this time written using Beamer as a basis:

```
1. import 'package:beamer/beamer.dart';
2. import 'package:flutter/material.dart';
3.
4. void main() {
5.   runApp(const ContactsApp());
6. }
7.
8. class Contact {
9.   final String name;
10.  final String email;
11.
12.  const Contact({
```

```
13.     required this.name,
14.     required this.email,
15.   });
16. }
17.
18. class ContactDetailsScreen extends StatelessWidget {
19.   final Contact contact;
20.
21.   const ContactDetailsScreen({super.key, required this.contact});
22.
23.   @override
24.   Widget build(BuildContext context) {
25.     return Scaffold(
26.       body: SafeArea(
27.         child: Padding(
28.           padding: const EdgeInsets.all(8.0),
29.           child: Column(
30.             mainAxisAlignment: MainAxisAlignment.start,
31.             children: [
32.               Text(
33.                 contact.name,
34.                 style: Theme.of(context).textTheme.headline6,
35.               ),
36.               Text(
37.                 contact.email,
38.                 style: Theme.of(context).textTheme.subtitle1,
```

```
39.           ),
40.           ],
41.           ),
42.           ),
43.           ),
44.       );
45.   }
46. }
47.
48. class ContactsApp extends StatefulWidget {
49.     const ContactsApp({super.key});
50.
51.     @override
52.     State<StatefulWidget> createState() => _ContactsAppState();
53. }
54.
55. class ContactsListScreen extends StatelessWidget {
56.     final List<Contact> contacts;
57.     final ValueChanged<Contact> onTapped;
58.
59.     const ContactsListScreen(
60.         super.key,
61.         required this.contacts,
62.         required this.onTapped,
63.     );
64.
```

```
65. @override
66. Widget build(BuildContext context) {
67.   return Scaffold(
68.     body: ListView(
69.       children: [
70.         for (final Contact contact in contacts)
71.           ListTile(
72.             title: Text(contact.name),
73.             subtitle: Text(contact.email),
74.             onTap: () => onTapped(contact),
75.           ),
76.         ],
77.       ),
78.     );
79.   }
80. }
81.
82. class UnknownScreen extends StatelessWidget {
83.   const UnknownScreen({super.key});
84.
85.   @override
86.   Widget build(BuildContext context) {
87.     return const Scaffold(
88.       body: Center(
89.         child: Text('Oops.'),
90.       ),

```

```
91.     );
92.   }
93. }
94.
95. class _ContactsAppState extends State<ContactsApp> {
96.   static const List<Contact> contacts = [
97.     Contact(name: 'Alice', email: 'alice@flutter.dev'),
98.     Contact(name: 'Bob', email: 'bob@flutter.dev'),
99.     Contact(name: 'Charlie', email: 'charlie@flutter.dev'),
100.   ];
101.
102.   final _routerDelegate = BeamerDelegate(
103.     setBrowserTabTitle: false,
104.     notFoundRedirectNamed: '/404',
105.     locationBuilder: RoutesLocationBuilder(
106.       routes: {
107.         '/': (context, _, __) => ContactsListScreen(
108.           contacts: contacts,
109.           onTapped: (contact) => context.beamToNamed(
110.             '/contact/${contacts.indexOf(contact)}',
111.             ),
112.             ),
113.             '/contact/:id': (_, state, __) {
114.               final id = int.parse(state.pathParameters['id']!);
115.               return BeamPage(
```

```
116.           key: ValueKey('contact-$id'),
117.           type: BeamPageType.slideRightTransition,
118.           popToNamed: '/',
119.           child: ContactDetailsScreen(contact: contacts[id]),
120.         );
121.       },
122.       '/404': (_, __, ___) => const UnknownScreen(),
123.     },
124.   ),
125.   guards: [
126.     BeamGuard(
127.       pathPatterns: ['/contact/*'],
128.       check: (_, beamLocation) {
129.         final beamState = beamLocation.state as BeamState;
130.         final idParameter = beamState.pathParameters['id'];
131.         if (idParameter != null) {
132.           final id = int.tryParse(idParameter);
133.           if (id == null || id < 0 || id >= contacts.length) {
134.             return false;
135.           }
136.         }
137.         return true;
138.       },
139.       beamToNamed: (_, __) => '/404',
```

```
140.          )
141.      ],
142.    );
143.    final _routeInformationParser = BeamerParser();
144.
145.    @override
146.    Widget build(BuildContext context) {
147.      return MaterialApp.router(
148.        title: 'Contacts App',
149.        routerDelegate: _routerDelegate,
150.        routeInformationParser: _routeInformationParser,
151.      );
152.    }
153. }
```

As you can see, the same application using Beamer simultaneously harnesses the power of the Router API while delivering something nearing the simplicity of Navigator 1.0.

Conclusion

Navigation and routing are ubiquitous in just about any application you will end up building. Having a firm understanding of how these systems work in Flutter is key to building any application which has more than a single screen (which will be most, if not all, of them!)

In the upcoming chapter, we will be looking at state management and the BLoC pattern (a commonly used state management system.) These systems will open a lot of new possibilities for us to build dynamic and complex experiences. So, refill your water, do some stretches, and when you are ready, let us dive into state management!

Questions

1. The concept of sending a user from one screen to another is known as what?
2. What is the difference between navigation and routing?
3. When navigating between screens in Flutter, what analogy is useful in visualizing how different screens and their states are maintained?
4. What is the difference between imperative and declarative routing?
5. What widget is used to route imperatively in Flutter? What widget is used to route declaratively?
6. What are some of the limitations of Navigator 1.0?
7. How do you configure named routes, then navigate to a route by name?
8. What's the difference between `Navigator.push` and `Navigator.pushReplacement`?
9. What happens if you use `Navigator.pushReplacement` followed by `Navigator.pop` on the new screen?
10. How do you pass data to a new route? How do you get a value back when calling `Navigator.pop`?
11. What useful things can you use `onGenerateRoute` for?
12. What widget can be used to animate a widget from one screen to another?
13. What is the correct name for Navigator 2.0?
14. How are the `ValueChanged` widget and `notifyListeners()` method used?
15. What is a `RouterDelegate`?
16. How do you change the transition animation when navigating to a new route using the Router API?

Key Terms

- **Declarative programming:** Code that provides a description of the expected outcome but not instructions to accomplish it.
- **Imperative programming:** Code that has step-by-step instructions to complete the task at hand.
- **Intent:** The expected outcome of a user's action.

Further Reading

- Animations in Flutter:
<https://docs.flutter.dev/development/ui/animations>
- All about Beamer: <https://beamer.dev/> and <https://pub.dev/packages/beamer>
- Learning Navigator 2.0 (the Router API):
<https://medium.com/flutter/learning-flutters-new-navigation-and-routing-system-7c9068155ade>

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](https://discord(bpbonline.com)



Chapter 9

State Management and the BLoC

Introduction

Every application has some sort of *state*. Whether that is the state in one of your stateful widgets or whether the user is logged in, there is *state* everywhere. Understanding what *state* is and how to manage it is essential for any application.

In this chapter, we are going to explore several approaches to state management in Flutter applications. We will look at just a couple of the more popular approaches, and even build an app with them so we can compare their differences.

Structure

In this chapter, we will discuss the following topics:

- Ephemeral versus app state
- Some prep work before we begin
- A brief aside to explain some new concepts
- Using inherited widgets to manage app state
- Using Provider to manage app state

- Using BLoC's Cubit to manage app state
- Using BLoC to manage app state

Objectives

After completing this chapter, you will have a firm understanding of the difference between the state of a widget and the state of an entire application. You will learn how to use Flutter's built-in widgets to manage app state before exploring several popular packages, which make working with app state much easier. Finally, you will build a simple application to use as a tool to compare the different approaches to state management.

Ephemeral Versus App State

When we are talking about state, there are two different kinds of state: ephemeral and app state. You are already familiar with the first, although you may not know it by name. This is the sort of state that is used to hold the necessary information to build widgets. Things like the current index of a `BottomNavigationBar`, the connection state of an API call, and whether a checkbox is checked are all examples of ephemeral state. These are states managed by a stateful widget.

Remember, though, that widgets are immutable. Once a widget is built, it cannot be changed. Instead, it is tossed away and rebuilt when its associated `State` object is updated. When the widget is disposed of, so too do we lose any associated `State` object. This is what we mean by ephemeral state.

App state, on the other hand, is state that is consumed throughout the application. If you have an app that allows a user to log in, the app state will hold the login state. If the user logs out, the login state would be updated and widgets throughout the application will be rebuilt (or disposed of) as necessary. The bulk of the widgets may get disposed of, and the user would be routed to the login screen, perhaps.

While you could create a stateful widget at the top of your widget tree that holds the state, you would need to pass that state down to every child widget, which needs to access it (or one of its children does). This is, generally, a horrible idea. Do not do this.

Well, what about inherited widgets, then? Is not the idea behind an inherited widget that you can access through the `BuildContext` from anywhere below it in the widget tree? Yep! That is right — and it is inherited widgets which we will use to manage the app state, if not directly, by utilizing packages that are wrappers around inherited widgets.

If you need a refresher on inherited widgets, now would be an excellent time to go back and review them in *Chapter 4: Introduction to Widgets*. Once you have brushed up on inherited widgets, let us look at how to use them to manage app state.

Some Prep Work Before We Begin

We are going to be building a very simple shopping cart in the examples in this chapter. This is not a fully featured shopping cart, but it will be enough to help us illustrate the concepts. Our shopping cart will have a list of items that are for sale and an `add item to cart` button. We will also have a shopping cart that contains the items we have added to the cart. Each item in the cart will have a multiplier next to it, so if you add several of a single item, the multiplier will increase. There will be a count of all the items in the cart, alongside the total cost of everything in the cart, plus a `clear cart` button.

Let us start by defining what an `Item` is. It should, at a minimum, have a cost and a name. You could spend more time adding a product image or other properties, but that is outside the scope of this chapter:

```
1. class Item {  
2.   final double cost;  
3.   final String name;  
4.  
5.   const Item({  
6.     required this.cost,  
7.     required this.name,  
8.   });  
9. }
```

Now that we have defined what an **Item** is in our store, we can use this class to build up an inventory:

```
1. const List<Item> inventory = [  
2.   Item(name: "Shirt", cost: 12.99),  
3.   Item(name: "Shoes", cost: 120.00),  
4.   Item(name: "Pants", cost: 63.74),  
5.   Item(name: "Hat", cost: 8.23),  
6. ];
```

For our main app, we are just going to create a **ListView** that has the inventory on top and the shopping cart on the bottom. We will make it a stateful widget, which we will come back to later:

```
1. class MyApp extends StatefulWidget {  
2.   const MyApp({super.key});  
3.   @override  
4.   State<MyApp> createState() => _MyAppState();  
5. }  
6.  
7.  
8. class _MyAppState extends State<MyApp> {  
9.   @override  
10.  Widget build(BuildContext context) {  
11.    return MaterialApp(  
12.      home: Scaffold(  
13.        body: Padding(  
14.          padding: const EdgeInsets.all(8.0),  
15.          child: ListView(
```

```
16.           children: [
17.             Inventory(),
18.             ShoppingCart(),
19.           ],
20.         ),
21.       ),
22.     ),
23.   );
24. }
25. }
```

Let us first look at the shopping cart. Both the shopping cart and inventory widgets will have a couple of nested `ListView`s, which we will need to be careful not to have scroll independently since we are wrapping the whole UI in its own `ListView` that will handle scrolling. For this reason, we apply the `ClampingScrollPhysics` class to the `ListView`s' to prevent them from scrolling. The net result is akin to using a `Column`, but with the performance benefits of using the `ListView`.

Here is the shopping cart:

```
1. class ShoppingCart extends StatelessWidget {
2.   const ShoppingCart({super.key});
3.
4.   @override
5.   Widget build(BuildContext context) {
6.     return ListView(
7.       shrinkWrap: true,
8.       physics: const ClampingScrollPhysics(),
9.       children: [
```

```
10.    Text(  
11.        'Shopping Cart',  
12.        style: Theme.of(context).textTheme.displayMedium,  
13.    ),  
14.    Row(  
15.        mainAxisAlignment:  
16.        MainAxisAlignment.spaceEvenly,  
16.        children: [  
17.            // TODO: Display the total number of items in the  
// cart  
18.            Text('Items in Cart: '),
19.            // TODO: Display the total cost of all items in the  
// cart
20.            Text('Total Cost: '),
21.            OutlinedButton.icon(  
22.                label: const Text('Clear cart'),  
23.                icon: const Icon(Icons.remove_shopping_cart),  
24.                onPressed: () {  
25.                    // TODO: Add a “clear cart” handler  
26.                },  
27.            ),  
28.        ],
29.    ),
30.    ListView.builder(  
31.        shrinkWrap: true,
```

```
32.           physics: const ClampingScrollPhysics(),  
33.           // TODO: Figure out how many unique items are in  
           // our cart  
34.           itemCount: 0,  
35.           itemBuilder: (context, i) => Card(  
36.             child: ListTile(  
37.               // TODO: Figure out how many of a specific item  
               // are in our cart  
38.               leading: Text("),  
39.               // TODO: Display the item's name  
40.               title: Text("),  
41.               // TODO: Display the item's price  
42.               subtitle: Text("),  
43.             ),  
44.           ),  
45.           ),  
46.         ],  
47.       );  
48.     }  
49. }
```

And here is the inventory:

```
1. class Inventory extends StatelessWidget {  
2.   const Inventory({super.key});  
3.  
4.   @override
```

```
5.     Widget build(BuildContext context) {  
6.         return ListView(  
7.             shrinkWrap: true,  
8.             physics: const ClampingScrollPhysics(),  
9.             children: [  
10.                 Text(  
11.                     'Inventory',  
12.                     style: Theme.of(context).textTheme.displayMedium,  
13.                 ),  
14.                 ListView.builder(  
15.                     shrinkWrap: true,  
16.                     physics: const ClampingScrollPhysics(),  
17.                     itemCount: inventory.length,  
18.                     itemBuilder: (BuildContext context, int i) => Card(  
19.                         child: ListTile(  
20.                             title: Text(inventory[i].name),  
21.                             subtitle:  
22.                             Text('$$ ${inventory[i].cost.toStringAsFixed(2)}'),  
23.                             trailing: IconButton(  
24.                                 icon: const Icon(Icons.add_shopping_cart),  
25.                                 onPressed: () {  
26.                                     // TODO: Add the item to the shopping cart.  
27.                                 },  
28.                             ),  
29.                         ),  
30.                     ),  
31.                 ),  
32.             ),  
33.         ),  
34.     );  
35. }
```

```
29.     ),
30.     ),
31.   ],
32. );
33. }
34. }
```

Before we move on to fixing all the TODOs in our code, let us look at a couple of new concepts that were just introduced.

A Brief Aside to Explain Some New Concepts

The two things here that may be new or unfamiliar are the `=>` operator and the `\${inventory...}` in the `Text` widget. Let us quickly go over these so you know what they mean and how to use them.

The `=>` operator is shorthand for *return this thing*. Take the following method, which returns a string:

```
1. String sayHello() {
2.   return "Hello";
3. }
```

Since this method only returns a value and does not have any complex logic, we can simplify it using the `=>` operator, like the following:

```
1. String sayHello() => "Hello";
```

And that is the gist of it. If your method needs to do some processing before returning the value, you will want to use the `return` style, otherwise you should be able to save a few keystrokes with the `=>` operator.

Now, let us investigate the `\${inventory...}` in our `Text` widget. There are two things happening here that we need to discuss: escaping and interpolation. Let us talk about escaping, first.

There are some characters that, when inserted into a string, are treated as special characters. These include, but are not limited to, the backslash (\), single and double quotes (' and ") and the dollar sign (\$). However, there are caveats and conditions as to when these are treated as special characters.

For example, the string "it's a wonderful life" has no problems whatsoever, even though there is a single quote in the middle of it. However, if we try to use single quotes around our string, such as with 'it's a wonderful life', we will run into a problem. Can you see it? The problem is that we contain our string within quotes — in this case, single quotes — but we have inserted a single quote in the middle. Effectively, we have told Dart that our string is simply 'it', and everything that comes after is going to cause Dart to complain. To get around this, we can use the backslash to escape the single quote, thereby telling Dart that we really do want a single quote in our string, like the following: 'it\'s a wonderful life'. That is escaping, in a nutshell.

Now, let us look at interpolation, then see why we have an escaped dollar sign. Interpolation is denoted by the following inside a string: \${}. Anything you put between the braces will be executed by Dart and the output will be inserted into the string as plain text. In our example, we are getting the price of an item, then running the method `.toStringAsFixed(2)` on it, which will ensure the cost always has two digits after the decimal, regardless of how the cost was originally set.

If you simply wanted the value of a variable to be inserted into your string, you could prefix the variable with a dollar sign: "I would like to say \$hello." is an example of this in action. If your hello variable is a class with properties, you *would* need to use interpolation: "I would like to say \${hello.toYou}". Since the dollar sign is used to denote that the characters which follow are to be treated as a variable and/or an interpolated string, inserting that dollar sign into our string to display a cost in dollars requires us to escape the dollar sign: '\\$100' because 100 is not a variable, but a string of text.

The final question remains: why do we need to interpolate in the first place? The answer is: because we are using the `=>` operator, we cannot assign a variable for the price string and simply use `$price` in our string.

Using Inherited Widgets to Manage App State

In *Chapter 4: Introduction to Widgets*, we learned about inherited widgets and that they can hold a value that we can then find anywhere below in the widget tree by referencing the current context. If we create an inherited widget with a list of items in the cart, we can reference that list anywhere by accessing the inherited widget using `BuildContext`. The trick is updating that list as we add items to and clear our cart. Otherwise, we are not breaking new ground here. In fact, the core of our inherited widget should look very familiar:

```
1. class ShoppingCartInheritedWidget extends InheritedWidget {  
2.   final List<Item> itemsInCart;  
3.   @override  
4.   const ShoppingCartInheritedWidget({  
5.     super.key,  
6.     required super.child,  
7.     required this.itemsInCart,  
8.   });  
9.   @override  
10.  bool updateShouldNotify(ShoppingCartInheritedWidget  
11.    oldWidget) {  
12.    return oldWidget.itemsInCart != itemsInCart;  
13.  }  
14.  
15.  static ShoppingCartInheritedWidget? of(BuildContext context) {  
16.    return context  
17.      .dependOnInheritedWidgetOfExactType<ShoppingCartI  
nheritedWidget>();  
18.  }
```

19. }

We pass in a list of items, and then we can access that list from anywhere. We are not quite done with our inherited widget, though. Right now, we can change our cart, but unless we specifically *ask* for those changes, we will never know that those changes happened. What we want to happen is: (1) we update the cart and (2) our `ShoppingCart` updates automatically. For this, we need to use a `ValueChanged`.

`ValueChanged` is not so much a widget, but a `typedef`. (We can specify our own types without creating a new class by using `typedef`. For example, we could say `typedef StringList = List<String>;` then use `StringList` in place of `List<String>`.) In fact, `ValueChanged` is defined as follows: `typedef ValueChanged<T> = void Function(T value)`. So, it is a function that takes in a value.

Let us quickly get that set up in our inherited widget. First, we need to add it as a property:

```
1. class ShoppingCartInheritedWidget extends InheritedWidget {  
2.   final List<Item> itemsInCart;  
3.   final ValueChanged<List<Item>> onListChanged;  
4.   @override  
5.   const ShoppingCartInheritedWidget({  
6.     super.key,  
7.     required super.child,  
8.     required this.itemsInCart,  
9.     required this.onListChanged,  
10.    });
```

Then we want to update the method to use it:

```
1.   @override  
2.   bool updateShouldNotify(ShoppingCartInheritedWidget  
oldWidget) {
```

```
3.     return oldWidget.itemsInCart != itemsInCart ||
4.         oldWidget.onListChanged != onListChanged;
5. }
```

That is all we need to do in our inherited widget, but we still need to create the method and pass it in. We will do that in the `_MyAppState`, which brings us back to why we are using a stateful widget and not a stateless widget.

Although we are using the inherited widget to access the state of our list anywhere in the application, it is the `MyApp` widget that truly holds the app state. This is where we will define a private variable to hold the items in our cart:

```
1. class _MyAppState extends State<MyApp> {
2.     List<Item> _itemsInCart = [];
3.
4.     @override
5.     Widget build(BuildContext context) {
```

And it is that variable that we will pass into our inherited widget. We also need to write our method to pass in. Let us do that now that we have our list set up:

```
1. class _MyAppState extends State<MyApp> {
2.     List<Item> _itemsInCart = [];
3.
4.     @override
5.     Widget build(BuildContext context) {
6.         ...
7.     }
8.
9.     void onCartUpdated(List<Item> items) => setState(() {
```

```
10.         _itemsInCart = items;  
11.     }));  
12. }
```

This method could not be any simpler: when a list of items is passed into the method, update the `_itemsInCart` variable to match. Next, we need to wrap our `ListView` with the inherited widget we have created and pass both the `_itemsInCart` variable as well as our `onCartUpdated` method into it:

```
1. @override  
2. Widget build(BuildContext context) {  
3.     return MaterialApp(  
4.         home: Scaffold(  
5.             body: Padding(  
6.                 padding: const EdgeInsets.all(8.0),  
7.                 child: ShoppingCartInheritedWidget(  
8.                     itemsInCart: _itemsInCart,  
9.                     onListChanged: (List<Item> items) =>  
10.                    child: ListView(  
11.                        children: [  
12.                            Inventory(),  
13.                            ShoppingCart(),  
14.                        ],  
15.                    ),  
16.                ),  
17.            ),  
18.        ),
```

```
19.    );
20. }
```

There is still something missing, though. How do we call the `onCartUpdated` method when we add an item to our cart or clear our cart? We are going to have to pass the method into our `Inventory` and `ShoppingCart` widgets and call them there:

```
1. child: ListView(
2.   children: [
3.     Inventory(onCartUpdated),
4.     ShoppingCart(onCartUpdated),
5.   ],
6. ),
```

You will also need to update each of the widgets to allow us to pass in a positional parameter. Here is the `ShoppingCart` as an example:

```
1. class ShoppingCart extends StatelessWidget {
2.   final Function(List<Item>) onCartUpdated;
3.
4.   const ShoppingCart(this.onCartUpdated, {super.key});
5. ...
6. }
```

Believe it or not, we are nearly finished. Let us focus next on the `Inventory` before moving on to the `ShoppingCart`. First, we need to figure out what is already in the cart so we can add more things to it. That is easy enough since we now have an inherited widget:

```
1. @override
2. Widget build(BuildContext context) {
3.   final List<Item> currentCart =
```

```
4.     ShoppingCartInheritedWidget.of(context)!.itemsInCart;  
5.     ...  
6. }
```

Now all that is left is to update the `onPressed` method of our `add to cart` button. This method needs to do two things:

- Take the current cart and add an item to it, and
- Invoke the `onCartUpdated` method with our new cart.

```
1. onPressed: () {  
2.     currentCart.add(inventory[i]);  
3.     onCartUpdated(currentCart);  
4. },
```

Note how we assigned the `currentCart` variable *inside* the build method. This will clear the list and rebuild the widget each time the cart changes, so the `currentCart` will always match what is in the inherited widget's cart. We could also use this logic to conditionally display an *out of stock* message on items in the inventory that have less in stock than the number of items in the cart (minus one).

At this point, the Inventory widget is complete. Pressing the `add to cart` button will update the cart in our inherited widget. The `ShoppingCart` is not being updated, however, so let us fix that now.

Just like with the Inventory, we are going to get the cart from the inherited widget. Once we have the cart, we can check off a couple of TODOs:

```
1. Row(  
2.     mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
3.     children: [  
4.         Text('Items in Cart: ${currentCart.itemsInCart.length}'),  
5.         // TODO: Display the total cost of all items in the cart  
6.         Text('Total Cost: ')),
```

```
7.     OutlinedButton.icon(  
8.         label: const Text('Clear cart'),  
9.         icon: const Icon(Icons.remove_shopping_cart),  
10.        onPressed: () => onCartUpdated([]),  
11.    ),  
12.  ],  
13. ),
```

It is easy enough to get the total number of items in the cart by getting the length of the list. It is also easy enough to clear the cart by passing an empty list to the `onCartUpdated` method. While we could certainly do some math here to figure out the total cost of all items in the cart, it might make more sense to do that in our inherited widget. In fact, let us write a little `getter` method to do just that:

```
1. double get totalCost {  
2.     double price = 0;  
3.     for (Item item in itemsInCart) {  
4.         price += item.cost;  
5.     }  
6.  
7.     return price;  
8. }
```

When we put this inside our inherited widget, we can simply call the property of the inherited widget to get the value. In practice, that would look like the following: `ShoppingCartInheritedWidget.of(context)!.totalCost`. Since we already assigned the inherited widget to a variable, we can use that same variable to get the total cost:

```
1. Text('Total Cost: \$\${currentCart.totalCost.toStringAsFixed(2)}'),
```

With that, we have checked another item off our TODO list. Looking at what is left of our TODOs, we know that we need to figure out how many unique items are in the cart. Just like we did with the `totalCost` getter, we can write a getter that will iterate over the items in our cart and return a unique list:

```
1. List<Item> get uniqueItemsInCart {  
2.     final List<Item> unique = [];  
3.     for (Item item in itemsInCart) {  
4.         if (!unique.contains(item)) {  
5.             unique.add(item);  
6.         }  
7.     }  
8.     return unique;  
9. }
```

Once we have got a method to tell us what the unique items in the list are, we have everything we need to finish off our `ShoppingCart`:

```
1. ListView.builder(  
2.     shrinkWrap: true,  
3.     physics: const ClampingScrollPhysics(),  
4.     itemCount: currentCart.uniqueItemsInCart.length,  
5.     itemBuilder: (context, i) => Card(  
6.         child: ListTile(  
7.             leading: Text(  
8.                 currentCart.itemsInCart  
9.                     .where((element) =>  
10.                         element == currentCart.uniqueItemsInCart[i])  
11.                     .length
```

```
12.           .toString() +  
13.           '×',  
14.       ),  
15.   title: Text(  
16.       currentCart.itemsInCart  
17.           .firstWhere((element) =>  
18.               element == currentCart.uniqueItemsInCart[i])  
19.           .name,  
20.       ),  
21.   subtitle: Text(  
22.       '\$\$\{currentCart.itemsInCart[i].cost.toStringAsFixed(2)\}  
23.       '),  
24.   ),  
25. ),
```

And that is how to manage app state using an inherited widget. As you can see, it takes quite a bit of work to use this method. There are *many* packages out there that wrap up all the complexity involved in using an inherited widget in this way to manage app state and make it much more straightforward to use. In the next section, we are going to look at Provider, which aims to do just that.

Using Provider to Manage App State

The Provider package (<https://pub.dev/packages/provider>) is a simple app state management solution that does all the heavy lifting of creating an inherited widget for us. To get started, add the latest version of Provider to your pubspec and run `flutter pub get`. Provider offers a bunch of extra functionalities that we are not going to cover here, so be sure to read over the documentation to familiarize yourself with the best practices and any functionality we do not cover.

Once you have imported Provider into your code, we can start using it. First, let us set up a provider for our shopping cart. This time, instead of creating an inherited widget, we will extend the `ChangeNotifier` class, which will trigger rebuilds when parts of the widget change (such as the items in our cart):

```
1. class ShoppingCartProvider extends ChangeNotifier {  
2.   final List<Item> _itemsInCart = [];  
3.  
4.   UnmodifiableListView<Item> get itemsInCart =>  
5.     UnmodifiableListView(_itemsInCart);  
6.  
7.   double get totalCost {  
8.     double price = 0;  
9.     for (Item item in _itemsInCart) {  
10.       price += item.cost;  
11.     }  
12.  
13.     return price;  
14.   }  
15.  
16.   List<Item> get uniqueItemsInCart {  
17.     final List<Item> unique = [];  
18.     for (Item item in _itemsInCart) {  
19.       if (!unique.contains(item)) {  
20.         unique.add(item);  
21.       }  
22.     }  
23.     return unique;  
24.   }  
25.  
26. }  
27.  
28. class Item {  
29.   String name;  
30.   double cost;  
31. }  
32.  
33. void main() {  
34.   runApp(MyApp());  
35. }  
36.  
37. class MyApp extends StatelessWidget {  
38.   @override  
39.   Widget build(BuildContext context) {  
40.     return MaterialApp(  
41.       title: 'Provider Example',  
42.       theme: ThemeData(  
43.         primarySwatch: Colors.blue,  
44.       ),  
45.       home: MyHomePage(),  
46.     );  
47.   }  
48. }  
49.  
50. class MyHomePage extends StatefulWidget {  
51.   @override  
52.   _MyHomePageState createState() => _MyHomePageState();  
53. }  
54.  
55. class _MyHomePageState extends State<MyHomePage> {  
56.   ShoppingCartProvider provider;  
57.  
58.   @override  
59.   void initState() {  
60.     provider = ShoppingCartProvider();  
61.     provider.addItem(Item(name: 'Widget', cost: 10));  
62.     provider.addItem(Item(name: 'Dart', cost: 20));  
63.     provider.addItem(Item(name: 'Flutter', cost: 30));  
64.     provider.addItem(Item(name: 'React Native', cost: 40));  
65.     provider.addItem(Item(name: 'Angular', cost: 50));  
66.     provider.addItem(Item(name: 'Node.js', cost: 60));  
67.     provider.addItem(Item(name: 'React', cost: 70));  
68.     provider.addItem(Item(name: 'Vue.js', cost: 80));  
69.     provider.addItem(Item(name: 'Svelte', cost: 90));  
70.     provider.addItem(Item(name: 'Ember.js', cost: 100));  
71.     provider.addItem(Item(name: 'Rails', cost: 110));  
72.     provider.addItem(Item(name: 'Rails', cost: 120));  
73.     provider.addItem(Item(name: 'Rails', cost: 130));  
74.     provider.addItem(Item(name: 'Rails', cost: 140));  
75.     provider.addItem(Item(name: 'Rails', cost: 150));  
76.     provider.addItem(Item(name: 'Rails', cost: 160));  
77.     provider.addItem(Item(name: 'Rails', cost: 170));  
78.     provider.addItem(Item(name: 'Rails', cost: 180));  
79.     provider.addItem(Item(name: 'Rails', cost: 190));  
80.     provider.addItem(Item(name: 'Rails', cost: 200));  
81.     provider.addItem(Item(name: 'Rails', cost: 210));  
82.     provider.addItem(Item(name: 'Rails', cost: 220));  
83.     provider.addItem(Item(name: 'Rails', cost: 230));  
84.     provider.addItem(Item(name: 'Rails', cost: 240));  
85.     provider.addItem(Item(name: 'Rails', cost: 250));  
86.     provider.addItem(Item(name: 'Rails', cost: 260));  
87.     provider.addItem(Item(name: 'Rails', cost: 270));  
88.     provider.addItem(Item(name: 'Rails', cost: 280));  
89.     provider.addItem(Item(name: 'Rails', cost: 290));  
90.     provider.addItem(Item(name: 'Rails', cost: 300));  
91.     provider.addItem(Item(name: 'Rails', cost: 310));  
92.     provider.addItem(Item(name: 'Rails', cost: 320));  
93.     provider.addItem(Item(name: 'Rails', cost: 330));  
94.     provider.addItem(Item(name: 'Rails', cost: 340));  
95.     provider.addItem(Item(name: 'Rails', cost: 350));  
96.     provider.addItem(Item(name: 'Rails', cost: 360));  
97.     provider.addItem(Item(name: 'Rails', cost: 370));  
98.     provider.addItem(Item(name: 'Rails', cost: 380));  
99.     provider.addItem(Item(name: 'Rails', cost: 390));  
100.    provider.addItem(Item(name: 'Rails', cost: 400));  
101.  }  
102.  
103.  @override  
104.  void dispose() {  
105.    provider.dispose();  
106.    super.dispose();  
107.  }  
108.  
109.  void _incrementCounter() {  
110.    provider.addItem(Item(name: 'Rails', cost: 410));  
111.  }  
112.  
113.  @override  
114.  Widget build(BuildContext context) {  
115.    return Scaffold(  
116.      appBar: AppBar(  
117.        title: Text('Provider Example'),  
118.      ),  
119.      body: Center(  
120.        child: Column(  
121.          mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
122.          children:   
123.            [  
124.              Text('Total Cost: ${provider.totalCost}'),  
125.              ElevatedButton(  
126.                onPressed: _incrementCounter,  
127.                child: Text('Add Item'),  
128.              ),  
129.              Text('Unique Items: ${provider.uniqueItemsInCart.length}'),  
130.              Text('Items In Cart: ${provider.itemsInCart.length}'),  
131.            ],  
132.          ),  
133.        ),  
134.      ),  
135.    );  
136.  }
```

```
22.      }
23.      return unique;
24.  }
25.
26.  void add(Item item) {
27.      _itemsInCart.add(item);
28.      notifyListeners();
29.  }
30.
31.  void emptyCart() {
32.      _itemsInCart.clear();
33.      notifyListeners();
34.  }
35. }
```

You will notice that a lot of the code is the same as our inherited widget version. In fact, our two getters, `totalCost` and `uniqueItemsInCart`, are the same. What is different, however, is how we access the items in the cart, add items to the cart, and empty the cart.

To get the items in the cart, this time, we have the getter `itemsInCart`, which returns an `UnmodifiableListView<Item>` value. An `UnmodifiableListView` is, as the name implies, a representation of an existing list that cannot be modified. We will get to why that is later. Also note that we were able to move the list holding our cart into this widget instead of using a stateful widget for our `MyApp`. It makes a lot more sense to have it here, since this is what is holding the actual cart state.

Finally, we have two methods for adding and clearing the cart. They are self-explanatory but have a new concept to look at: `notifyListeners()`. We have not come across this method call before. To explain it, we will first need to add our `ShoppingCartProvider` to the widget tree:

```
1. void main() {  
2.   runApp(  
3.     ChangeNotifierProvider(  
4.       create: (context) => ShoppingCartProvider(),  
5.       child: const MyApp(),  
6.     ),  
7.   );  
8. }
```

The **ChangeNotifierProvider** widget, which is part of the **Provider** package, will listen for events triggered by the **notifyListeners** method and rebuild itself accordingly. Without a listener of some sort (of which **ChangeNotifierProvider** is but one), calling **notifyListeners** has no effect.

So, how else does this change our app? Well, for starters, it significantly simplifies our **MyApp** widget. As previously mentioned, we no longer need to hold the app state, so we can use a stateless widget. Plus, since the cart is now in our **ShoppingCartProvider**, we do not need to have a variable for it there, which also means we do not need any sort of method to update the cart right here. Look at how much cleaner that makes our widget:

```
1. class MyApp extends StatelessWidget {  
2.   const MyApp({super.key});  
3.  
4.   @override  
5.   Widget build(BuildContext context) {  
6.     return MaterialApp(  
7.       home: Scaffold(  
8.         body: Padding(  
9.           padding: const EdgeInsets.all(8.0),
```

```
10.         child: ListView(
11.             children: const [
12.                 Inventory(),
13.                 ShoppingCart(),
14.             ],
15.         ),
16.     ),
17. ),
18. );
19. }
20. }
```

It also simplifies both the `Inventory` and `ShoppingCart` widgets. The `Inventory`, for example, becomes a consumer of the `ShoppingCartProvider`. The `Consumer` widget comes from the `Provider` package and takes in a builder, which provides the `BuildContext`, the Provider, and an optional child widget. The `child` parameter can be used to improve performance by limiting the number of widgets that redraw. See the documentation for more details.

This is what our new `Inventory` looks like, once we have updated it to use the provider we created:

```
1. class Inventory extends StatelessWidget {
2.     const Inventory({super.key});
3.
4.     @override
5.     Widget build(BuildContext context) {
6.         return Consumer<ShoppingCartProvider>(
7.             builder:
```

```
8.          (BuildContext context, ShoppingCartProvider cart,
9.           Widget? child) {
10.         return ListView(
11.           shrinkWrap: true,
12.           physics: const ClampingScrollPhysics(),
13.           children: [
14.             Text(
15.               'Inventory',
16.               style:
17.                 Theme.of(context).textTheme.displayMedium,
18.             ),
19.             ListView.builder(
20.               shrinkWrap: true,
21.               physics: const ClampingScrollPhysics(),
22.               itemCount: inventory.length,
23.               itemBuilder: (BuildContext context, int i) =>
24.                 Card(
25.                   child: ListTile(
26.                     title: Text(inventory[i].name),
27.                     subtitle:
28.                       Text('$$ ${inventory[i].cost.toStringAsFixed(2)}'),
29.                     trailing: IconButton(
30.                       icon: const Icon(Icons.add_shopping_cart),
31.                       onPressed: () => cart.add(inventory[i]),
32.                     ),
33.                   ),
34.                 ),
35.               ),
36.             ),
37.           ],
38.         );
39.       }
40.     ),
41.   );
42. }
```

```
31.     ),
32.   ],
33.   );
34. },
35. );
36. }
37. }
```

Note how this changes our **onPressed** method of the **add to cart** button. We are calling the method inside the provider we created when we tap the button. It is much cleaner than passing in a callback function and dealing with the change higher in the widget tree.

Like how using Provider has simplified the **Inventory**, it also simplifies the **ShoppingCart**:

```
1. class ShoppingCart extends StatelessWidget {
2.   const ShoppingCart({super.key});
3.
4.   @override
5.   Widget build(BuildContext context) {
6.     return Consumer<ShoppingCartProvider>(
7.       builder:
8.         (BuildContext context, ShoppingCartProvider cart,
9.          Widget? child) =>
10.        ListView(
11.          shrinkWrap: true,
12.          physics: const ClampingScrollPhysics(),
13.          children: [
```

```
13.     Text(  
14.         'Shopping Cart',  
15.         style:  
16.             Theme.of(context).textTheme.displayMedium,  
17.     Row(  
18.         mainAxisAlignment:  
19.             MainAxisAlignment.spaceEvenly,  
20.         children: [  
21.             Text('Items in Cart: ${cart.itemsInCart.length}'),  
22.             Text('Total Cost:   
23.                 \${cart.totalCost.toStringAsFixed(2)}'),  
24.             OutlinedButton.icon(  
25.                 label: const Text('Clear cart'),  
26.                 icon: const Icon(Icons.remove_shopping_cart),  
27.                 onPressed: () =>  
28.                     Provider.of<ShoppingCartProvider>(context,  
29.                         listen: false,  
30.                         ).emptyCart(),  
31.             ),  
32.             ListView.builder(  
33.                 shrinkWrap: true,  
34.                 physics: const ClampingScrollPhysics(),  
35.                 itemCount: cart.uniqueItemsInCart.length,
```

```
36.         itemBuilder: (context, i) => Card(
37.           child: ListTile(
38.             leading: Text(
39.               '${cart.itemsInCart.where((e) => e == cart.uniqueItemsInCart[i]).length} ×',
40.             ),
41.             title: Text(
42.               cart.itemsInCart
43.                 .firstWhere(
44.                   (e) => e == cart.uniqueItemsInCart[i]
45.                     ).name,
46.                     ),
47.             subtitle:
48.               Text('$$ ${cart.itemsInCart[i].cost.toStringAsFixed(2)}'),
49.             ),
50.             ),
51.             ),
52.           ],
53.           ),
54.         );
55.       }
56.     }
```

Take note of the `onPressed` parameter of the clear cart button. This time, rather than call the `emptyCart` method on the `cart` object, we have chosen to reference the Provider via context. We did this so that we can pass in `listen: false` because we do not need to rebuild the button after running the method.

This code was added as an illustration and only trivially changes the behavior of the application, if at all.

Outside of that change, everything else should be completely familiar. It is also all the last of the changes we need to use Provider to manage our app state. Using Provider means we had to write *far less* code than with the inherited widget. We no longer have a stateful widget anywhere and we are not passing methods around to the different child widgets.

Provider is an excellent solution if you need a simple app state management system. However, it lacks certain functionality that might be useful, such as providing *different* states (think an `adding to cart` state followed by an `added to cart` state or an `error adding to cart` state that might be useful when communicating with an API.) For that, let us look at a more powerful solution: BLoC.

Using BLoC's Cubit to Manage App State

BLoC, or *business logic component*, is an extremely popular app state management solution. The goal behind BLoC is to abstract out the logic of the app from the presentation of the app while still allowing you to maintain app state. To understand BLoC, we first need to look at the concept of a Cubit, which is akin to a *lite* version of BLoC. Both BLoC and Cubit are supplied by the `flutter_bloc` package (https://pub.dev/packages/flutter_bloc). To get started, first, add the package to your `pubspec`, then import it into your app.

A Cubit will feel very familiar since most of what is in a Cubit is what you would find in a Provider. The Cubit comes in two parts: the Cubit itself, and a state object, which holds the current state of the Cubit. First, look at the state class:

```
1. class CartState {  
2.   final List<Item> itemsInCart;  
3.  
4.   CartState(this.itemsInCart);  
5. }
```

That is all there is to the state object. It can hold other pieces of information, but it does not do any processing. That is where our Cubit comes into play:

```
1. class ShoppingCartCubit extends Cubit<CartState> {  
2.     ShoppingCartCubit() : super(CartState([]));  
3.  
4.     double getTotalCost {  
5.         double price = 0;  
6.         for (Item item in state.itemsInCart) {  
7.             price += item.cost;  
8.         }  
9.  
10.        return price;  
11.    }  
12.  
13.    List<Item> get uniqueItemsInCart {  
14.        final List<Item> unique = [];  
15.        for (Item item in state.itemsInCart) {  
16.            if (!unique.contains(item)) {  
17.                unique.add(item);  
18.            }  
19.        }  
20.        return unique;  
21.    }  
22.
```

```
23. void add(Item item) {  
24.     final List<Item> updatedCart = state.itemsInCart..add(item);  
25.  
26.     emit(CartState(updatedCart));  
27. }  
28.  
29. void clearCart() {  
30.     emit(CartState([]));  
31. }  
32. }
```

Most of this should be very familiar to you at this point. The parts that are new are going to be where we reference `state` to get at the `itemsInCart` variable, the double-dot operator on Line 24, and the `emit()` statements.

The `state` variable is managed by Cubit and points to that `CartState` class we created earlier. Each time we want to update the cart, the old `CartState` object is tossed out and replaced by a brand-new one. In order to use that object elsewhere in the app, we need to put a bow on it and send it off into the world by calling the `emit()` method on it.

As for the double-dot operator: this is Dart's *cascade notation*. This is a super useful shortcut that lets you perform multiple operations on the same item. Say, for example, you wanted to progressively build a new `Color`. We can start with a base `Color`, then change it one step at a time:

1. Color test = const Color(0xFFFFFFFF)
2. ..withAlpha(128)
3. ..withBlue(100)
4. ..withGreen(20)
5. ..withRed(72);

It is a silly example to be sure, but compare that to the alternative (but also technically correct) way of accomplishing the same task:

```
1. Color test = const Color(0xFFFFFFFF);  
2.     test = test.withAlpha(128);  
3. test = test.withBlue(100);  
4.     test = test.withGreen(20);  
5. test = test.withRed(72);
```

As you can see, the cascade notation is a wonderful time-saving utility. Ok, back to Cubit and BLoC.

Now that we have a Cubit and state class built, we need to provide access to them in the widget tree in the exact same way we did for Provider. The *only* difference is that we are going to use a **BlocProvider** class instead of a **ChangeNotifierProvider**:

```
1. void main() {  
2.   runApp(  
3.     BlocProvider(  
4.       create: (context) => ShoppingCartProvider(),  
5.       child: const MyApp(),  
6.     ),  
7.   );  
8. }
```

Just like Provider's **Consumer<Provider>** class, we have a **BlocBuilder<Cubit, State>** class:

```
1. class ShoppingCart extends StatelessWidget {  
2.   const ShoppingCart({super.key});  
3. }
```

```
4.    @override
5.    Widget build(BuildContext context) {
6.        return BlocBuilder<ShoppingCartCubit, CartState>(
7.            builder: (BuildContext context, CartState cart) {
8.                return ...
9.            }
10.       );
11.   }
```

This gives us access to the state object via the `cart` variable, assigned on Line 7. So, we could access the items in our cart by doing something like this:

```
1. Text('Items in Cart: ${cart.itemsInCart.length}'),
```

Since our cart state object only holds the list of items in the cart, we need another way to get information out of the Cubit. Luckily, we can access it via the `BuildContext`:

```
1. final List<Item> uniqueItemsInCart =
  context.read<ShoppingCartCubit>().uniqueItemsInCart;
```

Or if we want to add something to the cart, we could use the same logic:

```
1. IconButton(
2.   icon: const Icon(Icons.add_shopping_cart),
3.   onPressed: () => context.read<ShoppingCartCubit>()
4.     .add(inventory[i]),
4. ),
```

So, that is Cubit in a nutshell. See if you can update the Provider example to use a Cubit. The process should be quite straightforward with minimal challenges. Once you have got it working, or whenever you feel ready, we will dig into BLoC.

Using BLoC to Manage App State

While Cubit is great for simple app state management, sometimes you just need something *more*. BLoC is broken down into the BLoC itself, a state object, and events. Let us start building some of these out so you can see what we are talking about, starting with defining a couple of base classes. We will not use these directly, but we will extend them to build the rest of our events and states:

1. `@immutable`
2. `abstract class ShoppingCartEvent {}`
- 3.
4. `@immutable`
5. `abstract class ShoppingCartState {}`

Next, let us create a couple of possible states for the app to be in; first, an initial state where our cart is empty, then a state where we have items in the cart. While we are at it, let us create a waiting state that we can use later when we mimic the delay of calling an API and waiting for a response:

1. `class Initial extends ShoppingCartState {`
2. `Initial() : super();`
3. `}`
- 4.
5. `class CartWithItems extends ShoppingCartState {`
6. `final List<Item> itemsInCart;`
7. `CartWithItems({required this.itemsInCart}) : super();`
8. `}`
- 9.
10. `class Waiting extends ShoppingCartState {`
11. `Waiting() : super();`
12. `}`

Now that we have some states, let us create the events we will be using. Our app allows us to perform two actions: (1) add an item to the cart and (2) clear the cart contents. Let us create an event for each of those:

```
1. class AddItemToCart extends ShoppingCartEvent {  
2.     final Item item;  
3.     AddItemToCart({required this.item});  
4. }  
5.  
6. class ClearCart extends ShoppingCartEvent {  
7.     ClearCart();  
8. }
```

Finally, we can build our basic BLoC using these event and state objects:

```
1. class ShoppingCartBloc extends Bloc<ShoppingCartEvent,  
   ShoppingCartState> {  
2.     ShoppingCartBloc() : super(Initial()) {  
3.         on<ShoppingCartEvent>((event, emit) {  
4.             // TODO: implement event handler  
5.         });  
6.     }  
7. }
```

The general idea of how this will operate is: the BLoC is created and generates a state [the `super(Initial())` tells us that the state it starts with is `Initial()`], some sort of interaction will send an event to the BLoC, where the BLoC will process that event and emit one or more states. Let us walk through this each step of the way.

When the BLoC is initialized, the state is `Initial()`. Next, the user taps the **add item to cart** button, which looks like the following:

```
1. IconButton(  
2.   icon: const Icon(Icons.add_shopping_cart),  
3.   onPressed: () => context.read<ShoppingCartBloc>().add(  
4.     AddItemToCart(  
5.       item: inventory[i],  
6.     ),  
7.   ),  
8. ),
```

An event is added to the BLoC. That event, `AddItemToCart`, has a required `item` property, which is where we pass in the item we want to add to the cart. Next, the BLoC will process that event and emit a new state. Let us update the BLoC accordingly, to add the item to the cart and process the event:

```
1. class ShoppingCartBloc extends Bloc<ShoppingCartEvent,  
   ShoppingCartState> {  
2.   final List<Item> _itemsInCart = <Item>[];  
3.  
4.   ShoppingCartBloc() : super(Initial()) {  
5.     on<AddItemToCart>((event, emit) {  
6.       _itemsInCart.add(event.item);  
7.       emit(CartWithItems(itemsInCart: _itemsInCart));  
8.     });  
9.   }  
10. }
```

This will emit a state that we can build the UI based upon. We will look at how to do that in just a moment, but first, let us add that perceived delay to mimic the API call and add the rest of the functionality to our BLoC which

brings it up to par with what we have been working with in other state management solutions:

```
1. class ShoppingCartBloc extends Bloc<ShoppingCartEvent,  
   ShoppingCartState> {  
2.   final Cart _itemsInCart = <Item>[];  
3.  
4.   ShoppingCartBloc() : super(Initial()) {  
5.     on<AddItemToCart>((event, emit) async {  
6.       emit(Waiting());  
7.       await Future.delayed(const Duration(seconds: 2));  
8.       itemsInCart.add(event.item);  
9.       emit(CartWithItems(itemsInCart: _itemsInCart));  
10.    });  
11.    on<ClearCart>((event, emit) {  
12.      _itemsInCart.clear();  
13.      emit(Initial());  
14.    });  
15.  }  
16.  
17.  double getTotalCost {  
18.    double price = 0;  
19.    for (Item item in _itemsInCart) {  
20.      price += item.cost;  
21.    }  
22.  
23.    return price;
```

```
24.    }
25.    }
26.    Cart get uniqueItemsInCart {
27.        final Cart unique = [];
28.        for (Item item in _itemsInCart) {
29.            if (!unique.contains(item)) {
30.                unique.add(item);
31.            }
32.        }
33.        return unique;
34.    }
35. }
```

This BLoC has everything we need to build the rest of the application. Note how the event handler for adding an item to the cart emits multiple states. This is something you could also do with, say, Cubit. It allows us to build more dynamic interfaces for users by expressing to them the current *state* of the application.

Once you have updated your **BlocProvider** to provide this new BLoC and the **BlocBuilder** to use the new BLoC and state, we can update the variables to match. As an example:

```
1. class ShoppingCart extends StatelessWidget {
2.     const ShoppingCart({super.key});
3.     }
4.     @override
5.     Widget build(BuildContext context) {
6.         return BlocBuilder<ShoppingCartBloc, ShoppingCartState>(
7.             builder: (BuildContext context, ShoppingCartState state) {
```

```
8.     final Cart uniqueItemsInCart =  
9.         context.read<ShoppingCartBloc>  
          ().uniqueItemsInCart;  
10.    final double totalCost = context.read<ShoppingCartBloc>  
        ().totalCost;  
11.    ...  
12.};  
13.}  
14.}
```

This is where things start to get interesting, though. We can use the `state` variable to build different widgets depending on the current state of the application. For example, if there is nothing in the cart, we have no reason to show the `cart details` or `clear` button. Instead, we might want to tell the user that their cart is empty. Or, if we are waiting for the API to finish (that is, we are in a `Waiting` state), we might want to show a progress indicator. We just need to have a conditional based on what *type* of state the `state` is. That is why we created a base state class and extended it: all the states we created are still, technically, the base state.

When all is said and done, here is what the complete `ShoppingCart` looks like using the BLoC:

```
1. class ShoppingCart extends StatelessWidget {  
2.     const ShoppingCart({super.key});  
3.  
4.     @override  
5.     Widget build(BuildContext context) {  
6.         return BlocBuilder<ShoppingCartBloc, ShoppingCartState>(  
7.             builder: (BuildContext context, ShoppingCartState state) {  
8.                 final Cart uniqueItemsInCart =
```

```
9.           context.read<ShoppingCartBloc>
10.          ().uniqueItemsInCart;
11.
12.          return ListView(
13.            shrinkWrap: true,
14.            physics: const ClampingScrollPhysics(),
15.            children: [
16.              Text(
17.                'Shopping Cart',
18.                style: Theme.of(context).textTheme.displayMedium,
19.              ),
20.              if (state is Initial) const Text('Your cart is
empty.'),
21.              elseif (state is Waiting)
22.                Column(
23.                  children: const [
24.                    CircularProgressIndicator(),
25.                    Text('Please wait while we update your cart.'),
26.                  ],
27.                ),
28.              if (state is CartWithItems)
29.                Column(
30.                  children: [
31.                    Row(
```

```
32.                                     mainAxisAlignment:  
33.                                         MainAxisAlignment.spaceEvenly,  
34.                                         children: [  
35.                                             Text('Items     in     Cart:  
36.                                             ${state.itemsInCart.length}'),  
37.                                             Text('Total     Cost:  
38.                                             \$\$ ${totalCost.toStringAsFixed(2)}'),  
39.                                         OutlinedButton.icon(  
40.                                             label: const Text('Clear cart'),  
41.                                             icon: const Icon(Icons.remove_shopping_cart),  
42.                                             onPressed: () =>  
43.                                               context.read<ShoppingCartBloc>().add(  
44.                                                 ClearCart()  
45.                                             ),  
46.                                         ),  
47.                                         ],  
48.                                         ),  
49.                                         ),  
50.                                         ListView.builder(  
51.                                             shrinkWrap: true,  
52.                                             physics: const ClampingScrollPhysics(),  
53.                                             itemCount: uniqueItemsInCart.length,  
54.                                             itemBuilder: (context, i) {  
55.                                                 return Card(  
56.                                                   child: ListTile(  
57.                                                     leading: Text(  
58.                                                       '$ {state.itemsInCart.where((element) =>  
59.                                                       element == uniqueItemsInCart[i].length} ×'),
```

```
55.         title: Text(uniqueItemsInCart[i].name),  
56.         subtitle: Text(  
57.             '\$\${uniqueItemsInCart[i].cost.toStringAsFixed(2)  
58.         }),  
59.     );  
60.     },  
61.     ),  
62.     ],  
63.     ),  
64.     ],  
65.     );  
66.     },  
67. );  
68. }  
69. }
```

BLoC is a very powerful state management solution, capable of empowering you, as a developer, to build expressive and dynamic interfaces that truly convey the current state of the application to a user. It may be overkill in some scenarios where a Cubit or Provider would be just as effective, though. So, when choosing a state management solution, be sure to learn as much as you can about the different requirements for the task at hand and choose wisely.

Conclusion

We explored a lot of different methods of dealing with app state in this chapter. There is no single best solution, either. Each has its pros and cons. This also was not an exhaustive list of all possible state management

solutions that exist out there. We did not mention Flutter Hooks, Riverpod, Redux, MobX, or any of the other frameworks which are also quite popular. Chances are, when you come into a project, there may already be a state management solution being used. Fortunately, by understanding the basics of things like using inherited widgets, Provider, or Cubit/BLoC to manage app state, you will not have much trouble following along with whatever state management solution you come across.

In the upcoming chapter, we are going to discuss concepts such as how to make an interface that lays itself out dynamically based on the available screen space. Plus, we are going to dig into some of the things you will need to know about building applications for different platforms.

Questions

1. What is ephemeral state, and how is it different from app state?
2. Why is a stateful widget not a good option for managing app state?
3. If you have nested `ListView` widgets, how can you prevent the inner `ListView` from scrolling independently of its parent?
4. What is the `=>` operator used for? What are the benefits/drawbacks of using it?
5. What is escaping? What characters might need to be escaped?
6. What is interpolation?
7. What is `ValueChanged` used for? How is it defined?
8. What is a `typedef`?
9. What is `UnmodifiableListView` used for?
10. What are some differences and similarities between Provider, Cubit, and BLoC?
11. What is cascade notation?
12. What are some other popular app state management solutions?

Key Terms

- **=> operator:** Shorthand for returning a value.
- **App state:** State which is maintained throughout the application.

- **Ephemeral state:** State managed by a stateful widget.
- **Escaping:** The process of using a backslash (\) to denote the following character should not be interpreted as a special character in a string.
- **Interpolation:** A way of evaluating a variable or method and inserting the result directly in a string. Interpolation is denoted by the \${} characters, where the code inside the braces is to be evaluated.
- **typedef:** A way to define a new type, on the fly, without creating a new class. `ValueChanged`, for example, is a `typedef` defined as: `typedef ValueChanged<T> = void Function(T value)`

Further Reading

- Learn more about Dart's cascade notation: <https://dart.dev/guides/language/language-tour#cascade-notation>
- A list of state management approaches: <https://docs.flutter.dev/development/data-and-backend/state-mgmt/options>

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

Reactivity and Platform-Specific Considerations

Introduction

Only rarely do we have complete control over what devices our applications will run on. Users will bring a variety of devices that should all be supported; be they phones, Web browsers, desktop computers, tablets, or any other form of personal computing device. The differences between each of these platforms are sometimes subtle and other times extreme. That means that we must be smart about how we build applications, ensuring they will work flawlessly on every platform we plan to target. There are a variety of different techniques we can employ to ensure that our app looks, feels, and acts as it belongs on whatever screen it is running on. In this chapter, we are going to explore these techniques.

Structure

In this chapter, we will discuss the following topics:

- Building responsive layouts
- Benefits and limitations of dart:io
- App icons and splash screens
- Material, Cupertino, Yaru, and Fluent UI (oh my!)

- Platform-specific navigation paradigms
- Using packages to build cross-platform UIs

Objectives

After completing this chapter, you will have a firm understanding of responsive layouts, different UI styles and paradigms, and how to add icons and splash screens to our applications. You will learn about packages that provide differently styled widgets that match the native look and feel of different platforms, such as Windows and Linux. You will be able to describe some of the differences between Android and iOS, especially in terms of navigation within an application. Finally, you will learn how to build a UI that can swap between two sets of widgets, automatically, depending on what platform it is running on.

Building Responsive Layouts

Users will run your application on whatever device they have. Perhaps it is the latest iPhone, last year's Android device, a foldable phone, a Web browser on their laptop, or as a Linux desktop application. What works on a phone does not necessarily work on the desktop or in a Web browser. In such cases, it is appropriate to build a different layout. A hamburger menu or tab bar design might work well on the phone, but a different navigation paradigm is appropriate otherwise. So, how do we decide what sort of navigation paradigm to show?

For this, we will use the `LayoutBuilder` widget. Take the following simple example:

```

1. class My StatelessWidget extends StatelessWidget {
2.   const MyWidget({super.key});
3.
4.   @override
5.   Widget build(BuildContext context) {
6.     return Scaffold(
7.       body: LayoutBuilder(
```

```
8.     builder: (BuildContext context, BoxConstraints constraints) {
9.       return Center(
10.         child: AnimatedCrossFade(
11.           firstChild: const Text('This is a wide-screen
12.             layout'),
13.             secondChild: const Text('This is a small-screen
14.               layout'),
15.               crossFadeState: constraints.maxWidth > 600
16.                 ? CrossFadeState.showFirst
17.                   : CrossFadeState.showSecond,
18.                     duration: const Duration(milliseconds: 500),
19.                       ),
20.                     ),
21.                   );
22.   }
23. }
```

In this example, we will show different text based on the width of the application. Go ahead and try it out: run it as a desktop or Web application, then resize the window. You will see that as soon as you make the screen narrow or wide enough, the text will cross-fade. Let us look at the scenarios in which `LayoutBuilder` will update its children.

- The first time the widget is laid out.
- When the parent widget passes different layout constraints.
- When the parent widget updates this widget.

- When the dependencies that the builder function subscribes to change.

In short, the builder will run when the constraints or dependencies of the child widget update. This useful widget is perfect for building layouts based on different screen sizes. Let us look at our example of building different navigation styles. On mobile, we want a hamburger menu but on larger screens, we might want a vertical navigation bar on the edge of the screen. First, build your separate navigation options:

```

1. class DesktopNavigation extends StatelessWidget {
2.   final Widget body;
3.   const DesktopNavigation({super.key, required this.body});
4.
5.   @override
6.   Widget build(BuildContext context) {
7.     final double height = MediaQuery.of(context).size.height;
8.     final double width = MediaQuery.of(context).size.width;
9.     return SizedBox(
10.       height: height,
11.       width: width,
12.       child: Row(
13.         children: [
14.           Container(
15.             width: 200,
16.             height: height,
17.             color: Colors.grey,
18.             child: Column(
19.               children: const [

```

```
20.           Text('Desktop Navigation'),  
21.           ],  
22.           ),  
23.           ),  
24.           body,  
25.           ],  
26.           ),  
27.           );  
28.       }  
29.   }  
30.  
31. class MobileNavigation extends StatelessWidget {  
32.     final Widget body;  
33.     const MobileNavigation({super.key, required this.body});  
34.  
35.     @override  
36.     Widget build(BuildContext context) {  
37.       return SizedBox(  
38.         height: MediaQuery.of(context).size.height,  
39.         width: MediaQuery.of(context).size.width,  
40.         child: body,  
41.       );  
42.     }  
43. }
```

Then you can use a **LayoutBuilder** to choose which navigation option to use:

```
1. class My StatelessWidget extends StatelessWidget {  
2.     const My StatelessWidget({super.key});  
3.  
4.     @override  
5.     Widget build(BuildContext context) {  
6.         final double width = MediaQuery.of(context).size.width;  
7.         return Scaffold(  
8.             appBar: (width <= 600) ? AppBar() : null,  
9.             drawer: (width <= 600)  
10.                ? const Drawer(child: Text('Mobile Navigation'))  
11.                : null,  
12.             body: LayoutBuilder(  
13.                 builder: (BuildContext context, BoxConstraints  
14.                     constraints) {  
15.                         return constraints.maxWidth > 600  
16.                             ? const DesktopNavigation(  
17.                                 body: Text('This is a wide-screen layout'),  
18.                             )  
19.                         : const MobileNavigation(  
20.                             body: Text('This is a small-screen layout'),  
21.                         );  
22.                     ),  
23.                 );  
24.             }  
}
```

```
25. }
```

Not only will the navigation style change, but we also have control over what widgets are displayed in the app body. It is also possible to nest a `LayoutBuilder` within another `LayoutBuilder`. In our example here, we are passing different widgets into the body of the application. This is not strictly necessary, though. Since widgets themselves can have a `LayoutBuilder`, and we know that the `LayoutBuilder` can be nested, we can design the smallest of widgets to be reactive. Let us look at an example where we change the placement of a button's label based on the layout:

```
1. class My StatelessWidget extends StatelessWidget {  
2.   const MyWidget({super.key});  
3.  
4.   @override  
5.   Widget build(BuildContext context) {  
6.     return Scaffold(  
7.       body: LayoutBuilder(  
8.         builder: (BuildContext context, BoxConstraints constraints) {  
9.           return constraints.maxWidth > 600  
10.          ? Row(  
11.              children: const [  
12.                MyButton(),  
13.                Text('Wide layout'),  
14.              ],  
15.            )  
16.          : Column(  
17.              children: const [
```

```
18.           MyButton(),
19.           Text('Narrow layout'),
20.           ],
21.           );
22.       },
23.   ),
24. );
25. }
26. }
27.
28. class MyButton extends StatelessWidget {
29.     const MyButton({super.key});
30.
31.     @override
32.     Widget build(BuildContext context) {
33.         return IconButton(
34.             onPressed: () {},
35.             icon: const Icon(Icons.flutter_dash),
36.         );
37.     }
38. }
```

By combining these techniques, the possibilities are infinite. Elements of your UI can hide and show dynamically, based on screen constraints, whereas other widgets can adjust their layouts to match the space they are given more appropriately. And, since everything is a widget, that means we can change every aspect of the application based on constraints.

It may not always be ideal to add or remove widgets entirely when different constraints are present. Swapping navigation styles may be desirable but be

careful. Depending on how you build your navigation, you may end up duplicating code or, worst case scenario, forgetting to add a navigation option. In instances such as this, you may wish to define your navigation options as an `enum`, with whatever properties you need to properly navigate to a particular area of your application. Then, the `enum` itself can be passed into a builder which dynamically builds your navigation system. Look at the following example:

```
1. class My StatelessWidget extends StatelessWidget {  
2.   const MyWidget({super.key})  
3.  
4.   @override  
5.   Widget build(BuildContext context) {  
6.     return Scaffold(  
7.       body: LayoutBuilder(  
8.         builder: (BuildContext context, BoxConstraints constraints) {  
9.           return constraints.maxWidth > 600  
10.             ? const WideNavigation(  
11.                 navigationTargets: NavigationOption.values,  
12.             )  
13.             : const NarrowNavigation(  
14.                 navigationTargets: NavigationOption.values,  
15.             );  
16.           },  
17.         ),  
18.       );  
19.     }  
20. }
```

```
21.  
22. enum NavigationOption {  
23.   home(Home()),  
24.   settings(Settings()),  
25.   preferences(Preferences());  
26.  
27.   const NavigationOption(this.target);  
28.  
29.   final Widget target;  
30. }
```

From here, you can see how you could easily build a dynamic navigation system using the list of `enum` values provided. Maybe you would want to use a `ListView`:

```
1. class WideNavigation extends StatelessWidget {  
2.   final List<NavigationOption> navigationTargets;  
3.   const WideNavigation({super.key, required  
      this.navigationTargets});  
4.  
5.   @override  
6.   Widget build(BuildContext context) {  
7.     return ListView(  
8.       children: List<Widget>.generate(  
9.         NavigationOption.values.length,  
10.        (index) => ListTile(  
11.          title: Text(NavigationOption.values[index].name),  
12.          onTap: () => Navigator.of(context).push(
```

```
13.         MaterialPageRoute<void>(
14.             builder: (BuildContext context) =>
15.                 NavigationOption.values[index].target,
16.             ),
17.             ),
18.             ),
19.             ),
20.         );
21.     }
22. }
```

The idea, whatever you choose, is that the navigation system(s) you build are dynamic so that nothing is forgotten. Be considerate when swapping widgets in different components; do not accidentally remove important features without realizing it!

Benefits and Limitations of dart:io

There exists a package, `dart:io`, which can perform absolute magic... on *most* of your build targets. Let us look at this package, why it is useful, and when it *cannot* be used. The `dart:io` package is a library that deals with files, directories, processes, sockets, WebSockets, and HTTP clients and servers. If your application needs to read or write a file, list the contents of a directory, spawn a process, make a socket connection to a server, upgrade an HTTP session to a WebSocket, be a Web server, or connect directly to a Web server without using a different HTTP client package, you will likely be using `dart:io`. That is a long list of accomplishments for one small package, so let us break it down and look at how some of these can be helpful.

Let us start with files and folders. Specifically, let us list all the files and folders in a given directory:

```
1. class My StatelessWidget extends StatelessWidget {
```

```
2.     const MyStatelessWidget({super.key});  
3.  
4.     @override  
5.     Widget build(BuildContext context) {  
6.         const String path = "/";  
7.         final Directory directory = Directory(path);  
8.         return Scaffold(  
9.             body: FutureBuilder(  
10.                 future: listDirectoryContents(directory),  
11.                 builder: (  
12.                    BuildContext context,  
13.                     AsyncSnapshot<List<FileSystemEntity?>> snapshot,  
14.                 ) {  
15.                     if (snapshot.hasError) {  
16.                         return const Text('There was an error.');  
17.                     }  
18.  
19.                     if (!snapshot.hasData) {  
20.                         return const Text('There was no data');  
21.                     }  
22.                     return ListView.builder(  
23.                         itemCount: snapshot.data!.length,  
24.                         itemBuilder: (BuildContext context, int index) {
```

```
25.         final FileSystemEntity file = snapshot.data![index]!;
26.         final String filename = file.path.split(path)[1];
27.         if (file is File) {
28.             return Text(
29.                 '📄 $filename',
30.             );
31.         }
32.
33.         if (file is Directory) {
34.             return Text(
35.                 '📁 $filename',
36.             );
37.         }
38.         return const SizedBox();
39.     },
40. );
41. },
42. ),
43. );
44. }
45.
46. Future<List<FileSystemEntity>> listDirectoryContents(Directory
dir) {
47.     final List<FileSystemEntity> files = <FileSystemEntity>[];
48.     final Completer<List<FileSystemEntity>> completer =
```

```
49.         Completer<List<FileSystemEntity>>());
50.     final Stream<FileSystemEntity> lister = dir.list(recursive:
51.         false);
52.     lister.listen(
53.         (file) async => files.add(file),
54.         onError: (Object e) => log('$e'),
55.         onDone: () => completer.complete(files),
56.     );
57.     return completer.future;
58. }
59. }
```

This is only possible by leveraging the power of `dart:io` and its ability to read files and list directory contents. Let us expand this example further so that we can print out the contents of files that we can read. First, let us update the widget to read the contents of the file using a method we will create in a moment. We already have the ability to list the directory contents, so we can re-use that code:

```
1. class My StatelessWidget extends StatelessWidget {
2.     const My StatelessWidget({super.key});
3.
4.     @override
5.     Widget build(BuildContext context) {
6.         const String path = "/";
7.         final Directory directory = Directory(path);
8.
9.         return Scaffold(
```

```
10.     body: FutureBuilder<
11.           future: listDirectoryContents(directory),
12.           builder: (
13.            BuildContext context,
14.             AsyncSnapshot<List<FileSystemEntity?>> snapshot,
15.           ) {
16.             if (snapshot.hasError) {
17.               return const Text('There was an error.');
18.             }
19.
20.             if (!snapshot.hasData) {
21.               return const Text('There was no data');
22.             }
23.             return ListView.builder(
24.               itemCount: snapshot.data!.length,
25.               itemBuilder: (BuildContext context, int index) {
26.                 final FileSystemEntity file = snapshot.data![index]!;
27.                 final String filename = file.path;
28.
29.                 if (file is File) {
30.                   return TextButton(
31.                     style: const ButtonStyle(
32.                       alignment: Alignment.centerLeft,
33.                     ),
```

```
34.          onPressed: () async {
35.            final String? contents = await
36.              readFile(File(filename));
37.            if (contents != null) {
38.              log(contents);
39.            },
40.            child: Text('📄 $filename'),
41.          );
42.        }
43.      ),
44.      if (file is Directory) {
45.        return Text('📁 $filename');
46.      }
47.      return const SizedBox();
48.    },
49.  );
50. },
51. ),
52. );
53. }
54. }
```

Next, we can create the method we will use to read the contents of the file. Note that this code exists within our widget, just like our `listDirectoryContents` method:

1. Future<String?> readFile(File file) async {

```
2.     String? fileContents;  
3.     try {  
4.         fileContents = await file.readAsString();  
5.     } catch (e) {  
6.         log('Unable to read file contents');  
7.         return null;  
8.     }  
9.     return fileContents;  
10. }
```

Now, we can tap on any file in the directory listing to log its contents to the debug console. We can just as easily write to a file:

```
1. final File logFile = File('log.txt');  
2. IOSink sink = logFile.openWrite();  
3. sink.write('FILE ACCESSED ${DateTime.now()}\\n');  
4. await sink.flush();  
5. await sink.close();
```

And if you do not want to completely overwrite the contents of the file you are writing to:

```
1. IOSink sink = logFile.openWrite(mode: FileMode.append);
```

Or if you are trying to write binary data (such as an image or some other type of data), use the following:

```
1. sink.add(List<int> data);
```

We can list files and directories, read file contents, and write files, among many other operations by using `dart:io`, so what is the catch? It turns out, `dart:io` is *only* available on non-Web platforms. This makes sense if we think about it. We cannot read the contents of a directory from a Web browser. (That would be a *massive* security risk, and browsers do not allow you to do

this for such a reason.) Likewise, we cannot arbitrarily read and write files directly on the Web. Again, this would be a huge security risk, so browsers do not allow it. That means that despite all the myriad benefits of using `dart:io`, we are strictly limited in where we can deploy it.

This is not the only package that is platform dependent. Take, for example, the package `image_picker_for_web`, which is *only* available for the Web. When deciding the functionality of your application, it is important to explore the packages that you will be using to build said application. If the packages you want to use will not support a platform you want to target, it is important to understand that early in the process.

App Icons and Splash Screens

Branding your application with an appealing app icon (see *figure 10.1*) and splash screen is a vital finishing touch for every application. Each platform is a bit different, with varying requirements. Let us start with the app icon requirements, then we will talk about how to implement them before moving on to a splash screen:

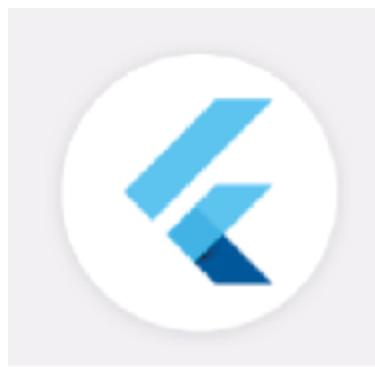


Figure 10.1: Flutter's default app icon on Android

The app icon requirements change from time to time as Google, Microsoft, Apple, and others update their design guidelines. It is important to stay informed about these design guideline changes so your app does not feel old or out of place to users. Here are some links to the app icon guidelines for various platforms:

- Google (Android):

<https://developer.android.com/distribute/google-play/resources/icon-design-specifications>

- Apple (iOS):
<https://developer.apple.com/design/human-interface-guidelines/foundations/app-icons>
- Microsoft (Windows):
<https://learn.microsoft.com/en-us/windows/apps/design/style/iconography/app-icon-design>
- Ubuntu (Linux):
<https://wiki.ubuntu.com/AppStream/Icons>

After you have designed an icon, you will need to tell each native platform you are targeting how to find your image. The official Flutter docs are useful in this, as they offer some additional guidance. You can find them here: <https://docs.flutter.dev/development/ui/assets-and-images#updating-the-app-icon>. Fortunately for us, Flutter includes default app icons. By simply overwriting the existing files while maintaining the same filename, our icons will be updated automatically. If you are changing the icon for Windows, you will need to navigate to the `<your_project>\windows\runner\resources` directory where you will find the `app_icon.ico` file. If you do not know how to create a `.ico` file, a quick online search ought to point you in the right direction (keywords: *png to ico* or *jpg to ico* should suffice). If you are looking to change the macOS icon, look in the `<your_project>/macos/Flutter/Runner/Assets.xcassets/AppIcon.appiconset` directory. Finally, for Web, look in the `web` directory for the `favicon.png` file. There are also additional icons of larger sizes in the `icons` subdirectory, which you will want to update. Provided you choose not to rename your app icon files, no further configuration is required.

Creating all these different files by hand can be quite tedious and prone to human error. You may forget to update one of the images, get the dimensions wrong, or have any number of other small issues, which could have a large impact on the final product. Rather than doing it all by hand, we can use the `flutter_launcher_icons` package. The documentation is quite straightforward and easy to follow, with plenty of examples in the git repository. Be sure to check this package out when it comes time to set your icons!

Let us look at splash screens next. Like before, here are some resources for designing splash screens on various platforms:

- Google (Android):
<https://developer.android.com/develop/ui/views/launch/splash-screen>
- Apple (iOS):
<https://developer.apple.com/design/human-interface-guidelines/patterns/launching#launch-screens/>
- Web:
<https://docs.flutter.dev/development/platform-integration/web/initialization>
- Desktop (Windows/macOS). Note: At time of writing, these platforms are as-of-yet unsupported. Check this GitHub issue for an up-to-date status: **<https://github.com/flutter/flutter/issues/41980>**

The requirements for each platform are as different as the app icon requirements, so reading over the documentation for the platform(s) you are targeting is mandatory. Apple, for example, requires you to have an Xcode storyboard. The default included one is `LaunchScreen.storyboard`, which can be customized any way you choose in Xcode. Android is quite different, however. You will be looking at the `AndroidManifest.xml` file to configure the activity style, which is what defines the look and feel of the splash screen. In addition to this, you will also be modifying the `styles.xml` file, which is where the styles are defined. Then, you will likely be creating image assets that the style refers to. The Flutter documentation has instructions on what to modify for each platform, here: **<https://docs.flutter.dev/development/ui/advanced/splash-screen>**.

Much like with the app icon, it is possible to use a package to help generate the splash screens. The `flutter_native_splash` package is highly configurable, allowing platform-specific configurations in addition to a global configuration. The configuration lives in `pubspec` or a separate, custom configuration file. Once configured, you can run the plugin from the command line to automatically generate the splash screens for you. This is a much easier way to implement a splash screen, so be sure to check out the package.

Material, Cupertino, Yaru, and Fluent UI (oh my!)

By now, you are familiar with both **Material** and **Cupertino** widgets. If you need a refresher, look way back in *Chapter 5: Handling User Input* at some of the difference between **Material** and **Cupertino** input widgets. These widgets are great for Android and iOS/macOS development, but did you know that they are not the only styles available to us?

If you have ever used Ubuntu Linux, you will be familiar with the **Yaru** style. Look at the screenshot to see how this style differs from **Material** and **Cupertino** (*figure 10.2*).

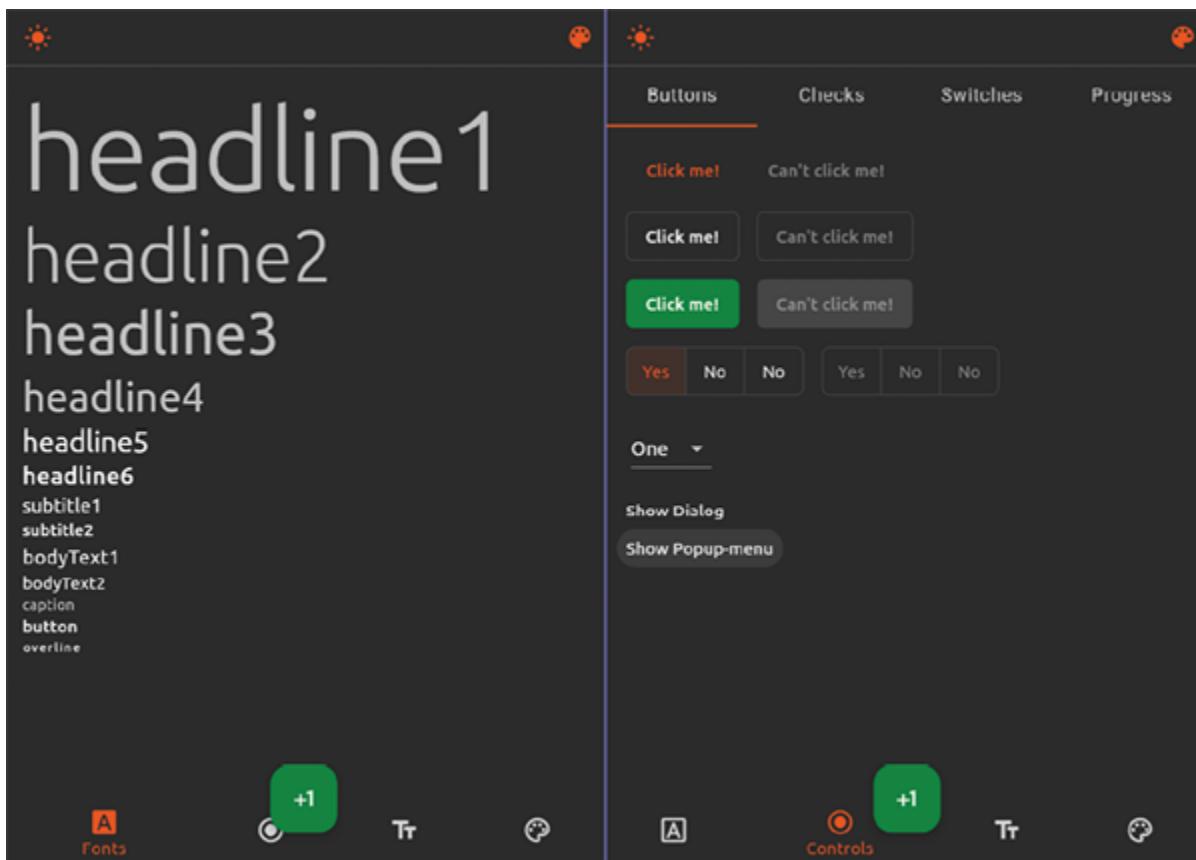


Figure 10.2: Yaru (Ubuntu) styled widgets

The **Yaru** package works by wrapping your **MaterialApp** widget with a **YaruTheme** widget, which has a builder that provides the appropriate themes for you to use in your **MaterialApp**. After importing the **Yaru** package, the setup is quite simple:

```
1. YaruTheme(  
2.   builder: (context, yaru, child) {  
3.     return MaterialApp(  
4.       theme: yaru.theme,  
5.       darkTheme: yaru.darkTheme,  
6.       home: Scaffold(  
7.         appBar: AppBar(  
8.           title: Text('Yaru Theme'),  
9.         ),  
10.        body: Container(),  
11.      ),  
12.    );  
13.  }  
14.);
```

From there on, you can simply use the Material widgets as normal to build your application. Since Yaru modifies the theme, there is no need to use custom widgets to reap the benefits of the package!

Maybe Yaru is not your style, though. Perhaps you prefer the Gnome desktop's Adwaita style (*figure 10.3*). There is certainly a package for that, too. Look no further than `libadwaita`. This package comes with replacement widgets, rather than a theme, however.

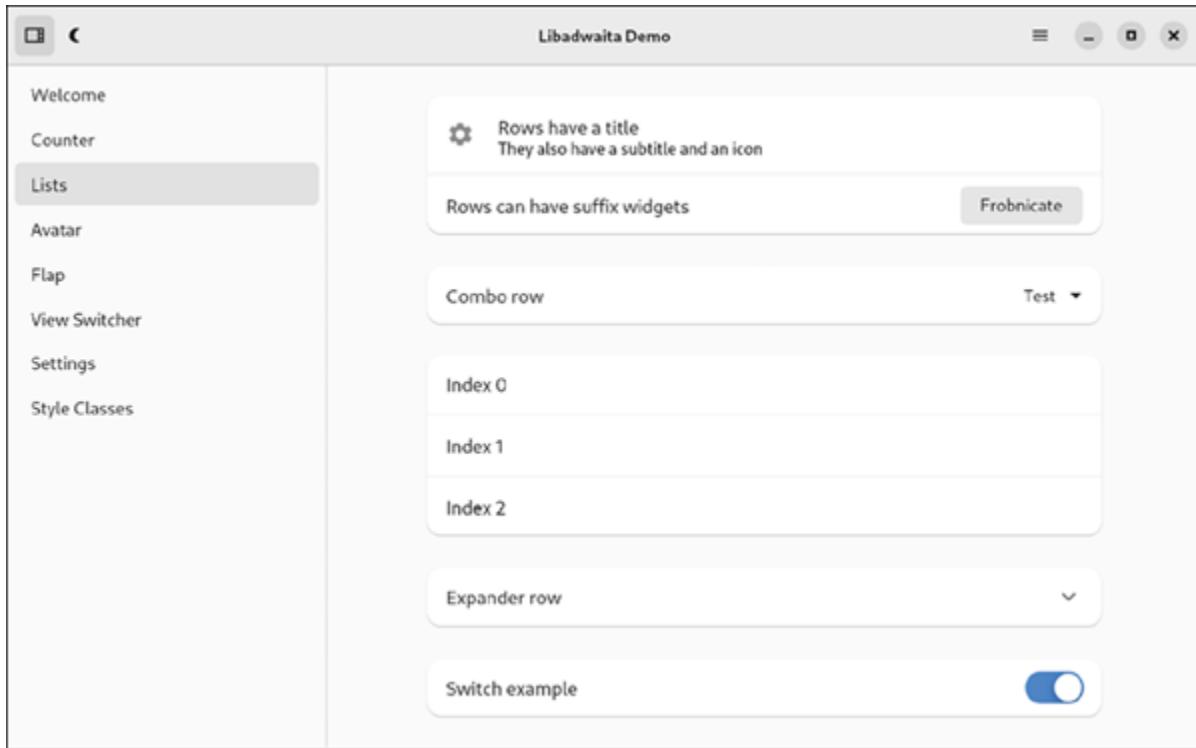


Figure 10.3: The Gnome Adwaita style

If you prefer Yaru's method of building a theme rather than replacing the widgets, try out the `adwaita` package. The setup is similar in that it builds a theme; however, it requires you to create a `ValueListenableBuilder` widget, instead:

```
1. class MyApp extends StatelessWidget {  
2.     final ValueNotifier<ThemeMode> themeNotifier = ValueNotifier(  
3.         ThemeMode.light  
4.     );  
5.     MyApp({super.key});  
6.     @override  
7.     Widget build(BuildContext context) {  
8.         return ValueListenableBuilder<ThemeMode>(  
9.             builder: (context, value, _) {  
10.                 return Scaffold(
```

```
11.         valueListenable: themeNotifier,
12.         builder: (_, ThemeMode currentMode, __) {
13.             return MaterialApp(
14.                 theme: AdwaitaThemeData.light(),
15.                 darkTheme: AdwaitaThemeData.dark(),
16.                 debugShowCheckedModeBanner: false,
17.                 home: MyHomePage(themeNotifier:
18.                     themeMode: currentMode,
19.                 );
20.             },
21.         );
22.     }
23. }
```

Here is what the `adwaita` package looks like once you have got it all set up (*figure 10.4*):

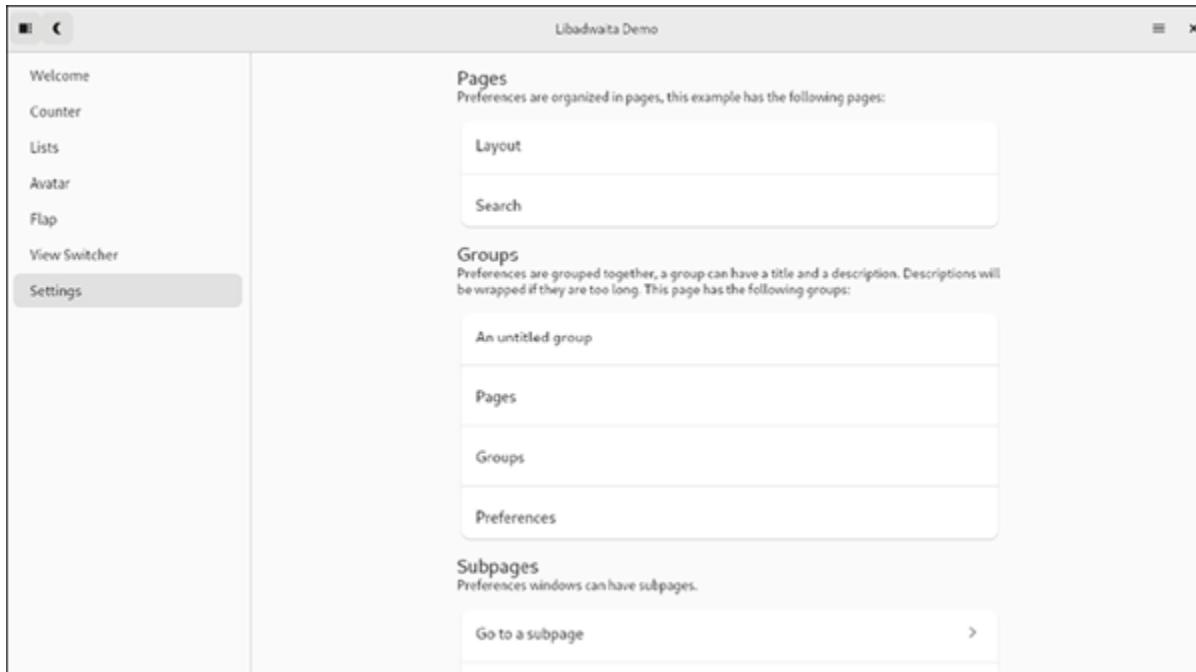


Figure 10.4: Material widgets styled using the Adwaita package

These styles are all beautiful, clean, concise, and easy to use. However, they only truly feel at home on a Linux desktop. If you are targeting a Windows environment, you will probably want something that feels more at home by using Microsoft's Fluent style.

Microsoft has published the Fluent design guidelines for developers to use when creating applications on the Windows platform. You can find the documentation here: <https://docs.microsoft.com/en-us/windows/uwp/design>. This style has been in use for the past couple of versions of Microsoft Windows and does not appear to be going anywhere any time soon. The package `fluent_ui` is an unofficial implementation of the Fluent design system, based on the official documentation. Check out the showcase to see what kind of widgets are available, what they look and feel like, and what their Material-equivalent widget is. You can find the showcase here: https://bdlukaa.github.io/fluent_ui/. Here is a sampling of what is on offer (*figure 10.5*):

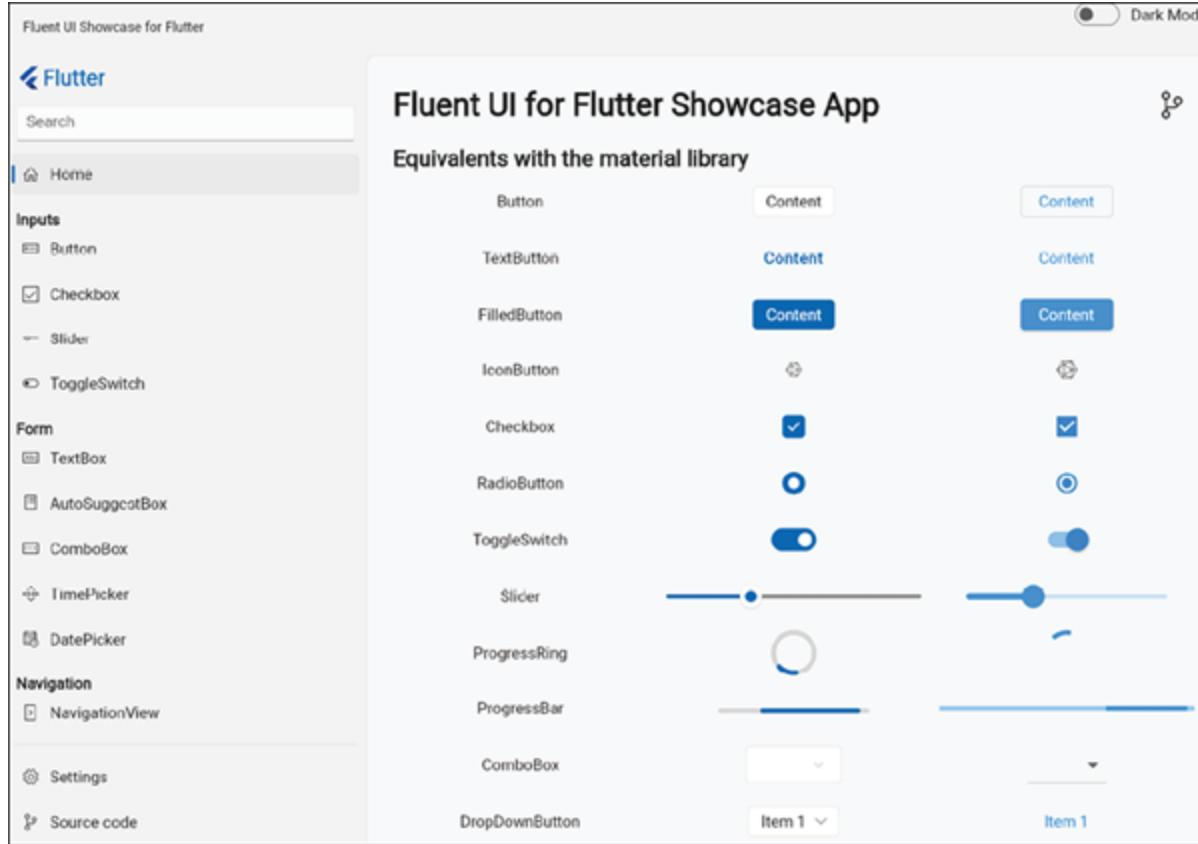


Figure 10.5: A selection of widgets available in the Fluent UI package

These widgets will feel right at home on a Windows computer. There is even a very similar package for macOS, called `macos_ui`. Check out the showcase screenshot (*figure 10.6*) and then look at the gallery for yourself here: <https://groovinchip.github.io/macossui/#/>.

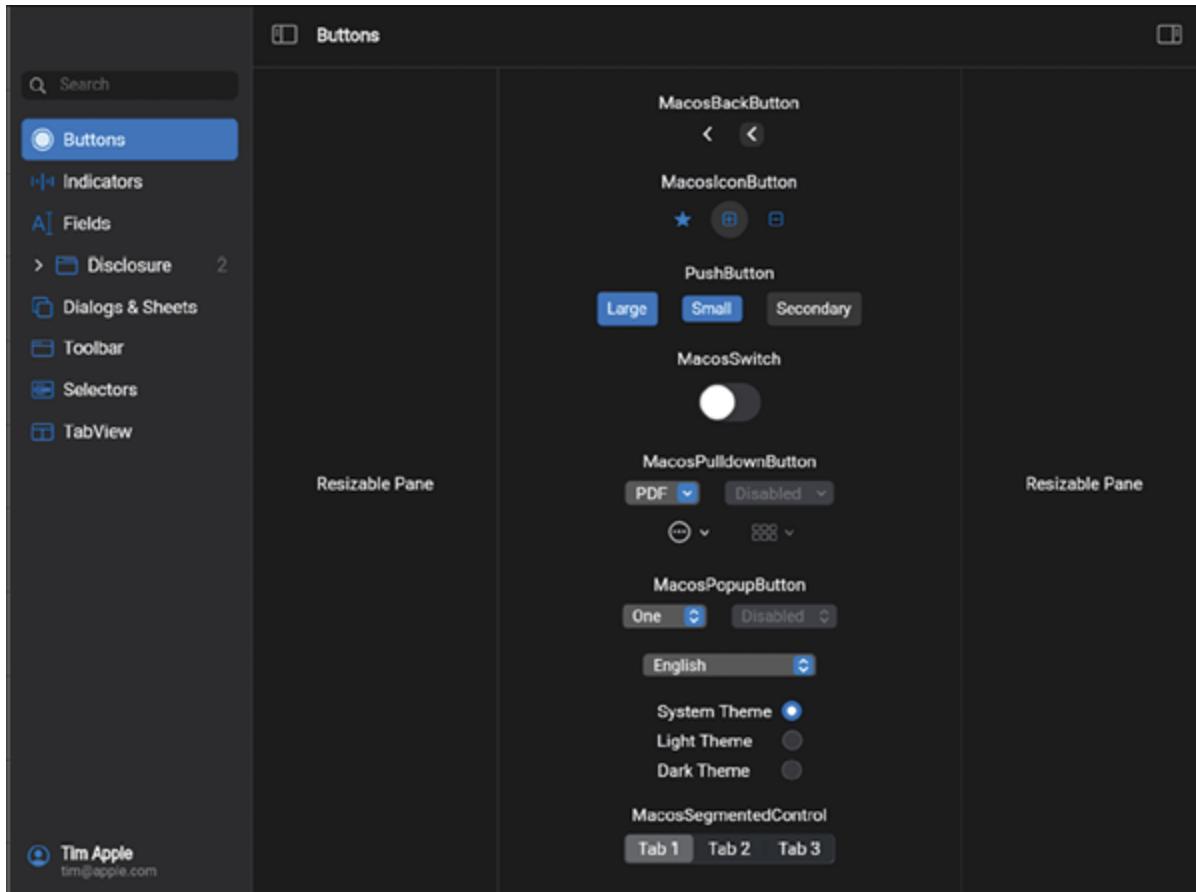


Figure 10.6: macOS style widgets from the macos_ui package

The style of your application's widgets matters. Whichever platform you end up on, you want the application to feel like it belongs there, with widgets that feel native to the environment they are running in. Even with a consistent style for the platform you are running on, there are still other considerations.

Platform-Specific Navigation Paradigms

We have all used software on computer, on the Web, and on our phones. There are certain navigation paradigms used on each platform that meet an agreed-upon standard. A Web page might have a sticky navigation bar across the top, whereas a desktop application may have a menu bar or sidebar with navigation options. Mobile applications have seen just as many paradigms as the Web and desktop, from a *hamburger menu* to a tab bar and iterations in between. The tab bar of a mobile app simply does not work on a larger screen. That is just not how people have been trained to interact

with their devices. Likewise, a sidebar such as a type you may find on the desktop or on a website would simply take up too much space on a mobile app, creating a navigation system that feels wrong and out of place.

The differences do not stop there, however. Let us look, specifically, at an Android/iOS paradigm: the back button. Since its inception, Android has had a universal back button, starting first as a hardware key (*figure 10.7*), later as a software button, and eventually as a gesture:



Figure 10.7: Note the hardware back button on the HTC Desire, circa 2010. Source: Wikipedia

Android's system-level back button (be it a button or gesture) is something any Android user is intimately familiar with. When the user navigates to a new screen, the back button will get them back to where they came from. At the application's main screen, the back button will exit the application. In fact, if you have navigated several screens deep into an application, repeatedly pressing the back button will eventually get you back to the main screen of the app, then exit the app, returning you to your phone's home screen.

Apple's iOS devices operate a bit *differently* in this regard. Apple opted not to provide any system-level back-button functionality within iOS, choosing instead to leave the option up to developers. You may find a **Done** text button, a chevron icon button, or simply choose to tap the icon of the already-selected tab to go back. Recently, Apple even introduced a swipe-from-the-side-to-go-back paradigm, although it is not always supported, even on their own applications. When it is, it is inconsistent as to which side of the screen you can swipe in from. Without a consistent mechanism,

Apple has created somewhat of a navigational nightmare for iPhone users. Unfortunately, this is the world we live in.

Now, we are not here to start a culture war. The Android versus iOS debate has been raging since 2009 and is not likely to end any time soon. The fact is, if you are an iOS user, you probably do not even *notice* the inconsistencies in navigation styles from app to app. And if you are on Android, you do not think twice about reaching for that universal back button or gesture. As an app developer, you will need to understand both platforms, however. With a universal back button on Android, it is impossible to get yourself *stuck* on a screen with no way back. On iOS, you must be careful to add back buttons for your users.

iOS has another paradigm that does not exist on Android: stacked navigation in tab views. The tab bar can be found both on iOS (*figure 10.8*) and Android (*figure 10.9*):

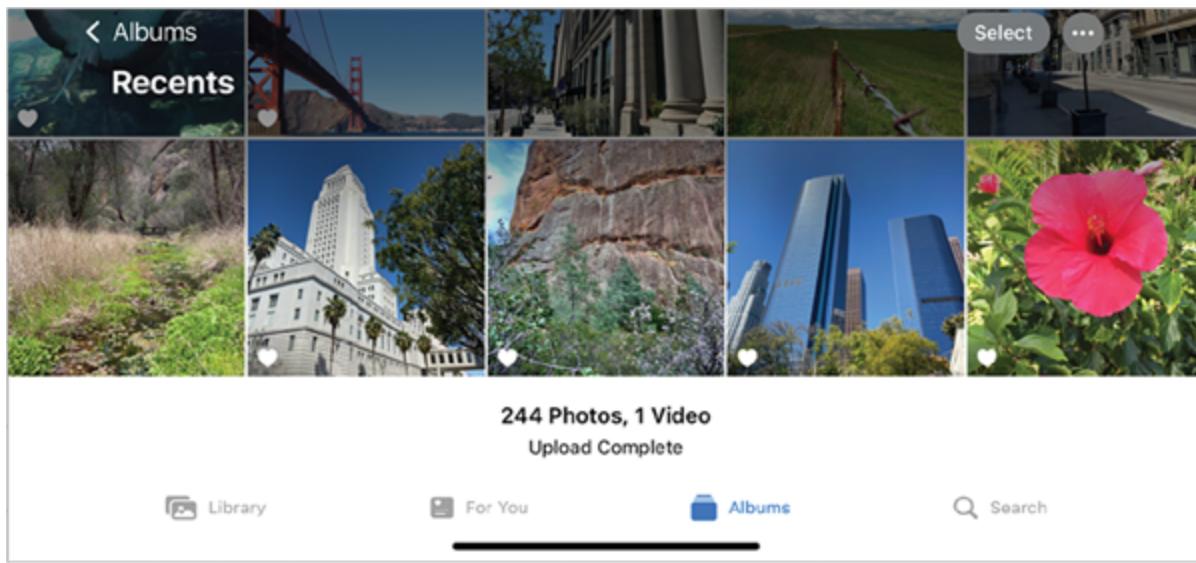


Figure 10.8: An example of the iOS tab bar. Source: Apple

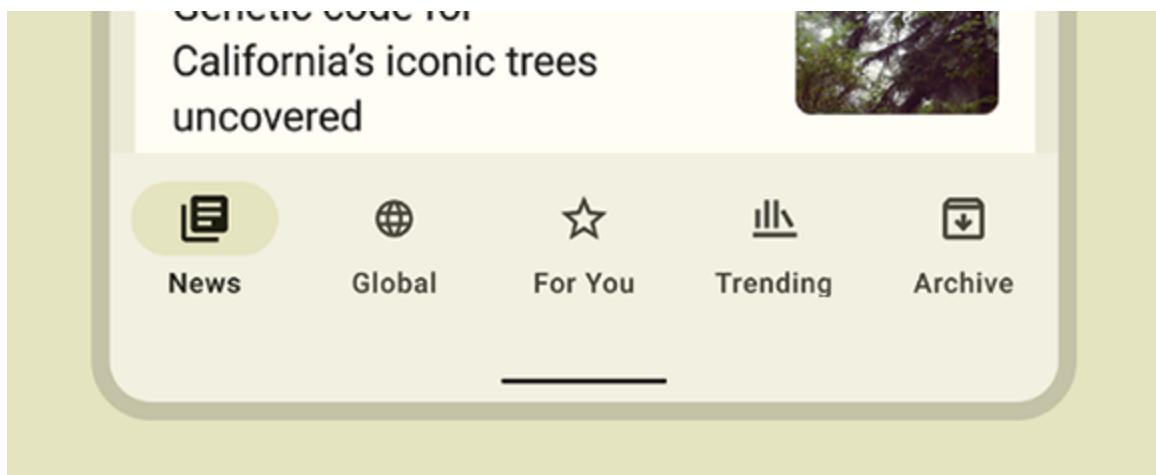


Figure 10.9: An example of a Material Design style tab bar. Source: Google

Ok, strictly speaking, the paradigm *can* exist. In practice, you generally will not find it, though. So, what are we talking about when we say, *stacked navigation*? To answer that, let us look at a couple of code examples. First, let us examine a typical tab bar using `BottomNavigationBar`, part of the Material package:

```
1. class MyStatefulWidget extends StatefulWidget {  
2.   const MyStatefulWidget({super.key});  
3.  
4.   @override  
5.   State<MyStatefulWidget> _MyStatefulWidgetState();  
6. }  
7.  
8. class _MyStatefulWidgetState extends State<MyStatefulWidget> {  
9.   int _selectedIndex = 0;  
10.  
11.  static const List<Widget> _widgetOptions = <Widget>[  
12.    Text('Home'),  
13.    Text('Business'),
```

```
14.      Text('School'),  
15.    ];  
16.  
17.    void _onItemTapped(int index) {  
18.      setState(() {  
19.        selectedIndex = index;  
20.      });  
21.    }  
22.  
23.    @override  
24.    Widget build(BuildContext context) {  
25.      return Scaffold(  
26.        body: Center(  
27.          child: _widgetOptions.elementAt(_selectedIndex),  
28.        ),  
29.        bottomNavigationBar: BottomNavigationBar(  
30.          items: const <BottomNavigationBarItem>[  
31.            BottomNavigationBarItem(  
32.              icon: Icon(Icons.home),  
33.              label: 'Home',  
34.            ),  
35.            BottomNavigationBarItem(  
36.              icon: Icon(Icons.business),  
37.              label: 'Business',  
38.            ),
```

```
39.         BottomNavigationBarItem(  
40.             icon: Icon(Icons.school),  
41.             label: 'School',  
42.         ),  
43.     ],  
44.     currentIndex: _selectedIndex,  
45.     selectedItemColor: Colors.purple[800],  
46.     onTap: _onItemTapped,  
47. ),  
48. );  
49. }  
50. }
```

When you tap a tab (Line 46), the body of the `Scaffold` updates by building a new widget (Line 27). The widget, which was previously the body of the `Scaffold`, is disposed of and a new widget is built. As we know, when a widget is disposed of, so is its state. If the widget that is being used as the body of the `Scaffold` contains a navigation stack of its own, that navigation stack will be disposed of, too. Therefore, tapping a tab on the tab bar causes the single navigation stack to be discarded and replaced. Now, let us look at the `CupertinoTabScaffold`, `CupertinoTabBar`, and `CupertinoTabView` to see how it differs:

1. class CupertinoTabBarApp extends StatelessWidget {
2. const CupertinoTabBarApp({super.key});
- 3.
4. @override
5. Widget build(BuildContext context) {
6. return const CupertinoApp(

```
7.         home: CupertinoTabBarExample(),  
8.     );  
9. }  
10.  
11.  
12. class CupertinoTabBarExample extends StatelessWidget {  
13.     const CupertinoTabBarExample({super.key});  
14.  
15.     @override  
16.     Widget build(BuildContext context) {  
17.         return CupertinoTabScaffold(  
18.             tabBar: CupertinoTabBar(  
19.                 items: const <BottomNavigationBarItem>[  
20.                     BottomNavigationBarItem(  
21.                         icon: Icon(CupertinoIcons.star_fill),  
22.                         label: 'Home',  
23.                     ),  
24.                     BottomNavigationBarItem(  
25.                         icon: Icon(CupertinoIcons.clock_solid),  
26.                         label: 'Recents',  
27.                     ),  
28.                     BottomNavigationBarItem(  
29.                         icon: Icon(CupertinoIcons.person_alt_circle_fill),  
30.                         label: 'Contacts',  
31.                     ),
```

```
32.           ],
33.           ),
34.           tabBuilder: (BuildContext context, int index) {
35.             return CupertinoTabView(
36.               builder: (BuildContext context) {
37.                 return Center(
38.                   child: Text('Content of tab $index'),
39.                 );
40.               },
41.             );
42.           },
43.         );
44.       }
45.     }
```

These three widgets work in conjunction to create something a bit different. The `CupertinoTabBar` works just like the `BottomNavigationBar`, but the `CupertinoScaffold` differs from the `Scaffold` in one key aspect: there is a `tabBuilder`. Since each tab is built via the `CupertinoTabScaffold`, they are not disposed of or replaced when a new tab is tapped. The state of the tab's body widget is retained, thereby allowing you to navigate deep into each tab's screens, switch to a different tab, then back to the first without losing your navigation stack.

This feature is undoubtedly familiar to iOS users but would be relatively foreign to Android users. For this reason, it is important to understand the differences between each platform's paradigms on which you intend to release your application. Furthermore, it is important to *test* your application on each platform. Designing, testing, and building for, say, *only* iOS would potentially leave your application feeling like it is an afterthought on Android. Not only does this erode consumer confidence, but

it simply does not make for a great experience, and when it comes to consumer confidence, the success of your app will hinge upon it. Why alienate half the potential users simply because you have not adequately tested on a platform?

Using Packages to Build Cross-Platform UIs

Using a `LayoutBuilder` will only get you part of the way to a truly cross-platform UI. Certainly, you can use it to build a responsive layout, but what about a layout that will adjust its widgets to match whether you are on iOS or Android? As we just discussed, maintaining consumer confidence is key to a successful application. Unless your application is designed by Apple (and even then, it is only barely acceptable), `Cupertino` widgets probably should not be in an Android application. Just the same, `Material` widgets would feel out of place on an iOS application. Would it not be nice if the application would adjust automatically, to match the platform it is on?

Fortunately, a package exists which will do just this: `flutter_platform_widgets`. This package will use the information from `Theme.of(context).platform` to build different widgets depending on whether it finds `TargetPlatform.android` or `TargetPlatform.ios`. However, this behavior can be overridden for any supported platform (Android, iOS, Web, macOS, Fuchsia, Windows, and Linux). Something important to keep in mind is that `flutter_platform_widgets` will only provide builders for the different platforms; what you actually *do* with those builders is entirely up to you: you need not use `Cupertino` or `Material` at all. Let us look at some code:

```
1. class MyApp extends StatelessWidget {  
2.   const MyApp({super.key});  
3.  
4.   @override  
5.   Widget build(BuildContext context) {  
6.     return const PlatformApp(  
7.       home: MyStatelessWidget(),  
8.     );
```

```
9.     }
10. }
11. 
12. class My StatelessWidget extends StatelessWidget {
13.   const My StatelessWidget({super.key});
14. 
15.   @override
16.   Widget build(BuildContext context) {
17.     return Scaffold(
18.       body: Center(
19.         child: PlatformWidget(
20.           cupertino: (_, __) => Column(
21.             mainAxisAlignment: MainAxisAlignment.center,
22.             children: const [
23.               Icon(CupertinoIcons.flag),
24.               Text('Using Cupertino widgets'),
25.             ],
26.           ),
27.           material: (_, __) => Column(
28.             mainAxisAlignment: MainAxisAlignment.center,
29.             children: const [
30.               Icon(Icons.flag),
31.               Text('Using Material widgets'),
32.             ],
33.           ),
```

```
34.           ),
35.           ),
36.       );
37.   }
38. }
```

In this example, if the platform is detected as iOS (or macOS), a `CupertinoIcons` flag will be displayed (along with a message telling you that we are using `Cupertino` widgets.) Likewise, if Android is detected, a `Material Icons` flag will be displayed. What happens if you are testing on an Android device (or writing your code on a non-Apple computer, and simply do not have access to an iOS simulator?) We can set a style with an option, although note that this is not a replacement for testing on the actual platform:

```
1. class MyApp extends StatelessWidget {
2.   const MyApp({super.key});
3.
4.   @override
5.   Widget build(BuildContext context) {
6.     return PlatformProvider(
7.       initialPlatform: TargetPlatform.iOS,
8.       builder: (context) => const PlatformApp(
9.         home: MyStatelessWidget(),
10.        ),
11.      );
12.    }
13. }
```

The `PlatformProvider` widget will also let us override styles for different platforms. For example, we could tell the application that when running on the Web, it should also use the `Cupertino` style:

```
1. class MyApp extends StatelessWidget {  
2.   const MyApp({super.key});  
3.  
4.   @override  
5.   Widget build(BuildContext context) {  
6.     return PlatformProvider(  
7.       settings: PlatformSettingsData(  
8.         platformStyle: const PlatformStyleData(  
9.           web: PlatformStyle.Cupertino,  
10.          ),  
11.          ),  
12.          builder: (context) => const PlatformApp(  
13.            home: MyStatelessWidget(),  
14.            ),  
15.          );  
16.    }  
17. }
```

With some basic configuration out of the way, you will undoubtedly be able to create a truly native-feeling application, regardless of what platform your code runs on. There are many available widgets in `flutter_platform_widgets` which take some of the legwork out of building a multi-platform UI. By leveraging some of the built-in platform widgets provided by the package, you can reduce the amount of code needed to write and maintain a feature. The less code you have, the less that can go wrong.

Conclusion

Not every application needs to have a different look and feel, depending on the platform it runs on. However, many will require adjustments, sometimes slight and sometimes more extreme. Being able to dynamically adjust for a variety of screen sizes, orientations, and native platforms takes just a little bit of extra work but generally tends to pay off in the end. If you have taken away nothing else from this chapter, take this away: it is important to understand the differences between the platforms your users will be running your application on. Build that trust and maintain it. Your users will thank you.

In the upcoming chapter, we are going to talk about what to do when things go wrong. We will look at the tools and techniques we have available to us when we need to debug, troubleshoot, and figure out why there might be performance problems, such as a laggy UI, frame drops when scrolling, jitter, and so on.

Questions

1. What is a *responsive layout*?
2. Which widget can we use to build a responsive layout?
3. When building a responsive layout, what considerations should you take when choosing how to build your screen-size-dependent widgets? What are some actions you can take to ensure you do not forget about accidentally removing important functionality?
4. What is `dart:io` used for?
5. What platform(s) can `dart:io` not be used on?
6. How do you update your application's icon?
7. Where can you find resources for designing a splash screen?
8. What packages can be used to help ensure you have properly updated your app icon and splash screen?
9. What are some packages that can be used to match the look and feel of a desktop operating system, such as Windows or Linux?
10. What are the key differences between Android and iOS in terms of navigation principles?

11. What is *stacked navigation*?
12. What inherited widget contains information about the currently targeted platform?
13. What package allows you to easily build platform-adaptive layouts?

Further Reading

- Read more about what `dart:io` has to offer:
<https://dart.dev/guides/libraries/library-tour#dartio>

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

Debugging, Troubleshooting, and Performance Considerations

Introduction

It is rare for software to come out the way we expect it to the first time we write our code. Rather than being a linear process of *write code* → *have working application*, it is a cycle of planning, writing, debugging, testing, and iterating. In this chapter, we are going to focus on the debugging, testing, and iterating portions of the cycle, with a focus on learning what to do when our code goes wrong and improving the overall responsiveness of our applications.

Structure

In this chapter, we will discuss the following topics:

- Debugging and troubleshooting concepts
- Logging output to the debug console
- Setting breakpoints and exploring runtime variables
- Handling exceptions
- Introduction to Flutter DevTools

- Defining performance issues
- Exploring profile mode
- When a ListView is not a ListView
- Optimizing widgets

Objectives

After completing this chapter, you will understand how to debug and troubleshoot your code when things are not going as expected. You will learn how to view variables as your code runs and manipulates them, and how to log output to a debug console. You will have learned about some of the different compilation modes for your application and the developer tools that ship with Flutter to take advantage of each of them. Finally, you will learn how to optimize your code to deliver a buttery-smooth experience for your users.

Debugging and Troubleshooting Concepts

Even the most seasoned developers spend a lot of time debugging and troubleshooting their code. It is a rare occurrence when your code works flawlessly the first time, so learning how to solve the problems you will come across is very important. Let us first start by defining what debugging and troubleshooting are.

Troubleshooting is solving a problem. You can think of troubleshooting as identifying and resolving a problem by fixing whatever issues prevented things from working correctly in the first place.

Debugging is the process of going step by step through a problem to identify and resolve any errors.

Every problem to be solved by debugging can be broken down into five steps:

1. Ensure the error is reproduceable. If the error is intermittent, you have not properly identified the error.
2. Be certain that the problem you are debugging is the problem you need to be solving.

3. Check all the obvious error sources. It is easy to waste time by debugging a *complex* problem that turns out to have a very simple and obvious solution.
4. Debug the problem: isolate the problem by dividing it into working versus non-working components. Think like a computer: step through the instructions one at a time to find where the operation fails.
5. If you are unable to identify the root cause of the problem, reassess the situation: what assumptions have you made that might be incorrect? Walk through the process again, taking notes and using knowledge learned from the last time.

Although these steps are seemingly quite simple, building the mindset and toolkit to approach and debug any problem takes time and practice. In each section of this chapter, we will discuss some of the different tools you can employ to help debug your Flutter project.

Logging Output to the Debug Console

Sometimes it is useful to print some messages to the console while we debug a problem. Let us look at some pseudo-code (a code-like representation, often non-functional, which is used to illustrate a solution or mock out a problem):

```
1. Future<MyObject?> getMyObject() async {  
2.   Map<String, dynamic> result = await getResult();  
3.   MyObject object = MyObject.fromJson(result);  
4.  
5.   if (object.name.isEmpty) {  
6.     return null;  
7.   }  
8.  
9.   return object;  
10. }
```

Our code will await a future result, build an object from it, check if the object's name property is null, and either returns a null object or the full object, respectively. What you might find is a situation in which this method *always* returns null, even when you are certain the object has a name.

The first and most obvious thing to do would be to ask Flutter to print out the name of the object, but how and where should we do that? First, we will need to import the `dart:developer` package to gain access to the `log()` method. Note: if you have imported the `dart:math` package, `log()` will already be defined as a method! In that case, do not fret, as you can use `import 'dart:developer' as dev;` to scope the package to a namespace. Then, use `dev.log()` to access the function. Do not worry about following along in your editor, since we have not defined `MyObject` or `getResult()`. Instead, use this as a thought experiment:

```
1. import 'dart:developer';
2.
3. Future<MyObject?> getMyObject() async {
4.   Map<String, dynamic> result = await getResult();
5.   MyObject object = MyObject.fromJson(result);
6.
7.   log("The object's name is '${object.name}'");
8.   if (object.name.isEmpty) {
9.     return null;
10. }
11.
12.   return object;
13. }
```

On Line 7, we have our log message. The rules for accessing variables in a log message are the same as printing them in a string: use a dollar sign to denote the variable and wrap the variable in curly braces if you need to access a property of the variable.

Now, imagine we ran the code again and checked the log message. Unfortunately, the log message never shows up. What is going on here? Let us add some more logging to see if we can figure out where things are breaking:

```
1. import 'dart:developer';
2.
3. Future<MyObject?> getMyObject() async {
4.   log('About to run getResult()');
5.   Map<String, dynamic> result = await getResult();
6.   log('The result is $result. Using it to create MyObject.');
7.   MyObject object = MyObject.fromJson(result);
8.   log('MyObject was created: $object');
9.
10.  log("The object's name is '${object.name}'");
11.  if (object.name.isEmpty) {
12.    return null;
13.  }
14.
15.  return object;
16. }
```

Now if we run the code, here is what we might see:

1. [log] About to run getResult()
2. [log] The result is {}. Using it to create MyObject.

It looks like we have isolated the problem using logging. Something about the `getResult()` method does not seem to be working. We will have to continue the troubleshooting with a different method, so let us head over to that method and figure out what is going on.

Setting Breakpoints and Exploring Runtime Variables

There are quicker and easier ways to debug than by inserting logging lines everywhere. Since you are unlikely to want to keep those lines in your code when you ship it, you will likely go back and delete them once you have fixed the problem. One way we can sidestep the whole process of writing and deleting code is by leveraging the tools of our Integrated Development Environment (IDE). In this case, our IDE is Visual Studio Code. Any decent and/or modern IDE will have similar functionality. While the details of how to use the tool changes from IDE to IDE, the concept is usually the same.

The concept we will be looking at is called the **breakpoint**. A breakpoint is a point at which you intentionally stop or pause code execution at a specific point to assist in debugging. By setting a breakpoint, we tell the IDE to run the code normally until it reaches the line of code we have set a breakpoint for. Then, the program will pause and display information about the current state of the application. You will have the ability to *step* through lines of code to look at what is happening in a much slower way than normal.

Let us take our code from before and add a breakpoint (*figure 11.1*). Once the breakpoint has been set, you will see a red dot next to the line number, indicating a breakpoint exists for that line of code. If the dot is grey instead of red, you will need to stop and restart your code. You will also see the breakpoint is added to the IDE's breakpoints list, including the filename and path of the file with a breakpoint, along with the line number. Breakpoints can be toggled on and off by clicking the red dot next to the line number and temporarily disabled by unchecking the box next to the breakpoint in the list. A temporarily disabled breakpoint will not pause the execution of the program but can be re-enabled easily later by checking the box.

A screenshot of the Visual Studio Code interface. On the left is a sidebar with icons for file operations, a search bar, and tabs for 'WATCH', 'CALL STACK', and 'BREAKPOINTS'. Under 'BREAKPOINTS', there is a list with a single entry: 'main.dart lib' with a red dot indicating a breakpoint is set. The main editor area shows Dart code:

```

Run | Debug | Profile
void main() {
  runApp(const MyApp());
}

Future<MyObject?> getMyObject() async {
  log('About to run getResult()');
  Map<String, dynamic> result = await getResult();
  log('The result is $result. Using it to create MyObject.');
  MyObject object = MyObject.fromJson(result);
  log('MyObject was created: $object');

  log("The object's name is '${object.name}'");
  if (object.name.isEmpty) {
    return null;
  }

  return object;
}

Future<Map<String, dynamic>> getResult() async {
  bool? future = await Future.delayed(
    Duration(seconds: 1),
    () => {
      ...
    }
  );
}

```

Two callout boxes highlight specific elements: one points to the red dot on the line 11 breakpoint with the text 'Breakpoints can be toggled on and off by clicking the dot.', and another points to the 'BREAKPOINTS' tab in the sidebar with the text 'Note the newly created breakpoint in the list.'

Figure 11.1: Setting a breakpoint in Visual Studio Code

Running our code with one or more breakpoints set will cause the program to pause once it hits the line(s) with the breakpoint(s). Once a line with a breakpoint has been reached, the debugger in the IDE will present us with some useful information. The current line that was being processed when a breakpoint was reached, will be highlighted. Additionally, the call stack will be populated with information about what code is currently being processed at the time the breakpoint has been reached. Each line (called a **stack frame**) shows what function was running when the previous line was invoked. Refer to *figure 11.2* for a visual of what happens when you reach a breakpoint.

A screenshot of the Visual Studio Code interface during debugging. The top status bar says 'demo' and 'Paused on breakpoint'. The main editor shows the same Dart code as Figure 11.1, but line 11 is highlighted with a yellow background, indicating it is the current line being executed. A callout box points to this line with the text 'The line which has the breakpoint has been highlighted'. Another callout box points to the toolbar above the editor with the text 'Buttons to resume, jump over, step into/out of code'. The bottom status bar shows the command 'Launching lib/main.dart on Windows in debug mode... lib/main.dart:1'. The bottom-left panel shows the 'CALL STACK' with the following entries:

- getMyObject package:demo/main.d...
- MyHomePage.build package:demo/...
- Show 18 More: from the Flutter framework
- Load More Stack Frames

Figure 11.2: A breakpoint has been reached.

You may have noticed when running your code before that there are several little arrows in the same bar where you can stop or restart your code. Typically, the first icon would be a pause icon (||); however, when the debug point is reached, it changes to a resume icon (ID). Let us talk about what each of the blue icons does, starting with the pause/resume icon.

The pause/resume icon is the most straightforward of the bunch. If your code is running as normal without triggering a breakpoint or pausing during an exception, you can press the button to pause processing and show all the same information that you would normally get from hitting a breakpoint. The resume icon resumes processing, regardless of how you entered a paused state. The next three icons we will look at are only available while code execution has been paused.

The second icon in the debugger toolbar is the **step over** icon. Pressing this button will run the next function in its entirety before pausing at the next function. The next two icons, the **step into** and **step out** icons, will run the next or previous line of code, respectively. This allows you to see the stack trace for each line of code, but also helps to isolate when a given line of code generates an error or other issue.

By using the step into and step out buttons, we can process the lines of code one-by-one until we come to the root of our problem. In *figure 11.3*, we see that the `getResult` method only returns an object with a `name` if the `future` is not `false`.

The screenshot shows a Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Project Bar:** RUN AND DEBUG, demo, framework.dart.
- Variables Sidebar:** Shows `Locals` with `future: false`. A tooltip "Variables currently being operated upon" is shown over the `future` variable.
- Code Editor:** Displays the following Dart code:

```
lib > main.dart > getResult
23
24 Future<Map<String, dynamic>> getResult() async {
25   bool? future = await Future.delayed(
26     Duration(seconds: 5),
27     () => Future.delayed(Duration(seconds: 1),
28       onTimeout: () {
29         return false;
30       },
31     ),
32   );
33   if (future != false) return {'name': 'my object'};
34   return {};
35 }
```
- Call Stack:** Shows `getResult` and `getMyObject`.
- Debug Console:** Shows "Restarted application in 776ms." and "[log] About to run getResult().

Figure 11.3: Using the step into and step out buttons, we can identify which line of code causes our problem.

Let us examine this code more closely:

```
1. Future<Map<String, dynamic>> getResult() async {  
2.     bool? future = await Future.delayed(  
3.         const Duration(seconds: 5),  
4.         ).timeout(  
5.             const Duration(seconds: 1),  
6.             onTimeout: () {  
7.                 return false;  
8.             },  
9.         );  
10.    if (future != false) return {'name': 'my object'};  
11.    return {};  
12. }
```

Here, we can see that the debugger tells us `future` is equal to `false`. Looking at the line of code which checks the value of the future, we confirm that a valid object will only be returned if `future` is equal to anything *other than* `false`. Looking at the preceding lines, we see that the future waits several seconds, but times out before it completes. When it times out, `false` is returned. By using the debugging tools, we have isolated the area of code where the problem exists.

Although our example is a bit silly and not something you would be likely to find in production code, it helps to illustrate some of the concepts you will need to use when debugging your code. In a more realistic scenario, imagine an API call with a timeout to prevent long-running network requests from causing issues, such as blocking widgets from showing a proper error state. Learning how to use the debug tools in your IDE is key to helping you solve any problem you come across. You can read more

about Visual Studio Code's debugger on their website, <https://code.visualstudio.com/docs/editor/debugging>.

Handling Exceptions

An exception occurs when something has gone wrong in our code and processing is unable to continue; otherwise, the program will crash. We can tell our code to **throw** exceptions when processing goes wrong, such as in this updated example from before:

```
1. Future<Map<String, dynamic>> getResult() async {  
2.     await Future.delayed(  
3.         const Duration(seconds: 5),  
4.         ).timeout(  
5.             const Duration(seconds: 1),  
6.             onTimeout: () {  
7.                 throw Exception('Operation timed out.');//  
8.             },  
9.         );  
10.    return {'name': 'my object'};  
11. }
```

Here, instead of setting the future to a boolean value, we assume it will complete successfully and do not even bother checking the condition before returning a value from the function. Instead, we tell the future to throw an exception if the code times out. See *figure 11.4* for how it looks when this happens:

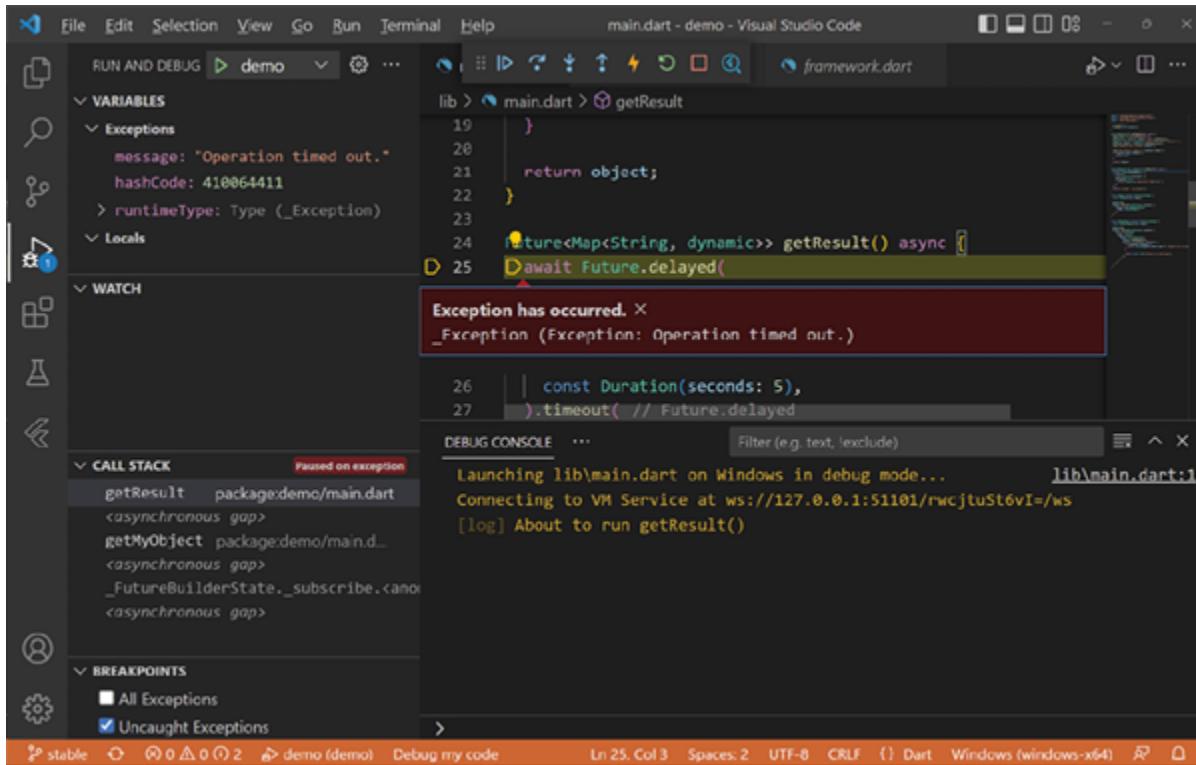


Figure 11.4: An exception has been thrown.

Now that we have an exception, we need to learn how to deal with them. One way we can deal with exceptions in our code is by using a `try/catch` block. The code within the `try` portion will run until an exception is thrown. In the case an exception is thrown, the code within the `catch` portion will run. Otherwise, the `catch` code will be skipped. In this example, we have wrapped the function call in a `try/catch` block:

```

1. Future<MyObject?> getMyObject() async {
2.   late Map<String, dynamic> result;
3.   try {
4.     result = await getResult();
5.   } catch (e) {
6.     log('An exception occurred while running `getResult`: $e');
7.     return null;
8.   }

```

```

9. 
10. MyObject object = MyObject.fromJson(result);
11. 
12. if (object.name.isEmpty) {
13.     return null;
14. }
15. 
16. return object;
17. }
```

Now, when we run our code, a log message is produced, but the exception does not stop the rest of our code from running (*figure 11.5*):

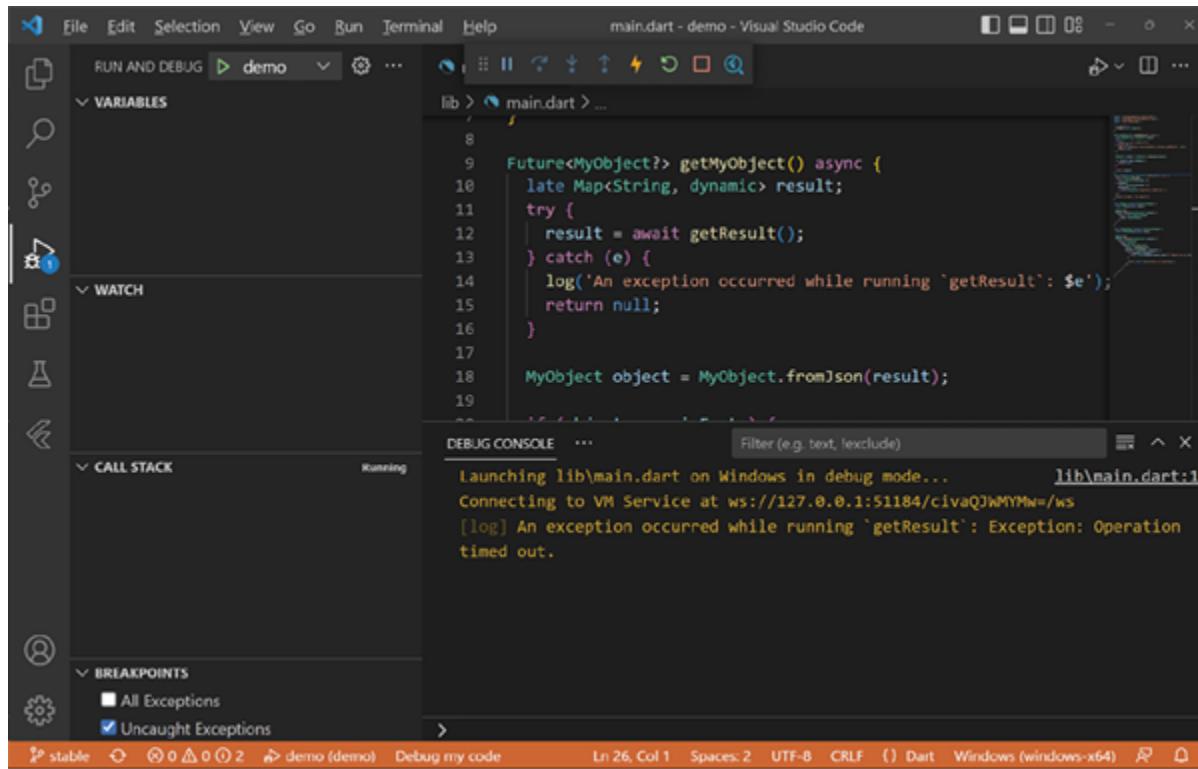


Figure 11.5: The exception has been handled, allowing our code to continue running.

Sometimes, however, using a `try/catch` can obfuscate problems entirely, leading to frustration while debugging. In this case, we included a log

message which logs the actual exception. If we had not done that, we would have no idea that an exception occurred. For this reason, you should consider using `try/catch` blocks sparingly. Instead, writing more robust code should prevent many exceptions from happening in the first place, being sure to run appropriate checks on variables throughout the process and handling any error states that could occur.

Introduction to Flutter's DevTools

Outside of your IDE's debugger, Flutter comes with a robust set of tools called **DevTools**. DevTools comes with many different components, from a visual widget tree, layout explorer, and widget details tree to tools that help you visualize when a widget is rebuilt, a network inspector, and the ability to tap on any widget to jump straight to it in the code and within DevTools. By default, opening the DevTools by pressing the icon (Q) in the toolbar will drop you into the Widget Inspector, side-by-side with your code. See *figure 11.6* for an example of the DevTools, after having been moved to its own tab:

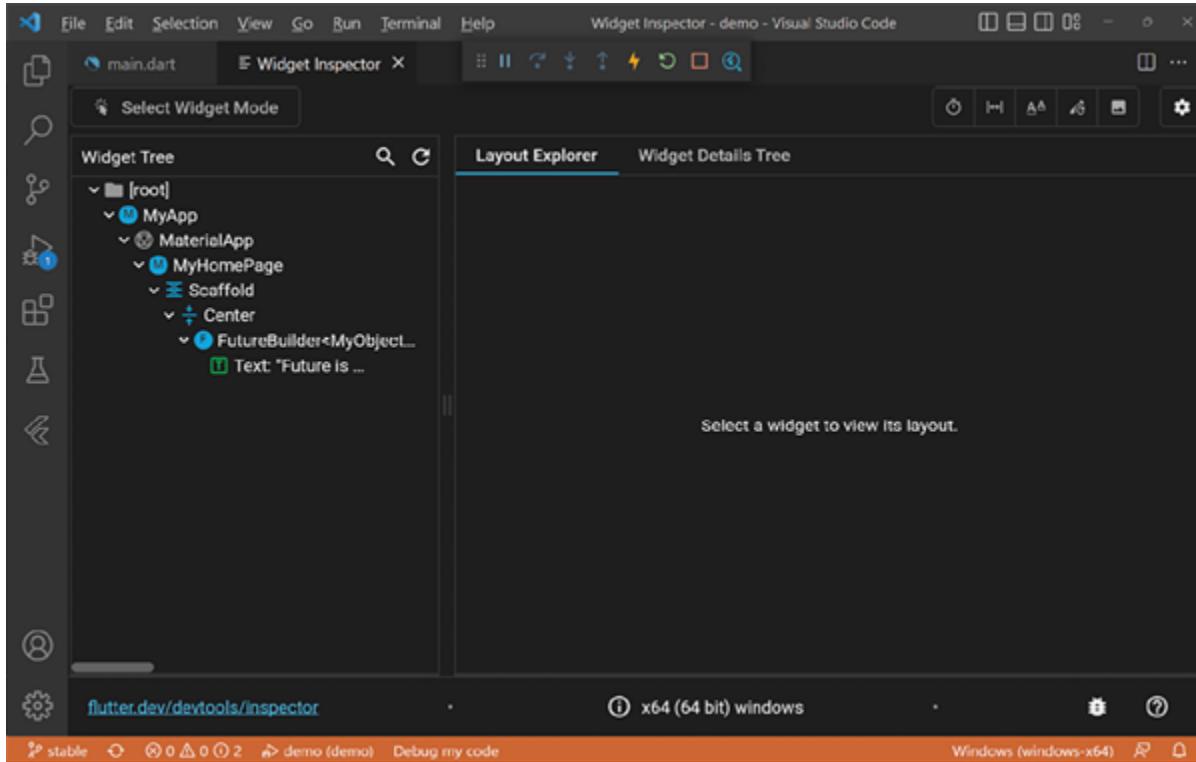


Figure 11.6: Flutter's DevTools

There are far too many tools available in DevTools for us to cover here, so let us look at some of the most common tools, starting with the layout explorer. Selecting a widget from the widget tree will jump to the code where that widget is defined in the editor tab, then display information about the widget in the layout explorer. The layout explorer will give information about the `RenderBox` that your selected widget is contained within, as well as information about the constraints and sizing. If your widget is of a `Flexible` type, such as a `Row` or `Column`, you will also be able to temporarily change the `flex` property to instantly see the changes in your app, all without having to write code.

Many applications will not have such a simple layout where it is quick and easy to find the exact widget you are looking for in the widget tree. If you have a particularly complex layout, you can search for a widget using the magnifying glass icon in the widget tree header (see *figure 11.7*). Using the search, you can search by things such as the widget type and properties:

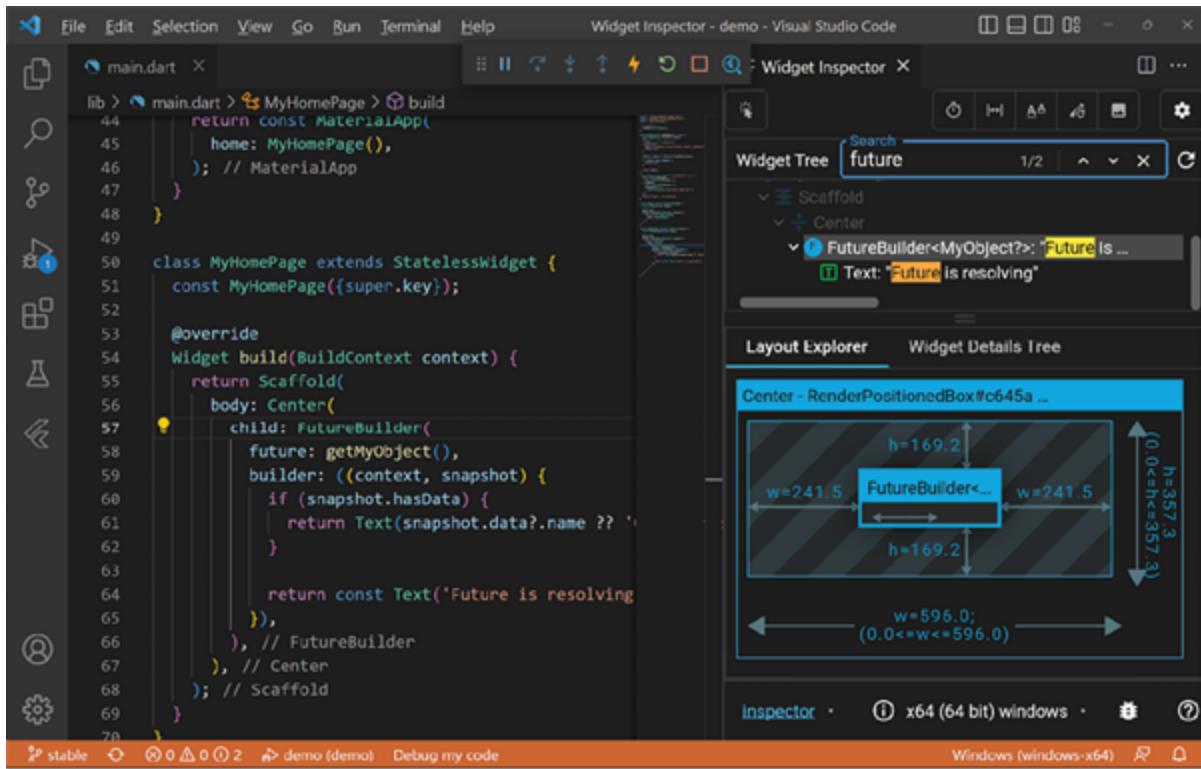


Figure 11.7: Searching for a Widget

Another useful tool to find widgets in your widget tree is the **Select widget mode**. Located to the top-left of the Widget Inspector panel is a button with a mouse cursor and, depending on the size of your window, a text label.

Clicking this button will give you the ability for you to click or tap on any widget from directly within your application to focus on it in the layout explorer and widget tree, as well as within the code itself.

Once you have selected a widget in DevTools, additional details can be found in the **Layout Explorer** tab and the **Widget Details Tree** tab. The Layout Explorer tab provides information about the widget's parent as well as its constraints and size. On some widgets, such as widgets that are the child of a **Row** or **Column**, additional information about the widget's flex, fit, and alignment is available. Each of these values can be manipulated in DevTools to see how a corresponding change in the code would impact the application's layout, in real-time. The **Widget Details Tree** tab lists all the properties a widget has, including all the properties that you have assigned to it. Furthermore, there is information about the render object and any associated state. These tools are invaluable when tracking down issues in your application, as they help to shed light on precisely how the widget is composed.

The DevTools Widget Inspector also includes a host of other useful tools, such as options to slow down animations, highlight widgets each time they repaint, display an overlay consisting of widget boundaries and other layout information, an option to show text baselines (to ensure text is aligned properly across widgets), and to highlight images which are oversized and using too much memory. The button toggles for each of these tools resides at the top of the Widget Inspector pane.

Flutter's DevTools are not limited to inspecting widgets. Although not immediately obvious at first, there are several other panes of useful tools. To access them, you will need to hover your cursor over the **Editor Language Status** field (which looks like `g`) at the bottom-left of Visual Studio Code, in the status bar. If the status item is not present, first check that your editor is focused on a Dart file (rather than, say, the Widget Inspector), and if all else fails, right click the status bar to ensure the **Editor Language Status** option is checked. From there, on the line which says **Dart DevTools**, click the **Launch** link. A popup will appear asking you which part of DevTools you would like to open. See *figure 11.8* for a visual reference:

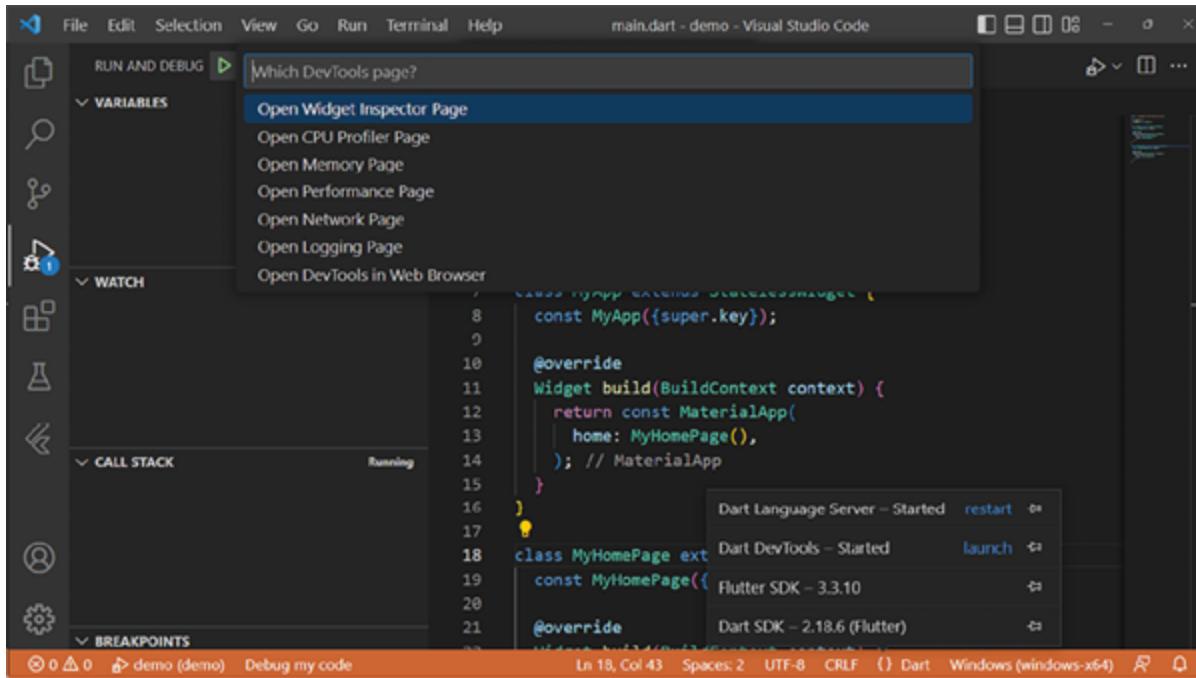
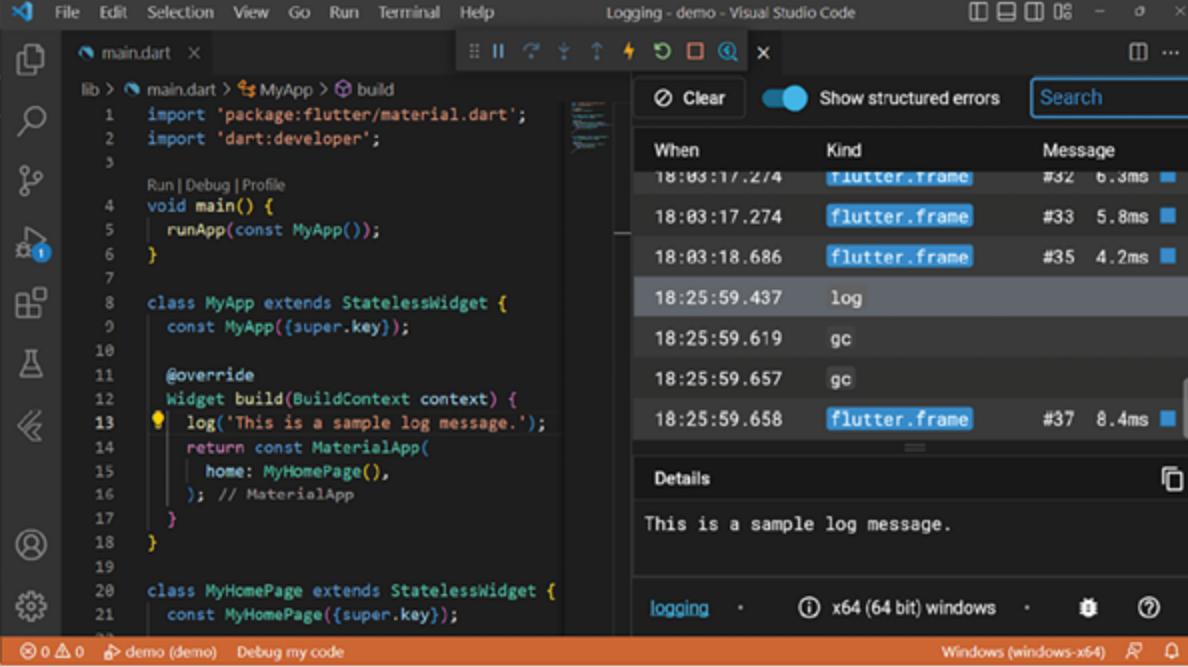


Figure 11.8: The process of launching different parts of DevTools requires you to hover over an item in the status bar then click launch before choosing which panel to open

The various panels available to us are the Widget Inspector, which we have already discussed at length; the CPU Profiler, Memory, and Performance panels, which we will look at later in this chapter; the Network panel, Logging panel; and the option to launch DevTools in your Web browser. Let us look at the Logging and Network panels.

The **Logging** panel (see *figure 11.9*) includes a timeline of messages logged by your code and the Flutter framework throughout the lifecycle of your application. Each line in the timeline can be clicked to view the contents of the message. When using the `dart:developer` package, the `print()` function, or the `debugPrint()` function to log messages, you can find the output in the Logging panel. This is especially useful if your debug console is flooded with output while your app runs, as you can easily find the logged message by using the search feature or scrolling through the list. Take note that the `print()` and `debugPrint()` functions write to the *standard output* and will therefore be labelled with the kind `stdout`. Additionally, do bear in mind that the `print()` and `debugPrint()` functions will only print the first 12,288 characters in a message. This can be especially frustrating if you are trying to log a large JSON response from an API, for example. If you find yourself frequently logging large pieces of data and not being able to see all of it in

the Logging panel, switch to the `log()` function and your woes will soon be gone.



The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Title Bar:** Logging - demo - Visual Studio Code.
- Left Panel:** Shows the project structure with `main.dart` selected. The code editor displays the following Dart code:

```
lib > main.dart > MyApp > build
1 import 'package:flutter/material.dart';
2 import 'dart:developer';
3
4 void main() {
5   runApp(const MyApp());
6 }
7
8 class MyApp extends StatelessWidget {
9   const MyApp({super.key});
10
11   @override
12   Widget build(BuildContext context) {
13     log('This is a sample log message.');
14     return const MaterialApp(
15       home: MyHomePage(),
16     ); // MaterialApp
17   }
18 }
19
20 class MyHomePage extends StatelessWidget {
21   const MyHomePage({super.key});
```

- Right Panel:** The Logging panel, part of Flutter's DevTools. It lists log entries:

When	Kind	Message
18:03:17.274	flutter.frame	#32 6.3ms
18:03:17.274	flutter.frame	#33 5.8ms
18:03:18.686	flutter.frame	#35 4.2ms
18:25:59.437	log	
18:25:59.619	gc	
18:25:59.657	gc	
18:25:59.658	flutter.frame	#37 8.4ms

- Bottom Status Bar:** Shows the current file (`main.dart`), the application name (`demo (demo)`), and the status `Debug my code`.
- Bottom Right:** Platform information (`x64 (64 bit) windows`) and other icons.

Figure 11.9: The logging tool, part of Flutter's DevTools

Now, let us look at the **Network** panel (figure 11.10). This is where you can find some (but not all — more on that in a minute) network requests made by your application. You will find information about the location of the request (often, the URL), the method of the request (whether it is a **GET**, **PUT**, **POST**, and so on), the status, port number, content type, number of bytes read and written, and other useful information which can be used to gain insight into network connections. If you were to make a request to an API, for example, you could find the API's response in the detailed view on the **Network** panel.

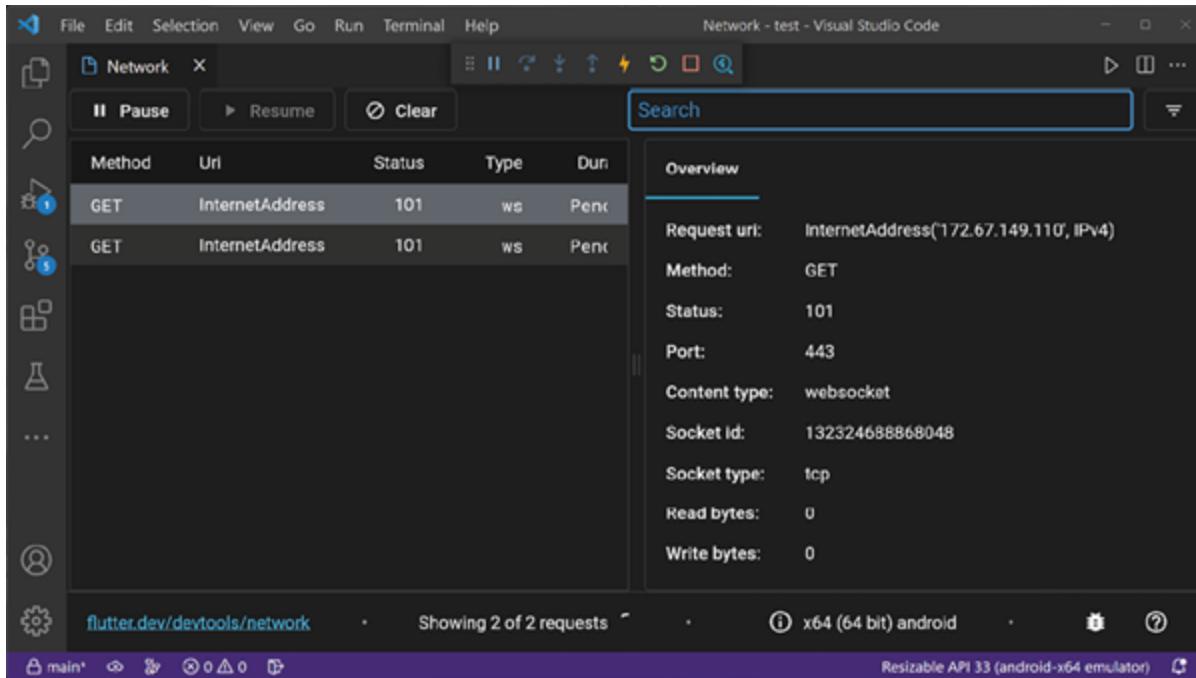


Figure 11.10: The Network panel of Flutter’s DevTools showing some activity

There are some limitations to be aware of with the Network panel, however. At the time of writing, only traffic generated using an `HttpClient` are visible (see the bug being tracked for this issue at <https://github.com/flutter/devtools/issues/4829>). There also appears to be an issue when developing in certain environments, such as within a Hyper-V Virtual Machine (see the bug filed here: <https://github.com/flutter/devtools/issues/4997>), but this should impact relatively few people. Still, it may be something worth considering if you are developing in such an environment.

Given the limitations of the Network panel in Flutter’s DevTools, it may be wise to *trust but verify* when it comes to the data presented in it. *If* the network requests are present, you can be assured that they are valid and correct. However, for the time being, do not trust that *all* network requests will be present here. Until the team has had more time to flesh out this component of the DevTools, you may want to find other ways to inspect the network traffic of your application, if possible.

Defining Performance Issues

Before we dive into what tools Flutter gives us to analyze the performance of our applications, we should define a few key terms and ideas. First, it is relevant to reiterate the fact that Flutter was always designed to be 60 frames-per-second (fps) from the very start, and each release of Flutter and the underlying Dart programming language often finds new ways to increase performance without requiring changes to your code. Flutter is very good at rendering large numbers of widgets on the screen, even animating them every frame without slowdowns. For this reason, it is important not to prematurely optimize your code: it likely is not necessary! However, if you are noticing some of the issues we are about to define and discuss, it is a great time to consider optimizing your code.

The performance of an application can be broadly split into two categories: *time* and *space*. Space-related performance issues are issues where your app may be using too much memory, or the application package is too large. Flutter's DevTools have a tool to evaluate an app's package size, which can help you determine what is taking up so much space. There are also tools to help you determine what is taking up memory, which we will talk about in a little bit. Time-related issues are a bit more complex, so let us talk about that.

Every device with a display will update that display at a fixed (or sometimes variable) refresh rate. For many devices these days, that is 60fps (which happens to be the target framerate of Flutter.) However, newer displays are capable of 100, 120, or even 240+ fps. That means the contents of the display are redrawn every 1/x seconds. For a 60fps display, the contents are redrawn every 1/60th of a second. Ok, but how does this relate to *performance*? Simply put, this means Flutter has 1/60th of a second to build and lay out all the widgets on the screen (called a *frame*) before the next screen refresh interval, called the `vsync`. Vsync is used to synchronize the refresh rate and the frame rate of the monitor to prevent tearing or when the screen displays information from multiple frames in a single screen draw. Tearing appears as a horizontal line across the screen where the image being displayed is split across one or more parts of the image. If a frame fails to build within `vsync` period, it will not be rendered until it is completed building.

Consider a spinning widget, rotating smoothly at 60fps. Suddenly, the widget appears to stop rotating, then a couple of frames later, snaps into the

position it would have been in had it continued rotating smoothly. This phenomenon is called *jank*. Jank occurs when a frame takes longer than the `vsync` period to build. If the frame has not been built yet, it cannot be displayed. In the following illustration (*figure 11.11*), the `vsync` period is marked by the dashed line, whereas the frame build time is denoted by rectangles:

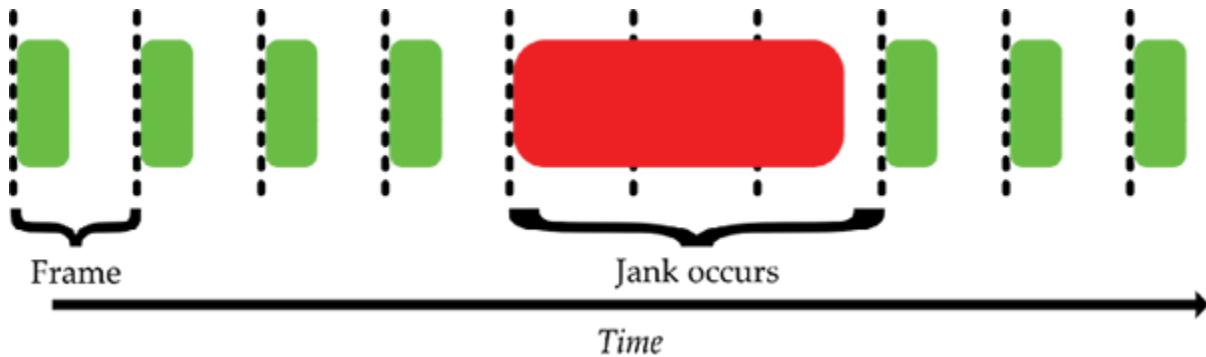


Figure 11.11: A visual representation of jank

However, even if you manage to avoid jank by ensuring your frames are built within the `vsync` period, you are not completely out of the water yet. If your frames take the *entire* `vsync` period to build, your app will consume a lot more energy. This will lead to shorter battery life on mobile devices, and generally contributes to global warming. We ought to strive to avoid this scenario. *Figure 11.12* represents what it might look like if we do not have jank but are taking too long to build our frames:

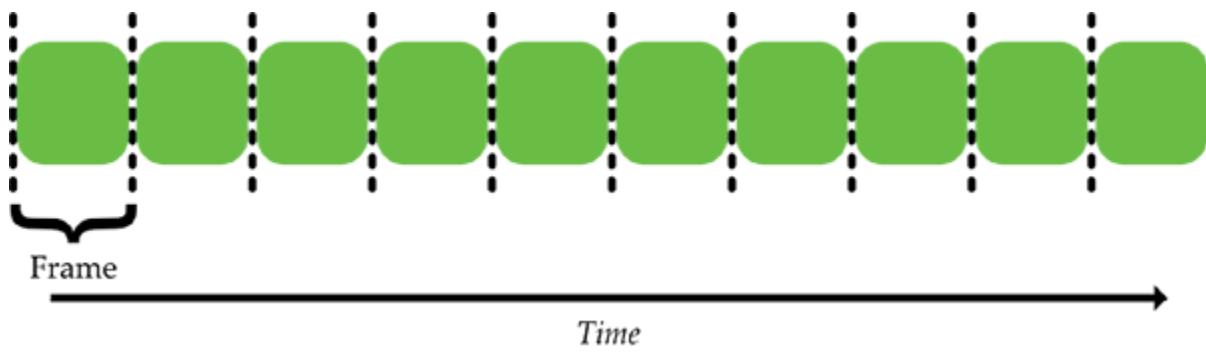


Figure 11.12: When a frame takes too long to build but still finishes within the vsync period, it will consume more energy.

So, if we can *see* that there is jank in our app, how do we verify it and identify the widget(s) causing the problem? Likewise, how do we identify whether our app is using too much memory and fix it? To do that, we will need to explore profile mode.

Exploring Profile Mode

Flutter can build your application in three different modes: debug mode, profile mode, and release mode (*figure 11.13*). Up to this point, we have been focused on debug mode, as it is the default for development and the most common mode you will be using while writing applications.

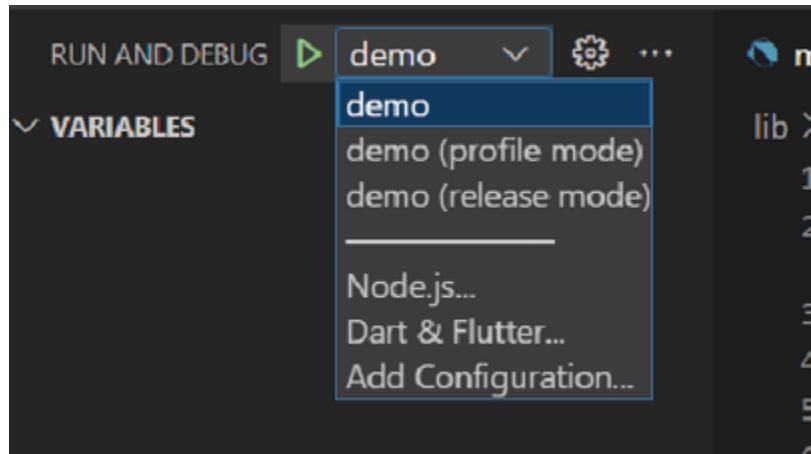


Figure 11.13: Flutter's three modes of building an application

To run your application in profile mode, either choose the profile mode option from the dropdown in Visual Studio Code, or use the following command: `flutter run --profile`. Once you are in profile mode, go ahead and open the **Performance** panel by pressing the leftmost button in the toolbar, next to the stop button (*figure 11.14*):

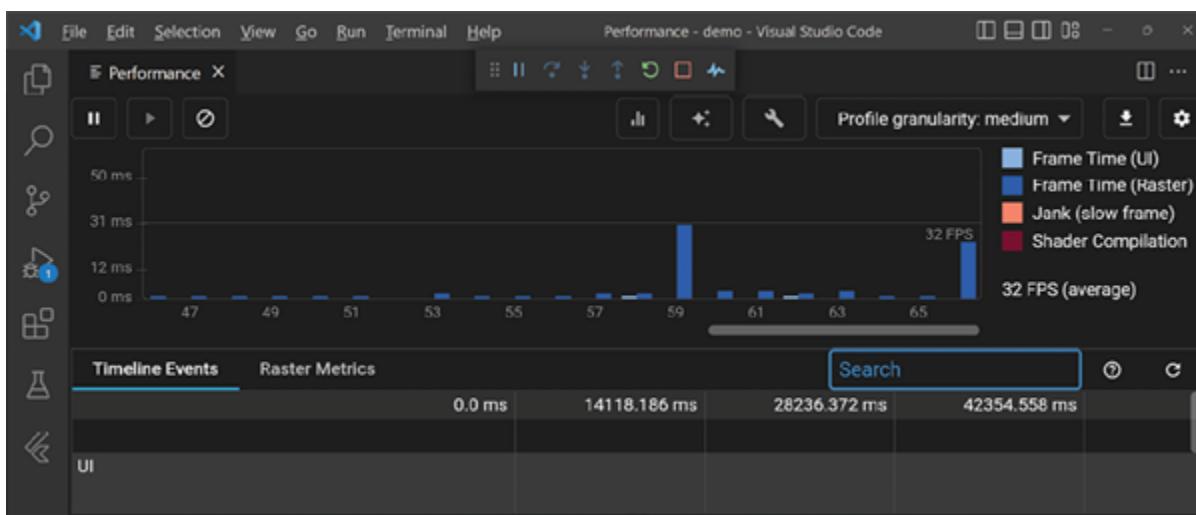


Figure 11.14: The Performance panel in Flutter's DevTools is available only when running in profile mode

On the top of this panel is a graph, where the X-axis is the frame number, and the Y-axis is a measure of time. The bars are color-coded to correspond to different aspects of the frame compilation process. Bars in orange and red are jank and shader compilation, respectively, and are generally not something we want to see. Additionally, there is a horizontal line in the middle of the graph, marking the point at which a frame needs to be finished to avoid jank. In *figure 11.14*, frame 59 comes in just below that line. In *figure 11.15*, frames 54 and 56 exceed it:

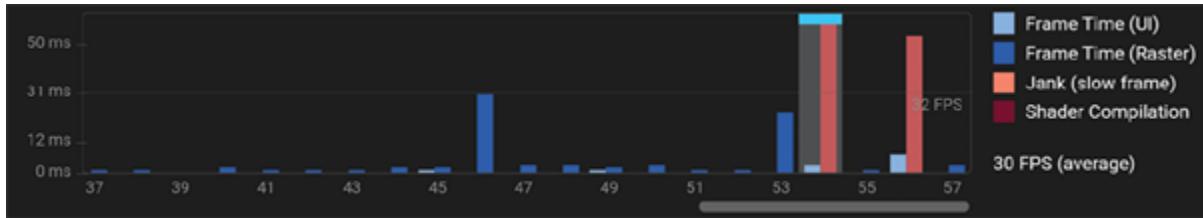


Figure 11.15: The performance graph

Tapping on a frame will display details in the panel below (*figure 11.16*) of what processes comprise a given frame compilation. Tapping on any event in the details view will give you even more details of that event, including precisely how many microseconds it took to run. By exploring the different processes that your widgets run, you can discover what operations are *expensive* (that is, they take a long time) and either avoid them entirely or use them sparingly.

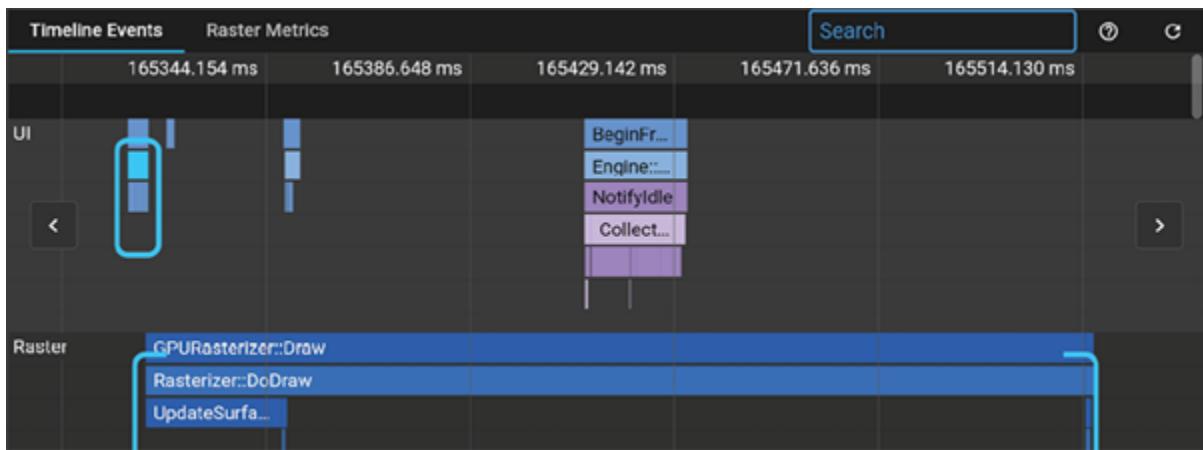


Figure 11.16: Details of a timeline event

By default, it may be difficult to determine exactly which widget(s) are taking a long time to build. We can remedy this by enabling a more expensive (and thereby slower) option to track widget builds (*figure 11.17*).

By enabling this option, specific widgets will appear in the timeline event details panel, thereby making it far easier to track precisely which widgets are causing your application to have performance issues. This operation will increase the frame build times, so bear that in mind as you're evaluating performance when the option is toggled on:

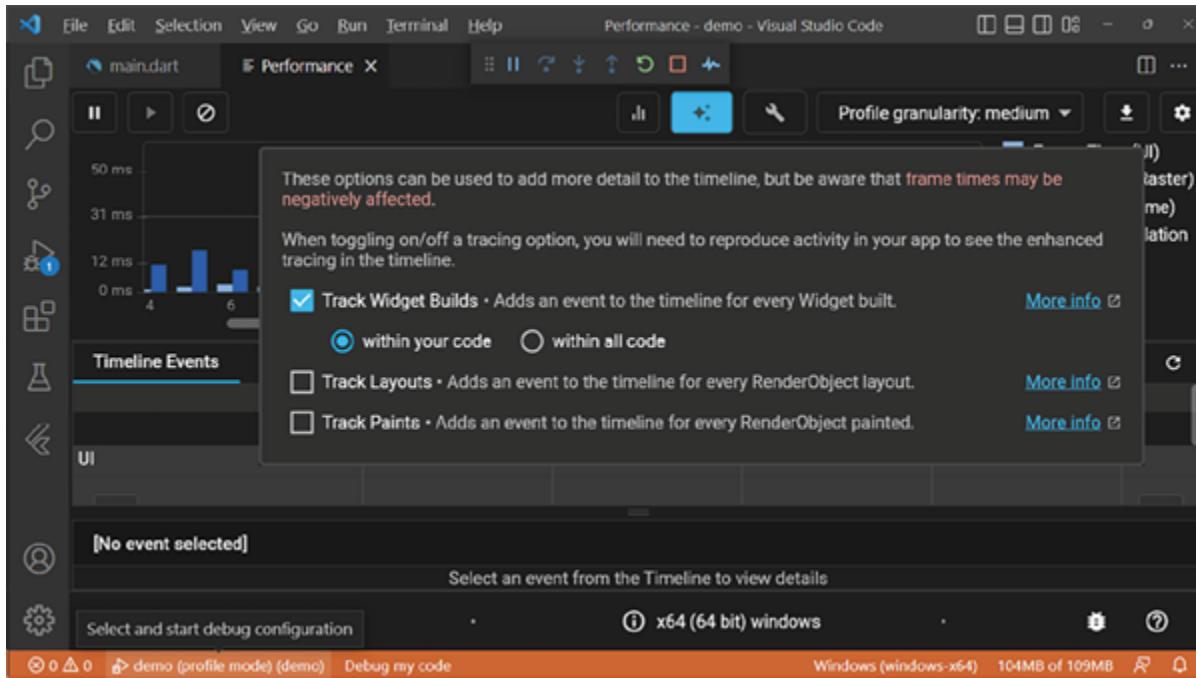


Figure 11.17: Enabling the option to track widget builds

Finally, there is a button to the left of the one we just pressed to enable tracking widget builds which will toggle a performance overlay directly on your application (*figure 11.18*). This can provide more immediate feedback while using the application to identify which touches, drags, taps, and other actions cause slowdowns and jank. It may be easier to visualize these slowdowns with the performance overlay on the application, so your focus is not divided between interacting with the application and the tools in your IDE.

There are two graphs that will appear. The one on the top is the raster graph, while the one on the bottom is the UI graph. The UI graph measures everything your code does, including the Flutter framework itself. This is code that runs inside the Dart virtual machine. When your application runs code here, a layer tree is generated, which contains the commands needed to paint the UI to the screen. The layer tree is then sent over to the raster thread, which is where Skia (or Impeller) is running. The raster thread will

convert the layer tree into GPU commands to draw the UI on the device display.

If the raster thread is busier than the UI thread, chances are high that you have built widgets that contain some expensive GPU commands. This could be things like having overlapping elements with different opacities, widgets that have a clipping pattern applied (such as with the `ClipRRect` widget), shadows, or by using one of Flutter's most expensive operations, `SaveLayer`. For more information regarding `SaveLayer`, the documentation found at <https://api.flutter.dev/flutter/dart-ui/Canvas/saveLayer.html> is an excellent resource.

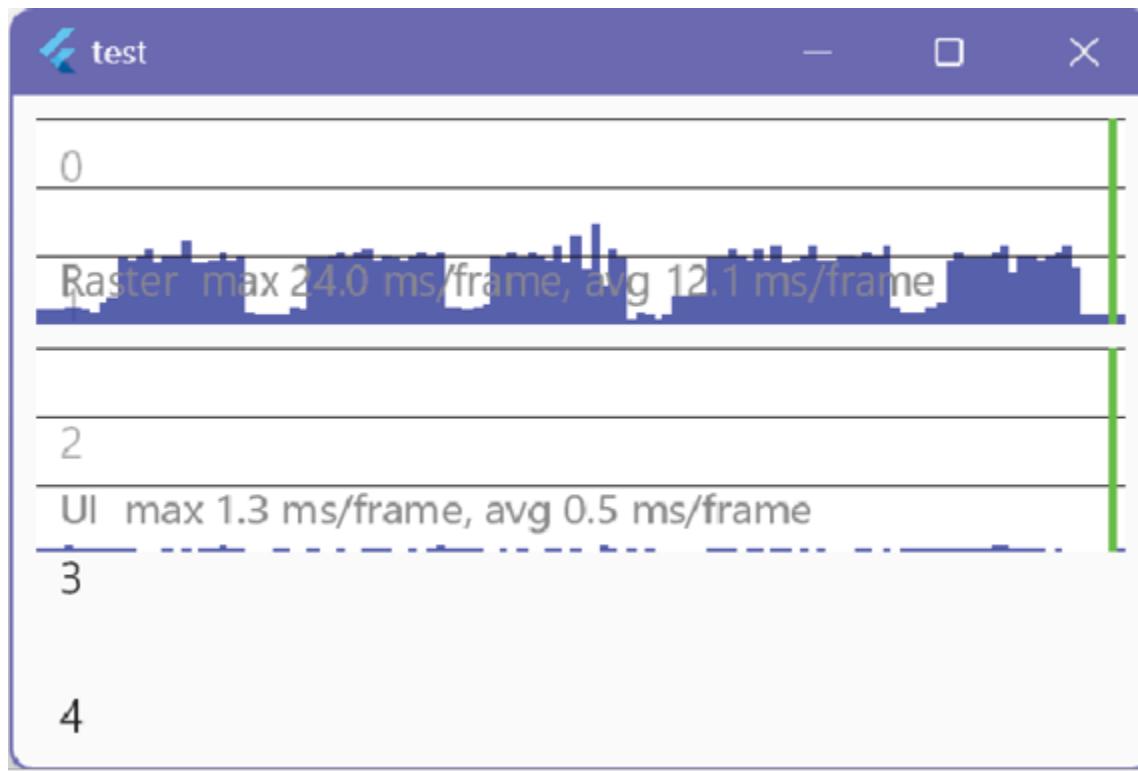


Figure 11.18: A performance overlay has been toggled on the running application

Next, we are going to be looking at the CPU Profiler, which will list all the various processes which occur during a recorded timeframe. To launch the CPU Profiler, you will use the same method we used before to launch the network and logging panels. The CPU Profiler is an expensive process to run, so by default it will be empty. To get started, press the circle **Record** button at the top, perform one or more actions within your application, then press the square **Stop** button. Depending on how long you recorded for, it could take a while to process the results.

Once the results have been processed, you will see a tree view of each method called by your application, many of which you have not explicitly called yourself, but are called by widgets and methods you have used (see *figure 11.19*). Each item is an expandable list, where you can continue to drill down into the subsequent methods that have been called by the previous method. By examining this tree, you will be able to identify the most expensive operations which your application has performed.

The default tab is the **Bottom Up** tab, which lists the method calls in reverse order from how they were performed. So, expanding a particular method call will show which method called it, the *callers*.

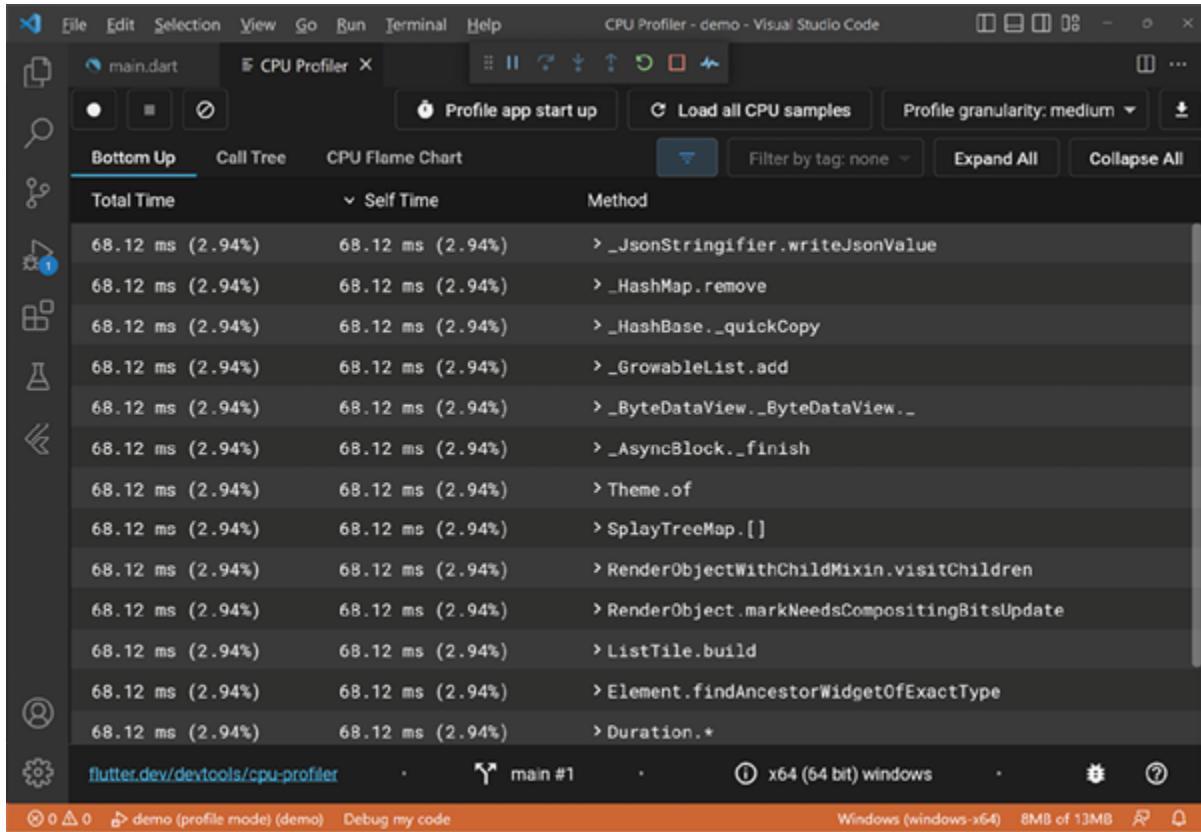


Figure 11.19: A sampling recorded by the CPU Profiler

If you would rather view a top-down view of method calls, switch to the **Call Tree** tab, where the methods calls will expand into the list of methods they have called, the *callees*.

The final tab is the **CPU Flame Chart** tab (*figure 11.20*). Much like the details displayed in the performance view, this shows a top-down call stack. In this instance, rather than being for a single frame, it is for the entire

period which was recorded. The longer the line, the more CPU time the process took to complete.

The flame chart can be interpreted as a top-down stack frame, much like the call tree. Rather than having an expandable list, however, the information is presented more visually. This view is an excellent tool for quickly identifying expensive operations, due to the correlation between the length of the bar and the amount of CPU time taken to perform that operation. At a glance, it is easy to pick out widgets and operations which should be investigated for performance improvements.

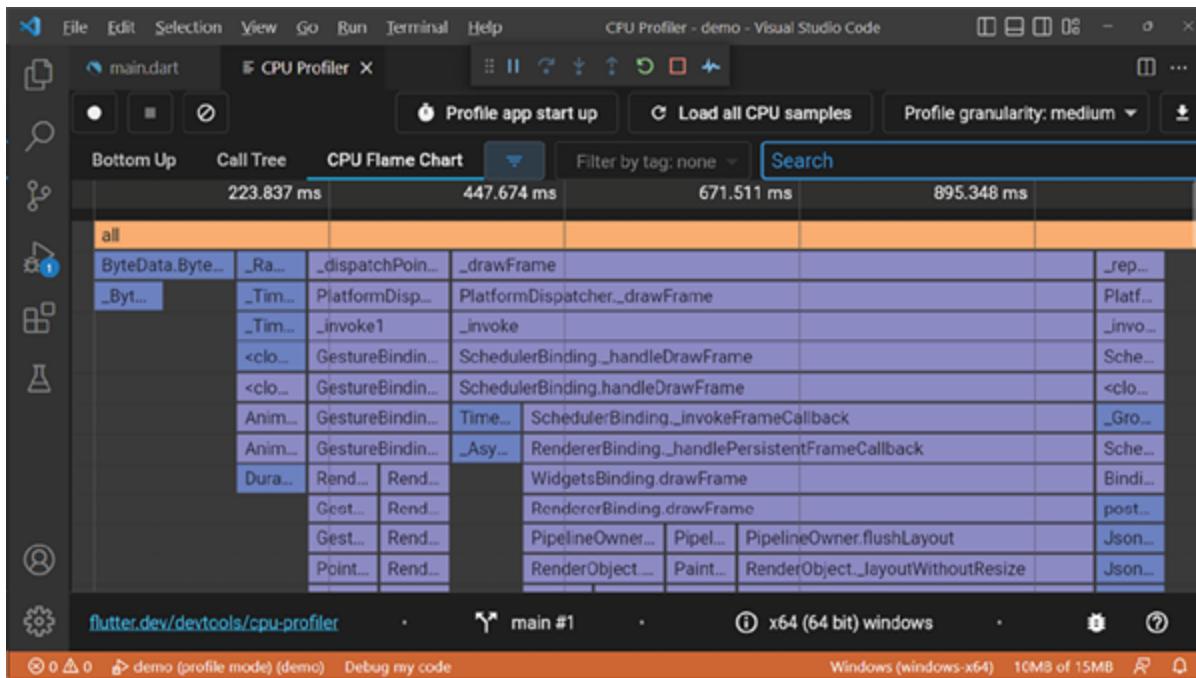


Figure 11.20: The CPU Flame Chart tab of the CPU Profiler

Lastly comes the memory tool. This tool can help you to identify which aspects of your application are taking up precious system memory. All computers have a finite amount of memory, also called **random access memory (RAM)**. This is *different* from system storage, which is used for the long-term storage of files, folders, and documents that persist even when the device is shut down or rebooted. Rather, memory is much more limited in quantity and *volatile*, meaning anything stored in system memory is erased when the device is shut down or rebooted.

Due to the nature of memory, it is one of the most precious resources on a computing device. Any action you want the CPU to take will require

loading information into the RAM before it can be processed. This means that when you want to read a file off the disk and display it on the screen, the contents of the file are loaded into RAM before being processed by the CPU and drawn to the display. Modern computing devices and programming languages have all sorts of tricks up their sleeve to optimize memory utilization, and Flutter is no different in this regard.

If, for example, you want to display a `ListView` of `ListTiles` with `Text` widgets containing an incrementing list of numbers, the list of numbers will need to be stored in memory (as well as everything else Flutter needs to perform the computation and layout.) In *figure 11.21*, we have opened the memory tool and found the `Paragraph` class, where we are able to see that there are 2,000 instances consuming a total of 93.8 KB of memory.

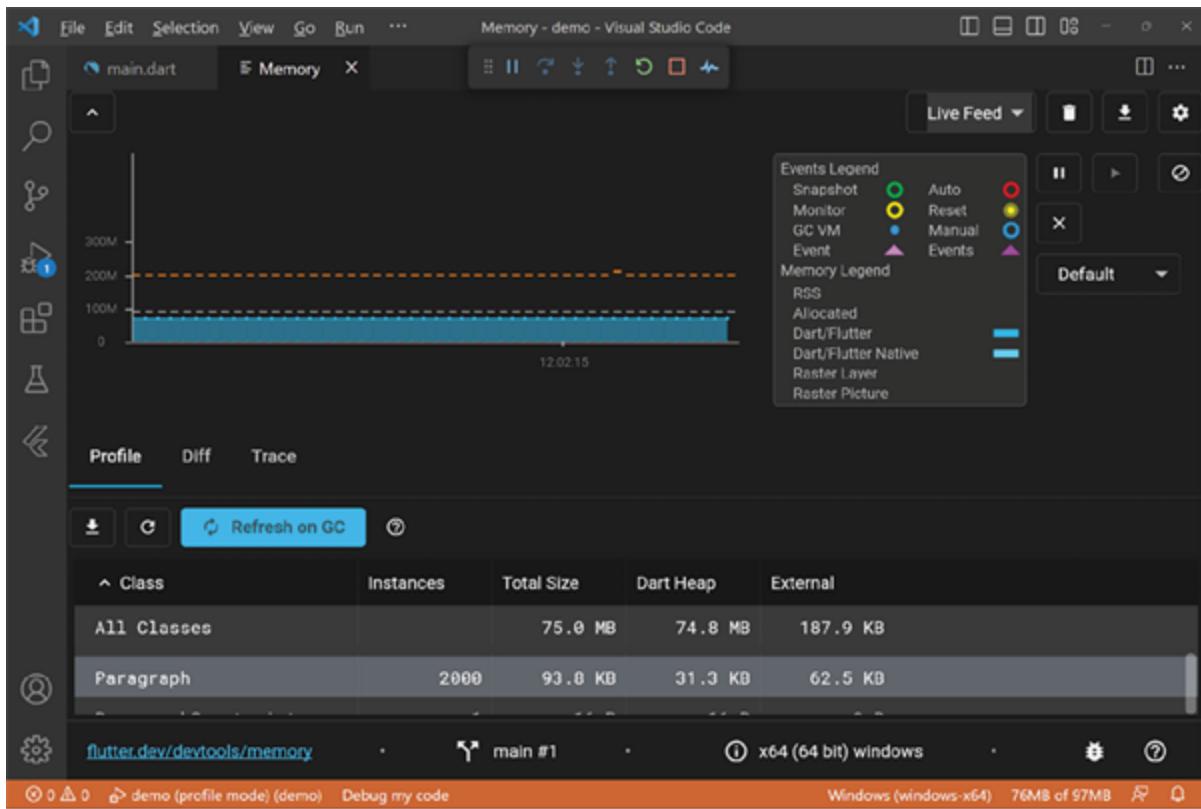


Figure 11.21: The Memory view of DevTools in Profile mode

Due to a bug in DevTools (<https://github.com/flutter/devtools/issues/5174>), we will need to temporarily switch back to debug mode for the next task. This bug may be resolved by the time you are reading this, so check the status of the bug before switching to debug mode. Continuing along, if we switch to the

Trace tab, we will get a list of all the objects used by the framework to build the screen. We can then track these objects by checking the box(es), interacting with the application, then pressing the **Refresh** button on the **Trace** tab, which will then place a dot in the graph when that object is created (*figure 11.22*). By selectively choosing potentially memory consuming widgets to track, you can get a sense of how many of these widgets are being created and whether it matches your expectations. More on this in a bit.

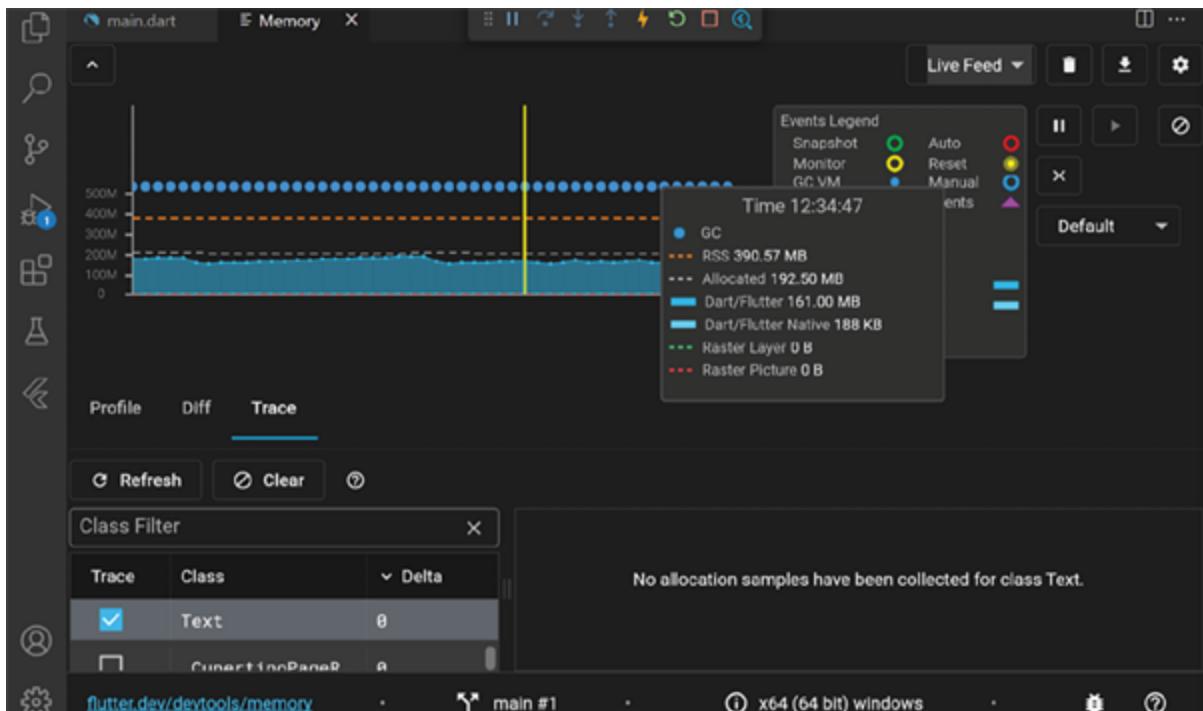


Figure 11.22: Tracking Text widget creation

It's unlikely that you will spend much time in performance mode or using the **CPU Profiler**, **Memory**, and **Performance** tabs while you are early in your Flutter career. These tools are often used by senior Flutter developers to analyze the application and determine which steps they and their team should take to improve the application. However, this very brief overview will familiarize you with what tools are available and how some of them can be used, hopefully planting a seed of curiosity, and giving you the tools necessary to answer questions in the future.

Some of these tools can feel overwhelming to use. There is a lot of information being thrown at you by them, and not all of it will make sense. Thankfully, as we covered at the beginning of this chapter, Flutter does a

great job of optimizing layouts right out of the box. The tools listed here are useful when you notice jank, high memory usage by your application, and other unexplained slowdowns. For the most part, unless your task is to explicitly find performance improvements to optimize your application, you likely will not be using these tools very often. However, the fact that they exist and knowing what they do is valuable information in and of itself.

When a ListView is not a ListView

Now that we have explored the various profiling tools available to us, let us put them to use. First, let us consider the section's title: When a **ListView** is not a **ListView**. To understand that we must first understand what a **ListView** is and why you would want to use it.

Using our earlier example of a list of integers, let us say we want to display them in a vertically scrolling list of widgets. First, let us generate a list of integers:

```
1. final List<int> integerList = List<int>.generate(2000, (index) =>  
    index);
```

This gives us a list of integers from 0 to 1,999, with a total of 2,000 integers. Next, let us try creating a series of **Text** widgets in a **Column** that we have made scrollable. Here is what that would look like:

```
1. class MyScrollingList extends StatelessWidget {  
2.     const MyScrollingList({super.key});  
3.  
4.     @override  
5.     Widget build(BuildContext context) {  
6.         final List<int> integerList = List<int>.generate(2000, (index)  
            => index);  
7.  
8.         return SingleChildScrollView(  
9.             child: Column(
```

```

10.         children: <Widget>[
11.             for (final int i in integerList)
12.                 ListTile(
13.                     title: Text('$i'),
14.                 ),
15.             ],
16.         ),
17.     );
18. }
19. }
```

Let us run this in profile mode and scroll the list a bit to see what the performance looks like. As you can see in *figure 11.23*, there is a lot of jank when scrolling this list. Even though our app is incredibly simple, consisting only of a column of 2,000 `ListTile` widgets, each displaying a single integer, scrolling the list causes jank. Running the same code on a less powerful machine than a high-end computer, such as a smart phone, would certainly result in even more jank.

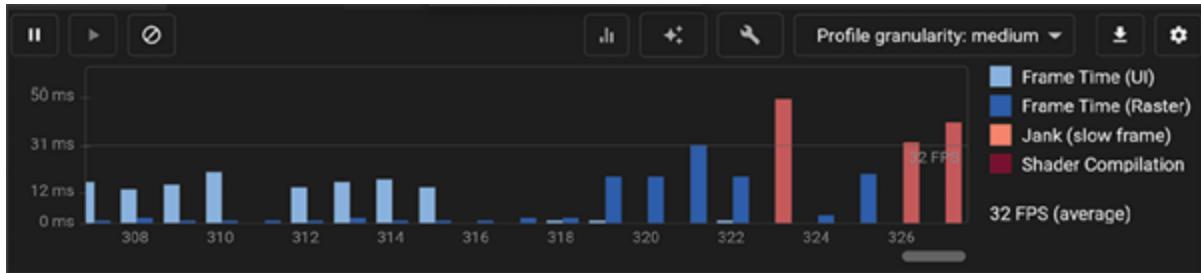


Figure 11.23: There's a lot of jank when scrolling

Now, let us rewrite this to use a `ListView` widget. In this example, we have removed the `SingleChildScrollView` and the `Column`, replacing both with the `ListView.builder()`. We tell the `ListView` how many items there will be in the list using the `itemCount` parameter, then use the `itemBuilder` to build the `ListTile` widgets. Note that there is no longer a for-loop in this code, as the builder takes care of this for us:

```
1. class MyScrollingList extends StatelessWidget {  
2.     const MyScrollingList({super.key});  
3.  
4.     @override  
5.     Widget build(BuildContext context) {  
6.         final List<int> integerList = List<int>.generate(2000, (index)  
=> index);  
7.  
8.         return ListView.builder(  
9.             itemCount: integerList.length,  
10.            itemBuilder: (BuildContext context, int i) {  
11.                return ListTile(  
12.                    title: Text('$i'),  
13.                );  
14.            },  
15.        );  
16.    }  
17. }
```

Now that we have updated our code, let us run the same test: scroll the list of integers and see what the performance looks like (*figure 11.24*). What we are looking for here is a performance improvement, ideally in the form of frames being rendered in less time and with no jank present.

Extraordinarily, there is absolutely no jank and frame times have been reduced to just a few *microseconds*, when compared to the tens of *milliseconds* that the previous solution offered. It is clear by this test that the `ListView` is far more efficient, but *why*?

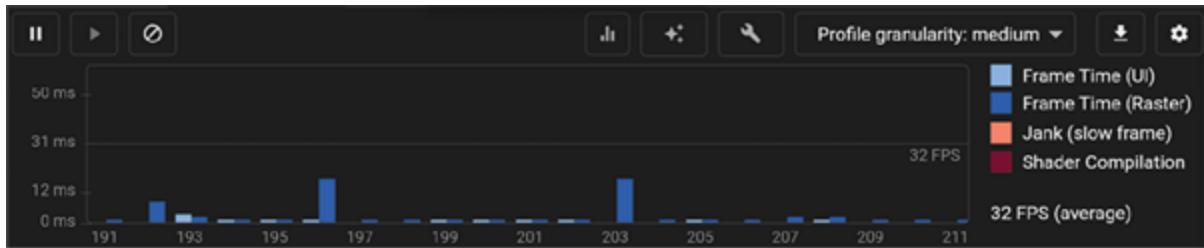


Figure 11.24: Scrolling performance of a `ListView`

When a `Column` is created, all its children are built each frame. When we have 2,000 `ListTile` widgets being built each frame, that is a lot of work for the CPU to handle, especially when we start scrolling. If we are not scrolling, Flutter is smart enough not to perform any unnecessary widget rebuilds because it knows that there is no animation happening, no state changes, and performing a rebuild of the widget would give us the *same* widget we started with. Thus, when no action is taken, the build process never happens. In that instance, it does not matter how many widgets we have built... or does it? It *does*, in fact, matter. Each widget consumes memory. If, instead of displaying an integer, we had a list of images, the memory size would balloon out of control as all 2,000 images were being loaded into memory when the `Column` was initially built, and then *every frame* as we scroll.

By contrast, a `ListView` will build *only the widgets displayed on screen* (plus a couple on either end, if applicable) at any given time. If the constraints of our list are such that only three items can fit on the screen at any given time, only about 5-10 or so widgets will be built. (That is the three that are on screen, and a couple each on the above and below.) When the list is scrolled, widgets falling outside of the possible range where they are either shown on screen or could immediately be shown on screen are removed from the widget tree, freeing memory in the process. This is precisely why the `ListView` is so much more performant than a `Column` when dealing with a large list of children. That is not to say a `Column` does not have its place — it absolutely does — but knowing the difference between the two can mean the difference between an application with jank and an application with smooth scrolling.

So, back to the original question: when *is* a `ListView` not a `ListView`? The answer is: when we *shrinkwrap* the `ListView`. Normally, the `ListView` will fill its constraints, culling off-screen widgets in the process. However, if we

specify the `shrinkWrap: true` property, the entire widget will act more like a `Column`. This could be useful if you needed a small number of widgets in a list, each being built using a builder rather than composed independently and inserted into a `Column`. It is not an impossible situation to imagine. However, not understanding the implications of setting the property on a much larger list can lead to problems. Let us look at an example:

```
1. class MyScrollingList extends StatelessWidget {  
2.   const MyScrollingList({super.key});  
3.  
4.   @override  
5.   Widget build(BuildContext context) {  
6.     final List<int> integerList = List<int>.generate(2000, (index)  
      => index);  
7.  
8.     return SingleChildScrollView(  
9.       child: ListView.builder(  
10.         shrinkWrap: true,  
11.         itemCount: integerList.length,  
12.         itemBuilder: (BuildContext context, int i) {  
13.           return ListTile(  
14.             title: Text('$i'),  
15.           );  
16.         },  
17.       ),  
18.     );  
19.   }  
20. }
```

Here, we have taken our highly performant `ListView` and added the `shrinkWrap: true` property, then wrapped the whole thing in a `SingleChildScrollView`. We will perform the same scrolling performance test as before, with the results as shown in *figure 11.25*:

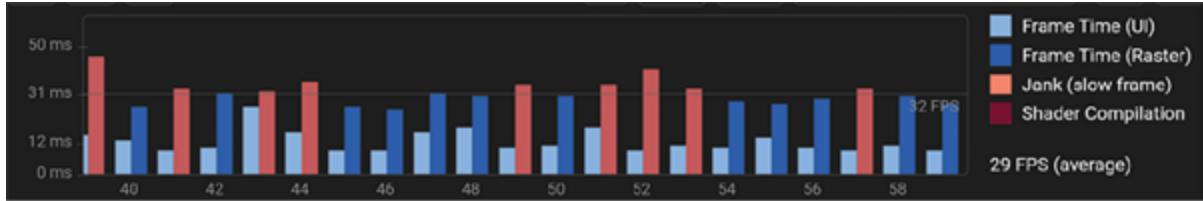


Figure 11.25: Jank introduced by treating a ListView like a Column

Without a doubt, there is a lot of jank, and the frame times are way up. As you can see, even though we are using the much more performant `ListView.builder()` rather than a `Column`, we have similar or worse performance. Understanding *how* widgets work and which to use in each scenario is key to building a highly performant application. However, this example was tailored specifically to showcase this single widget. How can we identify the problem in our own applications using these tools? Let us start by selecting one of the janky frames, then start examining the event details (*figure 11.26*):

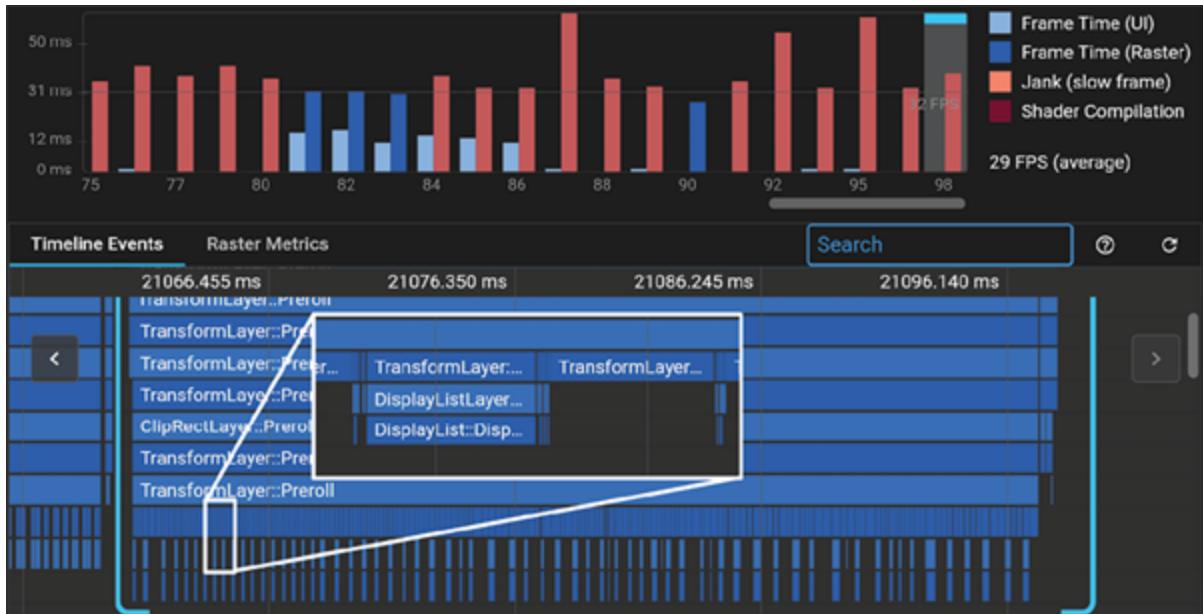


Figure 11.26: Zooming into the timeline details of one of our janky frames

By zooming in, we can see that there are probably about 2,000 `DisplayList` events *each frame*. Remember that you can also toggle on tracking widget

builds to make things even more clear. Using this information, we can tell that our list is rebuilding every item in the list, every frame. With that knowledge, we should have enough information to start tracking down the problem. If we rerun the application in debug mode to gain access to the widget inspector, we can very quickly identify the problem (*figure 11.27*):

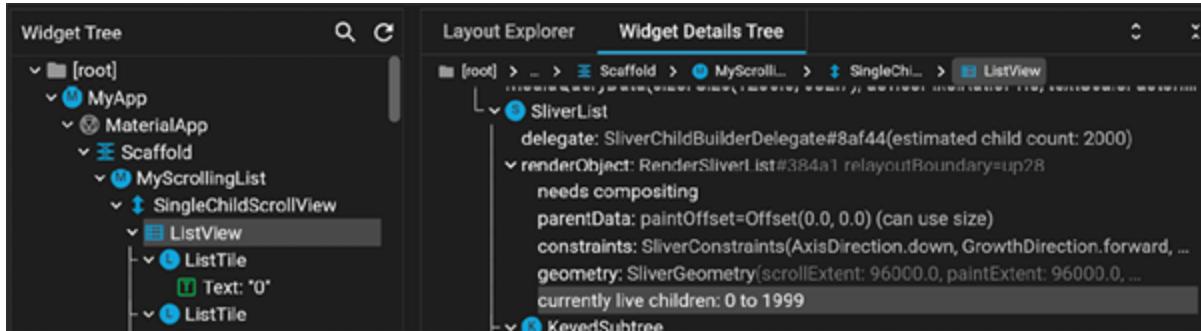


Figure 11.27: Our Widget Tree has 2,000 ListTiles in it with 2,000 live children.

The render object for our list will tell us precisely how many children are currently *live*, or actively being built. Note the estimated child count a couple of lines above, which states the estimated total child count. In contrast, if we look at our optimized ListView example without the `shrinkWrap: true` property being set, we can see there are still an estimated 2,000 children, but only 20 of them are currently live (*figure 11.28*):

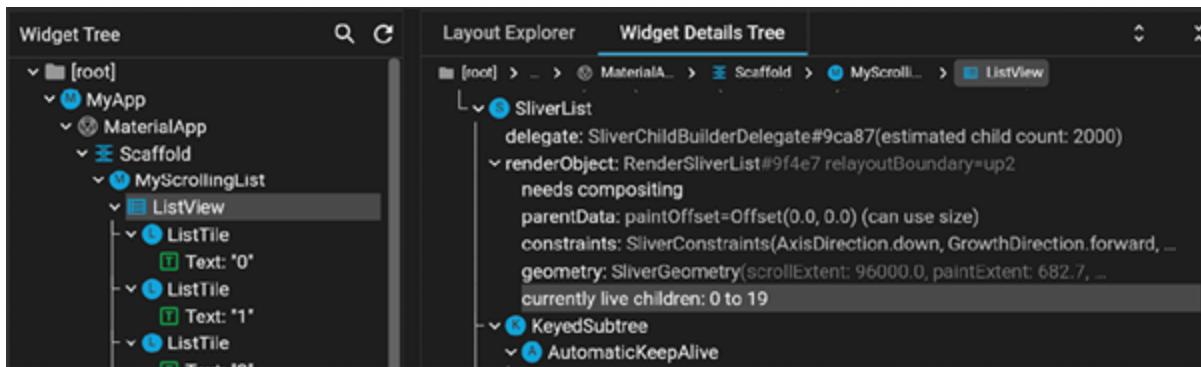


Figure 11.28: Our widget tree has only 20 ListTiles with 20 live children.

Here, we have looked at just one example of a built-in widget which can go awry if we are not careful. There are many other widgets which can have similar problems if we are not careful, but at least we have the tools to identify these issues when something goes wrong. But what are some best practices to keep in mind so that we do not put ourselves in this place to begin with?

Optimizing Widgets

As we know, the `build()` method of a widget is run each time the widget is rebuilt, which could be upwards of 60-120 times each second. If our aim is to make sure that we avoid jank, we need these rebuilds to happen as efficiently as possible. Thankfully, there are several things we can do to ensure that rebuilds are doing as little work as possible.

To start, you should avoid defining and running unnecessary functions within your build method. If you have a camel-case string that you want to split apart into individual words, then convert to lowercase, you *could* create a function in the build method which takes the string you want to convert in and returns the text in the format you want. Then, you can call that function whenever you need to display that text. This would cause the method you have created to be run repeatedly each time the widget is rebuilt, though. Since the output should always be the same, would it not make much more sense to pass that value into the widget before it is built, perhaps as an argument?

Another way to optimize your widgets is with caching. Let us say you want to display a user's profile picture on a navigation drawer's header whenever the drawer is opened. The image is stored online, so you will need to fetch that image from the internet to display it. Since the drawer header is rebuilt every time the drawer is opened, the image will be fetched from the internet every time the drawer is opened. Rather than constantly re-fetching the image, you could use a widget such as the `CachedNetworkImage` widget, which will store the image in the application's local cache upon fetching it for the first time. The next time the widget is rebuilt, the cache will be checked first and the image there will be used if it exists; otherwise, the image will be fetched from the internet.

Caching can be used in all sorts of ways, not just with images. It may be possible to cache some API calls, at least for a period, before invalidating the cache due to some sort of timeout. This means that a widget calling an API does not need to make dozens of calls each second, saving network bandwidth and speeding up the widget build.

One of the biggest things you can do to increase performance is to simply have smaller widgets. Instead of building a screen in a single widget tree,

consider logically segmenting different parts of the screen into smaller widgets. Your screen has a header? Make a header widget. Then, optimize the smaller widgets. Not only will this give you more re-usable code, but it will help prevent unnecessary widget rebuilds. If you have broken your header off into its own widget, it is far less likely to be rebuilt unnecessarily when part of the application's main content calls for a rebuild.

Your IDE likely has a line running down it by default, marking the 80-character line (*figure 11.29*). If the code for your widget tree is regularly bumping up to this line or even running past it, that should be a clue to you that your widgets might be nested too deeply, and it is time to refactor your code.

Not only will your code be more readable and easier to debug when the widgets are smaller, but you will be forced to think more deliberately about how you compose your widgets. The performance improvements will come as Flutter will only rebuild widgets which require a rebuild, allowing it to keep widgets that do not. If your entire application is a single widget, that widget will always be rebuilt in its entirety.



The screenshot shows a dark-themed code editor window for a Dart file named 'main.dart'. The code defines a main function that runs an MyApp widget. The MyApp class extends StatelessWidget and overrides the build method to return a MaterialApp with a Scaffold home. A horizontal white arrow points from the left margin of the code area towards the right side of the screen, indicating the 80-character line limit. The code is as follows:

```
lib > main.dart > MyScrollingList > build
1 import 'package:flutter/material.dart';
2
3 void main() {
4   runApp(const MyApp());
5 }
6
7 class MyApp extends StatelessWidget {
8   const MyApp({super.key});
9
10  @override
11  Widget build(BuildContext context) {
12    return const MaterialApp(
13      home: Scaffold(

```

Figure 11.29: Highlighting the 80-character line in Visual Studio Code

Finally, be conscientious about the parameters you are passing into your widgets. It may be convenient to pass in a large object to pull out multiple values for the widget you are building, but that might mean making a copy of that object in memory and will require more CPU time to process each time the widget is built than if you had simply passed in the variables you needed in the first place. If you are using BLoC, it is tempting to pass the BLoC state from a **BlocBuilder** into several of the child widgets being built.

However, that is unlikely to be an efficient way to build the widgets. Instead, pull out the variables you need from the state and pass them directly in as parameters. The resulting widgets will even be easier to read and debug, since you will know ahead of time if the parameters you are passing in are properly set, rather than doing checks inside that widget's build method.

These are only a few suggestions to keep in mind when building widgets optimally, and it is by no means an exhaustive list. It will take practice and experience to learn what causes jank, balloons memory usage, and contributes to an application that is underperforming your expectations. There are more suggestions in Flutter's online documentation concerning optimizing performance, which you can view here: <https://docs.flutter.dev/perf>.

Conclusion

Learning to use Flutter to build applications is not something you can do only by reading a book. It requires experimentation, failure, analysis, and eventual understanding. Fortunately, Flutter ships with a robust set of tools, DevTools, which give you everything you need to build your understanding. Even the most seasoned Flutter developers will use these tools constantly, having integrated them into part of their everyday workflow.

Flutter was designed from the very beginning to be highly performant, and while premature optimization of your widgets is not recommended, sometimes it is necessary to dig deep into your application to understand why it is not buttery smooth. When that happens, you have got the tools available to analyze and dig into precisely what is happening, thanks to Flutter's profile mode.

While these tools may seem daunting at first, you will soon find that they have become a part of your everyday toolkit when building applications. In the end, you will realize that you certainly would not want to write code without them! All it takes is practice and patience.

Up to this point in the book, we have been learning a lot about different *concepts* when it comes to developing applications using Flutter. In the upcoming chapter, we are going to take all these concepts and put them to

use. We will explore what it might be like to have a client with a set of ideas, business needs, and a design — then turn all of it into a fully-functioning application. This is what we have been building up to, and you have everything you need to be successful. So, let us go build an application!

Questions

1. What is the difference between debugging and troubleshooting?
2. What are the five steps involved in debugging?
3. What methods can you use to log information to the debug console?
4. What is a breakpoint, and how do you create one?
5. What is a stack trace? What is a stack frame?
6. What is an exception?
7. What risks do you run by using a `try/catch` block?
8. How do you access Flutter's DevTools?
9. What are the different tools available in Flutter's DevTools?
10. What are the two different categories of performance issues your application may encounter?
11. What is vsync and how does it relate to your application's performance?
12. What is jank?
13. What three compilation modes are available for Flutter applications? How do they differ?
14. What is the difference between storage and memory?
15. When is a `ListView` not a `ListView`?
16. When should you consider optimizing your application?
17. What are some steps you can take to optimize your application?

Key Terms

- **Breakpoint:** Intentionally stopping or pausing code from running at a specific point to assist in debugging.

- **Debugging:** The process of going step by step through a problem to identify and resolve any errors.
- **Jank:** A stuttering effect produced by one or more frames taking too long to complete.
- **Pseudo-code:** A code-like representation, often non-functional, which is used to illustrate a solution or mock out a problem.
- **Troubleshooting:** The process of solving a problem.

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

Creating Your First Application

Introduction

Throughout this book, we have spent a lot of time discussing the different widgets and concepts that are used to build an application. Now, it is time to put that knowledge to practical use. In this chapter, you, the reader, will play the role of a Flutter developer, who has just landed their first client. You will be given a design and set of requirements, and it will be up to you to figure out how to best deliver an application to your *client*.

We are going to use everything we have learned in the book so far to build the application in this chapter. A few new concepts will be introduced as well, such as *code generation*. This chapter will help guide you through the thought process of converting a design and requirements into an application, but it will be up to you to build it. If you get stuck, though, there will be code in the accompanying repository that you can use as a reference.

Structure

In this chapter, we will discuss the following topics:

- About the application (we will be building)
- Understanding the technical requirements
- Building the skeleton of your application

- The WeatherIcon class
- Code generation
- Building the API layer
- Finishing the application

Objectives

In this chapter, you will learn how to take a design and set of requirements and use them to create an application. You will also learn about how to organize your project files, automatically generate code, and use code decorators. Finally, you will have your very first, fully functioning application.

This is, by far, the most challenging chapter of the book. It is also, arguably, the most important. Be sure to take your time with this one, because the concepts you will learn here will be invaluable to you throughout your career!

About the Application

The application you will be building throughout this chapter is a weather application. Weather applications are ubiquitous and relatively simple, while offering ample opportunity for creativity in their design. Take a moment to look at a mockup of the main screen of the application in *figure 12.1*:



Figure 12.1: The weather application we will be building throughout this chapter

The design for this application leverages a set of free and open-source icons, created by Microsoft. However, you may use any set of icons which suit your tastes. The layout has been created in such a way as to pose a couple of unique challenges to solve. Take, for example, the **Today** section: the horizontally scrolling list clearly has padding to the left yet overflows the screen on the right. Were you to scroll all the way to the right, you would see padding on the right with the cards overflowing on the left. Another unique challenge exists on the seven-day forecast screen, as seen in *figure 12.2*:

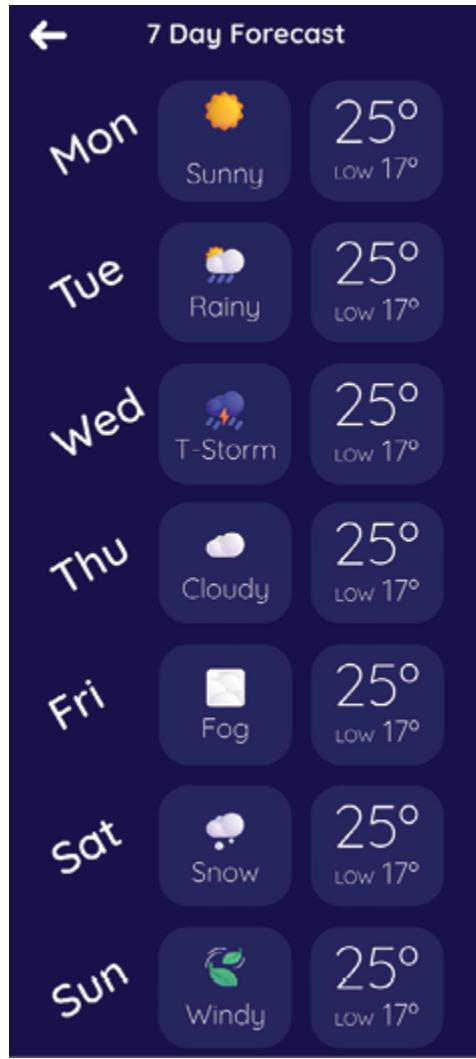


Figure 12.2: The 7-day forecast screen mockup

As you can see, the text has been rotated slightly. While not particularly difficult challenges to solve, these represent some real-world design challenges you may come across as a Flutter developer. Being able to quickly recognize and reconcile issues such as this is a skill you will need to develop. The final screen in the application is a very basic Settings screen, where the user can change between Imperial and metric display units (*figure 12.3*):



Figure 12.3: The settings screen

While basic, this screen could also be used to add further customizations should you so desire. Perhaps you would like to let the user specify which city to show the weather for. Or, maybe you would like to let the user swap between multiple icon sets, toggle between light and dark mode, adjust the update frequency, or swap between data sources (such as different weather providers). For now, we will leave this section quite Spartan, but feel free to experiment with it later.

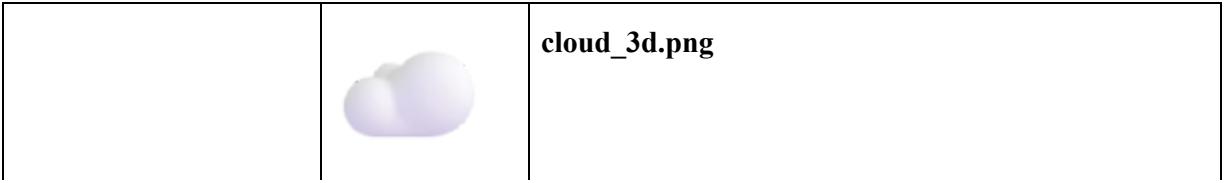
Let us explore some of the more technical details of the design, starting with the icons. As previously mentioned, these icons are from Microsoft's free and open-source icon set, which was published on GitHub in 2022, *making them free for creators to remix and build upon (Tom, 2022)*. The following is a selection of the emoji which Microsoft has open-sourced (*figure 12.4*):



Figure 12.4: A selection of Microsoft's free and open-source Fluent Emoji. Source: GitHub/Microsoft

You can download the full icon set, for free, from Microsoft's GitHub repository: <https://github.com/microsoft/fluentui-emoji>. From this emoji set, we will be pulling out several icons which specifically apply to weather, including moon phases, weather icons, and a couple of other related icons. The following is a table (*table 12.1*) of the images and relevant file names for the specific icons we need:

Weather



	<code>cloud_with_lightning_3d.png</code>
	<code>cloud_with_lightning_and_rain_3d.png</code>
	<code>cloud_with_rain_3d.png</code>
	<code>cloud_with_snow_3d.png</code>
	<code>fog_3d.png</code>
	<code>leaf_fluttering_in_wind_3d.png</code>
	<code>sun_3d.png</code>
	<code>sun_behind_cloud_3d.png</code>
	<code>sun_behind_large_cloud_3d.png</code>

		sun_behind_small_cloud_3d.png

Moon		first_quarter_moon_3d.png
		full_moon_3d.png
		last_quarter_moon_3d.png
		new_moon_3d.png
		waning_crescent_moon_3d.png
		waning_gibbous_moon_3d.png
		waxing_crescent_moon_3d.png
		waxing_gibbous_moon_3d.png

Miscellaneous		droplet_3d.png
		level_slider_3d.png
		thermometer_3d.png
		wind_face_3d.png

Table 12.1: The selection of Fluent Emoji chosen to be used in the example application.

Finally, let us explore the color palette that the design calls for. If you are reading the print version of this book, you may find that the colors are quite dark and difficult to differentiate. Once you get to building the application, you may find that the subtle shades of blue are quite pleasing to look at. Let us first start by defining what the colors are (*table 12.2*):

	Color in RGB	Color in Hex
Primary	rgb(15, 4, 76)	#0F044C
Secondary	rgb(20, 30, 97)	#141E61
Accent	rgb(120, 122, 145)	#787A91
Text	rgb(238, 238, 238)	#EEEEEE

Table 12.2: The color palette of our application

Let us try to put these colors into context by seeing where they are applied within our application (*figure 12.5*):



Figure 12.5: Applying the color scheme to the design

Now that we understand the visual components of the application, let us look at the technical requirements.

Understanding the Technical Requirements

At the heart of our application will be data provided by an API. There exist many different weather API providers, each offering slightly different services and levels of detail. Before we can go about choosing a provider, we need to start by identifying the data we will need to consume. It is also a good idea to look at the pricing that the provider offers. Often, there is a generous free tier, but if you exceed the limit, there will be a cost per API call. Do you plan to use the application only for yourself, or release it to the

world? If thousands of people are using your application, how many API calls do you expect each person to use per day? What are the potential costs that you will need to offset?

Costs aside, let us focus on the data we need from an API provider and assume that our usage will fall within their free tier. By examining the design screenshots, we can start to build up an idea of what we need our API provider to give us:

- Current weather, including temperature, conditions, humidity, and wind speed and direction.
- Current moon phase.
- Hourly forecast, including temperature and conditions.
- Seven-day forecast, including conditions and high/low temperatures.

A *nice to have* would be the ability to specify a measurement unit to the API so we do not have to do the math to figure out °C from °F and vice versa; however that is a relatively simple calculation, so it is not wholly necessary.

One weather API provider which meets all the needs outlined preceding, including our *nice to have* is Visual Crossing Weather (<https://www.visualcrossing.com/>). The documentation for Visual Crossing Weather's API can be found at the following location: <https://www.visualcrossing.com/resources/documentation/weather-api/timeline-weather-api/>. You will need to register an account to use this service, as only by registering an account will you be able to acquire an API key, which will be sent with every API request to authenticate your request (and track your usage so that you can be billed.) Take some time to look over the documentation and see if you can match up the application's requirements with specific parts of the API. The easiest way to do this is by using their query builder (<https://www.visualcrossing.com/weather/weather-data-services>). Enter a location into the builder, click the **JSON** button, and look over the JSON output of a sample weather query. Use the documentation to look up any field in the JSON response that you are not certain about. Eventually, we will be building Dart classes to handle this output, so understanding it is quite vital.

Moving on from the API for now, we need to further understand *how* our application will be built on a conceptual level. We know, of course, that there will be an API which we have to interact with. We also know that we will be giving the user an option to change the units which are displayed within the application. We can see that there is a set of data displayed on the main screen, with a second set of data being displayed on the seven-day forecast screen. Should all the data be retrieved in the same query? Should there be separate queries? Does it matter? These are all questions we need to be asking ourselves before we start writing code. The answer is up to you, the developer. However, here are some recommendations:

- We can utilize state management to toggle between different units of measurement.
- We can use local storage of some sort to save the user's unit of measurement preferences so subsequent app launches will reflect the last-chosen choice.
- Toggling between units of measurement can trigger a new API call to be performed, using the preferred unit of measurement as a parameter in the call.
- By using state management to perform our API queries, we can display status updates to the user such as *getting location*, *getting weather*, and in the event of a failure, *error*.
- Separating the state of the user's preferences from the state of the API calls is probably a good idea.

The final question you will need to answer for yourself (and there is no single, right answer) is how the weather will be fetched/updated after the app has been launched. Will the weather be updated *at all*? Will there be a pull-to-refresh? Will there be a timer to automatically update the weather on a given interval? (If so, what will the timer interval be? Should it be configurable?) Or will there be a combination of timer *and* pull-to-refresh?

Building an application from a given design and set of requirements can often be more planning than coding. Without adequate planning, you may end up wasting a significant amount of time by writing code only to realize that your API provider does not quite give you everything you need, requiring you to change providers and redesign all your classes. It could also mean wasting time in developing a state management system that you

end up throwing away in favor of a different solution, or something similarly time consuming. As the saying goes: measure twice, cut once. Spending more time understanding the requirements and expectations of an application and planning it all out in the beginning will make building the application itself much, much easier.

Now that we have a firm understanding of both the design and technical requirements of the application which we will be building, let us dive into the actual process of building the application itself.

Building the Skeleton of Your Application

As with every new Flutter project, we can use `flutter create weather` to create a new application. This will leave us with a `main.dart` that includes the standard counter application, which we can remove and replace with our own widgets. There are several components to our application which will need to be organized effectively. Up to this point, we have not discussed how to keep files organized in a project, but having an organized codebase is vital. There are no rules to define how your code should be organized, and if you are joining an established team with an existing codebase, try to understand and utilize their organization structure before making up your own. However, if you are starting from scratch, you will not have anything existing to adapt to. In that case, here is a suggestion for keeping your code organized:

```
📁 lib/
  |- 📁 common/
  |  |- 📄 theme.dart
  |  |- 📁 classes/ Common data classes used throughout the application
  |  \- 📁 widgets/ Widget(s) which are used in multiple features
  \- 📁 features/
    \- 📁 my_feature/
      |- 📁 api/ Feature-specific API files
      \- 📁 state (or bloc)/ Files related to state management
```

```
|   |-  classes/ Any data class(es) specific to the given feature  
|   |-  screens/ The screen(s) related to your feature  
|   |   L  feature_screen.dart  
|   L  widgets/ Widget(s) which are specific to your feature  
L  main.dart
```

If your application is simple, it is okay to forego the `common` and `features` directories, instead bringing their contents up a level to the `lib` directory. Either way, this is only a suggestion, and you may find a method which makes more sense to you.

When building out a new application or feature, often the easiest way is to use *dummy data* to get the layout right before ever making an API call. So, what does that mean and how does it look, in practice?

Let us say you are building the widget on the home screen which displays the current weather conditions (the large icon with temperature and conditions beneath it). Building that widget will require three pieces of information: the icon we need to display, the temperature, and the description of the current conditions. By taking those three parameters into the widget, we can temporarily pass in whatever data we want without making an API call.

By building our widget in such a manner, the widget does not care where the values come from, only that they exist. In contrast, if we opted not to pass any values into the widget and instead asked the widget to retrieve its values from our state management system, it would make the widget far more difficult to develop and test. When you eventually build out the portion of the application which supplies data from the API, you can easily swap the parameters used to build the widget from your dummy data to the live data. Additionally, when using dummy data to build the widget, you are easily able to test out different scenarios for wrong or missing data, which will help you build a more robust widget.

A keen-eyed reader may have noticed that one such parameter being supplied to this widget is of the class `WeatherIcon`. Let us spend some time looking over this class, how it was constructed, and what is unique about it.

The WeatherIcon Class

The Visual Crossing Weather API helpfully returns a weather icon identifier that we can use in our application. Let us start by exploring the documentation for this portion of the API: <https://www.visualcrossing.com/resources/documentation/weather-api/defining-icon-set-in-the-weather-api/>.

According to the documentation, there are two possible icon sets we can use in the application. The first icon set includes nine possibilities, while the second set includes 16. The documentation also gives us the possible icon identifiers which will be returned from the API and a description of when those icons will be used in relationship to the weather. We can use these icon IDs to map to image assets we include within our application. But, how? To explain this, we first need to discuss enums and generated code.

Enumerated types (for short, enumerations, or most commonly, simply **enums**) are a Dart type which extends the **Enum** class. An **enum** represents a fixed list of constants. Take this simple **enum** example: `enum Answers { yes, no, maybe }` wherein the list of possible answers can only ever be yes, no, or maybe. Referencing a value from the **enum** would work like this: `Answers.yes`. Enums are quite powerful in that all possible conditions are guaranteed. When building a widget that has a different state for a given set of enums, you can be *guaranteed* that the possible conditions in an **enum** will not change.

Enums are extremely powerful, but also quite limited. Since they only represent a constant list of options and nothing more, we have no way to assign values to the items within an **enum**. Or...do we?

In fact, we do. Dart offers *enhanced* enums, which allows us to extend the functionality beyond the base functionality provided. For example, we can create a getter which associates a value with an **enum** value:

```
1. enum Answers {  
2.     yes("Happy"),  
3.     no("Sad"),  
4.     maybe("confused");
```

```
5.         
6.     final String feeling;
7.         
8.     const Answers(this.feeling);
9. }
```

Here, we are mapping a `feeling` string to an answer. To get the feeling, we can simply call `Answers.yes.feeling` which would return `Happy`. Since we can use methods to return a value, we can create all sorts of complex getters or, really, do anything we want. When we create our `WeatherIcon` enum, we can map a specific value to the location of an image asset included within our application. Before we get to that, however, we need to talk about code generation.

Code Generation

When we are building classes for our application, especially when we expect to instantiate an instance of that class using JSON data from an API, we need to define a factory constructor which will take in the JSON data and convert it into an instance of our class. We might also want to create a method which converts the class *to* a JSON object, in case we want to send it to an API as a value. Doing this by hand requires us to maintain a careful mapping of each possible value, how it maps to our class, and necessitates us considering whether that value can be null. With a single parameter, and this is what building such a class might look like:

```
1. class MyClass {
2.     final String value;
3.         
4.     const MyClass({required this.value});
5.         
6.     factory MyClass.fromJson(Map<String, dynamic> json) {
7.                  return MyClass(
```

```
8.         value: json["value"],  
9.     );  
10.    }  
11.  
12.    @override  
13.    int hashCode {  
14.        return Object.hash(  
15.            runtimeType,  
16.            value,  
17.        );  
18.    }  
19.  
20.    @override  
21.    bool operator ==(Object other) {  
22.        return other is MyClass &&  
23.            other.runtimeType == runtimeType &&  
24.            other.value == value;  
25.    }  
26.  
27.    MyClass copyWith({  
28.        required String value,  
29.    }) {  
30.        return MyClass(  
31.            value: value,  
32.        );  
33.    }
```

```
34.  
35. Map<String, dynamic> toJson(MyClass myClass) {  
36.     return <String, dynamic>{  
37.         "value": myClass.value,  
38.     };  
39. }  
40.  
41. @override  
42. String toString() {  
43.     return 'MyClass('  
44.         'value: $value'  
45.     ')';  
46. }  
47. }
```

That is a whole lot of code just for a single value! Plus, the object we can create from this class lacks a whole lot of other, useful features. Creating all these methods for every class we create is needlessly time-consuming, error-prone, and not very fun. Would it not be nice if we could let Dart write all that code for us?

The `freezed` package, by the indomitable *Remi Rousselet* ([@rrousselGit](#) on GitHub), allows us to accomplish just such a task. Freezed is an extremely powerful and popular package with many features, all of which are geared toward the task of automatically generating code. To get started with Freezed, first enable it in the command line:

1. flutter pub add freezed_annotation
2. flutter pub add --dev build_runner
3. flutter pub add --dev freezed
4. # if using freezed to generate toJson/fromJson, also add:

5. flutter pub add json_annotation
6. flutter pub add --dev json_serializable

Then, we can rewrite our class using a decorator. You have seen a decorator before, even if you did not realize or understand it at the time. Both the `@immutable` and `@override` lines are decorators, for example.

Here is the exact same class we created by hand, but this time we are getting ready to use code generation:

1. `@freezed`
2. `class MyClass with _$MyClass {`
3. `const factory MyClass({`
4. `required String value,`
5. `}) = _MyClass;`
- 6.
7. `factory MyClass.fromJson(Map<String, dynamic> json) =>`
8. `_MyClassFromJson(json);`
9. }

Of course, we will still need to generate our code (then regenerate it each time we change the class definition), but that is a small price to pay. We will also need to import the generated file. A complete example would look like the following:

1. `import 'package:freezed_annotation/freezed_annotation.dart';`
- 2.
3. `part 'my_class.freezed.dart';`
4. `part 'my_class.g.dart';`
- 5.
6. `@freezed`
7. `class MyClass with _$MyClass {`

```
8. const factory MyClass({  
9.     required String value,  
10. }) = _MyClass;  
11.  
12. factory MyClass.fromJson(Map<String, dynamic> json) =>  
13.     _$MyClassFromJson(json);  
14. }
```

Then, to generate the file, we would run (from the root of our project) the following command: `dart run build_runner build --delete-conflicting-outputs`. Once `build_runner` has finished, our code will have been generated and we can use it as normal.

So, how does this relate to our `WeatherIcon` enum and why do we even *need* code generation in the first place? Let us start by taking another look at the possible icon identifiers that icon set 2 can return from the API:

- snow
- snow-showers-day
- snow-showers-night
- thunder-rain
- thunder-showers-day
- thunder-showers-night
- rain
- showers-day
- showers-night
- fog
- wind
- cloudy
- partly-cloudy-day
- partly-cloudy-night
- clear-day

- clear-night

Some of the possible IDs have a hyphen in them. Dart does not like hyphens in the names of things. Creating an `enum` for the weather IDs as-is would cause Dart to complain endlessly about those items with hyphens. Stringing together words with hyphens is called **kebab case**. Dart prefers to use Pascal case for class names and camel case for other types of names. What is the difference, you ask? This ought to give you an idea:

- camelCase
- PascalCase
- snake_case
- kebab-case

So, if we create an `enum` with items that exactly matches the values the API will return, Dart will get upset. Yet, we cannot change the values coming from the API. So, how do we map the kebab case variables to our camel case `enum`? This is where Freezed comes into play. Specifically, the `freezed_annotation` package, which is itself useless without Freezed.

This package will tell Freezed that a given value should be mapped to another value. This could be in a class definition when defining a property or in an `enum`, such as in our case. We use this by using a `@JsonValue` decorator on the property in question. Let us apply this to an `enum` to see how it works:

```
1. import 'package:freezed_annotation/freezed_annotation.dart';
2.
3. enum WeatherIcon {
4.   @JsonValue('clear-day')
5.   clearDay,
6.   @JsonValue('clear-night')
7.   clearNight,
8.   cloudy,
9.   fog,
```

```
10. @JsonValue('partly-cloudy-day')
11. partlyCloudyDay,
12. @JsonValue('partly-cloudy-night')
13. partlyCloudyNight,
14. rain,
15. @JsonValue('showers-day')
16. showersDay,
17. @JsonValue('showers-night')
18. showersNight,
19. snow,
20. @JsonValue('snow-showers-day')
21. snowShowersDay,
22. @JsonValue('snow-showers-night')
23. snowShowersNight,
24. @JsonValue('thunder-rain')
25. thunderRain,
26. @JsonValue('thunder-showers-day')
27. thunderShowersDay,
28. @JsonValue('thunder-showers-night')
29. thunderShowersNight,
30. wind,
31. }
```

Now, when we map the JSON output of the API to our `enum`, those pesky kebab case names will map nicely to our camel case names! Feel free to complete this by creating a getter which maps the image asset to the relevant `enum`. You may wish to map the enums to these strings:

1. WeatherIcon.clearDay: 'assets/images/weather/sun_3d.png',
2. WeatherIcon.clearNight: 'assets/images/moon/full_moon_3d.png',
3. WeatherIcon.cloudy: 'assets/images/weather/cloud_3d.png',
4. WeatherIcon.fog: 'assets/images/weather/fog_3d.png',
5. WeatherIcon.partlyCloudyDay:
'assets/images/weather/sun_behind_small_cloud_3d.png',
6. WeatherIcon.partlyCloudyNight:
'assets/images/weather/sun_behind_small_cloud_3d.png',
7. WeatherIcon.rain: 'assets/images/weather/cloud_with_rain_3d.png',
8. WeatherIcon.showersDay:
'assets/images/weather/sun_behind_rain_cloud_3d.png',
9. WeatherIcon.showersNight:
'assets/images/weather/sun_behind_rain_cloud_3d.png',
10. WeatherIcon.snow:
'assets/images/weather/cloud_with_snow_3d.png',
11. WeatherIcon.snowShowersDay:
'assets/images/weather/cloud_with_snow_3d.png',
12. WeatherIcon.snowShowersNight:
'assets/images/weather/cloud_with_snow_3d.png',
13. WeatherIcon.thunderRain:
'assets/images/weather/cloud_with_lightning_and_rain_3d.png',
14. WeatherIcon.thunderShowersDay:
'assets/images/weather/cloud_with_lightning_3d.png',
15. WeatherIcon.thunderShowersNight:
'assets/images/weather/cloud_with_lightning_3d.png',
16. WeatherIcon.wind:
'assets/images/weather/leaf_fluttering_in_wind_3d.png',

Now that you have created your `WeatherIcon` enum, you can use it to display an image in your layout. Go back to your `CurrentConditions` widget and plug in a value from your `enum` to see the image!

Code generation is an incredibly powerful tool which will save you countless hours of typing and tediously debugging code. Do not be afraid to use it often!

Building the API Layer

Our weather application will need to communicate with the Visual Crossing Weather API to get the current conditions and forecast. We previously discussed communicating with APIs in *Chapter 7, Working with APIs and Asynchronous Operations*. Now, we are going to expand upon the knowledge from that chapter to create our own **WeatherApi** class.

The idea behind our **WeatherApi** will be that we can create an instance of it and ask it to get the weather for us, like this: `WeatherApi().getWeather`. Any time we need to fetch the weather, we can call the same method. We expect that this call will return a **CurrentWeather** class, containing all the data we need from the API to build the widgets in our application. That means that our **WeatherApi** will do some sanity checking and error handling before returning a value to us.

There are several different ways to do error checking. It is possible to use a try-catch to account for errors. However, this can sometimes make debugging difficult. A try-catch would look something like the following:

```
1. try {  
2.     final Response response = await http.get('https://my-api');  
3.     final CurrentWeather weather =  
        CurrentWeather.fromJson(response.data);  
4.     return weather;  
5. } catch (e) {  
6.     throw Exception(e);  
7. }
```

The downside of using a try-catch here is that we do not know *what* broke. Was there an HTTP error in fetching the response from the API? Was there a problem with the returned data and we could not build a **CurrentWeather**

object from it? We just don't know. In some scenarios, a try-catch is useful, but it could be argued in this case that it is not. Let us try a different method, instead:

```
1. final Response response = await http.get('https://my-api');  
2. CurrentWeather? weather;  
3. if (response.statusCode == 200 && response.data != null) {  
4.     weather = CurrentWeather.fromJson(response.data);  
5. }  
6. return weather;
```

In this method, we first create a variable of type `CurrentWeather`, which is nullable (and null). Then, we check to see if our response had a successful status code (in this case, HTTP 200) and that there is data in the body of the response. If those checks pass, we build the new `CurrentWeather` class using that data. If we wanted to, we could also verify the object that we build is a valid object before returning it.

Let us take this knowledge and finish building out the rest of the API. We are going to need a few different things to do this properly. First, we will need some sort of HTTP client. Which one you choose to use is up to you. In this example, we will be using Dio. Next, we will need to know what URL to send our queries to. This can be found in the Visual Crossing Weather API documentation but will also be listed in the following code. We will also need the user's location, the API key to authenticate our requests to the API, and some options to send along to the API. Let us first look at the API key.

An API key is a string of text which uniquely identifies you to the API, much like a username and password. Often, this key will be included in the URL, although sometimes it can end up in the request headers, which are options passed along to the API but are not explicitly defined in the URL. What is important is that this token acts like credentials and should be stored safely and securely, and not checked into source control. One way we can safely store your API key and other secrets is to use an environment variable.

Environment variables are variables that are unique to the environment the code is being run on. That could be a variable set in the command line before running the application or a variable loaded from a file that does not get checked into source control. We will be using the second method. To get started, we need to create something called a dotfile. In Linux and macOS, unlike in Windows, files which begin with a period in the filename are considered hidden files — they are dotfiles. We are going to create a dotfile in the top-most level of our project, in the directory where you would find the `lib` folder. Name this file `.env` and then create a `.gitignore` file if it does not already exist, then add the following lines to it:

1. *# Environment Variables and Secrets*
2. `.env`
3. `*.env`
4. `**/.env`

This will ensure that any environment variables saved in a dotfile with a filename of `.env`, located anywhere in your project, are not checked into source control, preventing your secrets from being leaked. Next, open the `.env` file and add a line like this:

1. `API_KEY = "my-api-token"`

You can find the key to put in here by logging into your account at <https://www.visualcrossing.com/account> and then looking for the `key` section. Once your secret is set, head over to Pub and learn how to activate the `flutter_dotenv` package for your project. Next, head over to your `main.dart` and make the following change:

1. `Future<void> main() async {`
2. `await dotenv.load(fileName: ".env");`
3. `runApp(const MyApp());`
4. `}`

Finally, you can load up the variable in your API class like the following:

```
1. final String? apiKey = dotenv.env['API_KEY'];
```

Now that your secrets are safe, let us get the user's location. We can use the `location` package to easily get the user's location; however, there is a bit of extra setup work, so be sure to read and follow the documentation. Once you have got the package set up properly, you can get the location like the following:

```
1. final LocationData location = await getLocation();  
2. final double? lat = location.latitude;  
3. final double? lon = location.longitude;
```

If we cannot acquire the user's location, we will want to throw some sort of an error, so be sure to check for that. Otherwise, we have everything we need to craft the URL for our request. Again, be sure to reference the API documentation for exactly how to do this. What you will see in the following should work, however, you will need to understand what is being requested and why before you go about implementing it.

```
1. const String baseUrl =  
'https://weather.visualcrossing.com/VisualCrossingWebServices/rest/  
services';  
2. final Response response = await  
http.get('$baseUrl/timeline/$lat,$lon?  
unitGroup=$unit&include=hours%2Ccurrent&key=$apiKey&content  
Type=json');
```

Also note that we pass in a variable for the units. This could either be `us` or `metric`. If you recall, the designs had a preferences screen where the user can toggle between these units. Whatever way you decide to implement the preferences, you will want to read that variable from the user's preferences.

Finishing the application

The most challenging parts of the application are out of the way; the rest will be up to you to build. Have some fun with it! The important thing to keep in mind is that you are still learning, and this is the first time you have

been asked to build something from scratch. You *will* make mistakes. That is just part of the learning process.

If you get *really* stuck, feel free to reference the code at the end of this chapter, but try to figure things out on your own before resorting to doing so. If you get stuck on a particular subject, be sure to check out the previous chapters in this book, as the application was designed in such a way as to take advantage of the knowledge presented throughout this book with the only new knowledge needed to build the application having been presented in this chapter.

If you find that nothing else has worked, you may wish to start searching online for answers. The website Stack Overflow (<https://stackoverflow.com/>) is not only an excellent resource for asking questions and discovering whether your question has already been answered, but also has a reputation on the internet for being the only way programmers learn to write code (*figure 12.6*):



Figure 12.6: A running joke in the industry. Original artwork by Edwin Hoffman.

When you finish building your application, be sure to post screenshots online to gather feedback and showcase your hard work! Sharing the fruits of your labor is one of the most rewarding parts of software development. Plus, your friends and family will be awestruck by what you have accomplished!

Conclusion

Building your first application is no small thing. Even an application as simple as our weather application can be quite complex with a lot of challenges to solve. As you begin working in the industry, whether you are taking on clients or working for another company, you will end up applying many concepts from this chapter in your daily work. Being able to take a design and break down the technical requirements from it, plan out the work, and effectively organize your code will be a recurring theme for you. While it is easy to get carried away and just start writing code without any forethought or planning, doing so will only cause you frustration in the long run.

Without a doubt, this was the hardest chapter in the entire book. A lot was asked of you here, and by making it this far, you deserve to feel proud of what you have accomplished! In the upcoming chapter, we are going to deep dive into finding a job as a Flutter developer. Not only will we learn how to hack the system and make the jobs come to us, but we will get direct insights from several of the top Flutter recruiters. This is exciting stuff that you will not want to miss.

Questions

1. What does *open-source* mean? What are some different open-source licenses?
2. What are some factors to consider when selecting a data provider, such as a third-party API?
3. Why is it important to keep your code organized? What are some ways you can organize it?
4. What is an `enum`?

5. What is a decorator?
6. What is code generation?
7. What package can be used to generate code?
8. What case style does Dart prefer for class names? For variable names?
9. How can you keep secrets, such as an API key, safe?
10. What are some good resources for finding answers to programming questions?

Key Terms

- **Dotfile:** A file with a filename prefixed with a period. In Linux and macOS, dotfiles are hidden files.
- **Enum:** Enumerated types, or enumerations, are a Dart type which represents a fixed list of constants.
- **Hex:** Short for hexadecimal; a base-16 method of representing numbers. Numbers in hexadecimal range from 0-9 and A-F, with A being equivalent to 11.
- **Mockup:** A non-functioning representation of an application, often provided by an app designer, that depicts the desired look and feel of an application
- **Open-source:** Resources, often code, which are freely available to examine, use, and change. There exist many open-source licenses, such as GPL, BSD, and CC. Understanding the limitations and privileges of each is vital to not being sued for improper usage.
- **RGB:** Standing for Red, Green, and Blue, this is a numerical way to represent colors. Each color channel can be represented as a value from 0 (there is none of that color) to 255 (that color is at full intensity).

Command Reference

- `flutter run build_runner build --delete-conflicting-outputs`: (Re-)generates code using `build_runner`.

Further Reading

- Read more about case styles:
<https://betterprogramming.pub/string-case-styles-camel-pascal-snake-and-kebab-case-981407998841>
- For more information about decorators, it is worth looking over the Wikipedia page which describes the decorator pattern:
https://en.wikipedia.org/wiki/Decorator_pattern

References

- Tom, W. (2022, August 10). *Microsoft open sources its 3D emoji to let creators remix and customize them*. Retrieved from The Verge: <https://www.theverge.com/2022/8/10/23299527/microsoft-emoji-open-source-creators>

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](https://discord(bpbonline.com)



Chapter 13

Finding Flutter Jobs

Introduction

Congratulations, Flutter developer! By now, you should have all the necessary skills to write a Flutter application from scratch. It is time to turn that skill into a paying job. Whether you have worked in the industry before or this is your first time, the process can feel daunting. There is good news on the horizon, though: with the advice in this chapter, you will find that the jobs will come to you. Plus, we will get all our questions answered about things like compensation, job titles, and more!

Structure

In this chapter, we will discuss the following topics:

- How to hack the system and make the jobs come to you
- Question and answer time with top Flutter recruiters

Objectives

By the end of this chapter, you will know exactly what to do to find your first paid Flutter job! You will learn about the power of the Flutter community and how to put it to work for you. Then, we will have a question-and-answer session with some of the top Flutter recruiters out there, which will demystify things like what companies are looking for,

how to differentiate yourself, how to approach the conversation of compensation, and what sorts of things you can do to attract jobs and recruiters to you.

How to Hack the System and Make the Jobs Come to You

Job searching is not much fun. It is stressful and requires the same level of time and energy as a full-time job. *Nobody* enjoys the job hunt. Thankfully, there are more Flutter jobs than Flutter developers right now. More and more companies are seeing the benefits of using Flutter over developing native iOS and Android applications, which means more and more companies are building up their Flutter teams every day. With so many opportunities available, you might be surprised to hear that *many* of those jobs are never posted online (although many others *are*). So, if they are not posted online, how are you supposed to find them? And what does it mean to make the jobs come to you? Let us dig into this a bit deeper to understand what is going on.

The Flutter community is unlike any other developer community. There exists a genuine interest among Flutter developers in spending time helping the developers around them. What Flutter has is a very friendly, welcoming *community*. As Flutter is still relatively new to the market, many developers started learning at the same time, seeking answers from others, and crossing paths during their journeys. These connections often manifest in continued contact via LinkedIn, Discord, Matrix, Slack, or other social platforms. The result is that when a company needs a new developer, the existing developers will first reach out to their network of contacts, which they have cultivated throughout their journey. Most jobs are filled by candidates with a personal connection to other developers. It is only when nobody in the network of contacts is available for hire that other avenues are explored.

If a candidate is not found via internal referral, most companies prefer to outsource the hiring process to professional recruiters whose full-time job consists of connecting candidates with companies. Since recruiters get paid to place candidates in well-suited roles, it is in their best interest to get you hired with the right company, in the right role, as quickly as possible. Building lasting relationships with recruiters, then maintaining those

relationships once you have been hired will make any subsequent job searches extremely easy. In fact, the more relationships you build with recruiters (and the better those relationships are), the less work you will have to do when seeking a job. So, let us talk to some recruiters and see what they have to say about the subject.

Question and Answer Time with Top Flutter Recruiters

There exists no finer source of wisdom when it comes to finding a job as a Flutter developer than the recruiters who seek out Flutter developers all day. In speaking with some of today's top Flutter recruiters, we have got an exclusive opportunity to learn just exactly what they are searching for, how they find you, and what they need to get you hired. Let us make a rundown of the questions asked, as well as the collected answers provided by these talented individuals.

1. Where do you usually begin your search for Flutter developers?

Ans: The search for a Flutter candidate almost always begins with the internal company database. This database is a combination of individuals who have previously been placed, as well as those from sites such as LinkedIn, Indeed, Dice, and CareerBuilder. Sometimes, however, recruiters will turn to LinkedIn first. If a recruiter's initial search does not yield a suitable candidate, they will often reach out to personal connections made by talking to the network of Flutter developers that have been curated over time. As Flutter is still relatively young, the community of developers is quite intimate. When seeking out a candidate, personal connections to other, more established Flutter developers will help recruiters find less-established developers. Building these connections will not only expose you, the jobseeker, to new ideas and concepts but also to a network of developers who can, and often will, help you get hired. Many Flutter jobs are never posted to job boards or on LinkedIn because they are often filled by a candidate who is somebody's connection, without having to resort to opening the floodgates to external candidates.

2. How much do you find the adage, “it is not what you know but who you know,” still applies today?

Ans: This could not be truer today! The connections you build, both horizontally (with other Flutter developers) and vertically (with former managers, supervisors, and team leads), will be the key to unlocking new opportunities. If you know somebody who got connected with their job due to a recruiter, ask them about the recruiter and see if they will connect you. If you were on good terms with your former supervisor, ask them if they would be willing to act as a reference and/or write a review on your LinkedIn. Many companies will want to check your references, and this could be a make-it-or-break-it decision.

If there is a particular Flutter developer, you look up to, get connected and see if they can help mentor, teach, and assist you in developing your career as you learn. Be cognizant of their time, however. If you are asking them for a lot of their time for free, this could backfire. Instead, show them that you are self-motivated and that helping you is worth their time. Be pleasant, understanding that you are not their priority, and humble. Who knows what doors this could open!

When you talk to other Flutter developers who have a job, ask them questions. “How did you get into Flutter? How did you get this role? What other resources and experience did you find helpful once you started working? What would you recommend to someone like me to do to enhance my profile and land a job like yours? Is your team hiring? Can we stay connected so that when your team is hiring, you will keep me in mind?” Asking these questions to enough people, even if you are not actively looking for a role, will help you come across new projects and opportunities you otherwise would have never known existed!

3. When searching for Flutter developers for a role where you have multiple candidates available, what goes into the decision-making process when deciding which candidate to approach first?

Ans: Recruiters will first look at a candidate's latest experience to see how closely it aligns with the requirements of the job. Then, they will look at any adjacent technologies that you have got experience with, which might indicate further levels of expertise. Ideally, this experience would be gained while working on a large-scale application or at a bigger company (regardless of whether Flutter was involved), but it could also be gained through personal projects. What is important is that you show that you are working on things that the client cares about and demonstrate that you are always growing and learning.

Aside from the technical requirements, recruiters want to know that you can effectively communicate in a pleasant manner with timely responses. Recruiters have a job to do, too, and by showing respect to them and their time, you are much more likely to gain an ally that is willing to fight for you. Whether you are messaging on LinkedIn, communicating via e-mail, text, or talking on the phone, it is your responsibility to be professional, speak clearly and concisely, and respond quickly. After all, if you demonstrate this early that the recruiter and/or their client is not worth your time and respect, neither will want to work with you.

4. Outside of Flutter, what other skills are employers asking for? Are there any must-haves that are a deal-breaker if the candidate lacks them?

Ans: Communication skills are an absolute must, followed closely by soft skills. Soft skills are those skills that are non-technical in nature, such as critical thinking, problem-solving, adaptability, dependability, work ethic, time management, and conflict resolution, to name a few. The most brilliant developer in the world will never be able to hold a job long if they are not able to demonstrate these soft skills and get along with their peers. When communicating, speak slowly and clearly, especially if your accent differs from that of the person you are communicating with. It is normal to speak quickly when you are excited or nervous, so this skill will take practice, like any other. You will need to be able to demonstrate that you are actively listening to the questions being asked. The clearest

sign of a good communicator is one who asks questions to ensure they fully understand what they are being told.

You will also need to learn to communicate effectively at the level of the person you are talking to. While recruiters and HR professionals may have some limited understanding of the technologies you are working with, they do not have the depth of knowledge that you do. Being able to break down highly technical topics in such a way that anybody can understand, regardless of their technical aptitude, will show them that you understand the concepts well. You will need to learn to evaluate the technical level of the person you are talking to. You could start by asking them directly how familiar they are with the technology and what level they would like you to explain something at. You could also start by explaining a topic in simple terms, evaluate their response, and slowly move on to describing things in a more technical manner. Again, this is a skill you will learn to develop over time.

Another important skill that employers are asking for is the ability to learn new technologies and techniques. By picking up this book, you have already helped to prove to future employers that you have a growth mindset and interest in learning new technologies. This sort of attitude is very appealing to employers, as employees who are constantly growing and learning will consistently produce higher quality work than those who are complacent in life.

Cultural fit is very important for both candidates and employers. If you were the perfect candidate in every way but did not get along well with the team, nobody would be happy. Although not strictly a skill, it does qualify as a deal-breaker. It is not unheard of for candidates to be rejected for a role because the employer saw a less-than-savory post made by the candidate on LinkedIn, they demonstrated an unwillingness to share knowledge with the team or indicate in some way that they are not going to be a team player. If an employer is given a choice between a junior Flutter developer with a growth mindset, great communication skills, and a good cultural fit with the team and a mid-level Flutter developer with more experience but lacks soft skills, the employer is more likely to bring

on the junior developer and give them the opportunity to grow their skills.

Finally, it is important for you to be able to sell your skills. Do not be afraid of talking up projects you have worked on, whether they are personal or professional, and problems or challenges you have had to overcome. Speaking about your weaknesses and what you have done to learn and grow because of those weaknesses will speak volumes about your character. Likewise, it is acceptable to talk up your experience, even if it is not directly related to Flutter. (Maybe you have managed a team, even if it is not a software development team. An employer might want to know that you have those skills for the future!) Just remember not to be over-confident in your abilities. It is perfectly acceptable to say, *I do not know, but I am curious to learn!*

5. If a candidate has all the necessary skills for a role, what are some reasons a company might pass on them?

Ans: This question assumes the candidate is at fault, which is not always the case. Sometimes it is as simple as the budget for the project changing, so the role is no longer feasible to hire for. As a candidate, do not be discouraged and take it personally if you are passed over for a role because it may have nothing to do with you.

Assuming that the company is still hiring for the role and they have chosen to pass over you, there are some things you can do to not botch the interview. First and foremost, have a professional resume writer look over your resume. Your resume is the first impression the company will get of you, and first impressions are very important. A resume with spelling, punctuation, and grammar errors will give the impression that your work is sloppy, which is not a desirable trait. Is your resume worded concisely and professionally? Do you overuse buzzwords and jargon, or is it tailored toward the people who will be deciding whether to move forward with bringing you in for an interview? Does your job experience section accurately reflect what you worked on, or is it worded too generically? For example, a bad way to word your experience would be, *Added new features, maintained the app, and fixed bugs*. A much better way to word it

would be, *Worked on X team, where my focus was X. Added the X feature to the application, using X.*

During the interview, there are several things that employers will be looking for. Did you appear interested and engaged, or were you aloof and disinterested? Do you present yourself with an ego or air of superiority? Do you read off your resume, or are you able to talk about your experience without the need to reference other materials? Are you able to talk about the projects you have worked on outside of the specific areas you were responsible for? Do you use professional language when speaking, or do you speak casually, like you would around friends?

Of course, the biggest thing that your interviewer will be looking for is your industry experience. If you are interviewing for a junior role, you may not have much, if any, industry experience. The interviewer will be aware of this, but that does not absolve you from the responsibility of showcasing your experience, even if that experience is all a personal project. This is your time to shine! What the interviewer will be looking for is how your understanding has grown, whether you have been motivated to learn, and what your process has been. It does not matter what the projects you have worked on are, only that you have worked on them. Your experience will be the thing that employers will want to focus on most. Knowing this, you may want to build several applications and develop them openly on your public GitHub.

6. While searching for candidates to fill a Flutter role, what sort of keywords are you looking for?

Ans: Since Flutter is still considered a relatively niche technology, the search terms can be as broad as *Flutter*, *Dart*, and *mobile*. After that, it depends on the needs of the client, so additional keywords will be tailored to match the specific technologies they are working with.

7. What could jobseekers do to attract your attention?

Ans: LinkedIn is your best friend! Make sure your job experience is up-to-date and clearly illustrates your job experience in detail. Your resume

should match your LinkedIn experience and be no more than two well-organized pages. A resume longer than this may signal to the client that the candidate cannot articulate their experience, which could be a red flag. When discussing your experience, using *we* term as much as *I* terms will help you stand out as a team player. A professional photograph goes a long way in humanizing you, as well. Both your LinkedIn profile and your resume should link to your GitHub profile, where you will have example Flutter projects you have worked on. There are many Flutter-centric groups both on LinkedIn and off. Joining one or more of these groups and being active in the community will help set you apart from others who lack this sort of community contribution. Finally, do not be afraid to reach out to recruiters! They are generally a very nice bunch and are happy to help you in any way they can, whether that is placing you in a role or giving advice about things you can work on to make yourself more attractive to employers.

8. Sometimes companies will put up job postings where they require several years of native app development experience, but they are just interested in a Flutter developer. How common is it for companies to ask for one thing when what they want is something completely different?

Ans: This happens all the time! The hiring process generally starts with a team's needs being expressed to human resources. Human resources (HR) will do their best to translate the team's needs into a job posting. However, it is exceedingly rare for HR to speak the same lingo as the development team for which they are attempting to fill a position in; therefore, all they may know is, *we are building a mobile application, so we need a mobile developer*. The hiring manager may also have a list of *nice to haves* that they provide to HR, which could be misconstrued as *we need all these qualities, too*. This directly translates to job postings that cannot always be trusted to be accurate to the team's needs.

For this reason, you should not look too closely at the specific requirements of the position as a candidate. If you match, say, 70% of what they are looking for, you are probably going to be fine going

into an interview. There are only really two to three must-haves for most roles, meaning everything else can be learned on the job.

9. From your perspective, how much experience does a candidate need to have to be considered junior? Mid-range? Senior? Is that measured in a number of projects completed or months/years?

Ans: When it comes to Flutter, it is harder to establish clear boundaries in this regard. Since Flutter is still so new, most developers will have between one and four years of experience with it. Traditional time-based assumptions for seniority break down when the technology is so young. Instead, there are loose milestones that developers will hit along the way, which generally sorts them into these fuzzy categories.

- **Junior:** The candidate has a foundational understanding of Flutter and the platforms they will be deploying their code on (iOS, Android, Web, and so on). They understand the bigger picture and how the technology fits into the grander objectives of the business. They will still need mentoring and guidance while they get settled in and learn the codebase.
- **Mid-level:** The candidate will have been working in the industry for a couple of years and are comfortable/capable of clearly communicating what they are working on, what challenges they face, and what help they need when called on by other developers in a meeting. They will start to write more comprehensive tests for their code and can recognize (if not implement on their own) some common software design patterns.
- **Senior:** The candidate will be comfortable working on and completing tasks independently. They will feel comfortable reviewing other developers' code and pointing out where it could be improved. When an approach is too complex or inefficient, they are comfortable speaking up about it and defending their decisions. They will have a practical understanding of all aspects of the code they are working on, how to make it more efficient, and how to fully integrate it within the larger systems. They will be making decisions that affect other developers' code.

- **Principal/lead:** A principal or lead developer can manage all aspects of the project on a technical level, minus the administrative aspects (such as being a people manager). They can communicate with the product manager and product owner about the struggles, timeline, and roadblocks the team is facing, as well as the solutions the team is exploring or implementing.
- **Staff:** A candidate who is a staff engineer will architect the project from the ground up.

10. Do companies care about a candidate's open-source contributions or the projects they have on their GitHub/Bitbucket/GitLab/and so on?

Ans: Yes! Even if these projects do not accurately showcase the candidate's current state of knowledge (perhaps they wrote some code prior to working full-time and have learned a lot since writing that code), it still gives insight into how often and how much someone is writing code outside of work, and thereby showcasing their passion for the technology. If a candidate only writes code at work, that is fine, too, but demonstrating a true passion goes a long way.

11. How important is it for candidates to be active in the Flutter community? Does having Medium articles, LinkedIn posts, and so on factor into how they are viewed by companies looking to hire?

Ans: While not essential, this sort of activity will set you apart from other candidates by showcasing your passion—your self-driven willingness to learn and grow. This is not something you would have to do daily, weekly, or even monthly, but sharing what you have learned or what you know with others can only look favorable for you.

12. If you could describe the “perfect” junior Flutter developer that you would be able to place instantly, what would they look like to you?

Ans: The candidate would have at least two years of experience in software development (not just in Flutter, and can include internships), with experience with at least one large-scale client (again, it does not need to be Flutter-related). They should have at least one app that they have written using Flutter, either for work or a client or as a side project published to the app store. It would be great if they had a computer science degree! They should view the next role as a steppingstone to help with their career trajectory rather than be wholly focused on compensation. (That is not to say they should be paid less, but it is easier to sell a candidate to a company when they have less experience if their compensation expectations are in line with that.) There still is not a well-established market rate for Flutter developers, so being flexible on compensation will make a candidate much more attractive to companies who must work within a budget.

13. What are some things that stand out to you when you look at a resume?

Ans: Candidates who have several positions that use the skills and technologies that companies are looking for, as recent experience on their resume is a good sign. Where that experience comes from matters, too. Having experience at a more well-known company may make you look more desirable, as well as having long-term employment with companies rather than a brief employment period. Education is important, too: what degree(s) do you have? From which school(s)? Have you taken any classes, workshops, or certification programs post-graduation? A consistent path of learning and experience speaks volumes about the work ethic of a candidate.

People often make the mistake of believing that it is their resume that gets them the job. In reality, it is being able to articulate as clearly as possible what is on a resume that is most relevant to the position in question.

14. When you are speaking to a candidate on the phone, what are some things you are looking for in that conversation?

Ans: When a candidate is working with a recruiter, they become an ambassador for that recruiter and their company to their client. Recruiters are constantly assessing candidate behavior to evaluate whether they will reflect positively on themselves before ever putting them in front of a client. Part of this is building trust between the recruiter and the candidate.

Openness, honesty, and transparency are so, so important. If a candidate is unwilling to express why a position may not be right for them, the recruiter cannot effectively place them. This could be related to any number of things, ranging from compensation to whether you do not feel comfortable working for a company for some reason. Furthermore, candidates should be able to speak honestly and openly about their experience. The more the recruiter understands both your weaknesses and strengths, the more they can find a better-fitting role.

Being able to speak about your experience beyond what is just on your resume is something recruiters are looking for. Try to use these talking points as a guide:

- “I mostly used [*technologies*] to build [*feature*]. By doing so, I also learned how to use [*other technologies*] on a low level, so I have some exposure to it and how to learn new technologies on the fly.”
- “I am most proud of my work on _____ because _____.”
- “_____ was a really complex part of the project, which I worked on as a sole developer.”

Being able to effectively articulate these points will emphasize parts of the resume that might otherwise be overlooked.

Recruiters ask themselves several questions during a conversation with a candidate, with the goal of determining whether they are placeable at a company. Your answers will help to paint a picture of yourself for the recruiter. The recruiter will use that picture to determine that:

- a. You will be interested in the kinds of jobs they have.

- b. They will be able to get you in front of the client for interviews.
- c. The client will be interested in you.
- d. You will get an offer from the client.
- e. You will take the offer of employment (provided it matches your expectations).

Recruiters are there to work for you, to get you placed. Treat them with kindness, respect, honesty, and openness to build trust, and they will have a much easier time finding a great fit for you at a company.

15. From a candidate perspective, how would you go about handling the discussion of compensation with a recruiter?

Ans: The subject of compensation can be awkward for many people. In some societies, it feels taboo to discuss it with colleagues or speak about it publicly. In that context, the conversation of compensation may feel uncomfortable with a stranger, such as a recruiter. After all, compensation is often tied directly to your lifestyle and livelihood, making it a very personal, emotional subject. If this is your first time working in the industry, you may have no idea what *reasonable compensation* even looks like for somebody with your experience. It is normal to feel overwhelmed at first, but you will get the hang of it.

Start by looking on sites such as Glassdoor, Fishbowl, or Reddit to see what others with similar experience in a similar role are earning. Do not assume that you should be asking for the *same* compensation, as you are not them, nor are you working for their company. However, it should help to give a general idea of what the market rate is. Another viable option is to simply ask the recruiter. You may say, *I am not sure what the market rate is for somebody with my skills and experience. I know you see profiles like mine all the time. What do you think is reasonable compensation to expect for somebody with my experience? What have other candidates been submitted at for this position?* Do not just stop at asking a single recruiter, though. If you ask the questions, be sure to ask them from several different recruiters to figure out what the average is.

Having a general idea of what compensation looks like will help guide you in the next part of the process: the *conversation*. Talk to the recruiter openly about your needs. We all have bills to pay, a lifestyle to maintain (or improve), and a desire to save for the future. If you know how much money you *need*, you can more easily have a conversation that sounds something like the following:

- *I am targeting X. If they were to offer Y, I would not hesitate to accept, but Z is the lowest I'd even consider. It sounds like a good opportunity, but if I cannot pay my bills, there is really no reason to continue the conversation with them.*

This template gives the recruiter a wealth of knowledge to help find a role that fits you and will ignore anything that falls below your minimum compensation range. Once you have a bit more experience, you may wish to further discuss compensation for each role they present to you. Some roles will have more responsibilities than others, which will likely affect how much compensation you might want for doing the extra work.

Compensation is much more than just an hourly or salary wage, however. Many companies offer additional compensation in other ways, which is sometimes offset by a lower wage. Be sure to consider the benefits package when determining whether a given wage is acceptable, and if the added benefits are not interesting to you, it may be possible to have a conversation about giving up some of those benefits in exchange for a wage increase. If you are talking about a salary, try not to back yourself into a corner by focusing on the total compensation. Instead, establish a base compensation, then ask about what is possible from there. This could be adding stock options or other benefits, for example.

Your urgency in getting hired will likely factor into your compensation decision. If the company is willing to wait for the perfect candidate, but you need a new job immediately, the company has more effective control over the compensation conversation. Remember, though, that you do not have to take *any* job for *any* reason, and you can continue the conversation with other companies if you end up in a position of feeling uncomfortable for any reason.

At the end of the day, the conversation around compensation can feel more like a dance than a binary choice. Be open and honest with your recruiter about your wants and needs, and they can help you navigate the conversation. After all, they have done this many times before and are an incredible resource for you to leverage.

16. For someone who has never worked with a recruiter before, what is some general advice you would give?

Ans: **Be transparent:** Even if you have other interviews lined up or a job offer on the table, being transparent about where you are in the job hunt will only help you. A recruiter can take that knowledge and press the client to instill a sense of urgency, which may result in more money in your pocket just to secure you for the role. It will also help the recruiter to know when you are ready to decide so they are not putting you in a position of feeling pressured to accept or decline a role before you are ready by driving the conversation with the client. Make sure the recruiter knows your strengths and weaknesses, so they can effectively leverage this when finding the perfect role for you.

Communicate effectively: Be prompt in replying when you get an e-mail from a recruiter. (This cannot be overstated!) The job market moves quickly, and wasted time means missed opportunities. Do not let this happen to you! Keep in mind that recruiters are often talking to multiple people for the same role, and if you are the first to reply, it could mean getting the first interview.

Seek to understand: The job of a recruiter is to be an information gatherer and connector of people. Ask questions, even if you think they are silly. They *know* things.

Stay connected: Even if they could not place you in a role, they were able to work with you to bring several opportunities to your attention. Next time you are looking, or thinking of looking, for a role, chances are good that they will have more positions available. You will also be able to leverage these connections when a friend is looking for a new job, too. Likewise, when looking to connect with

recruiters, ask friends and former colleagues who they would recommend working with.

Be nice: The first rule of working with a recruiter is to be nice. They are people, too, and they spend all day working with a diverse range of people, not all of whom are pleasant. Do not be one of those people! If the recruiter finds you unpleasant to work with, you can expect that the quality of service you receive in return will suffer. Treat them the same way you would advocate for a friend that you are trying to get hired at your company versus someone you hardly know.

Conclusion

When it comes to finding a Flutter job, building your network, and participating in the Flutter community will get you a long way. Finding some recruiters that you work well with and keeping in touch with them as you learn and grow, whether employed or looking for a job, will pay dividends when the time comes that you are looking to get hired. Your passion has gotten you this far, and it will carry you through to employers being interested in hiring you, but we are not finished yet. Now that you have got an interview coming up, we need to prepare for it! In the upcoming chapter, we are going to look at what you will want to do to prepare for the interview, then some dos and do nots during your interview. We will make sure you are fully prepared for what is to come and nail the interview process. It's the home stretch!

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

Preparing for and Succeeding in the Job Interview

Introduction

Whether this is your first interview or your hundredth, it never hurts to be prepared. But how do you prepare for a Flutter interview if you have never had one? What can you expect the process to look like? What kinds of questions will you be asked? What kind of questions should *you* ask? How do you answer the question, *so, what are your compensation expectations?*

For many of us, the process can be daunting, but fear not! In this chapter, we are going to break down the entire process, start to finish, and make sure that you have got all the tools you need going into the interview to be successful — and land that job!

Structure

In this chapter, we will discuss the following topics:

- The interview process, start to finish
- What to expect during a technical interview
- Culture fit interview: Evaluating a company and being evaluated
- Negotiating compensation

- Answering some interview questions
- Accepting your first Flutter job

Objectives

After completing this chapter, you will understand what a common interview process looks like when interviewing for a technical role. We will also look at the nearly 150 interview questions (plus a couple of extras!) throughout this book and answer each of them, preparing you for most questions you could be asked during a technical interview.

The Interview Process, Start to Finish

Interviewing for a technical role, such as that of a Flutter developer, is quite different than interviewing for other types of non-technical jobs. Typically, the interview process will be broken down into about four to five discrete sections, spanning multiple days and often with several different people. This may sound daunting, but once we break it down, you will understand why this is.

The first part of the interview process is typically a brief, initial conversation where you will speak to a non-technical person, such as somebody from the human resources team. If you are collaborating with a recruiter, this step may be omitted as this exploratory phase would have been done between you and the recruiter, as well as between the recruiter and the company. This conversation is where interest between the employer and you, as a candidate, is established. You will be asked questions like, *so, how did you hear about our company?* They will typically leave room for you to ask questions, as well. You can expect this to be a light, casual conversation where you will talk about your educational and technical background at a high level, and you will hear more about the business, what it is they do, and where you might fit in with their team. Have fun with this one! Just remember to stay professional. Often, this first conversation will end with setting up a time for a first technical interview, although that may come later in some instances. It can sometimes take several days or a couple of weeks between each of the interview steps, so if you do not hear back immediately throughout the process, try not to worry — they are coordinating schedules between a lot of very busy people and they might be

interviewing other candidates as well (which means more scheduling and coordination!)

The next step of the interview process is usually the technical interview. This might be a single meeting, or it might be broken into two separate discussions with some coding done in between. We will talk more about this process in a little bit since it is going to be the bulk of the interview process.

After the technical interview usually comes a culture-fit conversation. By this point, the company is confident in your technical abilities and is interested in making sure that you will get along with the team. You will probably meet with several people from across the whole team that you will be working with should the offer of employment be extended. They might be interested in your hobbies and interests outside of Flutter. Primarily, what they are looking for is whether your personality, work style, ethics, and core values align with the company's. This may be the most important part of the interview, as working for a company where you do not share the same values, or you work in a way that is at odds with the rest of the company would make for a miserable experience for both you and the company. That is why we are going to talk about this much more in-depth later in this chapter.

The final part of the interview process is the job offer itself. This is where compensation and titles will be discussed, as well as any benefits the company may offer. Compensation is not just an hourly or salary wage, though. Depending on the role and company, it may include stock options, a 401(k) (retirement) plan, stipends for attending conferences, an allowance for purchases related to the job, or many other benefits. We will discuss this more later in the chapter, as well.

Again, the order of these conversations is bound to differ from company to company, so do not be surprised if they start with the compensation conversation, then the culture-fit, and finally, the technical interview. Now that you know what to expect broadly during the interview process, let us look more closely at the areas we called out earlier, starting with the technical interview.

What to Expect During a Technical Interview

The goal of a technical interview is to evaluate your critical thinking skills and whether you can translate your thought processes into code that is well-structured and reasonably correct. They want to know how you think and whether you can explain your thought process.

It is ok not to know the answer to every question they ask, but be careful about saying, *I do not know*. Doing so does not give the interviewer much to work with to help them get things back on track. Not knowing an answer is not a dealbreaker by any means, but not managing the stress and pressure of the interview process effectively could show them that you might not be able to do your job under pressure, either. If you get nervous, you might be tempted to guess at an answer. *Do not do this*.

You will be nervous, which means your brain probably is not working at peak efficiency. You are bound to forget things that you know well; it happens to the best of us. The absolute worst thing you can do is make up answers or guess at things. Giving a blatantly wrong answer will stick with the interviewer in a way that you would rather not have them remember you for. Instead, take a deep breath and count to three. Pause. Calm yourself down. Then, ask several thoughtful questions to have them supply clarification. Not only does this give you time to think, but it might unveil just the right information that you need to connect the dots and give a thoughtful, *correct* answer.

In your technical interview, they are likely to ask you to complete some sort of *task*, then expect you to talk through the problem, devise a solution, and write some code to accomplish it. There are several different ways by which this can be accomplished, such as by asking you to whiteboard the code (that is, to write the code on a whiteboard — sometimes literally, if you are interviewing in person), completing an online coding challenge (such as by asking you to complete a challenge on HackerRank), or perhaps even by allowing you to complete the coding exercise in your own time before the interview. In that case, you would be asked to submit the code several days prior to the interview so that they could look it over before talking to you. It is common for them to ask you to change the code during the live portion of the call if you have completed the coding exercise beforehand. If you are whiteboarding the issue, think aloud. Remember, they are there to evaluate your thought process and critical thinking skills. Do not let them guess what you are thinking.

Usually, the technical interview will involve two or more people from the company you are interviewing with. You should expect the hiring manager and a senior member of the team. They will be evaluating you for more than your technical abilities and critical thinking skills, so be personable and feel free to engage in the non-technical banter at the beginning of the interview process. This is designed to both put you at ease before asking you to delve into your technical ability, as well as to get to know you as a person. This is a wonderful time to talk about non-work-related things that you enjoy, such as sports, travel, cooking, or your favorite book.

The Culture Fit Interview: Evaluating a Company and Being Evaluated

The interview process is more than just a company evaluating *you*. It is also about you getting to know *them*. As we discussed before, the usual next step after a technical interview is a culture-fit conversation. This might be the most important part of the interview, because if you and your potential employer and/or new team do not work well together, things simply will not work out, and you will be back to looking for a new job in no time. It is easy to write this part of the interview off, especially when you are out of work, but give it the time and attention it deserves. Your mental health is not worth the paycheck.

So, how do you prepare for a culture-fit interview? It starts by understanding what is important to *you*. Take some time to sit and write down the answers to these questions:

- *What sort of environment do I need to do my best work?* (Is it a quiet, dimly lit environment? A cubicle where coworkers can pop in to ask a question at any time? Fully remote, so you can wear pajamas all day?)
- *What sort of company do I want to work for?* (Do you feel comfortable working for a company that contributes to accelerated climate change? Is it important for the company to contribute to the community, by offering volunteer opportunities, contributing to open-source projects, or offering public workshops?)
- *How many meetings are too many meetings?* (Some companies have a *lot* of meetings — how many is too much for you? Do you prefer a

structured environment where policy and procedure are all spelled out, or a fast and loose culture, where things are figured out on the fly?)

- *What sort of accommodations are important to me?* (Does the company need to be ADHD-friendly? LGBTQIA-friendly? Wheelchair accessible?)
- *How important is it for an employer to offer me a path to grow?* (Are you content being hired somewhere where you are never mentored or given the opportunity to expand your skillset? What about being offered a way to transition to a different type of role, such as management?)
- *What does my ideal boss look like?* (What qualities might they have? Are they hands-on or laid back?)
- *How do I prefer to get feedback?* (Do you prefer informal meetings or performance reviews?)
- *Do I prefer to work alone or as part of a team?* (If you prefer working on a team, what size team?)
- *How do I handle stress?* (Are you cool and calm under pressure, or do you totally break down? How might this affect how you interact with a team? What sorts of questions might you need to ask?)
- *How important is work-life balance to me?* (Some people are built to work *all the time*. Others have families and enjoy taking their evenings and weekends to spend time with friends or doing hobbies.)
- *What motivates me?* (Are you motivated by solving complex problems? Receiving a reward, such as praise? Do you enjoy teaching or learning?)
- *What is my ideal work schedule?* (Do you work best in the mornings? At night? Certain days during the week? Do you prefer to work all year and take a few weeks off during the summer for a long vacation?)
- *What is my long-term plan?* (As cliché as it sounds, think about what your 5-year and 10- year plans are. Where do you envision yourself?)

Hopefully, by answering these questions, you will spark further inspiration to ask yourself even more questions. Knowing yourself well enough to

know what it is you want, need, and what your goals are will help you evaluate a company to figure out if you would be a good fit there.

When the time comes to have a culture-fit conversation, you may wish to speak about a culture *add* rather than a culture *fit*. When a company looks for a fit to its existing culture, it could unintentionally lead to discrimination against candidates who do not think, act, or look like existing employees. In this context, it could be useful to talk about your ability to bring fresh, innovative ideas to the team. Culture *add* strengthens the company by adding new perspectives into the mix.

Prior to this part of the interview, it will be worth your time to explore the company's social media and any news about them. Visit their website: many companies have a section on the website about their mission. Check sites such as Glassdoor to see what employees are saying about working there. Reach out to current and past employees to ask for their perspectives. Anything you can do to understand the company and whether it shares your values, ethics, and vision will help prepare you for this part of the interview.

Negotiating Compensation

If both you and the employer walk away from the culture fit portion of the interview feeling confident, the next step is likely receiving a job offer. What sort of compensation you are offered will vary widely based on things such as geographic location, size of the company, whether the company is a startup or publicly traded, whether you are a contractor or full-time employee, and your own experience. You will either be offered an hourly or salary wage, and anything after that will vary. In some countries, with some employers, depending on the role, you may be offered a retirement plan, private health insurance, stock options, a wellness budget to join a gym, a conference allowance to travel to a relevant technology conference once a year, a technology budget for things like a laptop, an external display, a mouse/keyboard, or sometimes even a desk and chair.

You will have to figure out what part of a compensation package is most important to you. Are you being offered private health insurance but are covered by your spouse? You may be able to negotiate a higher salary instead. Perhaps you are interested in receiving more shares in the company. You could trade some of your salary for those additional shares. When it

comes to the wage, you will have to do your own research to understand the current market rates for a developer with your skillset in the position you are applying for. Sites like Glassdoor may list wages for other employees at the company in your role or an adjacent one. Try not to price yourself too high or too low. If you have been collaborating with a recruiter, you should be having these sorts of compensation conversations, and they will be able to help you navigate the process.

Negotiating your compensation might be the scariest part of interviewing with a company, but it is also extremely important. The wage you start at may follow you for years while you work at the company and could influence your sense of worth the next time you seek a position elsewhere. Take the time now to understand the market, the value you bring, and what the company is offering (as well as what they might be flexible on). Do not accept the offer on-the-spot, but give yourself time to think about it, discuss it with your recruiter, and really evaluate whether it is right for you.

Answering Some Interview Questions

Let us seek to answer some interview questions you will likely have to prepare for, starting with all the questions in this book.

1. What design paradigm does Flutter use?

Flutter uses an aggressive composability design paradigm. Widgets are built up using smaller and smaller widgets, down to the most basic widgets. Learn more about this by reviewing *Chapter 4: Introduction to Widgets*, and by reading the documentation found on Flutter docs website; <https://docs.flutter.dev/resources/inside-flutter#aggressive-composability>.

2. What method is called for each widget when Flutter wants to draw a widget to the screen?

The `build()` method is run each time Flutter needs to draw a widget to the screen. The `build()` method will always return a widget, which is generally composed of other widgets. The resulting widget will be added to the widget, element, and render trees.

3. What are three ways you can figure out what properties are available for a widget?

There are several ways to figure out what properties are available for any given widget. The quickest and easiest way is to simply hover your mouse cursor over the widget in question. By doing so, the widget's documentation will appear, outlining the available properties, as well as some other documentation. You can also refer to the documentation online by searching <https://flutter.dev> for the widget name. The third way is to go to the widget's definition by holding *Ctrl*/*⌘* and clicking on it. Once there, you will be able to explore not only the properties available, but how they are used within the widget, as well.

4. What is the difference between a positional parameter and a named parameter?

When passing parameters to a widget, if the widget has positional parameters, the order matters. If you have a widget that takes two positional parameters, one being a `String` and the other a `bool`, passing the `bool` before the `String` will cause the widget to fail to build. However, if those parameters are converted to named parameters, the order no longer matters.

5. When creating a widget, how do you specify that a parameter is either positional or named?

Named parameters are wrapped in curly braces `{}`. Positional parameters are not. In the following example, the parameter `text` is a positional parameter, whereas `isActive` is a named parameter. The value passed into either a positional or named parameter is called an **argument**.

1. class Test extends StatelessWidget {
2. final String? text;
3. final bool? isActive;
4. const Test(this.text, {this.isActive, super.key});

```
5.     
6. @override
7. Widget build(BuildContext context) {
8.     return Text('$text $isActive');
9. }
10. }
```

6. What does the question mark after a type confer?

The question mark tells Dart that the type is nullable. A `String?` can either be a `String` or `null`. However, a `String` can never be `null`.

7. When would you need to use a Key?

A `Key` is used by the framework to match a widget with other widgets when a rebuild is triggered. It can be useful to set a `Key` when you need to preserve a widget's state as it moves around the widget tree. In practice, it is rare to manually specify a `Key`; however, it can be a useful tool, especially when writing tests, as it will enable you to find a widget more easily in the widget tree programmatically. If you find yourself modifying a collection of widgets of the same type that hold some sort of state, it is time to add a `Key`. (Think: reordering a to-do list, where the state holds a checkbox.) There are several types of keys, such as `LocalKey`, `GlobalKey`, and `UniqueKey`. Learn more about `Keys` from the official Flutter documentation, which you can find here: <https://api.flutter.dev/flutter/foundation/Key-class.html>.

8. What is a private object and how is one defined?

A private object is something that cannot be used outside of the local library where it has been defined. When defining a private object, simply prepend an underscore to the name or type you are defining. The underscore conveys that the variable or method is scoped locally to the library in which it can be found. A library can be a single file or multiple files connected with the `part` and `part of` keywords.

9. What is the lifecycle of a stateful widget?

- `createState()`
- `mounted == true`
- `initState()`
- `didChangeDependencies()`
- `build()`
- `didUpdateWidget()`
- `setState()`
- `deactivate()`
- `dispose()`
- `mounted == false`

10. What is the difference between a stateless and stateful widget?

A stateless widget would not update once the widget is added to the widget tree. If the properties which were used to create it change by way of a parent widget passing down new properties, the widget will be discarded, the associated element in the element tree will be updated, and a new widget will be created before being inserted into the widget tree in place of the previously discarded widget. In contrast, a stateful widget can maintain its own state and allows for the mutation of its properties. A stateful widget will rebuild whenever the state is changed and `setState()` is called.

11. What is the purpose of an inherited widget? What is an example of one?

Inherited widgets are accessible via `BuildContext` and allow you to access their properties from any widget, which is an ancestor in the widget tree. Common inherited widgets you will find are `MediaQuery`, `Theme`, and `Navigator`.

12. What is `BuildContext`?

`BuildContext` contains information about where a widget is in the widget tree. When a widget is rendered, it is part of the widget tree and has a corresponding `RenderObject` in the element tree. `RenderObjects` are an implementation of `Element`, which implements `BuildContext`.

13. What is the difference between Material and Cupertino-styled widgets, and when would you use one over the other?

Material widgets are styled after Google's Material Design system. You can read more about Material Design's philosophies here: <https://m3.material.io/>. Cupertino (named after the city in which Apple is headquartered) looks to implement Apple's Human Interface design system. You can read more about Apple's Human Interface Guidelines here: <https://developer.apple.com/design/>.

14. What is a callback function?

A callback function is a function that is passed into another function or method (including the `build` method of a widget) as an argument to a parameter, which will be called from inside that function, but will run in the scope of the parent widget. For example, a button will have an `onPressed` callback function. When the button is pressed, the `onPressed` callback function will run.

15. What is the best way to add an icon to an OutlinedButton?

To add an icon to the `OutlinedButton`, use the `OutlinedButton.icon` constructor rather than by supplying the icon as a child of the button.

16. How do you style a button, including changing its color?

To style a button, you should either add a style to the button itself or add your style to the `Theme`. Button style parameters expect a `MaterialStateProperty` object, which will resolve the various states of a button (neutral, pressed, disabled, and so on) into a style. Refer to the documentation on the Flutter website for more information. You can find that here: <https://api.flutter.dev/flutter/material/ButtonThemeData-class.html>.

17. How do you disable a button?

Disabling a button is as simple as ensuring the `onPressed` method is `null`. This is typically done by checking some sort of state or variable, then returning a method when the appropriate conditions are met.

18. What widget can you use to convert an arbitrary widget, such as a Container, into a button?

`GestureDetector` not only allows you to turn a `Container` into a button but allows you to perform all sorts of swipe, tap, drag, pinch, and other gestures.

19. What widget can be used to detect a swipe gesture?

If you guessed `GestureDetector`, you would be right! There are other widgets that implement this functionality, too, such as `Dismissible`.

20. How do you change the type of keyboard presented to a user on a mobile device when they tap a text field? For example, how would you present the user with a keyboard which is more appropriate for numerical input?

Text inputs generally have a `keyboardType` property, which takes in a `TextInputType` object. To display a keyboard tailored for numerical input, use `keyboardType: TextInputType.number`.

21. What is a `TextEditingController` used for?

This allows you to access text which has been entered into a text input from outside the widget. It will also allow you to save, recall, and reset the contents of the field.

22. In what order are input formatters applied?

Input formatters are applied top to bottom, in the order defined in the `List<TextInputFormatter>` of your text input's `inputFormatters` property.

23. How can you prevent users from typing numbers in a text field, and instead only allowing letters?

Use a `FilteringTextInputFormatter` to only allow the characters you want, using a `RegExp`.

24. You have a `DateTime` object with a given date. How do you figure out what the date will be precisely one year in the future from that date?

`DateTime` objects can be added to or subtracted from using the `add` and `subtract` methods, which take a `Duration` object. The current date/time can be generated using the `now` constructor. `DateTime.now().add(const Duration(days: 365))` would result in a `DateTime` object with a time precisely one year in the future.

25. What is a key difference between the material date picker and the Cupertino date picker?

The Material date picker is much simpler to implement, requiring minimal setup work, whereas the Cupertino date picker needs to be placed within a modal popup to position it appropriately on the screen. The Material date picker can quickly switch between time and date options, while the Cupertino date picker needs to be told what mode to operate in. Finally, the Cupertino date picker can be used to select more than just a date/time. It is also capable of displaying arbitrary lists of data to pick from, such as choosing a day of the week, or a duration for a timer.

26. What does the `map` method of a List do?

This will allow you to perform an action on every object within the list, returning an iterable object, which can then be mapped back into a `List` if so desired. See *Chapter 5: Handling User Input* for an example.

27. What are some considerations when you want to use a group of Radio buttons?

Consider whether the look and feel are appropriate for the platform your application is running on. A Cupertino-styled radio button would feel out of place on an Android device, for example. Also, consider if there is a more appropriate widget: if your radio buttons toggle between two options, could you use a switch instead?

28. How many states does a Checkbox have?

The available states are `true` and `false`. However, if `tristate: true` is set, a third option of `null` becomes available. A checkbox will become disabled if the `onChanged` argument is `null`.

29. How can you limit the minimum and maximum values of a Slider?

By specifying the `min` and `max` properties.

30. What is a Form and what is the benefit of using one? What are some cases in which a Form might not be necessary?

A `Form` groups together several input widgets, allowing you to specify a `Key` to save, recall, reset, and validate all the inputs at once. If you only have one input, a `Form` is not necessary.

31. What is the difference between a TextField and a TextFormField?

A `TextFormField` is a `TextField` that is wrapped in a `FormField`, which allows you to use the `onSaved` and `validator` methods.

32. How do you validate an input?

Use a `validator` function on any widget with a `FormField`.

33. What is a regular expression?

Regular expressions are patterns used to match character combinations in strings of text.

34. What is pubspec.yaml used for?

This file contains basic information about your Flutter or Dart project, including the package name, description, and whether to publish it to pub.dev. It also defines what versions of Dart and Flutter to use, what packages are used by the project, and what external assets to include.

35. What language is pubspec.yaml written in?

The `pubspec.yaml` file is written in the YAML markup language.

36. What are the required directives for Flutter in pubspec.yaml, and what do they do?

- `name`: Used to name the package.

- **version**: Used to define what the application's version number is.
- **environment**: Used to define which version of Flutter/Dart the project relies upon

Other directives include `description`, `publish_to`, `dependencies`, and `dev_dependencies`. Both `dependencies` and `dev_dependencies` are required for Flutter projects. They define what packages are used within the project and which versions of those packages to use.

37. How do you update the package versions in `pubspec.yaml` with a single command?

Use `dart pub upgrade --major-versions` to upgrade your packages to the latest resolvable versions.

38. How do you add images, fonts, sounds, or other assets to your application?

Include them in the `assets` directive within the `flutter` directive.

39. What is Pub?

Pub is the package management ecosystem for Flutter and Dart. The website, pub.dev, is used to browse the available packages, whereas the command line tool(s) provided by Dart and Flutter are used to manage adding, removing, and upgrading the packages used by your project.

40. How does dependency version pinning work?

Dependencies can be pinned to a specific version by omitting the caret (^) in front of the version number. If the caret is present, the package can be upgraded to newer versions that fall within the dependency constraints. For example, a version of `^2.1.0` could be upgraded automatically to any version, provided it was still in the 2.1.x version range, such as 2.1.9. Learn more about package versioning here: <https://dart.dev/tools/pub/versioning>.

41. Where can one browse a list of packages that are available to add to a Flutter or Dart project?

Head on over to <https://pub.dev>!

42. What are Pub Points, and how are they calculated?

Pub Points are a measurement of the quality of a package based on numerous criteria. These criteria are ever-evolving, but as of writing they are as follows:

- Follow Dart file conventions (30 points total)
 - Provide a valid `pubspec.yaml` (10 points)
 - Provide a valid `README.md` (5 points)
 - Provide a valid `CHANGELOG.md` (5 points)
 - Use an OSI-approved license (10 points)
- Provide documentation (20 points)
 - Provide an example usage of the package (10 points)
 - 20% or more of the public API has `dartdoc` comments (10 points)
- Provide support for all 6 platforms (20 points)
- Pass static analysis (code has no errors, warnings, lints, or formatting issues) (30 points)
- Support up-to-date dependencies (20 points)
- Dart 3 and Flutter 3.10 compatibility (20 points)

The total number of points a package can have, as of writing, is 130. Find out more about how Pub Points are calculated here: <https://pub.dev/help/scoring#pub-points>.

43. How can one save a list of their favorite packages on Pub?

First, create an account and log in. Then, find a package you would like to save. Navigate to the package details and look for the thumbs up icon near the top. To find the list of packages you have saved, hover over **My pub.dev** in the header, then choose **Likes** from the list.

44. How can you play audio in your Flutter application?

Use a package such as `just_audio`.

45. Where can you find a list of popular Flutter and Dart packages?

The home screen of `pub.dev` includes a list of the most popular packages and the top packages on Pub.

46. What are some ways in which data can be persisted across app launches?

Use a package such as `shared_preferences`, `encrypted_shared_preferences`, or `isar` to store data on the device's local storage. Then, read that data the next time the app launches.

47. What YouTube playlist does the Flutter team use to showcase popular packages?

The **Flutter Package of the Week** playlist, which is available on YouTube here: https://www.youtube.com/playlist?list=PLjxrf2q8roU1quF6ny8oFHJ2gBdrYN_AK

48. What is a Future?

Any event which does not happen instantly will return as a `Future`. A future is a promise that, once the work has been completed, a value will be returned.

49. What is the difference between synchronous code and asynchronous code?

Synchronous code is *blocking*, whereas asynchronous code is *non-blocking*. Blocking code prevents the execution of other steps from taking place, whereas non-blocking code allows processing to continue while the non-blocking code is run in the background.

50. What happens to the application when there is a long-lived, synchronous task being executed?

The UI will freeze and become unresponsive. Avoid this by introducing non-blocking code where the blocking code currently exists.

51. What are the `async` and `await` keywords used for?

`async` is used to mark a method as a long-running, non-blocking process. The `await` keyword is used to wait for an `async` function to complete and return its value.

52. What type of value does an asynchronous operation return?

Asynchronous operations have a return type of `Future<T>`.

53. What is an API?

API stands for Application Programming Interface. It is a way in which computer programs can interact with each other without needing to know anything about how the other works. Many APIs are Web-based, using Representational State Transfer (REST) as the interface, which usually returns a JSON response, although other methods are available, such as GraphQL.

54. What package(s) can be used to communicate with an API?

There are many packages that can be used. The standard `http` package is a fine choice. If you need additional options which the `http` package does not offer, Dio is an excellent alternative.

55. What sort of data is generally returned from an API?

When working with a REST API, the most common type of data returned is JSON. However, there are other, less common types of data that could be returned from an API, such as XML. XML responses have fallen out of favor over the years, however.

56. What is an HTTP status code, and what are some common ones?

HTTP status codes are metadata returned from the remote endpoint to convey information about how a request was (or was not) completed. Status codes in the 100 range are informational messages, codes in the 200 range are success messages, codes in the 300 range are redirection messages, codes in the 400 range are client error messages, and codes in the 500 range are server error messages.

Read about HTTP status messages here:
<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>.

57. What is a guard clause?

A guard clause is used in conjunction with an `if` statement to catch errors early and halt processing as soon as one is found, rather than wasting time on further processing of data that is known to be invalid.

58. What is JSON, and what are some considerations that are important to consider when working with JSON data?

JSON, or JavaScript Object Notation, is a key-value-based representation of data. In Dart, it is either a `Map<String, dynamic>` or a `List<dynamic>`, depending on the JSON's structure. When using `print` or `log` to dump an API response to the console, any value which is a `String` will be stripped of its enclosing quotes. This can make it difficult to use these tools to analyze API responses. For this reason, prefer using DevTools to inspect responses.

59. When communicating with an API using an HTTP client, a “verb” is used to determine which kind of action the client should take. The verbs POST and GET are used for which types of operations? Extra credit: What about PUT, PATCH, and DELETE?

- **POST:** Used to send data to an API. Sending multiple `POST` requests would result in the server processing each request individually. (Think: making multiple orders at a restaurant.)
- **GET:** Used to retrieve data from the server.
- **PUT:** Nearly the same as the `POST` method, except that it is idempotent. Sending multiple `PUT` requests will have the same effect.
- **PATCH:** Used to modify an existing resource on the server. The resource must have been previously created with a `POST` or `PUT` before it can be modified with a `PATCH`.
- **DELETE:** Asks the server to remove a resource.

60. When sending a POST request to an API, which properties are required to be passed into the request?

Headers containing the content type being sent to the API must be specified. Additionally, a JSON-encoded body is often required by many APIs.

61. What does CRUD mean?

CRUD stands for Create, Read, Update, and Delete. These are the four methods available for objects in an API.

62. What HTTP verbs relate to which CRUD operations?

- C — POST or PUT
- R — GET
- U — PATCH
- D — DELETE

63. How can you generate List<int> containing all the odd numbers from 27 to 99?

Use `List<int>.generate` to generate the list. Then, check each value to verify it is odd:

```
List<int>.generate(37, (i) => 2 * i + 27).where((i) => i.isOdd).toList();
```

64. How can you tell if a FutureBuilder has completed its future?

The `builder` has an `AsyncSnapshot` value which will have a `connectionState` property equal to `ConnectionState.done`.

65. Why would somebody choose to use a ListView over a Column? Why would somebody choose to use a Column over a ListView?

If your list of widgets needs to be scrollable, prefer a `ListView` for performance reasons. If you are stacking a small number of related widgets which do not need to be independently scrollable, opt for a `Column`. Bonus: if `ListView` cannot return the different widgets you need to display, investigate Flutter's many `Sliver` widgets.

66. How do you display a dialog to a user?

Flutter offers a `showDialog` method where you can use the `builder` to display a `Dialog` object.

67. Given a dialog with two options, yes and no, how do you return either true or false from the dialog?

When popping the `Dialog`, pass the value back like this: `Navigator.pop(context, value);` where `value` is equal to a Boolean value that references your chosen option.

68. What is a sink? What is a stream?

A Stream is an object that provides a sequence of events that can be listened to, whereas a Sink is an object that allows you to push data into a stream.

69. What are the `async*` and `yield` operators used for?

These are the sink/stream versions of `async/await`. The `async*` keyword means the function will yield multiple values as a stream of events. This is called an `async` generator.

70. What is the `async*` equivalent to a `FutureBuilder`?

`StreamBuilder` is the `async*` equivalent to `async`'s `FutureBuilder`.

71. What library provides widgets which are used with sinks and streams?

The `dart:async` library provides widgets that work with sinks and streams.

72. How do you add an event to a sink?

First, assign a `StreamController` to a variable. Then, use the `.sink.add` method on the variable.

73. The concept of sending a user from one screen to another is known as what?

The concept of sending a user from one screen to another is called routing.

74. What is the difference between navigation and routing?

Routing is the concept of sending a user from one screen to another, whereas navigation is the action taken to accomplish this in the code.

75. When navigating between screens in Flutter, what analogy is useful in visualizing how different screens and their states are maintained?

You can think of navigating as a stack of cards. Each card can have its own state, and new cards can be added on top of the stack, or a card on the top of the stack can be replaced by a different card.

76. What is the difference between imperative and declarative routing?

Imperative routing has step-by-step instructions for how to route from one location to another. Declarative routing is a method of declaring intent on where you would like to be routed to.

77. What widget is used to route imperatively in Flutter? What widget is used to route declaratively?

Imperative routing is accomplished using the `Navigator` widget, whereas `Router` is used for declarative routing.

78. What are some of the limitations of Navigator 1.0?

`Navigator` is a much simpler method of routing than `Router`, making many of the more complex tasks that `Router` provides much more difficult to accomplish. While *technically* capable of doing anything `Router` can, as `Router` is built using `Navigator`, many tasks, such as nested routing and deep linking, are better suited for a more complex routing system.

79. How do you configure named routes, then navigate to a route by name?

Your named routes can be set in the `routes` property of your `MaterialApp`, and then you can switch from `Navigator.push` to `Navigator.pushNamed`.

80. What is the difference between `Navigator.push` and `Navigator.pushReplacement`?

`Navigator.push` will add the new route to the top of the routing stack. `Navigator.pushReplacement` will replace the current route with the new one.

81. What happens if you use `Navigator.pushReplacement` followed by `Navigator.pop` on the new screen?

It depends on what the previous route is prior to pushing the replacement route. If the initial route is replaced by a new route, then popped off the stack, this might cause the application to exit. However, if there was a previous route on the stack before pushing a replacement route, that previous route will become the eventual route you end up on.

82. How do you pass data to a new route? How do you get a value back when calling `Navigator.pop`?

You could either pass arguments into the screen itself or you could extract the arguments using the `onGenerateRoute` method. Getting data back from a route can be done by awaiting the navigated route and passing a value back from the `pop` method.

83. What useful things can you use `onGenerateRoute` for?

Since this method is run whenever you perform a page route, you can use it for passing arguments into a route or running any arbitrary code whenever a route happens. This makes it a prime place to introduce analytics into the application to understand how users are routing around the application.

84. What widget can be used to animate a widget from one screen to another?

The **Hero** widget can be used to animate a widget from one screen to another.

85. What is the correct name for Navigator 2.0?

When the Router API was first announced, it was referred to as Navigator 2.0.

86. How are the ValueChanged widget and notifyListeners() method used?

The **ValueChanged** widget will trigger a rebuild of the widget when a value is changed, then the `notifyListeners()` method is called. This is useful when you want to trigger a widget rebuild from another widget within your application.

87. What is a RouterDelegate?

A **RouterDelegate** is used with the Router API to define the behavior of how the **Router** learns about and responds to changes in the routing state.

88. How do you change the transition animation when navigating to a new route using the Router API?

You can override the `createRoute` method of a **Page** to return a **PageRouteBuilder** with the new animation being defined within the `pageBuilder` property.

89. What is ephemeral state, and how is it different from app state?

Ephemeral state is the state of values within a stateful widget that are not managed anywhere outside of the widget. App state is managed by some sort of state provider which widgets can consume and respond to.

90. Why is a stateful widget not a good option for managing app state?

When the state of a stateful widget changes, it will trigger a rebuild of itself, which could cause issues lower in the widget tree. There is

also no easy way to pass state between widgets, mutate the state from lower in the widget tree, or keep the code clean.

91. If you have nested ListView widgets, how can you prevent the inner ListView from scrolling independently of its parent?

To prevent a `ListView` from scrolling, pass both the `shrinkWrap: true` and `physics: const ClampingScrollPhysics()` parameters to it. Note that this will effectively turn your `ListView` into a `Column`, thereby eliminating any performance improvements gained by using a `ListView`.

92. What is the `=>` operator used for? What are the benefits/drawbacks from using it?

This is shorthand for *return this value*. It can make some code much more concise, such as when you are not processing a variable to decide what to return. However, if your method could return multiple values based on differing criteria, this operator could make it difficult to both read and write the code.

93. What is escaping? What characters might need to be escaped?

Escaping is a process by which special characters that would normally function as control characters in a `String` can be represented as part of that `String`. For example, a `String` with a dollar sign in it would normally cause Dart to evaluate the next word as a variable name, replacing both the dollar sign and word with the value of the variable. If you would rather have the dollar sign as part of your `String`, you would prepend the dollar sign with a backslash (\).

94. What is interpolation?

Interpolation is used to evaluate complex variables or methods directly inside a `String`. It is often used to evaluate a single property of an object, such as with "Hello, \${user.name.first}!" or by directly invoking a method call, like this: "Hello, \${getUsersFirstName(user)}!". It can also be used to escape a quote that would otherwise break the `String` definition, such as this: 'That\'s great news! '.

95. What is ValueChanged used for? How is it defined?

`ValueChanged` is a `typedef` defined as `typedef ValueChanged<in T> = void Function(T value)`. It is used to trigger rebuilds when a value changes, often in an `InheritedWidget`.

96. What is a `typedef`?

A `typedef` is a way of specifying a new type without defining an object class. The type could be a method call, a different object, or something similar. An example of a `typedef` is `typedef ListOfInts = List<int>;`. This would allow one to use `ListOfInts` in any place that `List<int>` could be used.

97. What is `UnmodifiableListView` used for?

This is a representation of `List` in which the contents cannot be modified. It guarantees that a List would not be changed, allowing for performance improvements, and preventing accidental discrepancies in data.

98. What are some differences and similarities between Provider, Cubit, and BLoC?

Each of these three systems enables state management through your application. A Cubit is like a *baby* BLoC, providing similar but reduced functionality. Provider uses `ChangeNotifierS` to notify listeners about updates to the state, compared to the events being emitted from BLoC and values being streamed from a Cubit.

99. What is cascade notation?

Cascade notation is a shorthand method of applying many properties to an object at once. You will recognize cascade notation when you see an object with multiple `..method` calls strung together. Note the double period, known as a cascade operator.

100. What are some other popular app state management solutions?

Flutter Hooks, Riverpod, Redux, and MobX, are some other popular state management solutions in Flutter.

101. What is a *responsive layout*?

A responsive layout is a layout that adapts dynamically to the platform and dimensions on which it is running. This could mean using a tab bar on the phone, switching automatically to a side bar on a tablet, then using a menu bar on a desktop.

102. Which widget can we use to build a responsive layout?

The `LayoutBuilder` widget can be used to build a responsive layout.

103. When building a responsive layout, what considerations should you take when choosing how to build your screen size-dependent widgets? What are some actions you can take to ensure you do not forget about accidentally removing important functionality?

Try defining important properties using an `enum`, then using a `switch` case to ensure you are do not forget any of the values.

104. What is `dart:io` used for?

This package is used to deal with files, directories, processes, sockets, WebSockets, and HTTP clients/servers.

105. What platform(s) can `dart:io` not be used on?

Due to the nature of the package and how it deals with system-level functions and objects, it is not available when targeting the Web as a platform.

106. How do you update your application's icon?

To manually update the app icon, you will be modifying resources in each native platform's configuration. The Flutter documentation has information on how to do this, here: <https://docs.flutter.dev/development/ui/assets-and-images#updating-the-app-icon>.

107. Where can you find resources for designing a splash screen?

The Flutter documentation has helpful resources for designing a splash screen. You can read more here: <https://docs.flutter.dev/development/ui/advanced/splash-screen>.

108. What packages can be used to help ensure you have properly updated your app icon and splash screen?

The `flutter_native_splash` package will help you update both your app icon and the splash screen.

109. What are some packages that can be used to match the look and feel of a desktop operating system, such as Windows or Linux?

The `yaru` package will help you build applications that feel at home on Ubuntu Linux, whereas the `adwaita` package will feel at home on any Linux distribution using Gnome as the desktop environment. Microsoft's modern design language, Fluent, can be replicated using the `fluent_ui` package. Apple's desktop-styled widgets can be found in the `macos_ui` package.

110. What are the key differences between Android and iOS in terms of navigation principles?

Android has a system-level back button, which is always available for the user. Sometimes it manifests as a gesture, while other times, it may be a physical or virtual button. iOS has no concept of a universal back button, leaving navigation to be performed within the applications themselves. An iOS app will commonly have some combination of **back** button (<) or **Done** button at the top of the screen. These navigation options are often omitted on Android, as they are unnecessary given the system back button.

111. What is *stacked navigation*?

When using a tab bar for navigation, each tab can have its own navigation stack. The tab bar will navigate to each individual navigation stack that corresponds to a given tab.

112. What inherited widget contains information about the currently targeted platform?

The `Theme` widget is an inherited widget that contains information about the currently targeted platform via `Theme.of(context).platform`.

113. What package allows you to easily build platform-adaptive layouts?

The `flutter_platform_widgets` package will allow you to build applications that use different widgets depending on which platform the application is running on.

114. What is the difference between debugging and troubleshooting?

Troubleshooting is identifying and solving a problem by fixing whatever issues prevented things from working in the first place. Debugging is the process of going step-by-step through a problem to identify and resolve any errors.

115. What are the five steps involved in debugging?

- I. Ensure the error is reproduceable. If the error is intermittent, you have not properly identified the error.
- II. Be certain that the problem you are debugging is the problem you need to solve.
- III. Check all the obvious error sources. It is easy to waste time by debugging a “complex” problem that turns out to have a very simple and obvious solution.
- IV. Debug the problem: isolate the problem by dividing it into working versus non-working components. Think like a computer: step through the instructions one at a time to find where the operation fails.
- V. If you are unable to identify the root cause of the problem, reassess the situation: what assumptions have you made that might be incorrect? Walk through the process again, taking notes and using knowledge learned from the last time.

116. What methods can you use to log information to the debug console?

You can log information to the debug console using the `print`, `debugPrint`, and `log` (from the `dart:developer` package) methods.

117. What is a breakpoint, and how do you create one?

A breakpoint will pause your code execution at a given line, allowing you to inspect the runtime variables. They are created directly in your IDE (not in code), generally by clicking next to the line number. Most IDEs will represent a breakpoint with a red dot.

118. What is a stack trace? What is a stack frame?

A stack trace is comprised of a call stack (a list of stack frames) and shows precisely what was happening in the application when a breakpoint was hit. A stack frame contains information about the current running method when the previous line of code was invoked.

119. What is an exception?

An exception occurs when something goes wrong in the application and processing is unable to continue. In Dart, there is an `Exception` class that can be `thrown` when an exception occurs. This class contains information about the problem.

120. What risks do you run by using a try/catch block?

It is possible to end up in a state where an exception occurs, but processing continues anyway unless the exception is properly handled within the `catch` portion of the `try/catch` block. A `try/catch` can also obfuscate problems if not written correctly, making debugging more difficult.

121. How do you access Flutter's DevTools?

If you are running your code from within Visual Studio Code, there will be a blue magnifying glass with a Flutter logo icon in the bar that pops up when you run the application. Running your code directly from the command line using the `flutter run` command will provide a URL you can access in your browser.

122. What are the different tools available in Flutter's DevTools?

The different pages available in DevTools are the Widget Inspector, CPU Profiler, Memory, Performance, Network, and Logging pages. Each of these pages has many tools available.

123. What are the two different categories of performance issues your application may encounter?

Time and Space. Space-related performance issues are issues where your application may be using too much memory or other system resources. Time-based issues are related to how long it takes for any given frame to be processed and displayed on the screen.

124. What is vsync, and how does it relate to your application's performance?

Vsync, or vertical sync, is a measure by which the contents of your device's display are updated at a given interval. A common refresh rate is 60fps, meaning that the screen is being redrawn every 1/60th of a second. If a frame is not processed and rendered within the `vsync` period, `jank` will occur, causing hitching in the interface.

125. What is jank?

Jank is when a frame takes more than the vsync period to process.

126. What three compilation modes are available for Flutter applications? How do they differ?

- **Debug mode:** The default mode for developing applications in. This mode allows full access to many of the debugging tools that Flutter provides, including hot reload.
- **Profile mode:** Used to assist in evaluating the performance of an application, unlocking the ability to use the Performance page in DevTools.
- **Release mode:** Strips out the debugging and performance tooling used by Flutter, maximizing the performance of the application in preparation for release.

127. What is the difference between storage and memory?

Storage is non-volatile and can be used to store files long-term. It is slower than memory but generally much more abundant. Memory is extremely fast, temporary (volatile) storage, which is used to hold data that the CPU is currently processing. Upon power loss, all data stored in memory is lost.

128. When is a ListView not a ListView?

When you pass the `shrinkWrap: true` property to a `ListView`, it essentially becomes akin to a `Column`, losing all performance benefits that a `ListView` would otherwise give you.

129. When should you consider optimizing your application?

Flutter is built in such a way that premature optimization of widgets is not necessary. Only when you notice jank or excessive battery drain should you really need to start digging into performance optimizations for your application.

130. What are some steps you can take to optimize your application?

Make sure the `build()` method is not doing more work than it needs to. This method can run on every single frame if you are scrolling or animating and should not run expensive operations. You can also use caching to store values that would otherwise be re-fetched from external sources. One of the biggest things you can do to optimize performance is to simply break down large widgets into smaller ones and ensure you are passing only the necessary data to build a widget, rather than a larger object.

131. What does *open-source* mean? What are some different open-source licenses?

Open source means the code is available to inspect and modify freely. Not all open-source code is licensed the same, however. Some common licenses are BSD, Creative Commons, GPL, and MIT. Find out more about open-source licenses here: <https://opensource.org/licenses/>.

132. What are some factors to consider when selecting a data provider, such as a third-party API?

Ask yourself the following questions: Does this API provide the data I require for my application? What is the cost of using this API? Is the provider reputable and expected to be around for a long time?

133. Why is it important to keep your code organized? What are some ways you can organize it?

Keeping your code well-organized will make it easier to develop and troubleshoot features, prevent other developers from cursing your name, and add structure to your application. While there are no *rules* on how to organize your code, a suggested file layout can be found in *Chapter 12: Creating Your First Application*.

134. What is an enum?

An `enum` is a Dart type that represents a fixed list of constants.

135. What is a decorator?

A decorator, such as `@override`, is used to add additional behavior to an object without affecting other objects of the same class.

136. What is code generation?

Code generation allows us to write less code to define a class. The “full” class will be generated for us automatically when we run the appropriate command.

137. What package can be used to generate code?

The `Freezed` package is one of the most popular packages that can be used to generate code, but it is not the only one.

138. What case style does Dart prefer for class names? For variable names?

Dart prefers using Pascal case for class names and using camel case for other types of names. Pascal case `LooksLikeThis`, whereas camel

case looksLikeThis.

139. How can you keep secrets, such as an API key, safe?

Secrets can be stored as environment variables, which are not stored in source control. At runtime, the variable can be read in and compiled automatically.

140. What are some good resources for finding answers to programming questions?

This book is a fantastic resource for many Flutter questions! There are many online resources dedicated to answering questions, too. Always start by reading the documentation before asking your questions to others, as it may save you (and others) a lot of time. If you are part of a developer community, you have already got a great resource at your fingertips. Otherwise, sites like Stack Overflow or even ChatGPT can answer your questions. Beware when using AI-assisted coding tools, however: they are known to generate non-functioning code that *looks* correct. Often, you will spend as much time debugging code that has been generated for you as you will by just reading the documentation and writing it yourself to begin with.

141. What is a singleton?

A singleton is an object which is designed to have only a single, global instance in the application. You might create a singleton to be the single point of access for your code to read and write files on the filesystem, for example. This would help prevent writing to a file that is currently being read elsewhere in the application by not allowing the two actions to be performed simultaneously.

142. Is async/await blocking?

No, `async/await` operations are not blocking. When an `async` function is invoked, Dart will mark it as inactive and continue processing other code until the function returns a value, at which point it will be marked as active again.

143. What is dependency injection?

Dependency injection allows you to decouple objects, rather than hard-coding them. For example, you may have a widget that takes in a parameter for an API that it uses to make an API call. When creating the widget, you could pass the API object directly in, thereby coupling the API directly to the widget. However, this would make it difficult, if not impossible, to write tests where the API response is mocked. By registering the API class in a dependency injector, you can pass the instance of the API, which has been registered into the widget. Then, when writing a test, you can inject a *different* API, which you can use to mock responses.

144. What are some software design patterns?

- **Model View Controller (MVC)**: Separates data (Model), user interface (View), and application logic (Controller).
- **Model-View-ViewModel (MVVM)**: Separates user interface (View) from business logic (Model) and presentation logic (ViewModel).
- **CLEAN**: A set of principles designed for building scalable, maintainable, and testable software. Read more here: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>.

145. Why does release mode run faster?

In release mode, assertions, service extensions, and debugging are disabled, debugging information is stripped from the code, and tree shaking is performed. Tree shaking removes dead and unused code, resulting in a smaller, leaner package.

146. Is Flutter imperative or declarative?

Flutter is declarative, rather than imperative. Read more in the documentation, here: <https://docs.flutter.dev/get-started/flutter-for/declarative>.

Conclusion

Interviewing for technical roles can be daunting, but with the tools and resources available to you in this chapter, you will be more than prepared to tackle it. Getting your first job offer as a Flutter developer is going to be a *huge* accomplishment for you, and after working so hard to get to that point, you deserve to feel proud! Once you have accepted your job, you have got a whole new journey ahead of you. In the final chapter, we are going to talk about what it takes to be successful in your new role, as well as what you will need to do to grow in your career.

Congratulations on your new position, and welcome to the ranks of *paid* Flutter developers!

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

Your Road Ahead

Introduction

Congratulations on your new role as a Flutter developer! It has been a long road to get here, no doubt. The journey is not over — rather, it is just beginning. In this final chapter, we are going to talk about what you can do to prepare for your new job, as well as look at how to collaborate with a software development team, what skills you will need to learn on your own that were not covered in this book, and we will look at some suggestions for furthering your career as a Flutter developer.

Structure

In this chapter, we will discuss the following topics:

- How to prepare for your new job
- Tips for collaboration with other developers
- Additional skills you will need to seek out
- Suggestions for furthering your career

Objective

Once you have completed this chapter, you will have a solid understanding of what to do before you start your job, what the first days/weeks of your

new role will be like, and what skills you have left to learn on your own. Then, we will look at some advice for furthering your career.

How to Prepare for Your New Job

The single greatest piece of advice for those starting a new role is this: *do not work before you start working*. You have already proven yourself enough to secure the position, so there is nothing left to prove. If your employer wanted you to start working sooner, they would have asked you to start sooner. So, despite your excitement, anticipation, and eagerness to start working: *don't*.

It is very easy to get burned out when doing software development. Whether it is working long hours by choice or because you are pushing to get the latest release out, at some point, we have all worked a bit later than we wanted or expected to. This might be fine occasionally, but repeatedly working extended hours is what leads to burnout. Your employer never wants you to reach a stage of burnout because your productivity will drop to near zero. It is also a miserable thing to experience, so you should try to avoid it at all costs. By not working more than you need to (for example, by not *working* before you start work), you will help prolong the amount of time you can work before starting to feel the first hints of burnout.

So, before you start working, *stop* working. Put down those personal Flutter projects you have been working on. Take some time to do hobbies you enjoy; read a book, go on a vacation, do a puzzle, or run a marathon. Whatever you choose to do, make sure it is not related to work in any way.

Once you have started your new job, after all the paperwork is finished and your laptop is set up with the tools you will need to do your job, you are going to want to familiarize yourself with the existing codebase. Figure out what kind of architecture you will be working with, and what packages are in use. Seek out any existing documentation that may exist to help you further understand the code. This may be in the form of a **README** file in the repository, or on some internal documentation website or application. You do not have to understand everything, but having a basic familiarity with the concepts will help you spin up quickly.

Learn about the team's development process. What guidelines are in place for writing, reviewing, and committing code to the codebase? How are tests

written? How is the code deployed? What different environments (for example, development, testing, production) exist? Learn about the version control system that the team is using. Is it Git or something else? How is work allocated to the team? Is there a tool for tracking tickets? What do the tickets look like?

Get to know the team you will be working with! You will have already met some of them during the interview process. Likely, this includes one of the senior developers on the team, who will be your primary point of contact for any questions you have during your early days on the job. Set up time with them to get to know them better and discuss their expectations from you. If there are other members of the team, set up a time during your first week to talk to them one-on-one, as well. Learn about the roles and responsibilities of those you will be working with and any special considerations to keep in mind. (Maybe they prefer they/them/their pronouns or would rather talk on the phone than over the company chat platform). Software development is a collaborative process, and getting to know your team is key to success. It will help you figure out who to turn to when something goes wrong (either with your code or with the product) and getting to know your team will help build trust with them, which is vital when working under pressure.

Consider what tools you will need to do your job. We are not talking about software but hardware. A carpenter cannot be expected to frame a building with a cheap hammer, so why should you limit yourself by using a cheap keyboard and mouse? These are tools you will be using daily, and they should be comfortable, ergonomic, and pleasant to use. Consider investing in a nice mechanical keyboard and high-end mouse if you have not already. Since you will be sitting most of the day, consider investing in a high-end office chair. Even better, consider a sit-stand desk and anti-fatigue mat. At the end of the day, your physical and mental health is more important than your code output. Be sure to drink lots of water, exercise, take regular breaks, and find ways to keep up your mental health.

Be sure to take notes as you begin this new chapter of your life. Someday, you will find yourself in a position where you will either need to refer to them to do your job, or you will be far enough along in your career that you will want to share the knowledge and wisdom with others. Starting on your first day, keep a notebook of important things you have learned and refer to

it often. There is no such thing as a dumb question, but there is such a thing as a question that shows you have not put in the appropriate level of effort to answer it on your own before seeking the help of others.

Tips for Collaboration with Other Developers

Many software development teams work with either a Scrum/Agile framework or a Waterfall methodology. These are two different styles of organizing a team and defining a process around the work that needs to be done. Whether your team operates under one (or a hybrid of both) of these methods or your team lacks structure (and might benefit from these methodologies), it is worth your time to understand them. Let us look at each of them to see where your team might fit in.

We will start by looking at the Agile framework using the Scrum methodology (*figure 15.1*). The idea behind this methodology is that it helps to organize the large amount of work that needs to get done by breaking it down into small pieces. There is a continuous feedback loop happening, which enables experimentation and iteration. Scrum teams are generally small — usually fewer than 10 people — and are led by a Scrum Master.

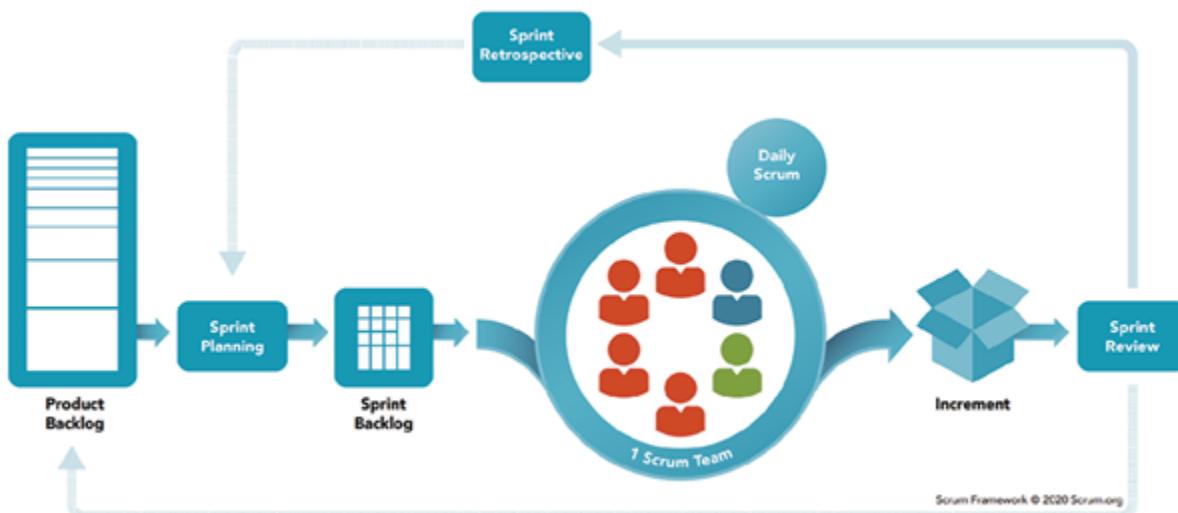


Figure 15.1: The Scrum process.
Source: <https://www.scrum.org/resources/what-is-scrum>

A Scrum team works in *sprints*, usually one to two weeks, where all the tasks to be accomplished are defined prior to the beginning of the sprint. A

Scrum board will keep track of who is assigned to what ticket (often called a *Story*) and what the status of that Story is. When it comes to breaking down work into smaller pieces, a new feature will be first broken down into one or more Epics. An Epic is composed of Stories. Each Story should be a small, well-defined unit of work. Once all the Stories in an Epic are completed, the feature can be shipped.

A Scrum team will meet regularly throughout the sprint, sometimes daily, other times two to three times per week, in a meeting called a *stand-up* and discuss the status of the Stories they are working on, including any challenges they are facing, or *blockers* preventing them from moving forward. Blockers are generally addressed by a discussion during or after the stand-up with one or more team members who can help to remove the blocker and get work moving ahead again.

At the end of each Sprint, a Sprint Review (or Sprint Retrospective) is generally held. This meeting is longer than the stand-ups, with a typical goal being to identify what worked well throughout the Sprint (and thus, what the team would like to continue doing in the future) and what did not work out (and the team would like to find a way to avoid, moving forward). Everybody on the team should feel welcome to provide feedback at all the meetings they attend.

There are other meetings to attend, as well. There are Sprint planning meetings, where the team decides which Stories from the backlog to work on during the next Sprint, and Story Grooming meetings, where feature requests and bugs that have been reported are turned into well-defined Stories and placed in the backlog.

This only scratches the surface of what it is like to work on a Scrum team. If you would like to learn more about the Scrum methodology, check out the guide available at <https://scrumguides.org/> or a brief introduction at the official Scrum website, which you can find here: <https://www.scrum.org/resources/what-is-scrum>.

Next, we will look at the Waterfall methodology (*figure 15.2*). This methodology differs from Scrum in several key aspects. First, each process of the Waterfall is completed before moving on to the next. Rather than a continuous flow of Stories from the backlog being added to the

development cycle, all Stories are created and defined up-front. Then, they are worked on until all of them are completed:

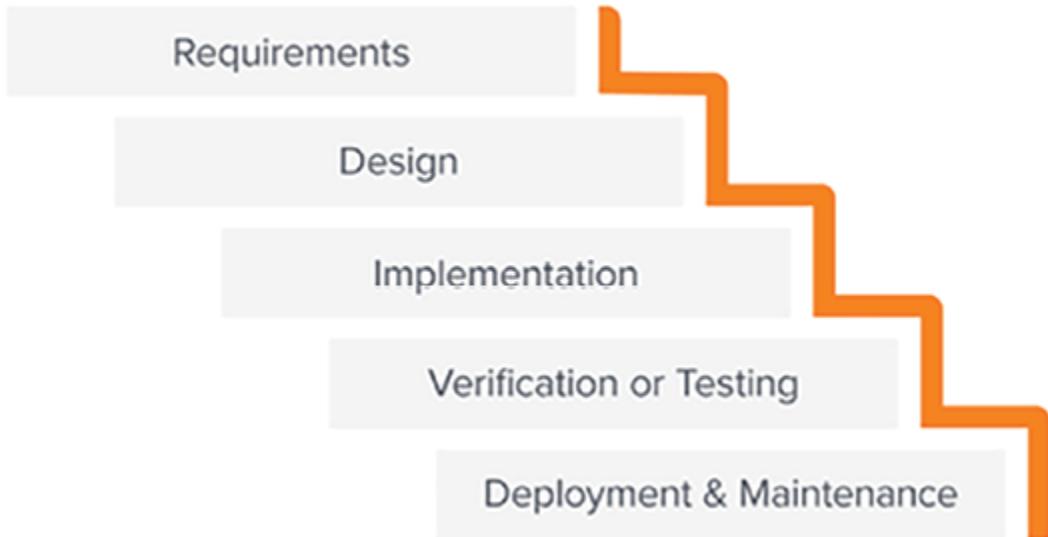


Figure 15.2: The waterfall methodology.

Source: <https://business.adobe.com/blog/basics/waterfall>

The waterfall process starts by gathering all the requirements for the project and fully understanding them before any work continues. The project manager will do much of this work before bringing it to the rest of the team as a written document. Once the requirements have been laid out, the team will discuss the design of the solution: what are the data classes which will need to be created? What does the data flow look like? How will the software be designed? Once a complete, logical representation of the entire application is built, the team will then move on to implementing the solution they have laid out. If the design they came up with is not working or requires significant changes, they will stop development and move back to the design phase.

Once the application has been fully implemented, it needs to be tested. This is where bugs are found and fixed, and an analysis of the final solution against the initial requirements is done. The project manager, in the requirements document, will have created user use cases. Those use cases can be used as a basis of evaluation to ensure the application does precisely what is required of it.

Finally, the application will be released to the end user(s). Feedback in the form of bug reports and feature requests will begin to flow in, and the

development team will need to respond to those requests. To learn more about the Waterfall method, check out the following link: <https://management.org/waterfall-methodology>.

There are benefits and disadvantages to both Agile/Scrum and Waterfall. It will be up to your team to decide which method works best for them — unless they already have! In that case, you will already be somewhat familiar with how the team operates, thanks to this primer. However, it is not all you need to know about working on a team. You will also need to learn how to contribute to the codebase.

Additional Skills You'll Need to Seek Out

As we have touched on throughout this book, you will need to have a solid understanding of whichever **version control system (VCS)** your team uses. The most popular these days is Git; however, there are others, such as Mercurial (hg) and Subversion (svn). These systems allow a team to contribute to the same source code independently while maintaining a history of changes (called a **changelog**). Figure out what VCS your team is using and seek out the documentation for it. You will need to spend time learning how to clone the repository, create a branch to do your work in, commit code to the branch, and push your code up to the online repository. You will also need to learn how to create a pull request — that is, a request for your colleagues to review and comment on your code before merging it with the main codebase.

There are plenty of things that can go wrong when trying to merge your code into the main codebase via a pull request. If more than one person is working on the same file, there may be an instance where the changes that have been made conflict with one another. In that instance, you will need to learn how to read, understand, and resolve a merge conflict. It is also not uncommon when you are first starting out to commit your code to the wrong branch. You will need to learn how to undo those commits without losing your work, then place those changes in a new branch.

Your mentor can help guide you through some of these advanced scenarios as they come up, but having a basic familiarity with the tools you are using is likely an expectation that your team has of you. Take the time to do some

research because no matter what type of code you are writing, if you are collaborating with others, you will need to understand these concepts.

You will also need to understand how to properly test your code. We did not cover writing tests in Flutter in this book, but that does not mean it is not vitally important to understand. When it comes to writing tests, there are several types in Flutter: widget tests, unit tests, BLoC tests, integration tests, and golden tests.

Widget tests test that a widget can be built correctly in various scenarios. This is where you would test whether a null value for a nullable parameter prevents the widget from building or whether the logic inside the build method of your widget produces the desired output for a given input.

Unit tests seek to test individual methods, rather than widgets. If you have created a helper function that validates whether somebody with a given birthday is over the age of 18, your unit test should verify that the function works as expected, including with edge cases. An edge case might be that the user's birthday is *today*, or the given date is in the future.

BLoC tests, as the name implies, are only relevant to the BLoC package. A BLoC test would verify that the expected states are emitted when given events are passed into the BLoC. This includes verifying not only the correct state, but that any data that was expected to mutate during the request has successfully done so.

Integration tests are the most complete tests that can be written. An integration test will launch your application and perform a series of pre-planned steps to verify that functionality works across many features. While the rest of the tests have focused on smaller units to test, such as a single widget or method, an integration test brings everything together and verifies interoperability between features.

Finally, there are golden tests. A golden test will verify that the UI has not changed from test run to test run. When a golden test is created, the screen(s) will be rendered into images that are then stored alongside the tests. When the tests are run subsequently, the same screen(s) will be built, and the resulting images will be compared to the stored images. If there is a deviation in the pixels, the test will fail.

Learning about version control systems and testing your code is a great start for the additional skills you will need to seek out as you grow and flourish in your new role. However, they are far from the only skills you will need to learn.

At some point, you will likely be asked to create animations in Flutter. Flutter has a wide variety of widgets available to help construct any type of animation you could possibly imagine. However, with so many options and so much complexity, you may wish to take a quick course to learn how to create animation in Flutter. Here (<https://flutter.thinkific.com/>) is a recommended course that will teach you everything from the basics to creating your own, custom animations using Flutter's `CustomPainter` widget. The course is inexpensive and relatively short. Readers of this book may use code `JOBSEEKERS` for 25% off at checkout. It is well worth your time, especially as you look to further your career. Speaking of furthering your career...

Suggestions for Furthering Your Career

Your new job is the first step in your career as a Flutter developer. Flutter is a rapidly evolving framework, resulting in several rewrites of chapters in this very book before its completion. In fact, by the time you are reading it, there are likely to be other parts of this book that are no longer accurate! You will want to stay up to date with all the goings on of Flutter as it continues to grow and evolve. Consider subscribing to the Flutter Public Announcements group (<https://groups.google.com/g/flutter-announce>) to get emails when new releases of Flutter and the associated tools, such as browser extensions, are released. This is the single best way to learn about recent changes to Flutter, including new features and breaking changes.

As we have discussed throughout this book, the Flutter team maintains a YouTube channel full of different types of content. Consider subscribing to their channel and watching some of the playlists that might be relevant to your interests. You can find the Flutter YouTube channel here: <https://www.youtube.com/@flutterdev>.

The Flutter developer community is one of the warmest, most welcoming developer communities around, and you should be a part of it! Whether you are on Matrix, Discord, Slack, Reddit, or any other platform you can think

of, there is a place for you. Do you identify as female or non-binary? If so, you may be interested in joining the Flutteristas (<https://flutteristas.org/>). Maybe you are interested in joining a meetup to talk about Flutter with other developers in your area? If so, check out <https://www.meetup.com/pro/flutter/>. Whichever platform you choose to engage in, no matter where it is, keep in mind the culture of the Flutter community (<https://flutter.dev/culture>). Networking with the Flutter community will open new opportunities for growth, as well as new job opportunities. Consider how you can contribute to new Flutter developers' growth as you grow in your own career.

Remain humble. It is a perilous trap to fall into when you start to gain knowledge and experience in a new domain to feel better than others or that, *they should know this because I know it, and it is really not that hard*. Take active steps to not be this person. Give freely to those who are following in your footsteps. Be patient and kind. Remember how you feel *now* because that is how they will feel when they come to you later.

Consider contributing to open-source projects or the Flutter framework itself. You do not have to spend all your time contributing to projects, nor do you have to write code to fix issues (although that is *always* appreciated!). Sometimes, filing a bug report or adding steps to reproduce an issue that somebody else reported is enough. The key here is giving back to the community, which is not only the right thing to do but also shines a light on you as a community contributor.

Learn aspects of software development outside of your current domain of knowledge. Explore color theory, the principles of UI/UX and the design process, database management, CI/CD (continuous integration/continuous development — the system used to automatically build, test, and deploy your code), and backend development. Learn about the business aspects of building software. Explore management to see if it is a path you would like to go down.

Whatever you do in your career, remember where you came from and be open to change, as change is the only constant in life. And, when the time is right, publish a book about Flutter for the next generation of developers.

Conclusion

This book was quite an adventure. Together, we learned about the history of application development, why Flutter was created, and some of the history of its development. We learned a whole host of concepts, from what a *widget* is to what is involved in the technical interview, and everything in-between. Whether you began this journey as a seasoned software developer with years of experience or were simply curious about getting started with app development, hopefully the concepts contained within this book were helpful to you. If you just landed your first job as a Flutter developer, a heartfelt *congratulations* are in order! (In that case, consider sharing this book with someone whom you feel could benefit from it!)

Wherever your journey takes you from here, *thank you* for purchasing this book. Always remember: if there is one thing that is certain it is that by completing this book you have already proven that you have the necessary skills to be successful in this industry.

Flutter Community Links

Join the Flutter community on:

- **Matrix:** <https://matrix.to/#/#flutter-devs:matrix.org>
- **Discord:** <https://discord.com/invite/N7Yshp4>
- **Slack:** <https://fluttercommunity.dev/joinslack>
- **Reddit:** <https://www.reddit.com/r/FlutterDev/>
- **Twitter:** <https://twitter.com/FlutterDev>

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

A

adwaita package

aggressive composition design paradigm

AMD Processors

Android Emulator

Android SDK Build-Tools

Android SDK Location

Android Studio

installing

Android Virtual Device (AVD)

API

data, requesting from

data, sending

API Layer

building

API Responses

classifying
app icons
guidelines
application
assets, adding
creating
finishing
images, adding
skeleton, building
technical requirements

Application Programming Interface
apps
abridged history

app state
managing, with BLoC
managing, with Cubit
managing, with inherited widgets
managing, with Provider package
versus ephemeral state

arguments
passing, into route

async
asynchronous code

AVD Manager

B

BLoC

used, for managing app state
breakpoint
setting

BuildContext
build() method

buttons

CupertinoButton

ElevatedButton

GestureDetector

OutlinedButton

C

CachedNetworkImage widget

callback function

callees

Call Tree tab

cascade notation

case studies, Flutter

Alibaba

BMW

ByteDance

eBay

Google

Toyota

Center widget

changelog

checkboxes

CMake

CocoaPods
code generation
collaboration tips
with developers
Column and Row widgets
community
console
debugging
Container widget
CPU Flame Chart tab
cross-platform UIs
building, with packages

CRUD
Cubit
used, for managing app state
culture fit interview
Cupertino pickers
Cupertino widget
custom stateless widget
creating

D

Dart
Dart 3
dart:io
benefits
limitations
data

running, from screen
debugging
debugPrint() function
declarative routing
decorator
default Flutter application
exploring
dependency injection
devices, with Apple Silicon
DevTools
URL
DevTools Widget Inspector
dropdowns
material dropdown

E

Editor Language Status field
enum
Environment Variables
ephemeral state
versus, app state
escaping
exception
handling

F

flame chart
Fluent UI
Flutter

- Alibaba case study
- BMW case study
- ByteDance case study
 - development experience
- eBay case study
 - features
- Google case study
 - growth, in marketplace
 - installing, via command line
 - installing, with PowerShell
 - rapid development
- Toyota case study
 - users
- Flutter 3.0
- Flutter 3.7
- Flutter 3.10
- Flutter application
 - building
 - developing, on macOS
 - developing, on Windows
 - running
- flutter_bloc package
- Flutter community
- Flutteristas
- Flutter Public Announcements group
- Flutter Recruiters
- question and answers

Flutter's asynchronous widgets

Flutter YouTube channel

FormField widget

Form widget

Future

welcoming

G

GestureDetector

Git

Google I/O

Gradle

guard clause

H

Hero widget

Hummingbird

preview

Hyper-V Virtual Machine

I

imperative routing

inherited widgets

integrated development environment (IDE)

Intel x86 Emulator Accelerator

interpolation

interview process

culture-fit interview

technical interview

J

Jank

Job

additional skills

preparing for

job searching

JSON

K

Key

Kotlin

L

ListView

not a ListView

ListView.builder

Logging panel

M

Material widget

Meta

Microsoft Visual Studio

Model View Controller (MVC)

Model-View-ViewModel (MVVM)

multi-platform app development

history

solution

N

named routes

for routing

Navigator

Navigator 1.0
Navigator 2.0
NDK (side by side)

P

package
for building cross-platform UIs
for helping with routing
using, for playing audio
Path variable
performance issues
defining
pickers
CupertinoDatePicker
CupertinoPicker
CupertinoTimerPicker
Material style date picker
Pixel 4
platform-specific navigation paradigms
PowerShell
using
print() function
private object
profile mode
exploring
Provider package
used, for managing app state
Pub

packages
packages, finding
Pub Package Management System
Pub Points
pubspec.yaml

R

radios
random access memory (RAM)
React Native
release mode
responsive layout
responsive layouts
 building
Rosetta 2
route
 arguments, passing
RouterDelegate
routing
 packages, using
 using named routes
Row widget
runApp() function
runtime variables
exploring

S

screen
 data, running from

Scrum team
SDK Manager
SDK Tools tab
segmented controls
Select widget mode
shared_preferences, Pub

dio
flutter_bloc
isar
url_launcher

shopping cart application
preparing for

Simple Signup Form
building

singleton

sinks

sliders

splash screens

stacked navigation

stack frame

stack trace

stateful widgets

stateless widgets

Center widget

Column and Row widgets

Container

custom stateless widget, creating

Text widget

state management
step into icon
step out icon
step over icon
Stream
StreamController
Supply Chain Levels for Software Artifacts (SLSA)
Swift
synchronous code
system
hacking
System Properties window

T

technical interview
TextField
text input
capturing
textInputAction
Text widget
Text widget creation
tracking
troubleshooting
typedef

U

user input
handling

V

ValueChanged widget
Visual Crossing Weather
Visual Studio Code
 configuring
 installing
Visual Studio Installer
Vsync

W

Waterfall methodology
WeatherApi
WeatherIcon Class
Widget Details Tree
widgets
 animating, from one route to another
 drawing, to screen
 inherited widgets
 optimizing
 stateful widgets
 stateless widgets
widget tree
Windows Developer Mode
Windows PowerShell
Windows PowerShell ISE

X

Xamarin
Xcode
 installing

Xianyu

Y

Yaru

YaruTheme widget