

Dedicated to Giorgia, my friends and my family.

Contents

1	Welcome	15
1.1	Introduction	15
1.1.1	Who is this book for	16
1.1.2	Author	16
1.1.3	Acknowledgments	17
1.1.4	Online resources and the quiz	17
1.2	Introduction to Dart	18
1.2.1	Supported platforms	18
1.2.2	Package system	22
1.2.3	Hello World	22
1.3	Intorduction to Flutter	24
1.3.1	How does it work	24
1.3.2	Why Flutter uses Dart	28
1.3.3	Hello world	29
I	The Dart programming language	33
2	Variables and data types	35
2.1	Variables	35
2.1.1	Initialization	36
2.1.2	final	37
2.2	Data types	38
2.2.1	Numbers	39
2.2.1.1	Good practices	40
2.2.2	Strings	41
2.2.3	Enumerated types	43
2.2.3.1	Good Practices	44

2.2.4	Booleans	44
2.2.5	Arrays	45
2.3	Nullable and Non-nullable types	46
2.4	Data type operators	49
2.4.1	Arithmetic operators	49
2.4.2	Relational operators	50
2.4.3	Type test operators	51
2.4.4	Logical operators	52
2.4.5	Bitwise and shift operators	52
3	Control flow and functions	54
3.1	If statement	54
3.1.1	Conditional expressions	54
3.1.2	Good practices	55
3.2	switch statement	56
3.3	for and while loops	58
3.3.1	for-in loop	59
3.4	Assertions	60
3.5	Good practices	61
3.6	The basics of functions	61
3.6.1	The Function type	63
3.7	Anonymous functions	64
3.8	Optional parameters	67
3.8.1	Named parameters	67
3.8.2	Positional parameters	69
3.9	Nested functions	70
3.10	Good practices	71
3.11	Using typedefs	72
4	Classes	74
4.1	Libraries and visibility	76
4.1.1	Encapsulation	78
4.1.2	Good practices	80
4.2	Constructors	80
4.2.1	Initializer list	83
4.2.2	Named constructors	84
4.2.3	Redirecting constructors	85
4.2.4	Factory constructors	85

Contents

4.2.5	Instance variables initialization	86
4.2.6	Good practices	88
4.3	const keyword	90
4.3.1	const constructors	92
4.3.2	Good practices and annotations	93
4.4	Getters and setters	95
4.4.1	Good practices	97
4.5	Operators overload	98
4.5.1	callable classes	99
4.6	Cloning objects	101
5	Inheritance and Exceptions	105
5.1	Inheritance	105
5.1.1	super and constructors	108
5.1.2	Abstract classes	109
5.1.3	Interfaces	111
5.1.3.1	extends vs implements	113
5.1.4	Mixins	116
5.1.5	Good practices	120
5.2	Extension methods	121
5.2.1	Good practices	123
5.3	The Object class	124
5.3.1	Comparable<T>	127
5.4	Exceptions	128
5.4.1	on and catch	129
5.4.2	finally	131
5.4.3	Good practices	132
6	Generics and Collections	136
6.1	Generic types	136
6.1.1	Introduction	136
6.1.2	Type safety	137
6.1.3	Usage	139
6.2	Collections	141
6.2.1	List	141
6.2.1.1	Collection statements	142
6.2.1.2	Implementation	143
6.2.2	Set	145

6.2.2.1	Implementation	147
6.2.3	Map	147
6.2.3.1	Implementation	149
6.3	Good practices	150
6.3.1	operator== and hashCode	150
6.3.2	Transform methods	153
7	Asynchronous programming	157
7.1	Introduction	157
7.2	Futures	158
7.2.1	Comparison	161
7.2.2	async and await	163
7.2.3	Good practices	165
7.3	Streams	166
7.3.1	Streams and generators	168
7.3.2	Subscribers	171
7.3.3	Differences	174
7.3.4	Using a controller	175
7.4	Isolates	179
7.4.1	Multiple isolates and Flutter	184
8	Coding principles with Dart	186
8.1	SOLID principles	187
8.1.1	Single Responsibility Principle	187
8.1.2	Open closed principle	188
8.1.3	Liskov Substitution Principle	190
8.1.4	Interface Segregation Principle	191
8.1.5	Dependency Inversion Principle	193
8.2	Dependency Injection	194
8.2.1	Constructor injection	196
8.2.2	Method injection	198
II	The Flutter framework	201
9	Basics of Flutter	203
9.1	Structure and tools	203
9.1.1	Folder structure	205
9.1.2	The pubspec.yaml file	206

Contents

9.1.3	Hot Reload	210
9.1.4	Linter rules	211
9.1.5	Tree shaking and constants	213
9.2	Widgets and State	215
9.2.1	Basic widgets	218
9.2.1.1	Text	218
9.2.1.2	Row	219
9.2.1.3	Column	220
9.2.1.4	ListView	222
9.2.1.5	Container	223
9.2.1.6	Stack and Positioned	225
9.2.2	Stateless and StatefulWidget widgets	226
9.2.2.1	Good practices	231
9.2.3	Keys	233
9.3	Rebuilds and optimization	236
9.3.1	const constructor	237
9.3.2	Prefer widget composition over functions	238
9.4	Architecture	240
9.4.1	Element and RenderObject	243
9.4.2	Foreign Function Interface	249
9.4.3	Method channels	251
10	Building UIs in Flutter	255
10.1	Material	255
10.1.1	Scaffold	257
10.1.2	Material widgets	260
10.1.2.1	Buttons	261
10.1.2.2	Dialogs	262
10.2	Cupertino	266
10.2.1	CupertinoPageScaffold	268
10.2.2	Cupertino widgets	270
10.3	Building layouts	272
10.3.1	Platform support	272
10.3.1.1	Single OS	272
10.3.1.2	Multiple OSes	274
10.3.2	Responsive UIs	276
10.3.2.1	LayoutBuilder	277
10.3.2.2	MediaQuery	280

10.3.2.3	Good practices	280
10.3.3	Scrolling and constraints	283
10.3.4	Using themes	287
11	State management	290
11.1	Updating the UI	291
11.1.1	Considerations	293
11.1.2	Good practices	298
11.2	Passing the state with Provider	299
11.2.1	Considerations	303
11.2.1.1	Provider class	303
11.2.1.2	Consumer class	304
11.2.2	Good practices	307
11.3	Alternative to setState: BLoC pattern	312
11.3.1	Considerations	318
11.3.1.1	BlocListener class	321
11.3.2	BlocObserver class	323
11.3.3	Persisting the state with HydratedBloc	324
11.3.4	Undo and redo with ReplayBloc	328
11.3.5	The internals of Bloc: Cubit	330
11.3.6	Good practices	333
11.4	Good practices for state management	336
12	Routes and navigation	339
12.1	Basics of navigation and routing	339
12.1.1	Creation of routes	340
12.1.2	The main.dart file	341
12.1.3	The routes.dart file	342
12.1.4	Navigating between pages	345
12.1.5	Good practices	346
12.1.6	Navigator 2.0	348
12.2	Passing data between pages and widgets	350
12.2.1	The Navigator class	351
12.2.2	Passing data with Navigator	352
12.2.3	Passing data with provider	355
12.3	Other routing techniques	359
13	Localization and internationalization	363

Contents

13.1	Introduction	363
13.2	Manual internationalization	368
13.2.1	AppLocalization	368
13.2.2	Localization delegate	371
13.2.3	Backward compatibility	373
13.3	Internationalizing using intl	375
13.3.1	AppLocalization	375
13.3.2	Localization delegate	379
13.3.3	Plurals and data interpolations	380
13.4	Considerations	382
14	Animations	385
14.1	Implicit animations	385
14.2	The animation library	392
14.2.1	AnimatedWidget	392
14.2.2	AnimationBuilder	397
14.2.2.1	Curves	400
14.2.2.2	Tweens	402
14.3	Custom animations	405
14.4	Good practices	410
14.4.1	Hero animations	411
14.4.2	Custom route transitions	413
15	Working with JSON and other formats	417
15.1	Parsing JSON	417
15.1.1	Manual parsing	418
15.1.1.1	Parsing lists	421
15.1.1.2	Parsing nested objects	422
15.1.2	Automatic parsing	423
15.1.2.1	Parsing lists	427
15.1.2.2	Parsing nested objects	427
15.2	Parsing XML	429
15.2.1	Parsing strings	431
15.2.2	Building XML strings	434
16	Testing and profiling apps	437
16.1	Testing Flutter apps	437
16.1.1	Unit Test	438

16.1.1.1	Testing asynchronous code and streams	442
16.1.1.2	Mocking dependencies	443
16.1.1.3	Unit testing blocs	447
16.1.2	Widget Test	449
16.1.2.1	Testing blocs and providers	452
16.1.2.3	Integration testing	453
16.2	Testing performances	458
16.2.1	DevTools	459
16.2.2	Using the Flutter inspector	459
16.2.3	Using the Timeline view	461
16.2.4	Using the Memory view	464
16.2.5	Using the Network view	466
16.2.6	Using the Logging view	467
16.2.7	Monitoring widget rebuilds	468
III	Practical Flutter examples	471
17	Networking	473
17.1	Making HTTP requests	473
17.1.1	GET requests	473
17.1.2	POST requests and headers	478
17.1.3	Good practices	479
17.2	Working with data	481
17.2.1	Downloading data	483
17.2.2	Uploading data	487
17.2.3	Good practices	490
17.3	Advanced REST API calls	491
17.3.1	Model classes	492
17.3.2	Parsing JSON	494
17.3.3	HTTP Client	497
18	Assets, images and multimedia	502
18.1	Assets and images	502
18.2	Working with images	505
18.2.1	Loading from the network	506
18.3	Scalable vector graphics	510
18.3.1	Working with SVG files	512

Contents

18.3.2 Loading from the network	513
18.4 Audio and video with Flutter	517
18.4.1 Playing a video	517
18.4.2 Listening to music	522
19 Forms and gestures	529
19.1 Forms and validation	529
19.1.1 Keyboard and overflows	534
19.1.2 Getting the text from a text field	537
19.1.3 Constraining the input	539
19.2 Gestures	543
19.2.1 Swipe to dismiss	544
19.2.2 Dragging items	548
20 Interacting with the device	555
20.1 Taking a picture	555
20.2 Working with sensors	560
20.3 Working with Geolocation	563
20.4 Platform-specific packages	567
20.4.1 Battery level	568
20.4.2 Device info	568
20.4.3 Internet connectivity	569
20.4.4 Shared preferences	572
21 Widgets showcase	573
21.1 Material	573
21.1.1 Drawer	573
21.1.2 BottomNavigationBar	577
21.1.3 NavigationRail	579
21.1.4 TabBar	581
21.1.5 Stepper	584
21.1.6 DataTable	586
21.2 Cupertino	593
21.2.1 CupertinoDatePicker	593
21.2.2 CupertinoActionSheet	594
21.2.3 CupertinoSegmentedControl	595
21.3 Community widgets	596
21.3.1 Flutter Hooks	597

21.3.2 State notifier	603
22 Using Firebase with Flutter	606
22.1 Installation	606
22.2 Using Firestore as backend	610
22.2.1 Building the backend	612
22.2.2 Building the frontend	615
22.2.3 Working with data	619
22.2.4 Transactions and batches	622
22.3 Monetizing your apps with AdMob	623
22.3.1 Ad banners	625
22.3.2 Rewarded video ads	632
22.4 Flutter ML Kit	633
22.4.1 Detecting faces	634
22.4.1.1 CustomPainter and the canvas	641
22.4.2 Firebase vision kit	644
22.5 Push notifications with FCM	645
22.5.1 Handling push notifications	648
22.5.2 Sending push notifications	649
22.6 Authenticating with Firebase	653
22.6.1 Authentication features	658
23 Publishing packages and apps	661
23.1 Publishing packages on pub.dev	661
23.1.1 Creating the package	662
23.1.2 Documenting the code	664
23.1.3 Reviewing and publishing	667
23.1.4 Scores and good practices	668
23.1.5 Verified publishers and Flutter favorite	670
23.2 Publishing apps on the stores	672
23.2.1 Releasing Android apps	673
23.2.2 Releasing iOS apps	677
23.2.3 Splash screens	679
23.2.4 Doing CI/CD for Flutter	681
23.2.4.1 GitHub actions	682
24 Complete Flutter project example	686
24.1 Preparing the project	686

Contents

24.1.1	Folder structures and basic setup	687
24.2	State management and model classes	689
24.2.1	Authentication bloc	693
24.2.2	Credentials bloc	696
24.2.3	Localization files	701
24.3	Building the UI	702
24.3.1	Creating the login form	707
24.3.2	Creating the welcome page	711
24.4	Testing the code	712
24.4.1	Unit tests	712
24.4.2	Widget tests	714
A	Dart Appendix	717
A.1	The I/O library	717
A.1.1	Files	717
A.1.2	Directories	718
A.1.3	Server side Dart	719
A.2	Date and time	721
A.3	Obfuscating Dart code	724
B	Flutter Appendix	727
B.1	Riverpod	727
B.1.1	Usage	728
B.1.2	Combining providers	734
B.1.3	Testing	736
B.2	Local databases	738
B.2.1	Hive (NoSQL)	738
B.2.2	SQLite (SQL)	741
B.3	Initializing data at startup	743
B.4	Accessibility	745
B.5	The Flutter community	748
B.6	Flutter SDK management	751
B.6.1	Web and Desktop	752
Index		755

Flutter and the related logo are trademarks of Google LLC. We are not endorsed by or affiliated with Google LLC.

1 | Welcome

1.1 Introduction

Thank you for having put your faith on this book. If you want to learn how to use a powerful tool that allows developers to quickly create native applications with top performances, you've chosen the right book. Nowadays companies tend to consider cross-platform solutions in their development stack mainly for three reasons:

1. **Faster development**: working on a single codebase;
2. **Lower costs**: maintaining a single project instead of many (N projects for N platforms);
3. **Consistency**: the same UI and functionalities on any platform.

All those advantages are valid regardless the framework being used. However, for a complete overview, there's the need to also consider the other side of the coin because a cross-platform approach also has some drawbacks:

1. **Lower performances**: a native app can be slightly faster thanks to the direct contact with the device. A cross-platform framework might produce a slower application due to a necessary *bridge* required to communicate with the underlying OS;
2. **Slower releases**: when Google or Apple announce a major update for their OS, the maintainers of the cross-platform solution could have the need to release an update to enable the latest features. The developers must wait for an update of the framework, which might slow down the work.

Every framework adopts different strategies to maximize the benefits and minimize or get rid of the drawbacks. The perfect product doesn't exist, and very likely we will never have one, but there are some high quality frameworks you've probably already heard:

- **Flutter**. Created by Google, it uses Dart;

- **React Native.** Created by Facebook, it is based on javascript;
- **Xamarin.** Created by Microsoft, it uses the C#;
- **Firemonkey.** Created by Embarcadero, it uses Delphi.



Flutter



React Native



Xamarin



Firemonkey

During the reading of the book you will see how Google tries to make the cross-platform development production-ready using the Dart programming language and the Flutter UI framework. You will learn that Flutter renders everything by itself¹ in a very good way and it doesn't use any intermediate *bridge* to communicate with the OS. It compiles directly to ARM (for mobile) or optimized JavaScript (for web).

1.1.1 Who is this book for

To get the most out of this book, you should already know the basics of object-oriented programming and preferably at least an "*OOP language*" such as Java or C#. Our goal is trying to make the contents of this book understandable for the widest possible range of developers. Nevertheless, you should already have a minimum of experience in order to better understand the concepts.

If you already know what is a class, what is inheritance and what is nullability, part 1 of this book is going to be a walk in the park. Foreknowledge aside, we will talk about both Dart and Flutter "from scratch" so that the reader can understand any concept regardless the expertise level.

1.1.2 Author

Alberto Miola is an Italian software developer that started working with Delphi (Object Pascal) for desktop development and Java for back-end and Android apps. He currently works in Italy where he daily uses Flutter for mobile and Java for desktop and back-end. Alberto graduated

¹For example, it doesn't use the system's OEM widgets

in computer science at University of Padua with a thesis about cross-platform frameworks and OOP programming languages.

1.1.3 Acknowledgments

This book owes a lot to some people the author has to mention here because he thinks it's the minimum he can do to express his gratitude. They have technically supported the realization of this book with their fundamental comments and critiques that improved the quality of the contents.

- **Rémi Rousselet.** He is the author of the famous "provider" ² package and a visible member in the Flutter/Dart community. He actively answers on stackoverflow.com helping tons of people and constantly works in the creation of open source projects.
- **Felix Angelov.** Felix is a Senior Software Engineer at Very Good Ventures. He previously worked at BMW for 3 years and is the main maintainer of the bloc state management library. He has been building enterprise software with Flutter for almost 2 years and loves the technology as well as the amazing community.
- **Matej Rešetár.** He is helping people get prepared for real app development on resocoder.com and also on the Reso Coder YouTube channel. Flutter is an amazing framework but it is easy to write spaghetti code in it. That's why he's spreading the message of proper Flutter app architecture.

Special thanks to my friends Matthew Palomba and Alfred Schilken which carefully read the book improving the style and the quality of the contents.

1.1.4 Online resources and the quiz

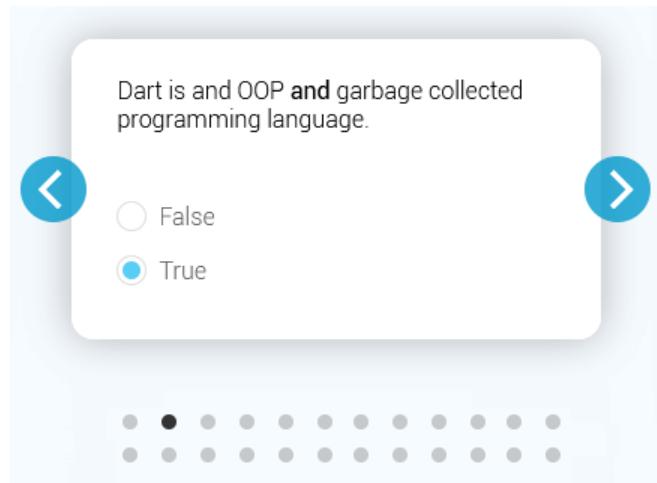
The official website of this book ³ contains the source code of the examples described in Part III. While reading the chapters you might encounter this box:

- (Resources > Chapter 16 > Files download)

²<https://pub.dev/packages/provider>

³<https://fluttercompletereference.com>

It indicates that if you navigate to the *Resources* page of our website, you'll find the complete source code of the example being discussed at *Chapter 16 > Files download*. In addition, you can play the "Quiz game" which will test the Dart and Flutter skills you've acquired reading this book.



At the end, the result page will tell you the exact page of the book at which you can find an explanation of the answer.

1.2 Introduction to Dart

Dart is a client-optimized, garbage-collected, OOP language for creating fast apps that run on any platform. If you are familiar with an object oriented programming language such as Java or C# you might find many similarities with Dart. The first part of this book aims to show how the language can help you solving problems and the vastness of its API.



1.2.1 Supported platforms

Dart is a very flexible language thanks to the environment in which it lives. Once the source code has been written (and tested) it can be deployed in many different ways:

- **Stand-alone.** In the same way as a Java program can't be run without the Java Virtual Machine (JVM), a stand-alone Dart program can't be executed without the Dart Virtual Machine (DVM). There's the need to download and install the DVM which to execute Dart in a command-line environment. The SDK, other than the compiler and the libraries, also offers a series of other tools:
 - the pub package manager, which will be explored in detail in chapter 23;
 - dart2js, which compiles Dart code to deployable JavaScript;
 - dartdoc, the Dart documentation generator;
 - dartfmt, a code formatter that follows the official style guidelines.

In other words, with the stand-alone way you're creating a Dart program that can only run if the DVM is installed. To develop Flutter apps for any platform (mobile, web and desktop), instead of installing the "pure" Dart SDK, you need to install Flutter⁴ (which is basically the Dart SDK combined with Flutter tools).

- **AOT compiled.** The Ahead Of Time compilation is the act of translating a high-level programming language, like Dart, into native machine code. Basically, starting from the Dart source code you can obtain a single binary file that can execute natively on a certain operating system. AOT is really what makes Flutter fast and portable.



With AOT there is **NO** need to have the DVM installed because at the end you get a single binary file (an *.apk* or *.aab* for Android, an *.ipa* for iOS, an *.exe* for Windows...) that can be executed.

- Thanks to the Flutter SDK you can AOT compile your Dart code into a native binary

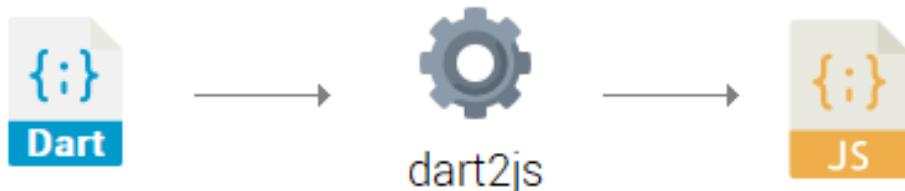
⁴<https://flutter.dev/docs/get-started/install>

for mobile, web and desktop.

- As of Flutter 1.21, the Dart SDK is included in the Flutter SDK so you don't have to install them separately. They're all bundled in a single install package.
- Starting from version 2.6, the `dart2native` command (supported on Windows, macOS and Linux) makes AOT compiles a Dart program into x64 native machine code. The output is a standalone executable file.

AOT compilation is very powerful because it natively brings Dart to mobile/desktop. You'll end up having a single native binary which doesn't require a DVM to be installed on the client in order to run the application.

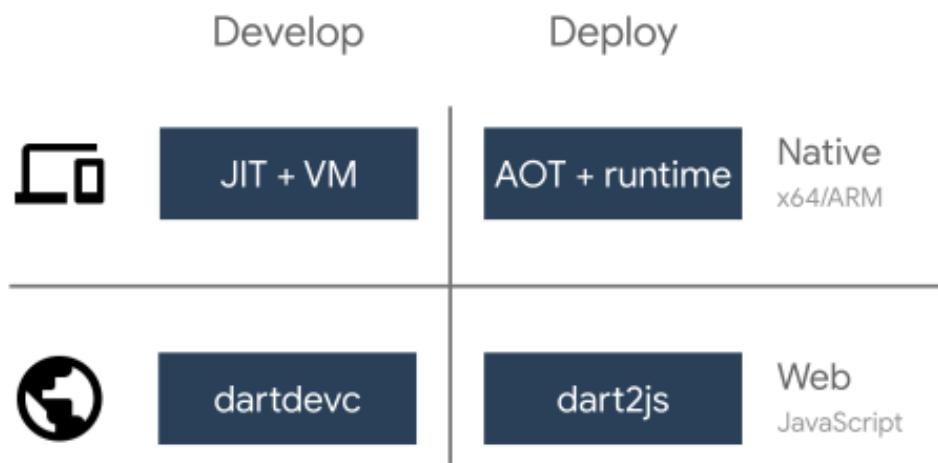
- **Web.** Thanks to the `dart2js` tool, your Dart project can be "transpiled" into fast and compact JavaScript code. By consequence Flutter can be run, for example, on Firefox or Chrome and the UI will be identical to the other platforms.



AngularDart⁵ is a performant web app framework used by Google to build some famous websites, such as "AdSense" and "AdWords". Of course it's powered by Dart!

So far we've covered what you can do with Dart when it comes to deployment and production-ready software. When you have to debug and develop, both for desktop/mobile and web, there are useful some tools coming to the rescue.

⁵<https://angulardart.dev/>



This picture sums up very well how the Dart code can be used in development and deployment. We've just covered the "Deploy" side in the above part, so let's analyze the "Develop" column:

- **Desktop/mobile.** The Just In Time (JIT) technique can be seen as a "real time translation" because the compilation happens while the program is executing. It's a sort of "dynamic compilation" which happens while the program is being used.

JIT compilation, combined with the DVM (JIT + VM in the picture), allows the dispatch of the code dynamically without considering the user's machine architecture. In this way it's possible to smoothly run and debug the code everywhere without having to mess up with the underlying architecture.

- **Web.** The Dart development compiler, abbreviated with **dartdevc**, allows you to run and debug Dart web apps on Google Chrome. Note that dartdevc is for development only: for deployment, you should use **dart2js**. Using special tools like *webdev*⁶ there's the possibility to edit Dart files, refreshing Chrome and visualizing changes almost immediately.

As you've just seen, Dart can run literally everywhere: desktop, mobile and web. This book will give you a wide overview of the language (Dart version 2.10, with null safety support) and all the required skills to create easily maintainable projects.

⁶<https://dart.dev/tools/webdev#serve>

1.2.2 Package system

Dart's core API offers different packages, such as `dart:io` or `dart:collection`, that expose classes and methods for many purposes. In addition, there is an official online repository called `pub` containing packages created by the Dart team, the Flutter team or community users like you.



If you head to <https://pub.dev> you will find an endless number of packages for any purpose: I/O handling, XML serialization/de-serialization, localization, SQL/NoSQL database utilities and much more.

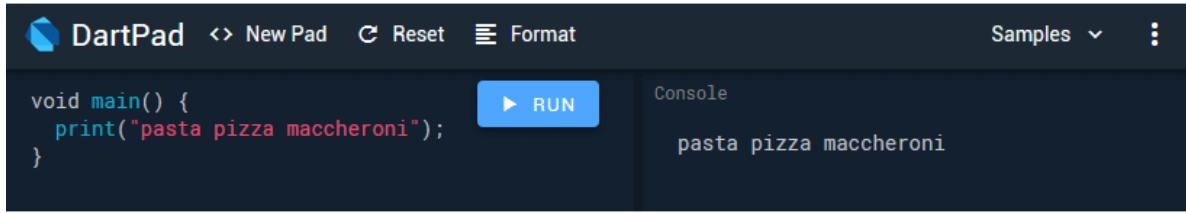
1. Go to <https://pub.dev>, the official repository;
2. Let's say you're looking for an equation solving library. Type "equations" in the search bar and filter the results by platform. Some packages are available only for Dart, others only for Flutter and a good part works for both;
3. The page of the package contains an installation guide, an overview and a guide so that you won't get lost.

You should check the amount of *likes* received by the community and the overall *reputation* of the package because those values indicate how mature and healthy the product is. You will learn how to properly write a library and how to upload it to the pub.dev repository in order to give your contribution to the growth of the community.

1.2.3 Hello World

The simplest way you have to run your Dart code is by opening DartPad⁷, an open-source compiler that works in any modern browser. Clicking on "New Pad" you can decide whether creating a new Dart or Flutter project (with latest stable version of the SDK).

⁷<https://dartpad.dartlang.org/>



It's the perfect tool for the beginners that want to play with Dart and try the code. If you're new to the language, start using DartPad (which is absolutely **not** an IDE). It always has the latest version of the SDK installed and it's straightforward to use.

```
void main() {
    // Best food worldwide!
    print("pasta pizza maccheroni");
}
```

Like with Java and C++, any Dart program has to define a function called `main()` which is the entry point of the application. Very intuitively the `print()` method outputs to the console, on the right of the DartPad, a string. Starting from chapter 2, you'll begin to learn the syntax and the good practices that a programmer should know about Dart.



Android Studio



Visual Studio Code

When you develop for real world applications, you're going to download the whole SDK and use an IDE like IntelliJ IDEA, Android Studio or VS Code. DartPad doesn't give you the possibility to setup tests, import external packages, add dependencies and test your code.

1.3 Intorduction to Flutter

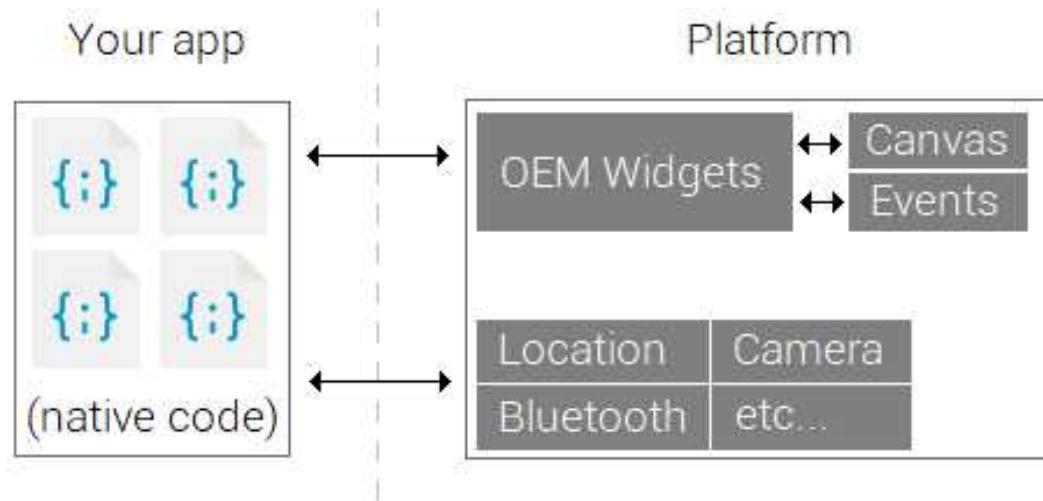
Flutter is an UI toolkit for building natively compiled applications for mobile, desktop and web with a single codebase. At the time of writing this book, only Flutter for mobile is stable and ready for production. Web support is currently in beta while desktop (macOS, Linux and Windows) is in early alpha: they will be covered in a future release of this reference once they will be officially released as stable builds.



Being familiar with *Jetpack compose* or *React Native* is surely an advantage because the concepts of reactive views and "components tree" are the fundamentals of the Flutter framework.

1.3.1 How does it work

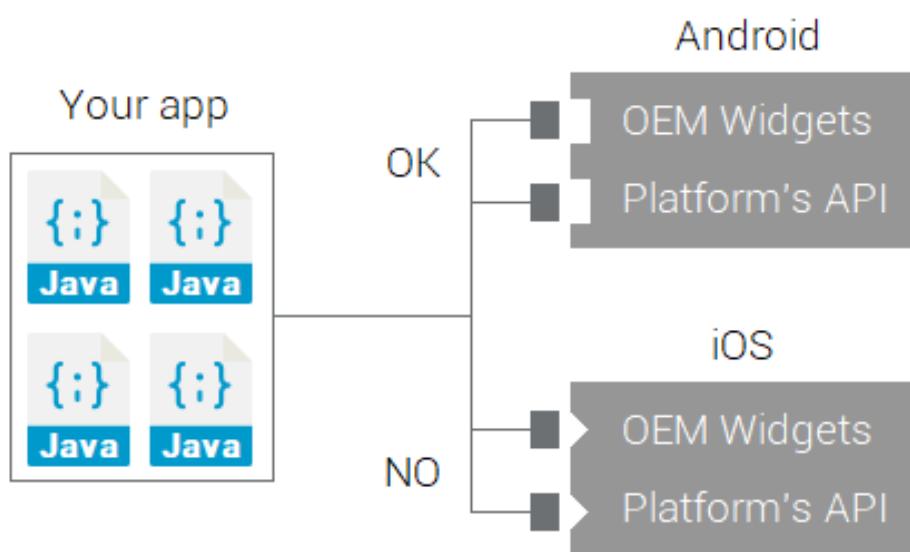
This picture shows how a native app interacts with the OS, whether it's been written in Kotlin (or Java) for Android or Swift (or Objective-C) for iOS. We're going to use these 2 platforms as examples in this section.



1. The platform, which can be Android or iOS, exposes a series of OEM widgets used by the app to build the UI. Those widgets are **fundamental** because they give our app the capabilities to paint the UI, use the canvas and respond to events such as finger taps.
2. If you wanted to take a picture from your app or use the bluetooth to send a file, there would be the need to communicate with the native API exposed by the platform. For example, using OS-specific APIs, you could ask for the camera service, wait for a response and then start using it.

The cross-platform approach is different and it **has** to be like so. If you want your app to run on both Android and iOS with the same codebase, you can't directly use OEM widgets and their API because they come from different architectures. They are **NOT** compatible. On the hardware side however, both are based on the ARM architecture (precisely, v7 and v8) and the most recent versions have 64-bit support. Flutter AOT compiles the Dart code into native ARM libraries.

i **ARM** is a family of RISC microprocessors (32 and 64 bit) widely used in embedded systems. It dominates the mobile world thanks to its qualities: low costs, good heat dissipation and a longer battery life thanks to a low power consumption.



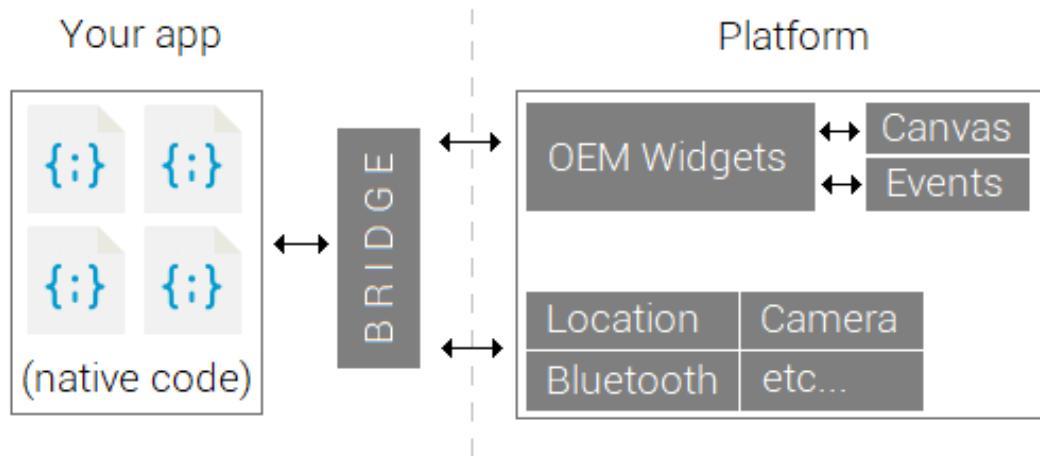
The picture has rectangles on Android and triangles on iOS to indicate that OEM widgets and

APIs have differences in how they are structured, in how they interact with the app and in how you have to use them. For this reason, cross-platform apps cannot directly "talk" to the underlying environment: they must speak a language that everyone can understand.

- ➊ Try to only think about the runtime environment for a moment. If you wrote a Java Android app, it would be compiled to work with the ART ecosystem (**A**ndroid **R**un**T**ime): how could the same binary file work with iOS architecture which is completely different and has no ART?

In the above image, squares represents calls made by Java to interact with the ART which is available only in Android and not on iOS. This compatibility problem is solved by cross-platform frameworks.

ReactJS is a Reactive web framework which tries to solve the above problem by adding a *bridge* in the middle that takes care of the communication with the platform. With this approach, the bridge becomes the real starring of the scene it acts like a translator:

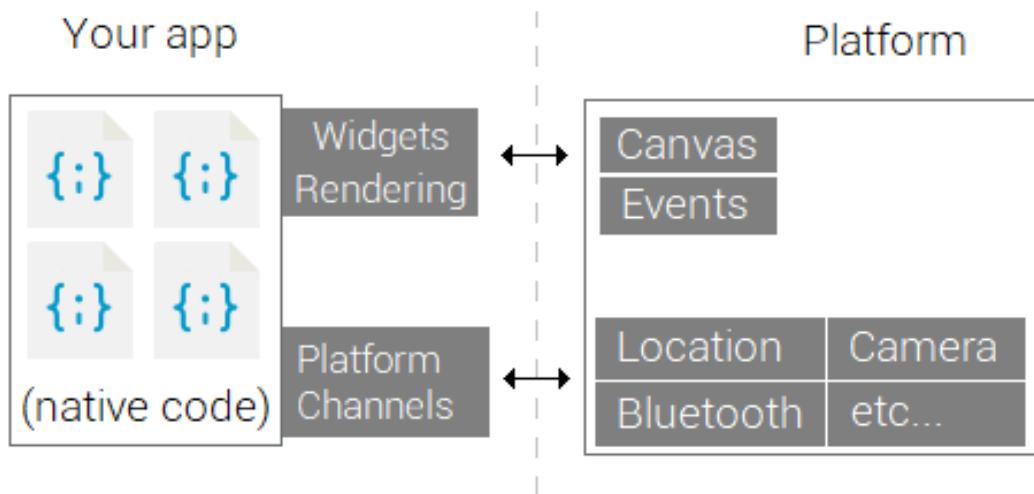


1. The bridge **always** exposes the same interface to the app so that it doesn't care anymore about the OS it's running on;
2. The bridge has an implementation of OEMs and APIs for each platform to allow the app to correctly work in many environments. In this way, you have a native app in the sense that it uses the native tools given by the OS, but there's still an "adapter" in the middle.

As you can see from the picture, the *bridge* is an abstraction layer between the app the OS in which it's hosted. Of course, there has to be a bridge for each supported platform but that's not something you have to deal with because the developers of the framework will take care of creating all of them.

- ❶ If you used a cross-platform framework, you'd just need to care about creating the app with the code and the API exposed by the framework. The implementation of the bridge is already in the internals of the SDK and it's automatically "attached" to the app in the build phase. You don't have to create the bridge.

The *bridge* approach is quite popular, but it could be a potential bottleneck that slows down the execution and thus the performances might drop. If you think about animations, swipes or transitions, widgets are accessed very often and many of them running at the same time could slow down the app. Flutter adopts a completely different strategy:



It uses its own very efficient rendering engine, called *Skia*, to paint the UI so that OEM widgets are not needed anymore. In this way, the app doesn't rely on the instruments the OS exposes to draw the interface and you can freely control each single pixel of the screen.

- Flutter produces **native** ARM code for the machine;

- when launched, the app loads the Flutter library. Any rendering, input or event handling, and so on, is delegated to the compiled Flutter and app code. This is much faster than having a bridge.
- A minimal Flutter app is about 4.4 MB on Android and 10.9 MB on iOS (depending on the architecture, whether it be ARM 32 or 64 bit)⁸

The true power of Flutter lies on the fact that apps are built with their own rendering stuff and they are not constrained to paint the UI following the rules "imposed" by OEM widgets. You're free to control the screen and manipulate every single pixel.

1.3.2 Why Flutter uses Dart

There are many reasons behind the decision made by Google to choose Dart as language for the Flutter framework. At the time of writing this book, the latest **stable** version of Dart is 2.9.2 (Dart 2.10 is on beta, but downloadable anyway). Here's a summary⁹ of what brought them to make this choice.

1. **OOP style.** The vast majority of developers have object-oriented programming skills and thus Dart would be easy to learn as it adopts most of the common OOP patterns. The developer doesn't have to deal with a completely new way of coding; he can reuse what he already knows and integrate it with the specific details of Dart.
2. **Performances.** In order to guarantee high performances and avoid frame dropping during the execution of the app, there's the need of a high performance and predictable language. Dart can guarantee to be very efficient and it provides a powerful memory allocator that handles small, short-lived allocations. This is perfect for Flutter's functional-style flow.
3. **Productivity.** Flutter allows developers to write Android, iOS, web and desktop apps with a single codebase keeping the same performances, aspect and feeling in each platform. A highly productive language like Dart accelerates the coding process and makes the framework more attractive.
4. Both Flutter and Dart are developed by Google which can freely decide what to do with them listening to the community as well. If Dart was developed by another company, Google probably wouldn't have the same freedom of choice in implementing new features and the language couldn't evolve at the desired pace.

Another important aspect is that Dart is **strongly typed**, meaning that the compiler is going

⁸<https://flutter.dev/docs/resources/faq#how-big-is-the-flutter-engine>

⁹<https://flutter.dev/docs/resources/faq#why-did-flutter-choose-to-use-dart>

to be very strict about types; you'll have both less runtime surprises and an easier debugging process. In addition, keep in mind that Dart is a complete swiss-knife because it has built-in support for:

- tree-shaking optimization;
- hot reload feature;
- a package manager with mandatory documentation and the possibility to play with the code using DartPad;
- *DevTools*, a collection of debugging and performance tools;
- code documentation generator tool;
- support for JIT and AOT compilation.

By owning two home-made products, Google can keep the entire projects under control and decide how to integrate them in the best way possible with quick development cycles. Dart evolves together with Flutter and as time goes by: they help each other maximizing productivity and performances.

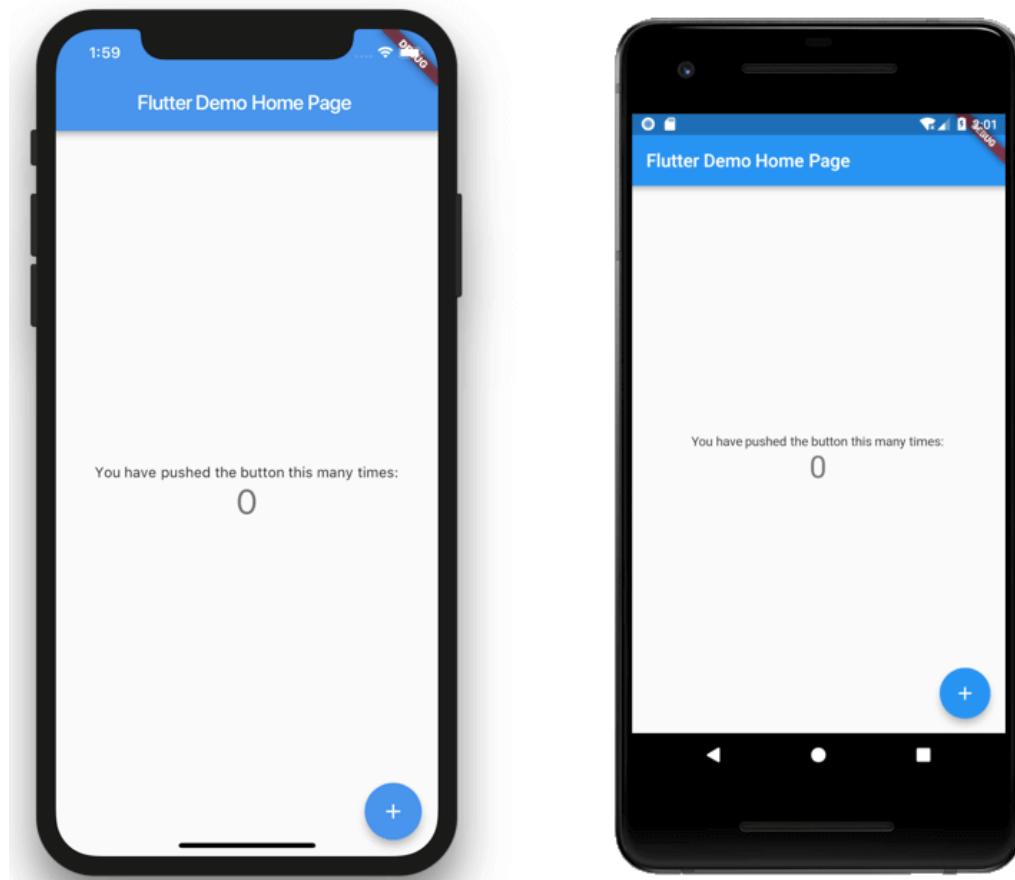
1.3.3 Hello world

When creating Flutter apps for the production world, you should really consider using Android Studio or VSCode and install the respective plugins. They offer a debugger, hints, a friendly UI and powerful optimization tools we will explore in detail.

```
void main() {  
    runApp(MyApp());  
}  
  
class MyApp extends StatelessWidget {  
    const MyApp();  
  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                body: Center(  
                    child: Text("Flutter app!"),  
                ),  
            ),  
    ),
```

```
    );  
}  
}
```

This is a very simple example of a minimal Flutter application. You can notice immediately that there is a `void main() { ... }` function, required by Dart to define the entry point of the program. An UI is a composition of *widgets* that decorate the screen with many objects; you will learn how to properly use them to create efficient and beautiful designs.



This is an example of how a simple Flutter app looks identical in two different platforms. In this book we will focus on Android and iOS mobile apps but **everything** you're going to learn is also valid for web and desktop because it's always Flutter. Once you have the code ready, open the console...

Chapter 1. Welcome

```
$ flutter build appbundle
```

```
$ flutter build ios
```

```
$ flutter build web
```

```
$ flutter build macos
```

```
$ flutter build windows
```

```
$ flutter build linux
```

...and it's just a matter of running different *build* commands to get different native binaries of the same app. For more info on Flutter for web and desktop, see the appendix B at the bottom of the book.

PART I

THE DART PROGRAMMING LANGUAGE

"I'm not a great programmer; I'm just a good programmer with great habits."

MARTIN FOWLER

2 | Variables and data types

2.1 Variables

As in any programming language, variables are one of the basics and Dart comes with support for type inference. A typical example of creation and initialization of a variable is the following:

```
var value = 18;  
var myName = "Alberto"
```

In the example, `value` is an integer while `myName` is a string. Like Java and C#, Dart is able to **infer** the type of the variable by looking at the value you've assigned. In other words, the Dart compiler is smart enough to figure out by itself which is the correct type of the variable.

```
int value = 18;  
String myName = "Alberto"
```

This code is identical to the preceding example with the only difference that here the types have been typed explicitly. There would also be a third valid way to initialize variables, but you should almost never use it.

```
dynamic value = 18;  
dynamic myName = "Alberto"
```

`dynamic` can be used with any type, it's like a "jolly": any value can be assigned to it and the compiler won't complain. The type of a `dynamic` variable is evaluated at runtime and thus, for a proper usage, you'd need to work with checks and type casts. According with the Dart guidelines and our personal experience you should:

1. Prefer initializing variables with `var` as much as you can;
2. When the type is not so easy to guess, initialize it explicitly to increase the readability of the code;

3. Use `Object` or `dynamic` only if it's really needed but it's almost never the case.

Actually, we could say that `dynamic` is not really a type: it's more of a way to turn off static analysis and tell the compiler you know what you're doing. The only case in which you'll deal with it will come in the Flutter part in regard to JSON encoding and decoding.

2.1.1 Initialization

The official Dart guidelines¹ state that you should prefer, in most of the cases, the initialization with `var` rather than writing the type explicitly. Other than making the code shorter (programmers are lazy!) it can increase the readability in various scenarios, such as:

```
// BAD: hard to read due to nested generic types
List<List<Toppings>> pizza = List<List<Toppings>>();
for(List<Toppings> topping in pizza) {
    doSomething(topping);
}

// GOOD: the reader doesn't have to "parse" the code
// It's clearer what's going on
var pizza = List<List<Toppings>>();
for(var topping in pizza) {
    doSomething(topping);
}
```

Those code snippets use generics, classes and other Dart features we will discuss in depth in the next chapters. It's worth pointing out two examples in which you want to explicitly write the type instead of inferring it:

- When you don't want to initialize a variable immediately, use the `late` keyword. It will be explained in detail later in this chapter.

```
// Case 1
late List<String> names;

if (iWantFriends())
    names = friends.getNames();
else
    names = haters.getNames();
```

¹<https://dart.dev/guides/language/effective-dart/design#types>

If you used `var` instead of `List<String>` the inferred type would have been `null` and that's **not** what we want. You'd also lose the type safety and readability.

- The type of the variable is not so obvious at first glance:

```
// Is this a list? I guess so, "People" is plural...
// but actually the function returns a String!
var people = getPeople(true, 100);

// Ok, this is better
String people = getPeople(true, 100);
```

However, there isn't a golden rule to follow because it's up to your discretion. In general `var` is fine, but if you feel that the type can make the code more readable you can definitely write it.

2.1.2 final

A variable declared as `final` can be set only once and if you try to change its content later, you'll get an error. For example, you won't be able to successfully compile this code:

```
final name = "Alberto";
name = "Albert"; // 'name' is final and cannot be changed
```

You can also notice that `final` can automatically infer the type exactly like `var` does. This keyword can be seen as a "restrictive var" as it deduces the type automatically but does not allow changes.

```
// Very popular - Automatic type deduction
final name = "Alberto";
// Generally unnecessary - With type annotation
final String nickName = "Robert";
```

If you want you can also specify the type but it's not required. So far we've only shown examples with strings, but of course both `final` and `var` can be used with complex data types (classes, enums) or methods.

```
final rand = getRandomInteger();

// rand = 0;
// ^ doesn't work because the variable is final
```

The type of `rand` is deduced by the return statement of the method and it cannot be re-assigned in a second moment. The same advice we've given in "2.1.1 Initialization" for `var` can be applied here as well.

-  Later on in the book we will analyze in detail the `const` keyword, which is the "brother" of `final`, and it has very important performance impacts on Flutter.

While coding you can keep this rule in mind: use `final` when you know that, once assigned, the value will **never** change in the future. If you know that the value might change during the time use `var` and think whether it's the case to annotate the type or not. Here's an example in which a `final` variable fits perfectly:

```
void main() {  
    // Assume that the content of the file can't be edited  
    final jsonFile = File('myfile.json').readAsString();  
  
    checkSyntax(jsonFile);  
    saveToDisk(jsonFile, 'file.json');  
}
```

In this example the variable `jsonFile` has a content that doesn't have to be modified, it will always remain the same and so a `final` declaration is good:

- it won't be accidentally edited later;
- the compiler will give an error if you try to modify the value.

If you used `var` the code would have compiled anyway but it wouldn't have been the best choice. If the code was longer and way more complicated, you could accidentally change the content of `jsonFile` because there wouldn't be the "protection" of `final`.

2.2 Data types

Types in Dart can be initialized with "literals"; for example `true` is a boolean literal and "`test`" is a string literal. In chapter 6 we will analyze *generic* data types that are very commonly used for collections such as lists, sets and maps.

2.2.1 Numbers

Dart has two type of numbers:

- **int**. 64-bit at maximum, depending on the platform, integer values. This type ranges from -2^{63} to $2^{63}-1$.
- **double**. 64-bit double-precision floating point numbers that follow the classic IEEE 754 standard definition.

Both **double** and **int** are subclasses of **num** which provides many useful methods such as:

- `parse(string)`,
- `abs()`,
- `ceil()`,
- `toString()...`

You should always use **double** or **int**. We will see, with generic types, a special case in which **num** is needed but in general you can avoid it. Some examples are always a good thing:

```
var a = 1; // int
var b = 1.0; // double

int x = 8;
double y = b + 6;
num z = 10 - y + x;

// 7 is a compile-time constant
const valueA = 7;
// Operations among constant values are constant
const valueB = 2 * valueA;
```

From Dart 2.1 onwards the assignment **double** `a = 5` is legal. In 2.0 and earlier versions you were forced to write `5.0`, which is a *double* literal, because `5` is instead an *integer* literal and the compiler didn't automatically convert the values. Some special notations you might find useful are:

1. The exponential representation of a number, such as `var a = 1.35e2` which is the equivalent of 1.35×10^2 ;

2. The hexadecimal representation of a number, such as `var a = 0xF1A` where `0xF1A` equals to `F1A` in base 16 (3866 in base 10).

2.2.1.1 Good practices

Very likely, during your coding journey, you'll have at some point the need to parse numbers from strings or similar kinds of manipulations. The language comes to the rescue with some really useful methods:

```
String value = "17";  
  
var a = int.parse(value); // String-to-int conversion  
var b = double.parse("0.98"); // String-to-double conversion  
var c = int.parse("13", radix: 6); // Converts from 13 base 6
```

You should rely on these methods instead of writing functions on your own. In the opposite direction, which is the conversion into a string, there is `toString()` with all its variants:

```
String v1 = 100.toString(); // v1 = "100";  
String v2 = 100.123.toString(); // v2 = "100.123";  
String v3 = 100.123.toStringAsFixed(2); // v3 = "100.12";
```

Since we haven't covered functions yet you can come back to this point later or, if you're brave enough, you can continue the reading. When converting numbers from a string, the method `parse()` can fail if the input is malformed such as `"12@4.49"`. You'd better use one of the following solutions (we will cover nullable types later):

```
// 1. If the string is not a number, val is null  
double? val = double.tryParse("12@.3x_"); // null  
double? val = double.tryParse("120.343"); // 120.343  
  
// 2. The onError callback is called when parsing fails  
var a = int.parse("1_6", onError: (value) => 0); // 0  
var a = int.parse("16", onError: (value) => 0); // 16
```

Keep in mind that `parse()` is deprecated: you should prefer `tryParse()`. What's important to keep in mind is that a plain `parse("value")` call is risky because it assumes the string is already well-formed. Handling the potential errors as shown is safer.

2.2.2 Strings

In Dart a string is an ordered sequence of UTF-16 values surrounded by either single or double quotes. A very nice feature of the language is the possibility of combining expressions into strings by using `${expr}` (a shorthand to call the `toString()` method).

```
// No differences between s and t
var s = "Double quoted";
var t = 'Single quoted';

// Interpolate an integer into a string
var age = 25;
var myAge = "I am $age years old";

// Expressions need '{' and '}' preceeded by $
var test = "${25.abs()}"

// This is redundant, don't do it because ${} already calls toString()
var redundant = "${25.toString()}";
```

A string can be either single or multiline. Single line strings are shown above using single or double quotes, and multiline strings are written using triple quotes. They might be useful when you want to nicely format the code to make it more readable.

```
// Very useful for SQL queries, for example
var query = """
    SELECT name, surname, age
    FROM people
    WHERE age >= 18
    ORDER BY name DESC
""";
```

In Dart there isn't a `char` type representing a single character because there are only strings. If you want to access a particular character of a string you have to use the `[]` operator:

```
final name = "Alberto";

print(name[0]); // prints "A"
print(name[2]); // prints "b";
```

The returned value of `name[0]` is a `String` whose length is 1. We encourage you to visit ² the online Dart documentation about strings which is super useful and full of examples.

```
var s = 'I am ' + name + ' and I am ' + (23).toString() + ' y.o.';
```

You can concatenate strings very easily with the `+` operator, in the classic way that most programming languages support. The official Dart guidelines ³ suggest to prefer using interpolation to compose strings, which is shorter and cleaner:

```
var s = 'I am $name. I am ${25} years old';
```

In case of a string longer than a single line, avoid the `+` operator and prefer a simple line break. It's just something recommended by the Dart for styling reasons, there are no performance implications at all. Try to be as consistent as possible with the language guidelines!

```
// Ok
var s = 'I am going to the'
        'second line';

// Still ok but '+' can be omitted
var s = 'I am going to the' +
        'second line';
```

Since strings are immutable, making too many concatenations with the `+` operator might be inefficient. In such cases it'd be better if you used a `StringBuffer` which efficiently concatenates strings. For example:

```
var value = "";

for(var i = 0; i < 900000; ++i) {
    value += "$i ";
}
```

Each time the `+` operator is called, `value` is assigned with a `new` instance which merges the old value and the new one. In other words, this code creates for 900000 times a new `String` object, one for each iteration, and it's not optimal at all. Here's the way to go:

```
var buffer = StringBuffer();
```

²<https://dart.dev/guides/libraries/library-tour#strings-and-regular-expressions>

³<https://dart.dev/guides/language/effective-dart/usage#prefer-using-interpolation-to-compose-strings-and-values>

```
for(var i = 0; i < 900000; ++i)
    buffer.write("$i ");

var value = buffer.toString();
```

This is much better because `StringBuffer` doesn't internally create a new string on each iteration; the string is created only **once** at the moment in which `toString()` is called. When you have to do long loops that manipulate strings, avoid using the `+` operator and prefer a buffer. The same class can also be found in Java and C# for example.

2.2.3 Enumerated types

Also known as "enums", enumerated types are containers for constant values that can be declared with the `enum` keyword. A very straightforward example is the following:

```
enum Fruits { Apple, Pear, Grapes, Banana, Orange }

void main() {
    Fruits liked = Fruits.Apple;
    var disliked = Fruits.Banana;

    print(liked.toString()); // prints 'Fruits.Apple'
    print(disliked.toString()); // prints 'Fruits.Banana'
}
```

Each item of the enum has an associated number, called **index**, which corresponds to the zero-based position of the value in the declaration. You can access this number by using the `index` property.

```
enum Fruits { Apple, Pear, Grapes, Banana, Orange }

void main() {
    var a = Fruits.Apple.index; // 0
    var b = Fruits.Pear.index; // 1
    var c = Fruits.Grapes.index; // 2
}
```

Note that when you need to use an `enum` you always have to fully qualify it. Using the name only doesn't work.

2.2.3.1 Good Practices

When you need a predefined list of values which represents some kind of textual or numeric data, you should prefer an `enum` over a primitive data type. In this way you can increase the readability of the code, the consistency and the compile-time checking. Look at these 2 ways of creating a function (`///` is used to document the code):

```
enum Chess { King, Queen, Rook, Bishop, Knight, Pawn }

/// METHOD 1. Checks if the piece can move in diagonal
bool diagonalMoveC(Chess item) { ... }

/// METHOD 2. Checks if a piece can move in diagonal: [item] can only be:
/// 1. King
/// 2. Queen
/// 3. Rook
/// 4. Bishop
/// 5. Knight
/// 6. Pawn
/// Any other number is not allowed.
bool diagonalMoveS(int item) { ... }
```

This example should convince you that going for the first method is for sure the right choice.

- `diagonalMoveC(Chess item)`. There's a big advantage here: we're guaranteed by the compiler that `item` can only be one of the values in `Chess`. There's no need for any particular check and we can understand immediately what the method wants us to pass.
- `diagonalMoveS(int item)`. There's a big disadvantage here: we can pass any number, not only the ones from 1 to 6. We're going to do extra work in the body because we don't have the help of the compiler, so we need to manually check if `item` contains a valid value.

In the second case, we'd have to make a series of `if` conditions to check whether the value ranges from 1 to 6. Using an `enum`, the compiler does the checks for us (by comparing the types) and we're guaranteed to work with valid values.

2.2.4 Booleans

You can assign to the `bool` type only the literals `true` or `false`, which are both compile-time constants. Here there are a few usage examples:

```

bool test = 5 == 0; // false
bool test2 = !test; // has the opposite value of test

var oops = 0.0 / 0.0; // evaluates to 'Not a Number' (NaN)
bool didIFail = oops.isNaN;

```

2.2.5 Arrays

Probably you're used to create arrays like this: `int[] array = new int[5]`; which is the way that Java and C# offer. In Dart it doesn't really work like that because you can only deal with collections: an "array" in Dart is represented by a `List<T>`.

i `List<T>` is a generic container where `T` can be any type (such as `String` or a class). We will cover generics and collections in detail in chapter 6. Basically, Dart doesn't have "arrays" but only generic containers.

If this is not clear, you can look at this comparison. In both languages there is a generic container for the given type but only Java has "primitive" arrays.

- Java

```

// 1. Array
double[] test = new test[10];
// 2. Generic list
List<double> test = new ArrayList<>();

```

- Dart

```

// 1. Array
// (no equivalent)
// 2. Generic list
List<double> test = new List<double>();

```

In Dart you can work with arrays but they are intended to be instances of `List<T>`. Lists are 0-indexed collections and items can be randomly accessed using the `[]` operator, which will throw an exception if you exceed the bounds.

```

//use var or final
final myList = [-3.1, 5, 3.0, 4.4];
final value = myList[1];

```

A consequence of the usage of a `List<T>` as container is that the instance exposes many useful methods, typical of collections:

- `length`,
 - `add(T value)`,
 - `isEmpty`,
 - `contains(T value)`
- ... and much more.

2.3 Nullable and Non-nullable types

Starting from Dart 2.10, variables will be **non-nullable by default** (`nnbd`) which means they're not allowed to hold the `null` value. This feature has been officially introduced in June 2020 as *tech preview* in the dev channel of the Dart SDK.

```
// Trying to access a variable before it's been assigned will cause a
// compilation error.
int value;
print("$value"); // Illegal, doesn't compile
```

If you don't initialize a variable, it's automatically set to `null` but that's an error because Dart has non-nullability enabled by default. In order to successfully compile you have to initialize the variable as soon as it's declared:

```
// 1.
int value = 0;
print("$value");

// 2.
int value;
value = 0;
print("$value");
```

In the first case the variable is assigned immediately and that's what we recommend to do as much as possible. The second case is still valid because `value` is assigned **before** it's ever accessed. It wouldn't have worked if you had written this:

```
// OK - assignment made before the usage
```

Chapter 2. Variables and data types

```

int value;
value = 0;
print("$value");

// ERROR - usage made before assignment
int value;
print("$value");
value = 0;

```

Non-nullability is very powerful because it adds another level of type safety to the language and, by consequence, lower possibilities for the developer to encounter runtime exceptions related to `null`. For example, you won't have the need to do this:

```

String name = "Alberto";

void main() {
    if (name != null) {
        print(name)
    }
}

```

The compiler guarantees that it can't be `null` and thus no null-checks are required. To sum up, what's important to keep in mind while writing Dart 2.10 code (and above) is:

- By default, variables cannot be `null` and they must **always** be initialized before being used. It would be better if you immediately initialized them, but you could also do it in a second moment before they ever get utilized.
- Don't do null-checks on "standard" non-nullable variables because it's useless.

In Dart you can also declare *nullable* types which doesn't require to be initialized before being accessed and thus they're allowed to be `null`. Nullables are the counterpart of non-nullable types because the usage of `null` is allowed (but the additional type safety degree is lost).

```

int? value;
print("$value"); // Legal, it prints 'null'

```

If you append a question mark at the end of the type, you get a nullable type. For safety, they would require a manual null checks in order to avoid undesired exceptions but, in most of the cases, sticking with the default non-nullability is fine.

```

// Non-nullable version - default behavior

```

```
int value = 0;
print("$value"); // prints '0'

// Nullable version - requires the ? at the end of the type
int? value;
print("$value"); // prints 'null'
```

Nullable types that support the index operator [] need to be called with the ?[] syntax. `null` is returned if the variable is also `null`.

```
String? name = "Alberto";
String? first = name?[0]; // first = 'A';

String? name;
String? first = name?[0]; // first = 'null';
```

We recommend to stick with the defaults, which is the usage of non-nullable types, as they're safer to use. Nullables should be avoided or used only when working with legacy code that depends on `null`. Last but not least, here are the only possible conversions between nullables and non nullables:

- When you're sure that a nullable expression isn't null, you can add a ! at the end to convert it to the non-nullable version.

```
int? nullable = 0;
int notNullable = nullable!;
```

The ! (called "bang operator") converts a nullable value (`int?`) into a non-nullable value (`int`) of the same type. An exception is thrown if the nullable value is actually `null`.

```
int? nullable;
// An exception is thrown
int notNullable = nullable!;
```

- If you need to convert a nullable variable into a non-nullable subtype, use the typecast operator as (more on it later):

```
num? value = 5;
int otherValue = value as int;
```

You wouldn't be able to do `int otherValue = value!` because the bang operator works only when the type is the same. In this example, we have a `num` and an `int` so there's the need for a cast.

Chapter 2. Variables and data types

- Even if it isn't a real conversion, the operator ?? can be used to produce a non-null value from a nullable one.

```
int? nullable = 10;
int nonNullable = nullable ?? 0;
```

If the member on the left (`nullable`) is non-null, return its value; otherwise, evaluate and return the member of the right (0).

Remember that when you're working with nullable values, the member access operator (.) is not available. Instead, you have to use the **null-aware** member access operator (?.):

```
double? pi = 3.14;

final round1 = pi.round();    // No
final round2 = pi?.round();  // Ok
```

2.4 Data type operators

In Dart expressions are built using operators, such as + and - on primitive data types. The language also supports operator overloading for classes as we will cover in chapter 4.

2.4.1 Arithmetic operators

Arithmetic operators are commonly used on `int` and `double` to build expressions. As you already know, the + operator can also be used to concatenate strings.

Symbol	Meaning	Example
+	Add two values	2 + 3 //5
-	Subtract two values	2 - 3 //-1
*	Multiply two values	6 * 3 //18
/	Divide two values	9 / 2 //4.5
~/	Integer division of two values	9 ~/ 2 //4

`%`

Remainder (modulo) of an int division

`5 % 2 //1`

Prefix and postfix increment or decrement work as you're used to see in many languages.

```
int a = 10;
++a; // a = 11
a++; // a = 12
```

```
int b = 5;
--b; // b = 4;
b--; // b = 3;
```

```
int c = 6;
c += 6 // c = 12
```

As a reminder, both postfix and prefix increment/decrement have the same result but they work in a **different** way. In particular:

- in the prefix version (`++x`) the value is first incremented and then "returned";
- in the postfix version (`x++`) the value is first "returned" and then incremented

2.4.2 Relational operators

Equality and relational operators are used in boolean expression, generally inside `if` statements or as a stop condition of a `while` loop.

Symbol	Meaning	Example
<code>==</code>	Equality test	<code>2 == 6</code>
<code>!=</code>	Inequality test	<code>2 != 6</code>
<code>></code>	Greater than	<code>2 > 6</code>

Chapter 2. Variables and data types

<	Smaller than	<code>2 < 6</code>
<code>>=</code>	Greater or equal to	<code>2 >= 6</code>
<code><=</code>	Smaller or equal to	<code>2 <= 6</code>

Testing the equality of two objects *a* and *b* always happens with the `==` operator because, unlike Java or C#, there is no `equals()` method. In chapter 6 we will analyze in detail how classes can be properly compared by overriding the equality operator. In general here's how the `==` works:

1. If *a* or *b* is null, return `true` if both are null or `false` if only one is null. Otherwise...
2. ... return the result of `==` according with the logic you've defined in the method override.

Of course, `==` works only with objects of the same type.

2.4.3 Type test operators

They are used to check the type of an object at runtime.

Symbol	Meaning	Example
 <code>as</code>	Cast a type to another	<code>obj as String</code>
<code>is</code>	True if the object has a certain type	<code>obj is double</code>
<code>is!</code>	False if the object has a certain type	<code>obj is! int</code>

Let's say you've defined a new type like `class Fruit {}`. You can cast an object to `Fruit` using the `as` operator like this:

```
(grapes as Fruit).color = "Green";
```

The code compiles but it's unsafe: if `grapes` was `null` or if it wasn't a `Fruit`, you would get an exception. It's always a good practice checking whether the cast is doable before doing it:

```
if (grapes is Fruit) {  
    (grapes as Fruit).color = "Green";  
}
```

Now you're guaranteed the cast will happen only if it's possible and no runtime exceptions can happen. Actually, the compiler is smart enough to understand that you're doing a type check with `is` and it can do a *smart cast*.

```
if (grapes is Fruit) {  
    grapes.color = "Green";  
}
```

You can avoid writing the explicit cast (`grapes as Fruit`) because, inside the scope of the condition, the variable `grapes` is automatically casted to the `Fruit` type.

2.4.4 Logical operators

When you have to create complex conditional expressions you can use the logical operators:

Symbol	Meaning
<code>!expr</code>	Toggles true to false and vice versa
<code>expr1 && expr2</code>	Logical AND (true if both sides are true)
<code>expr1 expr2</code>	Logical OR (true if at least one is true)

2.4.5 Bitwise and shift operators

You'll never use these operators unless you're doing some low level data manipulation but in Flutter this never happens.

Chapter 2. Variables and data types

Symbol	Meaning
a & b	Bitwise AND
a b	Bitwise OR
a ^ b	Bitwise XOR
~ a	Bitwise complement
a >> b	Right shift
a << b	Left shift

3 | Control flow and functions

3.1 If statement

This is probably the most famous statement of any programming language and in Dart it works exactly as you would expect. The `else` is optional and it can be omitted when not needed. You can avoid using brackets in case of one-liner statements.

```
void main() {
    final random = 13;

    if (random % 2 == 0)
        print("Got an even number");
    else
        print("Got an odd number");
}
```

Conditions must be boolean values. In C++ for example you can write `if (0) {...}` where zero is evaluated to `false` but in Dart it doesn't compile; you have to write `if (false) {...}`.

3.1.1 Conditional expressions

In Dart there are two shorthands for conditional expressions that can replace the if-else statement:

- `valueA ?? valueB`. If `valueA` is non-null, `valueA` is returned; otherwise `valueB` is evaluated and then returned. If the definition is too verbose, you can understand this syntax by looking at the following example.

```
String? status; // This is null
```

```
// isAlive is a String declared somewhere before
if (status != null)
    isAlive = status;
else
    isAlive = "RIP";
```

Basically we want to know whether `status` is null or not and then decide the proper value to assign. The same logic can be expressed in another, more concise way:

```
String? status; // This is null
String isAlive = status ?? "RIP";
```

In the example `isAlive` doesn't need to be nullable as it's guaranteed to be initialized with a string. The `??` operator automatically checks if `status` is null and decides what to do:

- `status` is **not** null: return `status`;
- `status` is null: return the provided "default value" at the right of `??`

It's a very helpful syntax because it guarantees that a variable is properly initialized avoiding unwanted operations with `null`.

- `condition ? A : B;`. If `condition` is true `A` is returned, otherwise you get `B`. It's a pretty common pattern among modern languages so you might already be familiar with it.

```
String status;

if (correctAns >= 18)
    status = "Test passed!";
else
    status = "You didn't study enough..."
```

If it looks a bit too verbose, you can rewrite the logic in a more concise way:

```
String status = (correctAns >= 18) ?
    "Test passed!" :
    "You didn't study enough...";
```

We could call this the "*shorter if*" syntax in which you replace the `if` with the question mark (?) and the `else` with the colon (:). You can omit parenthesis.

3.1.2 Good practices

Simple boolean expressions are easy to read but complicated ones might require documentation and might also not fit well inside a single `if` statement like the following:

```
if ( (A && B || C && !A) || (!(A && C) || B) ) { ... }
```

You might get a headache while trying to figure out what's going on and there are also no comments at all. In such cases, you probably want to make the code more readable by splitting the conditions:

```
final usefulTestName1 = A && B || C && !A;  
final usefulTestName2 = !(A && C)  
  
if (usefulTestName1 || usefulTestName2 || B) { ... }
```

For sure it's more understandable and another programmer, or yourself in the future, will be very grateful. We also recommend to not underestimate the usefulness of variable names.

i The point is that you have to keep expressions short and easy to read. Break down long conditions into smaller pieces and give the variables good names to better understand what you want to check.

We also recommend the usage of the *short if* syntax only when there's one condition or at maximum two short ones. The longer the line is the harder it is to understand.

3.2 switch statement

When you have a series of cases to take into account, instead of using a long chain of if-elses you should go for the `switch` statement. It can compare many types:

1. compile-time constants
2. enums
3. integers
4. strings
5. classes

Classes must not override `==` if they want to be compared with this statement. At the bottom there's a `default` label used as fallback if none of the previous cases matches the item being compared.

Chapter 3. Control flow and functions

```
enum Status { Ready, Paused, Terminated }

void main() {
    final status = Status.Paused;

    switch (status) {
        case Status.Ready:
            run();
            break;
        case Status.Paused
            pause();
            break;
        case Status.Terminated
            stop();
            break;
        default
            unknown();
    }
}
```

If the body of the `case` is **NOT** empty you must put a `break` otherwise your code won't compile. When you just want a fall-through to avoid code-replication, leave the body empty. Here's a few examples:

- This code is not going to compile because the first `case` has a body, containing `start()`, but there isn't a `break`.

```
switch (status) {
    case Status.Ready:
        start();
        //missing "break;" here
    case Status.Paused
        pause();
        break;
}
```

- This code instead is fine because the `case` doesn't have a body; the method `pause()` is going to be called when `status` is ready or paused.

```
switch (status) {
  case Status.Ready:
  case Status.Paused
    pause();
    break;
}
}
```

The above code is equivalent to...

```
switch (status) {
  case Status.Ready:
    pause();
    break;
  case Status.Paused
    pause();
    break;
}
}
```

... but you should avoid code duplication which is always bad in terms of code maintenance. When you have two or more cases that must execute the same action, use the fall-through approach.

3.3 for and while loops

The iteration with a `for` loop is the most traditional one and doesn't need many explanations. You can omit brackets in case of one-liner statements. The index cannot be nullable (using `int? i = 0` doesn't work).

```
for(var i = 0; i <= 10; ++i)
  print("Number $i");
```

As you'd expect, the output prints a series of "Number (*i*)" in the console. An equivalent version can be written with a classic `while` loop:

```
var i = 0;

while (i <= 10) {
  print("Number $i");
```

```
    ++i;
}
```

The language also has the `do while` loop that always executes at least **one** iteration because the condition is evaluated only at the end of the cycle.

```
var i = 0;

do {
    print("Number $i");
    ++i;
} while (i <= 10)
```

The difference is that the `while` evaluates the condition at the beginning so the loop could never start. The `do while` instead runs at least once because the condition check is placed at the end. If you wanted to alter the flow of the loop you could use:

- `break`. It immediately stops the loop in which it is called. In case of nested loops, only the one whose scope contains `break` is stopped. For example:

```
for (var i = 0; i <= 3; ++i) { // 1.
    for(var j = 0; j <= 5; ++j) { // 2.
        if (j == 5)
            break;
    }
}
```

In this case only loop *2* is terminated when *j* is 5 but loop *1* executes normally until *i* reaches 3. In practical terms, we can say `break` stops only 1 loop.

- `continue`. It skips to the next iteration and, like we've seen before, in case of nested loops it does the jump only for the loop containing it, not the others.

3.3.1 for-in loop

There are some cases in which you want to completely traverse a string or a container and you don't care about the index. Look at this very easy example:

```
final List<String> friendsList = ["A", "B", "C", "D", "E"];

for(var i = 0; i < friendsList.length; ++i)
    print(friendsList[i]);
```

That's perfectly fine but you're using `i` just to retrieve the element at the i -th position and nothing more. There are no calculations based on the index as it's just used to traverse the list. In such cases you should do the following:

```
List<String> friendsList = ["A", "B", "C", "D", "E"];  
  
for(final friend in friendsList)  
    print(friend);
```

This version is less verbose and clearer. You're still traversing the entire list but now, instead of the index `i`, you have declared `final friend` that represents an item at each iteration.

3.4 Assertions

While writing the code you can use assertions to throw an exception¹ if the given condition evaluates to false. For example:

```
// the method returns a json-encoded string  
final json = getJSON();  
  
// if length > 0 is false --> runtime exception  
assert(json.length > 0, "String cannot be empty");  
  
// other actions  
doParse(json);
```

The first parameter of `assert` must be an expression returning a boolean value. The second parameter is an optional string you can use to tell what's gone wrong; it will appear in the IDE error message window if the condition evaluates to `false`.

- ❶ In release mode, every `assert` is **ignored** by the compiler and you're guaranteed that they won't interfere with the execution flow. Assertions work only in debug mode.

When you hit **Run** on Android Studio or VS Code your Flutter app is compiled in debug mode so assertions are enabled.

¹Exceptions will be discussed in detail in chapter 5

3.5 Good practices

Sometimes you have to implement a complicated algorithm and you don't want to make it even more complex by writing code hard to understand. Here's what we recommend.

- Try to always use brackets, even if they can be omitted, so that you can avoid unexpected behaviors. Imagine you had written this code...

```
// Version 1
if ("A" == "A")
    if ("B" == "B")
        print("Oh well!");
else
    print("Oops...");
```

... but in reality you wanted to write this, with a better indentation:

```
// Version 2
if ("A" == "A")
    if ("B" == "B")
        print("Oh well!");
else
    print("Oops...");
```

There's a high possibility that, at first glance, in version 1 you associated the `else` to the first `if` but it'd be wrong! While it may seem obvious, in a complex architecture with thousands of lines you might misread and get tricked.

- When you have to traverse an entire list and you **don't** care about the position in which you are during the iteration, use a `for-in` loop. As we've already said, it's less verbose and so more understandable.

Use assertions, in particular when you create Flutter apps, to control the behavior of your software. Don't remove them when you're ready to deploy the code to the production world because they will be automatically discarded.

3.6 The basics of functions

Functions in Dart have the same structure you're used to see in the most popular programming languages and so you'll find this example self-explanatory. You can mark a parameter with `final` but in practice it does nothing.

```
bool checkEven(int value) {  
    return value % 2 == 0  
}
```

When the body of the function contains only one line, you can omit the braces and the `return` statement in favor of the "arrow syntax". It works with **expressions** and not with statements.

```
// Arrow syntax  
bool checkEven(int value) => value % 2 == 0;  
  
// Arrow syntax with method calls  
bool checkEven(int value) => someOtherFunction(value);  
  
// Does NOT work  
bool checkEven(int value) => if (value % 2 == 0) ... ;
```

The second example is a conditional statement so it doesn't work with the arrow syntax. The first example instead is still a condition but it's written as **expression** and so it works fine.

```
// 1. This function does not return a value  
void test() {}  
  
// 2. No return type so this function returns dynamic. Don't do this.  
test() {}
```

When you don't need a function to return a value, simply make it `void` like you'd do in Java for example. If you omitted the return type like in (2.), the compiler would automatically append `return dynamic` at the end of the body.

```
void test() => print("Alberto");
```

Interestingly, if you have a void function with an one-liner body, you can use the arrow syntax. The function doesn't return anything because of the `void` but you're allowed to do it anyway.

- ⓘ Try to **always** specify the return type or use `void`. Avoid ambiguity; you could avoid the return type for laziness (you just don't want to write `void`) but someone else could think you're returning `dynamic` on purpose.

3.6.1 The Function type

Dart is truly an OOP language because even functions are objects and the type is called... **Function!** A return type is required while the parameters list is optional:

```
// Declare a function
bool checkEven(int value) => value % 2 == 0;

void main() {
    // Assign a function to a variable
    bool Function(int) checker = checkEven;

    // Use the variable that represents the function
    print(checker(8)); // true
}
```

It's nothing new: you're just writing a type (`bool Function(int)`), its name (`checker`) and then you're assigning it a value (`checkEven`). You may find this declaration a bit weird because it's made up of many keywords but it's a simple assignment. This is a comparison to clarify the idea:

- 28: It's an integer and its type is `int`.
- "Pizza": It's a string and its type is `String`.
- `bool checkEven(int value) => ...`: It's a function and its type is `bool Function(int)`.

This particular syntax is very expressive; you have to declare the return type and the **exact** order of the type(s) it takes. In other words, signatures must match. If you think it's too verbose, you can use the typical automatic type deduction you're getting used to see:

```
bool checkEven(int value) => value % 2 == 0;

void main() {
    final checker1 = checkEven;
    var checker2 = checkEven;

    print(checker1(8)); // true
    print(checker2(8)); // true
}
```

Both `var` and `final` will be evaluated to `bool Function(int)`. There's still something to say

about this type but you'll have to wait until the next chapter where we'll talk about classes and the special `call()` method.

- When you declare a variable you can only write `Function(int) name` without the return type. However, automatic type deduction is generally the best choice because it reduces a lot the verbosity.

It might not seem very useful and we'd agree with you because there is no usage context, at the moment. When you'll arrive at part 2 of the book you'll see that the `Function` type is super handy in Flutter because it's used to create "function callbacks".

3.7 Anonymous functions

So far you've only seen *named functions* such as `bool checkEven(int value)` where `checkEven` is the name. Dart gives you the possibility to create nameless functions called anonymous functions.

```
void main() {  
    bool Function(int) isEven = (int value) => value % 2 == 0;  
  
    print(isEven(19)); //false  
}
```

This syntax allows you to create functions "*on the fly*" that are immediately assigned to a variable. If you want an anonymous function with no parameters, just leave the parenthesis blank `()`. Of course you can use `final` and `var` to automatically deduce the type.

- Single line.** You can use the arrow syntax when you have one-liner statements. This example declares a function with no parameters that returns a double.

```
final anon = () => 5.8 + 12;
```

- Multiple lines.** Use brackets and `return` when you have to implement a logic that's longer than one line.

```
final anon = (String nickname) {  
    var myName = "Alberto";  
    myName += nickname;
```

```
    return myName;  
};
```

We recommend to always write down the type of the parameter even if it's not required by the compiler. You can decide whether the type has to appear or not. Using `final` and `var` is allowed but it doesn't make much sense.

```
String Function(String) printName = (String n) => n.toUpperCase();  
String Function(String) printName = (final n) => n.toUpperCase();  
String Function(String) printName = (var n) => n.toUpperCase();  
String Function(String) printName = (n) => n.toUpperCase();
```

Any variant compiles with success but none of them is the best option, the decision is up to your discretion. Before moving on, we're going to show a simple scenario you'll encounter many times in Flutter.

```
// 1.  
void test(void Function(int) action) {  
    // 2.  
    final list = [1, 2, 3, 4, 5];  
  
    // 3.  
    for(final item in list)  
        action(item);  
}  
  
void main() {  
    // 4.  
    test(  
        // 5.  
        (int value) { print("Number $value"); }  
    );  
}
```

The `action` parameter commonly known as *callback* because it executes an action given from the outside.

1. This function doesn't return a value because of the `void`. The parameter, called `action`, accepts a `void` function with a single integer value.
2. It's a simple list of integer values

3. We iterate through the entire list and, for each item, we call the function.
4. `test(...);` is how you normally call a function
5. This is an anonymous function returning nothing (`void`) and asking for a single integer parameter.

The flexibility of callbacks lies on the fact that you can reuse the same function `test()` with different implementations. The caller doesn't care about the body of the anonymous function, it just invokes it as long as the signature matches.

```
// The same method (test) outputs different values
// because anonymous functions have different bodies
test( int value ) => print("$value" );
test( int value ) => print("${value + 2}") ;
```

You will often encounter the `forEach()` method on collections, which accepts a callback to be executed while elements are traversed. Again, the same function (`forEach()`) is reused multiple times regardless the implementation (thanks to callbacks).

```
void main() {
  // Declare the list
  final list = [1, 2, 3, 4, 5];
  // Iterate
  list.forEach((int x) => print("Number $x"));
}
```

This is an even shorter way that doesn't use a `for-in` loop. You pass an anonymous function to the method and it executes the given action for every item. Pay attention because the documentation suggests to avoid using anonymous functions in `forEach()` calls.

- i** A very handy feature you'll see very often is the possibility to put an underscore when one or more parameters of a function aren't needed. For example, in Flutter the `BuildContext` object is often given as a callback param but it's not always essential.

```
builder: (BuildContext context) {
  return Text("Hello");
}
```

Since the variable `context` isn't used, but it must be there anyway to match the method signature, you can use an underscore to "hide" it:

```
builder: (_){  
    return Text("Hello");  
}
```

It's less code for you to write and the reader focuses more on what's really important.
In case of multiple values that you don't use, just chain a series of underscores.

```
builder: (_, value, __) {  
    return Text("\$value");  
}
```

3.8 Optional parameters

In Dart function parameters can be **optional** in the sense that if you don't provide them, the compiler will assign `null` or a default value you've specified.

3.8.1 Named parameters

In the simplest case, a function can have optional parameters whose names must be explicitly written in order to be assigned. Pay attention to null-safety in case you don't plan to give the variables a default value.

Declaration

```
void test({int? a, int? b}) {  
    print("\$a");  
    print("\$b");  
}
```

Calling

```
void main() {  
    // Prints '2' and '-6'  
    test(a: 2, b: -6);  
}
```

When calling a function with optional named parameters, the order **doesn't** matter but the names of the variables names must be explicit. For example, you could have called `test(b: -6, a: 2)`; and it would have worked anyway. When a parameter is missing, the default value is given:

Declaration

```
void test({int? a, int? b}) {
  print("$a");
  print("$b");
}
```

Calling

```
void main() {
  // Prints '2' and 'null'
  test(a: 2);
}
```

Calling `test(a: 2);` initializes only `a` because `b`, which is omitted, is set to `null` by the compiler. `null` is the default value of nullable types. You can manually give a default value to an optional named parameter just with a simple assignment:

Declaration

```
void test({int? a, int b = 0}) {
  print("$a");
  print("$b");
}
```

Calling

```
void main() {
  // Prints '2' and '0'
  test(a: 2);
}
```

Note that `b` doesn't need to be nullable anymore thanks to the default value. In Dart 2.9 (and lower) nnbd was not enabled so you were able to successfully compile this code, which initializes both `a` and `b` to `null`:

Declaration

```
// Dart 2.9 and lower
void test({int a, int b}) {
  print("$a");
  print("$b");
}
```

Calling

```
// Dart 2.9 and lower
void main() {
  // Prints 'null' and 'null'
  test();
}
```

The `required` modifier, introduced in Dart 2.10 with nnbd, forces an optional parameter to be set. You won't be able to compile if a required parameter is not used when calling the function.

```
void test({int a = 0, required int b}) {
  print("$a");
  print("$b");
```

```

}

void main() {
  test(a: 5, b: 3); // Ok
  test(a: 5);      // Compilation error, 'b' is required
}

```

Even if you had written `required int? b` you'd have to assign `b` anyway because it's required. Version 2.9 of Dart and lower didn't have this keyword: you had to use instead an *annotation* which just produced a warning (and not a compilation error) by default.

```
// Dart 2.9 and lower
void test({int a = 0, @required int b}) {...}
```

In Flutter, for a better readability, some methods only use named optional params together with `required` to force the explicit name in the code. You can mix optional named parameters with "classic" ones:

Declaration

```
void test(int a, {int b = 0}) {
  print("$a");
  print("$b");
}
```

Calling

```
void main() {
  // Prints '2' and '3'
  test(2, b: 3);
}
```

Optional parameters must stay at the end of the list.

```
// it compiles
void test(int a, {int? b}) {}

// it doesn't compile
void test({int? a}, int b) {}
```

3.8.2 Positional parameters

You can also use optional parameters without being forced to write down the name. Optional positional parameters follow the same rules we've just seen for named params but instead of using curly braces (`{ }`) they're declared with square brackets (`[]`).

Declaration

```
void test([int? a, int? b]) {
    print("$a");
    print("$b");
}
```

Calling

```
void main() {
    // Prints '2' and '-6'
    test(2, -6);
}
```

All the examples we've made for named parameters also apply here. They really have the same usage but the practical difference is that, in this case, the name of the parameter(s) doesn't have to be written in the function call.

3.9 Nested functions

The language allows you to declare functions inside other functions visible only within the scope in which they're declared. In other words, nested functions can be called only inside the function containing them; if you try from the outside, you'll get a compilation error.

- From a practical side, the **scope** is the "area" surrounded by two brackets { }

This example shows how you can nest two functions, where `testInner()` is called "outer function" and `randomValue()` is called "inner function".

```
void testInner(int value) {
    // Nested function
    int randomValue() => Random().nextInt(10);

    // Using the nested function
    final number = value + randomValue();
    print("$number");
}
```

As we've just seen, functions are types in Dart so a "nested function" is nothing more than a `Function` type assignment. Given this declaration, we're able to successfully compile the following:

```
void main() {
```

```
// testInner internally calls randomValue
testInner(20);
}
```

An error is going to occur if we try to directly call `randomValue` from a place that's not inside the scope of its **outer** function.

```
void main() {
    // Compilation error
    var value = randomValue();
}
```

3.10 Good practices

Following the official Dart guidelines ² we strongly encourage you to follow these suggestions in order to guarantee consistency with what the community recognizes as a good practice.

- Older versions of Dart allow the specification of a default value using a colon (:); don't do it, prefer using =. In both cases, the code compiles successfully.

```
// Good
void test([int a = 0]) {}

// Bad
void test([int a : 0]) {}
```

The colon-initialization *might* be removed in the future.

- When no default values are given, the compiler already assigns `null` to the variable so you don't have to explicitly write it.

```
// Good
void test({int? a}) {}

// Bad
void test({int? a = null}) {}
```

In general, you should **never** initialize nullables with `null` because the compiler already does that by default.

²<https://dart.dev/guides/language/effective-dart/usage#functions>

Dart gives the possibility to write only the name of a function, with no parenthesis, and automatically pass proper parameters to it. It's a sort of "method reference". We are going to convince you with this example:

```
void showNumber(int value) {
    print("$value");
}

void main() {
    // List of values
    final numbers = [2, 4, 6, 8, 10];

    // Good
    numbers.forEach(showNumber);

    // Bad
    numbers.forEach((int val) { showNumber(val); });
}
```



The bad example compiles but you can avoid that syntax in favor of a shorter one.

- The `forEach()` method asks for a function with a single integer parameter and no return type (`void`).
- The `showNumber()` function accepts an integer as parameter and returns nothing (`void`).

The signatures match! If you pass the function name directly inside the method, the compiler automatically initializes the parameters. This **tear-off** is very useful and you might already have seen it somewhere else under the name of "method reference" (Java).

3.11 Using typedefs

The `typedef` keyword simply gives another name to a function type so that it can be easily reused. Imagine you had to write a callback function for many methods:

```
void printIntegers(void Function(String msg) logger) {
    logger("Done.");
}

void printDoubles(void Function(String msg) logger) {
```

```
    logger("Done. ");
}
```

Alternatively, rather than repeating the declaration every time, which leads to code duplication, you can give it an alias using the `typedef` keyword.

```
typedef LoggerFunction = void Function(String msg);

void printIntegers(LoggerFunction logger) {
    logger("Done int.");
}

void printDoubles(LoggerFunction logger) {
    logger("Done double.");
}
```

You are going to encounter this technique very often, especially in Flutter, in callbacks for classes or methods. For instance, `VoidCallback`³ is just a function alias for a void function taking no parameters.

```
typedef VoidCallback = void Function();
```

In a future version of Dart, probably later than 2.10, `typedef` will also be used to define new type names. At the moment it's not possible, but in the future there will be the possibility to compile the following code:

```
typedef listMap = List<Map<int,double>>;
```

The reason is that generic types can become very verbose and so an alias could improve the readability. Currently, `typedef` only works with functions.

³<https://api.flutter.dev/flutter/dart-ui/VoidCallback.html>

4 | Classes

Up to now you've seen us saying many times the claim that Dart is an OOP language and now we're finally going to prove it. There are a lot of similarities with the most popular programming language so you probably are already familiar with the concepts.

```
class Person {  
    // Instance variables  
    String name;  
    String surname;  
  
    // Constructor  
    Person(String name, String surname) {  
        this.name = name;  
        this.surname = surname;  
    }  
}
```

This syntax is almost identical to Java, C# or C++ and that's very good: if you're going to learn the language, there's nothing you've never seen before. Some keywords might be different but the essence is always the same.

- ❶ Every object is an instance of a class. Dart classes, even if it's not explicitly written in the declaration, descend from `Object` and in the next chapter you will see the benefits. In Delphi and C# as well, any class implicitly derives from `Object`.

In any class you have methods (it's the OOP way to call *functions*) which can be public or private. The keyword `this` refers to the current instance of the class. Dart has **NO** method overload so you cannot have more than a function with the same name. For this reason, you'll

Chapter 4. Classes

see how named constructors come to the help.

```
class Example {
    // Doesn't compile; you have to use different names
    void test(int a) {}
    void test(double x, double y) {}
}
```

You might be able to write this in other programming languages because methods have the same name but different signature. In Dart it's not possible, every function name (in the same class) must be unique. Before going into the details of classes, look at the *cascade notation*.

```
class Test {
    String val1 = "One";
    String val2 = "Two";

    int randomNumber() {
        print("Random!");
        return Random().nextInt(10);
    }
}
```

Given this class, you have two ways to give a value to the variables:

```
// first way, the "classic" one
test.val1 = "one";
test.val2 = "two";

// second way, using the cascade operator
test..val1 = "one"
    ..val2 = "two";
```

It's just a shorthand version you can use when there are multiple values of the same objects that has to be initialized. You can do the same even with methods but the returned value, if any, will be **ignored**. For this reason, the cascade notation is useful when calling a series of **void** methods on the same object.

```
Test()..randomNumber()
    ..randomNumber()
    ..randomNumber();
```

Here the integer returned by `randomNumber()` is discarded but the body is executed. If you

run the snippet, you'll get `Random!` printed three times in the console. In case of nullable values...

```
MyClass? test = MyClass();  
  
test?..one()  
  ..two()  
  ..three();
```

... the cascade notation has to start with `?..` in order to be null-checked before dereferencing. In Dart there **cannot** be nested classes.

4.1 Libraries and visibility

In the Dart world, when you talk about a "library" you're referring to the code inside a file with the `.dart` extension. If you want to use that particular library, you have to reference its content with the `import` keyword.

-  If you aren't sure you've understood the above statement, here's an example. Let's say there's the need to handle fractions in the form *numerator / denominator*: you are going to create a file named `fraction.dart`.

```
// == Contents of fraction.dart ==  
class MyFraction {  
  final int numerator;  
  final int denominator;  
  
  //... other code  
}
```

Congratulations, you have just created the `fraction` library! It can be used by any other `.dart` file that references it via `import`.

```
import 'package:fraction.dart';  
  
void main() {  
  // You could have written 'new Fraction(1, 2)' but  
  // starting from Dart 2.0 'new' is optional  
  final frac = Fraction(1, 2);
```

Chapter 4. Classes

```
}
```

You can also use the `library` keyword to "name" your library as you prefer. This keyword really just names the library as you like, nothing more, and it's not required.

```
library super_duper_fraction;

class MyFraction {
    final int numerator;
    final int denominator;
    const MyFraction(this.numerator, this.denominator);
}
```

The `import` directive accepts a string which must contain a particular scheme. For built-in libraries you have to use the `dart:` prefix followed by the library name:

```
import 'dart:math';
import 'dart:io';
import 'dart:html';
```

Everything else that doesn't belong to the Dart SDK, such as a custom library created by you or another developer in the community, must be prefixed by `package`.

```
import 'package:fraction.dart';
import 'package:path/to/file/library.dart';
```

It could happen that two different libraries have implemented a class with the same name; the only possible technique to avoid ambiguity is called "library aliases". It's a way to reference the contents of a library under a different name.

```
// Contains a class called 'MyClass'
import 'package:libraryOne.dart';
// Also contains a class called 'MyClass'
import 'package:libraryTwo.dart' as second;

void main() {
    // Uses MyClass from libraryOne
    var one = MyClass();

    //Uses MyClass from libraryTwo.
```

```
    var two = second.MyClass();
}
```

You can selectively import or exclude types using the `show` and `hide` keywords:

- `import 'package:libraryOne.dart' show MyClass;` Imports only `MyClass` and discards all the rest.
- `import 'package:libraryTwo.dart' hide MyClass;` Imports everything except `MyClass`.

At the moment, you have the basics of libraries (simple creation and usage). In chapter 23 you'll learn how to create a *package* (a collection of libraries and tools) and how to publish it at <https://pub.dev>.

4.1.1 Encapsulation

You may be used to hide implementation details in your classes using *public*, *protected* and *private* keywords but there's no equivalent in Dart. Every member is public by default unless you append an underscore (`_`) which makes it private to its library. If you had this file...

```
// === File: test.dart ===
class Test {
    String nickname = "";
    String _realName = "";
}
```

... you could later import the library anywhere:

```
// === File: main.dart ===
import 'package:test.dart';

void main() {
    final obj = Test();

    // OK
    var name = obj.nickname;
    // ERROR, doesn't compile
    var real = obj._realName;
}
```

The variable `nickname` is public and everyone can see it but `_realName` can be seen **ONLY** inside `test.dart`. In other words, if you put the underscore in front of the name of a variable, it

Chapter 4. Classes

is visible only within that file.

```
// === File: main.dart ===
class Test {
    String nickname = "";
    String _realName = "";
}

void main() {
    final obj = Test();

    // Ok
    var name = obj.name;
    // Ok, it works because 'Test' is in the same file
    var real = obj._realName;
}
```

We've moved everything in the same file: now both `Test` and `main()` belong to the **same** library and so `_realName` is not private anymore.

- ➊ The same rules on package private members also apply to classes and functions. For example, `void something()` is visible from the outside while `void _something()` is private to its library.

```
// Inside a file called users.dart
class Users { }

class _UsersHelper { }
```

In this case, `Users` is visible while `_UsersHelper` is package-private (exactly as it happens with variables and methods).

In Dart everything is "public" by default; if you append an underscore at the front, it becomes "private". There is no way to define "protected" members or variables (where `protected` is a typical OOP keyword that makes a member or variable accessible only by subclasses of a certain type.).

4.1.2 Good practices

We strongly recommend to **NOT** put everything in a single file (or a few ones) for the following reasons:

- Dealing with thousands of lines of code containing literally everything is not good at all. Maintenance is going to be hard and you're on the good way to become a professional "*spaghetti code*" writer!
- If you placed everything in the same file, you'd expose private details of the class. We've seen that private members exist only if classes are put in separated libraries (files).

Try to have one file per class or at maximum a few classes that are closely related (they should have the same purpose). When you write a Dart program, in general you have this folder structure:

```
root
| -- lib
|   | -- main.dart
|   | -- routes
|   | -- other_folders
| -- tests
| -- tools
```

You'll learn throughout the chapters how to create a robust folder hierarchy. The minimal Dart/Flutter project is made up of a `lib/` folder and a `main.dart` entry point file.

4.2 Constructors

The constructor is a special function with the same name of the class and doesn't carry a return type. To invoke it, the syntax is the most common one you can imagine:

```
final myObject = new MyClass();
```

Starting from Dart 2, the keyword `new` can be omitted. It's something you're always going to do, especially while writing Flutter apps.

```
final myObject = MyClass();
```

As example, in this chapter we are creating a library to work with fractions and rational numbers. To get started, there's the need to create a file called `fraction.dart` with the following contents:

```
class Fraction {
    int? _numerator;
    int? _denominator;

    Fraction(int numerator, int denominator) {
        _numerator = numerator;
        _denominator = denominator;
    }
}
```

In this case variables must be nullables because they are initialized, by default, to `null`. The body of the constructor is called **after** the variables initialization. If you want to get rid of nullables there are two options:

1. Declare variables as `late final` to tell the compiler to not emit an error. They are going to be initialized later but anyway before being accessed for the first time ever.

```
class Fraction {
    late final int _numerator;
    late final int _denominator;

    Fraction(int numerator, int denominator) {
        _numerator = numerator;
        _denominator = denominator;
    }
}
```

Because of the `final` modifier, variables cannot be changed anymore after their initialization. You could have only used `late` but variables would be mutable then:

```
class Fraction {
    late int _numerator;
    late int _denominator;

    Fraction(int numerator, int denominator) {
        _numerator = numerator;
        _denominator = denominator;
    }
}
```

In both cases, members are **NOT** initialized immediately because the body of the construc-

tor is executed after the variable initialization phase.

2. The Dart team recommends going for the "initializing formal"¹ approach as it's more readable and it initializes the variables immediately.

```
class Fraction {  
    int _numerator;  
    int _denominator;  
  
    Fraction(this._numerator, this._denominator);  
}
```

It's just syntactic sugar to immediately assign values to members. In this case, variables initialization is executed first so no need to use nullable types or `late`. This kind of initialization happens **before** the execution of the constructor's body.

Keep in mind that constructor bodies are executed **after** the variable initialization phase. The second approach is very common and you should get used to it, even if your class only declares `final` fields.

```
class Fraction {  
    final int _numerator;  
    final int _denominator;  
  
    Fraction(this._numerator, this._denominator);  
}
```

If your class doesn't define a constructor, the compiler automatically adds a *default* constructor with no parameters and an empty body. With the "initializing formal" you can still declare a body to perform additional setup for the class.

```
class Fraction {  
    final int _numerator;  
    final int _denominator;  
    late final double _rational;  
  
    Fraction(this._numerator, this._denominator) {  
        _rational = _numerator / _denominator;  
        doSomethingElse();  
    }  
}
```

¹<https://dart.dev/guides/language/effective-dart/usage#do-use-initializing-formals-when-possible>

```
    }
};
```

You could only write `Fraction(this._numerator)` to initialize exclusively `_numerator` but then `_denominator` would be set to `null` by the compiler. Keep in mind that you cannot have **named** optional parameters starting with an underscore.

```
// Doesn't compile
Fraction({this._numerator, this._denominator});
```



However, you can have **positional** parameters starting with an underscore:

```
// Ok but pay attention to non-nullability
Fraction([this._numerator, this._denominator]);
```

4.2.1 Initializer list

When using the initializing formal approach, the names of the variables must match the ones declared in the constructor. This could lead to an undesired exposure of some internals of the class:

```
class Test {
    int _secret;
    double _superSecret;
    Test(this._secret, this._superSecret);
}
```



What if you wanted to keep `int _secret` (private) but with a different name in the constructor? Use an initializer list! It's executed before the body and thus variables are immediately initialized. No need for nullable types or late.

```
class Test {
    int _secret;
    double _superSecret;

    Test(int age, double wallet)
        : _secret = age,
          _superSecret = wallet;
}
```

In this way the constructor is asking you for `age` and `wallet` but the user has no idea that

internally they're treated as `_secret` and `_superSecret`. It's basically a way to "rename" internal private properties you don't want to expose.

4.2.2 Named constructors

Named constructors are generally used to implement a default behavior the user expects from your class. They are the only alternative to have multiple constructors since Dart has no method overload.

```
class Fraction {  
    int _numerator;  
    int _denominator;  
  
    Fraction(this._numerator, this._denominator);  
  
    // denominator cannot be 0 because 0/0 is not defined!  
    Fraction.zero() :  
        _numerator = 0,   
        _denominator = 1;  
}
```

At this point you can use the named constructor in your code like if it were a static method call.

```
void main() {  
    // "Traditional" initialization  
    final fraction1 = Fraction(0, 1);  
  
    // Same thing but with a named constructor  
    final fraction2 = Fraction.zero();  
}
```

In general constructors aren't inherited by a subclass so, if they are needed across the hierarchy, every subclass must implement its own named constructor. If we had written a named constructor with a body...

```
class Fraction {  
    int? _numerator;  
    int? _denominator;
```

```

        Fraction.zero() {
            _numerator = 0;
            _denominator = 1
        }
    }
}

```

... we would have had to use nullable instance variables as constructors' bodies are always executed **after** variables' initialization.

4.2.3 Redirecting constructors

Sometimes you might have a constructor that does almost the same thing already implemented by another one. It may be the case to use redirecting constructors in order to avoid code duplication:

```

Fraction(this._numerator, this._denominator);
// Represents '1/2'
Fraction.oneHalf() : this(1, 2);
// Represents integers, like '3' which is '3/1'
Fraction.whole(int val) : this(val, 1);

```

Where `Fraction.oneHalf()` is just another way to call `Fraction(1, 2)` but you've avoided code repetition. This feature is very powerful when mixed with named constructors.

4.2.4 Factory constructors

The `factory` keyword returns an instance of the given class that's not necessarily a new one. It can be useful when:

- You want to return an instance of a subclass instead of the class itself,
- You want to implement a singleton (the Singleton pattern), 
- You want to return an instance from a cache.

Factory constructors are like static methods and so they don't have access to `this`. There cannot be together a factory and a "normal" constructor with the same name.

```

class Test {
    static final _objects = List<BigObject>();

    factory Test(BigObject obj) {

```

```
    if (!objects.contains(obj))
        objects.add(obj);

    return Test._default();
}

// This is a private named constructor and thus it can't be called
// from the outside
Test._default() {
    //do something...
}
```

In the example, since `BigObject` requires a lot of memory and the list is very very long, we've declared `objects` as `static`. This technique is often used to save memory and reuse the same object across multiple objects of the same type.

- ❶ If the list weren't `static` (just a normal instance variable), it would be created **every** time that a `Test` object is instantiated. It'd be a waste of memory and a performance problem; in this way we're guaranteed that there's an unique list created only once.

In this case the factory constructor is essential because it takes care of updating the `_objects` cache. Factories are called "normally" like if they were a regular constructor:

```
// Calls the factory constructor
final a = Test();
```

4.2.5 Instance variables initialization

As we've already seen, "normal" non-nullable variables have to be initialized either via initializing formal or initializer list. In any other case, they're set to `null` because constructor bodies run after the instance initialization phase.

```
// Initializing formal
class Example {
    int _a; // Ok - 'a' initialized by the constructor
    Example(this._a);
```

Chapter 4. Classes

```

    }

class Example {
    int _a; // Error - 'a' not initialized
    Example(int a) {
        _a = a;
    }
}

```

If variables aren't initialized immediately and you want them to be non-nullsafe, you can use the `late` modifier. It works like a "lazy initialization" because with this keyword you allow a nullable to be initialized later (but anyway before it gets accessed for the first time ever).

```

// For Dart 2.10 and earlier versions, 'late' does not exist so just
// remove it. Not initialized variables will be set to 'null' by default.
class Example {
    late int a;

    void printExample() {
        a = 5;
        print("$a");
        a = 2;
        print("$a");
    }
}

```

If you tried to use the variable before it gets assigned for the first time, you would get a compilation error. Always be sure to have the initialization done before using it.

```

class Example {
    late int a;

    void printExample() {
        // Compilation error
        print("$a");
        a = 5;
    }
}

```

Using `a` like above causes an error because it's accessed before being initialized. There's also the

possibility to declare a `late final` variable which behaves in the same way but with the only exception that it can be assigned only once.

```
class Example {
    late final int a;

    void printExample() {
        a = 5;
        print("$a");
        //a = 6; <-- This would be an error
    }
}
```

Once `a` is set with `a = 5;` you can't re-assign it anymore because of the `final` modifier. Instead, if it were a simple `late int a;`, you could have re-assigned it multiple times.

```
late final double a = takesLongTime();
```

Thanks to the usage of the `late final` combination you can lazily initialize a variable that is going to hold a value computed from a function. You can assign it immediately or, as we've just seen, in a second moment. The function `takesLongTime()` will only be called once `a` is accessed.

4.2.6 Good practices

Following what the official documentation² suggests, here's some tips you should consider while writing constructors for your classes:

1. Prefer using the "initializing formal" approach rather than initializing variables directly in the body of the constructor. Doing so, you'll avoid the usage of nullable types or `late`.
2. When you use the initializing formal, the types of the variables are deduced automatically to reduce the verbosity. Omit the types because they're useless.

```
// Ok but useless
Constructor(String this.a, double this.b) {}
// OK
Constructor(this.a, this.b);
```

3. When you have an empty constructor with no body, use the semicolon instead of the empty brackets.

²<https://dart.dev/guides/language/effective-dart/usage#constructors>

```
// Bad
Constructor(this.a, this.b) {}
// Good
Constructor(this.a, this.b);
```

4. Do not use the `new` keyword when creating new instances of objects. In modern Dart and Flutter code, `new` never appears.
5. We recommend using redirecting constructors to avoid code duplication. It makes maintenance easier.
6. Try to not use the `late` keyword because it could lead to hard maintenance (you'd have to **manually** keep an eye on the initialization of variables). Whenever possible, initialize variables as soon as they are declared.
7. In the second part of the book we'll show a case where `late` or `late final` are required. They'll be used to initialize values inside the state of a Widget.

```
class Example extends State {
    late int value;

    @override
    void initState() {
        super.initState();
        value = 0;
    }
}
```

In short, subclasses of `State` cannot define a constructor and you have to perform the initialization of the variables in the `initState()` method. In order do to this, `late` is essential.

The syntax for private constructors in Dart might seem a bit weird at first. Generally, a private constructor is used in conjunction with a `factory` that returns a subtype or an instance of the actual object with certain criteria.

```
class Example {
    final a;
    // Private constructor
    Example._(this.a);

    factory Example(int value) {
```

```

    final c = value * 3;
    return Example._(c);
}
}

```

A private constructor is declared using the `._()` notation. In the example, the class can still be instantiated but only because we've defined a `factory`. In this case...

```

final ex = Example(10);
... we're not calling the "normal" constructor (because it's package private) but instead the
factory one.

```

4.3 const keyword

The `const` keyword can be used when you have to deal with `compile-time constant` values such as strings or numbers. It can automatically deduce the type.

```

// type of 'number' is int
const number = 5
// explicitly write the type
const String name = 'Alberto';

const sum = 5.6 + 7.34;

```

It's true that `final` and `const` are very similar at first glance and they also share the same syntax style. A very intuitive way to determine which one you can choose is ask yourself: is this value already well defined? is it known at compile time? Let's find out why.

- `final`. Use it when the value is not known at compile time because it `will be evaluated/obtained at runtime`. Common usages are I/O from the disk or HTTP requests. For example, this is how you read a text file from the disk (more on this in A.1):

```

final contents = File('myFile.txt').readAsString();

// const contents = File('myFile.txt').readAsString();
// ^ does not compile!

```

The compiler doesn't know in advance which is the content of `myFile.txt` because it will be read only when the program will be running (so *after* the compilation). For this reason, you can only use `final`.

Chapter 4. Classes

- **const**. Use it when the value is computed at compile time, for example with integers, doubles, Strings or classes with a *constant constructor* (more on it in the next section).

```
const a = 1;
// final a = 1 -> it works as well
```

If it works with **const**, it works also with **final** because anything that is **const** is also **final**.

Instance variables can only be declared as **final** while **const** can be applied in combination with the **static** keyword.

```
class Example {
    // OK
    final double a = 0.0;
    // NO, instance variables can only be 'final'
    const double b = 0.0;

    // OK
    static const double PI = 3.14;
    // OK but without type annotation
    static const PI = 3.14;
}
```

The real power of **const** comes when combined with constructors and, in Flutter, it can lead to an important performance boost.

- ➊ Variables and methods marked with the **static** modifier are available on the class itself and not on instances. In practice it means that you can use them without having to create an object.

```
class Example {
    static const name = "Flutter";
    static String test() => "Hello, I am $name!";
}

void main() {
    final name = Example.name;
    final text = Example.test();
}
```

Both variables are strings and they've been retrieved without creating an instance of `Example`; `static` members belong to the "class scope" and you cannot use `this`.

```
class Example {
    int a = 0;
    static void test() {
        // Doesn't compile
        final version = this.a;
        print("$version");
    };
}
```



Since you don't have to create objects to call static methods, `this` cannot work because it refers to the current instance that gets never created.

4.3.1 const constructors

In Dart you can append the `const` keyword in front of a constructor only if you're going to initialize a series of `final` (immutable) variables.

```
// Compiles
class Compiles {
    final int a;
    final int b;
    const Compiles(this.a, this.b);
}

// Does not compile because a is mutable (not final)
class DoesNot {
    int a;
    final int b;
    const DoesNot(this.a, this.b);
}
```



If your class only has `final` variables it's said to be an "immutable class" and you should really instantiate it with a `const` constructor. The compiler can perform some optimizations.

```
final example1 = const Compiles(); // (1) constant object
final example2 = Compiles();      // (2) not a constant object!
```

In example (1) we're calling the constant constructor but in (2) we're **not**. Even if your class only has constant constructors, objects can be instantiated as constants only with the `const` keyword. When put in front of a collection, such as a list, everything inside that container will automatically be `const` (if it is allowed to be constant).

```
class Test {
    // constant constructor
    const Test();
}

const List<Test> listConst = [Test(), Test()]; // (1)
final List<Test> listConst2 = [Test(), Test()]; // (2)
```

In (1) everything inside the list is automatically "converted" into a `const` value while in (2) it doesn't happen, meaning that the contents of `listConst2` aren't constant. The `final` keyword in front of a list just makes it impossible to change the reference assigned to `listConst2` but does **NOT** call the constant constructor (while example (1) does).

```
// Bad
const List<Test> list1 = [const Test(), const Test()]; // (1)

// Good
const List<Test> list2 = [Test(), Test()]; // (2)
```

You should avoid version (1) because calling `const Test()` is not necessary. Any constant collection initializes its children calling their `const` constructor (if any, otherwise a compilation error occurs).

4.3.2 Good practices and annotations

If you know that variables in your class will never change, you really should make them `final` and use a `const` constructor. As we've already said, constant constructors play a very important role in Flutter because they allow "caching" on instances.

i Don't get "obsessed" by immutable classes and `const` constructors because you simply can't use them in every situation. If your instance variables cannot be `final` that's perfectly fine; the environment and the compiler are very powerful and your final product won't suffer of speed degradations due to the lack of immutability.

Any class with a constant constructor can be used as **annotation**: they're are generally put

before the name of a class or method. An annotation is preceded by the "at" sign (@). In the next section, we will see that overriding methods is usually done in the following way:

```
class MySubclass extends SuperClass {  
  @override  
  void defineMethod() {}  
}
```

The `@override` annotation does nothing in practice: it just tells the developer that `defineMethod` has been overridden. If you looked at how the override annotation is declared in the Dart SDK, you'll find simply the following:

```
// This class has a constant constructor and so it can be used as annotation  
class _Override {  
  const _Override();  
}  
  
// the actual "@override" annotation  
const Object override = _Override();
```

The class has been made private (`_Override`) because its instantiation is useless as it does nothing. However, thanks to the `const Object` `override` variable being public, there's an "alias" of the `_Override` class which can be used as annotation. They're generally used for:

- reminding the developer about something, such as in the case of `@override`;
- before Dart 2.10, the `@required` annotation was used by the IDE to bring the developer's attention to the fact that a named optional parameter is required;
- some packages, such as `json_serializable` we're going to cover in chapter 15, rely on annotations to add additional information about a class, a method or a member. Annotations can be used to pass data to code generation tools.

Annotations can also have parameters but in this case you aren't doing the above "trick" of declaring a global variable exposing a private class. Just create a normal class, with a `const` constructor and use it as follows:

```
// Use this as annotation but it takes a param  
class Something {  
  final int value;  
  const Something(this.value);  
}
```

```
@Something(10)
class Test {}
```

4.4 Getters and setters

When a public variable is declared, anyone can freely manipulate it but it may not a good idea because the class partially loses control of its members. If we had written this...

```
class Fraction {
    int numerator;
    int denominator;
    Fraction(this.numerator, this.denominator);
}
```

... at a certain point someone could have changed the numerator and the denominator without any control introducing unexpected behaviors due to a wrong internal state.

```
void main() {
    final frac = Fraction(1, 7);

    frac.numerator = 0;
    frac.denominator = 0;
}
```

Having set both numerator and denominator to 0 there will be problems at runtime due to an invalid division operation. We can fix this problem using a *getter*, which makes the variables read-only.

```
class Fraction {
    int _numerator;
    int _denominator;
    Fraction(this._numerator, this._denominator);

    // Getters are read-only
    int get numerator => _numerator;
    int get denominator {
        return _denominator;
    }
}
```

The getter `numerator` returns an `int`, `_numerator`; the getter `denominator` does the same thing with an equivalent syntax. Like it happens with methods, when there is an one-liner expression or value to return, the `=>` (arrow) syntax can be used.

```
void main() {
    final frac = Fraction(1, 7);

    // Compilation error, numerator is read-only
    frac.numerator = 0;
    // No problems here, we can read its value
    final num = frac.numerator;
}
```

The code is now safe because we can expose both numerator and denominator but it's guaranteed that they cannot be freely modified. Internally, `_numerator` and `_denominator` are "safe" because they aren't visible from the outside. Just as example, we're going to see how to write a `setter` for the denominator. So far it's read-only but with a setter it becomes editable:

```
class Fraction {
    int _numerator;
    int _denominator;
    Fraction(this._numerator, this._denominator);

    // getters
    int get numerator => _numerator;
    int get denominator => _denominator;

    // setter
    set denominator(int value) {
        if (value == 0) {
            // Or better, throw an exception...
            _denominator = 1;
        } else {
            _denominator = value;
        }
    }
}
```



There can be the same name for a setter and a getter so that a property can be read/written using the same identifier. **Setters should be used to make "safe edits" on variables; they often contain a**

validation logic which makes sure that the internal state of the class doesn't get corrupted.

```
void main() {
    final frac = Fraction(1, 7);

    var den1 = frac.denominator; // den1 = 7
    frac.denominator = 0; // the setter changes it to 1
    den1 = frac.denominator // den1 = 1
}
```

To sum it up, getters and setters are used to control the reading/writing on variables. They are methods under the hood but with a "special" syntax that uses the `get` and `set` keywords.

4.4.1 Good practices

Our recommendation is to keep the body of getters and setters as short as possible in benefit of code readability. You shouldn't put any loop that might slow down the assignment/retrieval of a value. The official documentation³ also has something to say:

- When a variable has to be both public and read-only, just mark it as `final` without associating a getter to it.

```
// Bad
class Example {
    final _address = "https://fluttercompletereference.com";
    String get address => _address;
}

// Good
class Example {
    final address = "https://fluttercompletereference.com";
}
```

If it's `final`, it's already a read-only variable because nothing can change its content. In this case a getter is simply useless.

- Avoid wrapping public variables with getters and setters if there's no validation logic.

```
class Example {
    var _address = "https://fluttercompletereference.com";
```

³<https://dart.dev/guides/language/language-tour#getters-and-setters>

```

    String get address => _address;
    set address(String value) => _address = value;
}

```

It compiles but there's no point in doing that: both getter and setter don't perform any particular logic as they just serve the variable as it is. Prefer doing this:

```

class Example {
    var address = "https://fluttercompletereference.com";
}

```

In general, use getters when you want to expose a variable but in "read-only mode" and setters when you want to filter/check the value that is going to be assigned.

4.5 Operators overload

When you deal with primitive types you use operators very often: `5 + 8` is a sum between two `int` types happening with the `+` operator. We are going to do the same with `Fraction` class.

```

class Fraction {
    Fraction operator+(Fraction other) =>
        Fraction(
            _numerator * other._denominator +
            _denominator * other._numerator,
            _denominator * other._denominator
        );

    Fraction operator-(Fraction other) => ...;

    Fraction operator*(Fraction other) => ...;

    Fraction operator/(Fraction other) => ...
}

```

Operator overloading gives the possibility to customize the usage of operators in your classes. We have overloaded the `+` operator so that we can easily sum two fractions instead of having to create an `add(Fraction value)` method, like it happens with Java.

```
void main() {
```

```
// 2/5
final frac1 = Fraction(2, 5);
// 1/3
final frac2 = Fraction(1, 3);

// 2/5 + 1/3 = 11/15
final sum = frac1 + frac2
}
```

They work like normal methods with the only exception that the name must be in the form `operator{sign}` where `sign` is a supported Dart operator:

- **Arithmetic** operators like `+`, `-`, `*`, or `/`.
- **Relational** operators such as `>=`, `<=`, `>` or `<`.
- **Equality** operators like `!=` and `==`

And many more. There are no restrictions on the types you can handle with the operators, meaning that we could also sum fractions with integers: `operator+(int other)`. You cannot overload the same operator more than once in the same class.

4.5.1 callable classes

There is a special `call()` method which is very closely related to an operator overload because it allows **classes to be called like if they were functions with the () operator**.

- i** You can give `call()` as many parameters as you want as there are no restrictions on their types. The function can return something or it can simply be `void`.

Let's give a look at this example.

```
class Example {
    double call(double a, double b) => a + b;
}

void main() {
    final ex = Example();           // 1.
    final value = ex(1.3, -2.2);   // 2.
```

```
    print("$value");
}
```

1. Classic creation of an instance of the class
2. The object `ex` can act like if it were a function. The `call()` method allows an object to be treated like a function.

Any class that implements `call()` is said to be a **callable class**. In Dart, everything is an object and you've seen in 3.6.1 that even functions are objects. You can now understand why with the following example:

```
void test(String something) {
    print(something);
}
```



This is a typical `void` function asking for a single parameter. Actually, the above code can be converted into a callable class that overrides `call()` returning nothing and asking for a string.

```
// Create this inside 'my_test.dart' for example
class _Test {
    const _Test();

    void call(String something) {
        print(something);
    }
}

const test = _Test();

// Somewhere else, for example in main.dart
import 'package:myapp/my_test.dart';

void main() {
    test("Hello");
}
```

The function is nothing more than a package private class that overrides `call()` with a certain signature. Thanks to `const test = _Test();` in the last line we're "hiding" the class and exposing a callable object to be used as function.

4.6 Cloning objects

Even if it's not mentioned in the official Dart documentation, there is a standard "*pattern*" to follow when it comes to cloning objects. Unlike Java, there is no `clone()` method to override but still you might need to create deep copies of objects:

```
class Person {  
    final String name;  
    final int age;  
    const Person({  
        required this.name,  
        required this.age,  
    });  
}  
  
void main() {  
    const me = const Person(  
        name: "Alberto",  
        age: 25  
    );  
    const anotherMe = me;  
}
```

As you already know, the variable `anotherMe` just holds a reference to `me` and thus they point to the same object. Changes applied to `me` will also reflect on `anotherMe`. If you want to make deep copies in Dart (cloning objects and making them independent), this is the way:

```
class Person {  
    final String name;  
    final int age;  
    const Person({  
        required this.name,  
        required this.age,  
    });  
  
    Person copyWith({  
        String? name,  
        int? age,  
    }) => Person(
```

```

        name: name ?? this.name,
        age: age ?? this.age
    );
}

@Override
String toString() => "$name, $age";
}

```



This method is called `copyWith()` by convention and it takes the same number (and name) of parameters required by the constructor. It creates a **new**, independent copy of the object (a clone) with the possibility to change some parameters:

```

const me = const Person(
    name: "Alberto",
    age: 25
);

// Create a deep copy of 'me'.
final anotherMe = me.copyWith();

// Create a deep copy of 'me' with a different age.
final futureMe = const.copyWith(age: 35);

print("$me");      // Alberto, 25
print("$anotherMe"); // Alberto, 25
print("$futureMe"); // Alberto, 35

```

Both `anotherMe` and `futureMe` have **no** side effects on `me` because the reference is not the same. In fact, `copyWith()` returns a fresh new instance by copying internal data. Let's take a look at this line:

```
name: name ?? this.name,
```

Thanks to the `??` operator, if `name` is `null` then initialize the clone with value of `this.name` taken from the instance. In other words, if you don't pass a custom `name` to `copyWith()`, by default a copy of `this.name` is made. Pay attention to generic containers and objects in general:

```

class Skills {...}

class Person {
    final List<Skills> skills;
}

```

```

const Person({
    required this.skills
});

Person copyWith({
    List<Skills>? skills,
}) => Person(
    skills: skills ?? this.skills
);
}

```

This code doesn't do what you'd expect because `List<T>`, like any other generic container, is an object and not a primitive type. With the above code you're just copying **references** and not making copies. The correct solution is the following:

```

Person(
    skills: skills ?? this.skills.map((p) => p.copyWith()).toList();
)

```



In this way you're making a copy of the entire list rather than passing a reference. The above code is just an one-liner way to iterate on each element of the source, making deep copies using `copyWith` and returning a new list. However, when the list is made up of primitive types, you could use a shortcut:

```

class Person {
    final List<int> values;
    const Person({
        required this.values
    });

    Person copyWith({
        List<int>? values,
    }) => Person(
        values: values ?? []..addAll(this.values)
    );
}

```

Primitive types are automatically copied so instead of using `map()` (which would be perfectly fine as well) we can use `addAll()` for a shorter syntax. There is no difference however because it still iterates on every element of the source list. The same example also applies to `Map<K, V>` and



Set<K>. To sum it up, what you have to keep in mind is:

- Deep copies in Dart are made using the `copyWith()` method. You can give it any other name but you'd better follow the conventions.
- When making copies, be sure that classes (like generic containers) are deep copied using the convenient `map((x) => x.copyWith())` strategy.
- If you have a list of primitive types (like `doubles` or `int`) you can use the `[]..addAll()` shortcut. Do this **only** with primitive types.

5 | Inheritance and Exceptions

5.1 Inheritance

In any OOP language you can create hierarchies of classes and Dart is no exception. Here's the most basic example we can imagine.

```
class A {}  
class B extends A {}
```

As you might expect, *A* is called superclass while *B* is the subclass (or "child class"). Methods can be overridden in subclasses because they all are "virtual" by default.

ⓘ The term *virtual* indicates the possibility to redefine the behavior of a method in the subclasses. It's a very common OOP concept.

Actually Dart doesn't have the `virtual` keyword but it's "implicit" because any method can be overridden along the hierarchy. The annotation `@override` is optional but you'd better always use it. Starting from Dart 2.9 onwards, implicit downcasts are **not** allowed:

```
class A {  
    double test(double a) => a * 0.5;  
}  
  
class B extends A {  
    @override  
    double test(double a) => a * 1.5;  
}
```

```

void main() {
    A obj1 = A();
    A obj2 = B(); // Upcast
    B obj3 = obj1; // Downcast - ERROR from Dart 2.9
    
    print("${obj1.test(1)}"); // Prints 0.5
    print("${obj2.test(1)}"); // Prints 1.5
}

```

This behaves in the classic OOP way; `obj1` calls the method in `class A` while `obj2` calls the overridden version in `class B`. When overriding, you can reference the original method definition in the superclass:

```

class B extends A {
    @override
    double test(double a) {
        final original = super.test(a);
        return original * 1.5;
    }
}

```

The special keyword `super` holds a reference to the super class. The usage of `super.test(a)` calls the `test()` method defined in the superclass. In Java for example, you get the same behavior.

-  You cannot block inheritance since every class can have the `extends` modifier.
In Java for example you can write `final class A {}` to say "hey, you can't inherit from me" but in Dart there is no equivalent.



You're allowed to also override setters (`set`) and getters (`get`) other than regular methods. Dart doesn't support multiple inheritance, like C++ does for example, meaning that `extends` works only with a single class:

```

// Ok
class A {}
class B extends A {}

// Doesn't work
class A {}

```

Chapter 5. Inheritance and Exceptions

```
class B {}
class C extends A, B {}
```

When overriding a parameter's type with a subtype, the compiler emits an error. Thanks to the `covariant` keyword, you can turn off static analysis for this kind of error to tell the compiler you know you're doing this intentionally. Here's an example:

```
abstract class Fruit {}
class Apple extends Fruit {}
class Grape extends Fruit {}
class Banana extends Fruit {}

abstract class Mammal {
    void eat(Fruit f);
}

class Human extends Mammal {
    // Ok
    void eat(Fruit f) => print("Fruit");
}

class Monkey extends Mammal {
    // Error
    void eat(Banana f) => print("Banana");
}
```

Instead of `eat(Banana f)` we should have written `eat(Fruit f)` inside `Monkey` because the superclass method is asking for `Fruit`. However, we can allow the definition of a subtype in an overridden method with `covariant`:

```
class Monkey extends Mammal {
    void eat(covariant Banana f) => print("Banana");
}
```

Now the code compiles. Usually the superclass method is the best place where you could use `covariant` because it removes the "subtype constrain" along the entire hierarchy.

```
abstract class Mammal {
    void eat(covariant Fruit f);
}

class Human extends Mammal {
    // Ok
    void eat(Fruit f) => print("Fruit");
```

```
}

class Monkey extends Mammal {
    // Ok
    void eat(Banana f) => print("Banana");
}
```

Thanks to `covariant Fruit` at the top, any subclass is allowed to override `eat()` with `Fruit` or a subtype. This keyword can also be applied to setters and fields.

5.1.1 super and constructors

Every subclass in Dart automatically tries to call the default constructor of the superclass. If there isn't one, you **must** call the superclass constructor manually in the initializer list.

```
class Example {
    int a;
    Example(this.a);
}

class SubExample extends Example {
    int b;
    // If you don't call 'super(b)' the compilation will fail
    // because the father class has NO default constructor
    SubExample(this.b) : super(b);
}
```

The superclass constructor must be called in `SubExample` because the compiler has to somehow initialize the variable `a` of `Example`. However `super` is not needed when the class has a default constructor:

```
// The compiler automatically generates the default constructor
// which is just 'Example();'
class Example {
    int a = 0;
}

class SubExample extends Example {
    int b;
    // No need to call super
    SubExample(this.b);
```

```
}
```

Which is equivalent to...

```
class Example {
    int a = 0;
    Example();
}

class SubExample extends Example {
    int b;
    SubExample(this.b) : super();
}
```

... but you shouldn't do this because the compiler adds the missing pieces for you. You should call the superclass constructor only when there are parameters to pass. The call to `super()`, in case of an initializer list, always goes last:

```
// Compiles
MyClass(int a) : _a = a, super(a*a);

// Doesn't compile
MyClass(int a) : super(a*a), _a = a;
```

5.1.2 Abstract classes

The `abstract` keyword defines a class that cannot be directly instantiated: only its derived classes can. An abstract class can define one (or more) constructors as usual.

```
abstract class Example {
    // This is an abstract method
    void method();
}
```

Usually abstract classes contain abstract methods which can be defined putting a semicolon (;) instead of the body. You cannot define an abstract method in a class that's not been marked with the `abstract` modifier.

```
// Wrong: doesn't compile
class Example {
    // This method is abstract (no body) BUT the
```

```
// class doesn't have the 'abstract' modifier
void method();
}

// Correct: it compiles
abstract class Example {
    // Good job, this abstract method is in an
    // abstract class
    void method();
}
```

If your class contains **at least** one method with no body, then it must be **abstract** and the children must provide an implementation. You could also do this:

```
abstract class Example {
    void method();
}

class ExampleTwo extends Example {}
```

`ExampleTwo` is an abstract class too because it doesn't contain an implementation of `method()`; yet. A class is concrete (and thus not abstract) when every method has a body.

```
abstract class Example {
    void method();
}

class ExampleTwo extends Example {
    @override
    void method() {
        print("I'm not abstract!");
    }
}
```

Now `ExampleTwo` is a concrete class. As we've already said, getters and setters can also be abstract and they're declared in the same way as methods (without a body).

```
abstract class Example {
    final int _a;
    const Example(this._a);
```

```
// Abstract getter  
int get calculate;  
  
// Abstract method  
int doSomething();  
}
```

5.1.3 Interfaces

In contrast to other programming languages, Dart does not have an `interface` keyword and you have to use classes to create interfaces. Your class can implement more than a single interface.

```
abstract class MyInterface {  
    void methodOne();  
    void methodTwo();  
}  
  
class Example implements MyInterface {  
    @override  
    void methodOne() {}  
  
    @override  
    void methodTwo() {}  
}
```

The keyword is `implements` and, differently from a regular subclass, here you must override **every** method defined by the class/interface. The official documentation states the following:

- ❶ Every class implicitly defines an interface containing all the instance members of the class and of any interfaces it implements. If you want to create a class A that supports class B's API without inheriting B's implementation, class A should implement the B interface.

In Dart when you use the term *interface* you are referring to a class that is going to be used by others along with `implements` because it only provides method signatures. The concept is the same you can find in Java, Delphi or C# with the only difference that Dart doesn't have a

dedicated keyword.

```
// Instead of the 'interface' keyword, 'abstract class' is used
abstract class OneInterface {
    void one();
}

abstract class TwoInterface {
    void two();
}

class Example implements OneInterface, TwoInterface {
    @override
    void one() {}

    @override
    void two() {}
}
```

While `extends` can be used with only one class, `implements` works with one or more classes, which you should treat as interfaces (methods with no body). You could also do the following:

```
class OneInterface {
    void one() {}
}

class TwoInterface {
    void two() {}
}

class Example implements OneInterface, TwoInterface {
    @override
    void one() {}

    @override
    void two() {}
}
```

Classes and abstract classes can both be treated as interfaces with the difference that in concrete classes there must be at least an empty body. For this reason, in the above example we've put

an empty body.

- ❶ We recommend to use abstract classes as interfaces so that you can write only the name of the method, like `void test();`. If you used a regular class you'd have to write `void test() {}` which is identical but it has an unnecessary empty body.

5.1.3.1 extends vs implements

You've just seen that `extends` is for subclasses and `implements` is for classes treated like if they were interfaces. It would be very fair if you were puzzled about the situation. Let's start with the technical difference:

- When you use `class B extends A {}` you are **NOT** forced to override every method of class *A*. Inheritance takes place and you can override as many methods as you want.
- When you use `class B implements A {}` you must override **every** method of class *A*. Inheritance does **NOT** take place because methods just provide an API, a "skeleton" that the subclass must concretize.

In practical terms instead:

- `extends`. This is the typical OOP inheritance that can be used when you want to add some missing features in a subclass. In chapter 8 we will see that it's a good practice deriving only abstract classes and not concrete classes.
- `implements`. Interfaces are useful when you don't want to provide an implementation of the functions but just the API. It's like if the interface was wall socket and the class was the plug that adapts to the holes.

While multiple inheritance is not allowed, you can `extends` a class and `implements` more than one. This is also how Java and C# behave, for example.

```
// Error
class A extends B, C, D {}

// Valid
class A extends B implements C, D {}
```

It might be useful if we made two final examples so that you can visualize the difference.

- When you have a common behavior for **every** children of your hierarchy, create an abstract class and use `extends`. Say that you want to read many popular document files such as `.pdf`, `.docx` and `.txt`. Inheritance might be a very good idea:¹

```
abstract class Reader {  
    bool fileExists(String path) {  
        // code...  
    }  
  
    double size(File file) {  
        // code...  
    }  
  
    String readContents();  
}
```

Deciding whether a file exists or not and getting its size is something common we always want to be able to call. Reading contents instead is specific to the implementation, it cannot be shared along the hierarchy, so it has to be abstract.

```
class PDFReader extends Reader {  
    @override  
    String readContents() {  
        // code...  
    }  
}  
  
class DocxReader extends Reader {  
    @override  
    String readContents() {  
        // code...  
    }  
}  
  
class TxtReader extends Reader {  
    @override  
    String readContents() {  
        // code...  
    }  
}
```

¹See appendix A for more info on the `File` class

```
    }
}
```

It's been the right choice because `fileExists()` and `size()` have been defined only once, so no code duplication, and they're available in every subclass "for free". If we used `implements` we'd have had to define all of methods in the subclasses, including the "shared" ones leading to code duplication.

- When you don't have methods implementations to share/reuse in the subclasses and you just need to give the signature, use `implements`. Say that you want to create a few sorting algorithms by yourself: using interfaces is the way to go (this is known as "*Strategy pattern*").

```
// This is going to be the "interface"
abstract class Sorter {
    void sort();
    String averageComplexity();
}

class MergeSort implements Sorter {
    @override
    void sort() {...}

    @override
    String averageComplexity() => "n*log(n)";
}

class InsertionSort implements Sorter {
    @override
    void sort() {...}

    @override
    String averageComplexity() => "n^2";
}
```

Any sorting algorithm has its own implementation and a particular name, so there's no common behavior. You just know that every concrete implementation needs a sorting method (`void sort()`) and the average time complexity (`String averageComplexity()`). Thanks to `implements` subclasses have to override every method and adopt a common API.

In summary, if you have common methods along the hierarchy go for subclasses otherwise use classes as interfaces. The usage of `implements` forces you to override every method; if this is not the behavior you're looking for, use `extends`.

5.1.4 Mixins

There's another important concept in Dart, which is the "cousin" of abstract classes and interfaces. A `mixin` is simply a `class with no constructor` that can be "attached" to other classes to reuse the code without inheritance.

```
mixin Swimming {
    void swim() => print("Swimming");
    bool likesWater() => true;
}

mixin Walking {
    void walk() => print("Walking");
}
```

If you use the special `with` keyword on a class, it acquires any method defined in the mixin. You could see mixins like a "copy/paste" tool to reuse methods.

```
class Human with Walking {
    final String _name;
    final String _surname;

    Human(this._name, this._surname);
    void printName() => "${_name} ${_surname}";
}

void main() {
    final me = Human("Alberto", "Miola");

    // prints "Alberto Miola"; method is defined in the class
    me.printName();
    // prints "Walking"; method is not defined in the class
    // but it's "copied" and "pasted" from the mixin.
    me.walk();
}
```

Chapter 5. Inheritance and Exceptions

The class `Human` doesn't have a `walk()` method inside but it's automatically imported from the `Walking` mixin. Since mixins are just classes without a constructor, you could also successfully compile the following code:

```
// use 'class' instead of 'mixin'  
class Walking {  
    void walk() => print("Walking");  
}  
  
class Human with Walking {}
```



In general it'd better if you used the `mixin` keyword since it helps both yourself, to remind what you want to do, and the IDE, which can give you useful hints. Furthermore, using the `mixin` keyword is less confusing than `class`.

- ❶ Note that your class must have no constructors declared in order to be used as mixin.

```
class Walking {  
    Walking();  
    void walk() => print("Walking");  
}  
// Does not compile  
class Human with Walking {}
```

Even if you've declared the empty non-argument constructor (`Walking()`) this class is not treated as a mixin. You should prefer `mixin` over `class` when you intend to accomplish this purpose.

Some other important features offered by mixins are:

1. A class can have more than a single mixin associated to it and its subclasses inherit the imported methods.

```
mixin Walking {  
    void walk() {}  
}  
mixin Breathing {  
    void breath() {}
```

```
}

mixin Coding {
    void code() {}
}

// Human only has walk()
class Human with Walking {}

// Developer has walk() inherited from Human and also
// breath() and code() from the two mixins
class Developer extends Human with Breathing, Coding {}
```

2. There's the possibility to constrain the usage of a `mixin` to subclasses of a certain type. In this way you can make a sort of "restrictive reusability":

```
// Constrain 'Coding' so that it can be attached only to
// subtypes of 'Human'
mixin Coding on Human {
    void code() {}
}

// All good
class Human {}
class Developer extends Human with Coding {}

// NO, 'Coding' can be used only on subclasses
class Human with Coding {}

// NO, 'Fish' is not a subclass of 'Human' so
// you cannot attach the 'Coding' mixin
class Fish with Coding {}
```

Mixins are different from abstract classes and interfaces because they do not involve inheritance at all: they don't create hierarchies. They are just a way to reuse code without having to deal with superclasses, overrides and so on.

-  There is no rule of thumb to follow when it comes to the question "when should I prefer mixins over interfaces/abstract classes?" because the answer would be "It depends!". However, here's a general idea that you can follow to take your decision.

In general your first choices should be abstract classes or interfaces, with all their implementations in the various subclasses. Mixins are very handy when different pieces of architecture have the same identical code which cannot be shared due to a lack of inheritance. Let's say you have these classes in a folder:

```
import 'dart:math';

abstract class FootballTeam {
    String name();
    void playsWith() => print("Ball");
    double ballVolume(double radius) {
        const values = 4/3 * 3.14;
        return values * pow(radius, 3);
    }
}

class RealMadrid extends FootballTeam { ... }
class LiverpoolFC extends FootballTeam { ... }
```

And then, still in the same project, you have in another folder this hierarchy:

```
abstract class VolleyballTeam {
    String nameAndAbbreviation();
    void playsWith() => print("Ball");
    double ballVolume(double radius) {
        const values = 4/3 * 3.14;
        return values * pow(radius, 3);
    }
}

class TeamA extends VolleyballTeam { ... }
class TeamB extends VolleyballTeam { ... }
```

You see two separated hierarchies (which is good) with a few identical methods. You could create a superclass for `VolleyballTeam` and `FootballTeam` but it couldn't be possible due to the structure itself of your architecture. A good solution is the usage of a `mixin` which provides code sharing with no inheritance.

```
mixin BallSports {
```

```
void playsWith() => print("Ball");

double ballVolume(double radius) {
    const values = 4/3 * 3.14;
    return values * pow(radius, 3);
}
```

At this point you can get rid of the redundant methods and use the `mixin` to share the implementation (across the hierarchy as well). In this way you've removed code duplication, which is always a problem in terms of maintenance.

```
// Somewhere in your project...
abstract class FootballTeam with BallSports {
    String name();
}

// Somewhere else in your project...
abstract class VolleyballTeam with BallSports {
    String nameAndAbbreviation();
}
```

No more code duplication because the `mixin` solved the problem. Both hierarchies are separated because they are two different things but they have something in common. A `mixin` makes methods reusable without having to rely on inheritance.

5.1.5 Good practices

Generally, `factory` constructors are used to return "default" implementations of a certain class. For example, if you had a series of encryption algorithms, you could define a default one in this way:

```
abstract class EncryptionAlgo {
    // 'AESEncryption' is the default encryption algorithm
    factory EncryptionAlgo() {
        return AESEncryption();
    }

    void decrypt(String filePath);
}
```

```
class AESEncryption extends EncryptionAlgo { ... }
class RSAEncryption extends EncryptionAlgo { ... }
class BlowfishEncryption extends EncryptionAlgo { ... }
```

With this technique, you can use `EncryptionAlgo` to return an instance of `AESEncryption`:

```
// 'encrypt' is actually 'AESEncryption' because you're calling the
// factory constructor
final encrypt = EncryptionAlgo();
```

You are **NOT** creating an instance of an abstract class (which is impossible): you're just calling the factory constructor of `EncryptionAlgo` which returns a `AESEncryption`. This technique is used to return "default" instances of a certain hierarchy. You could use an even shorter syntax:

```
abstract class EncryptionAlgo {
    factory EncryptionAlgo() = AESEncryption;

    void decrypt(String filePath);
}
```

This kind of *constructor assignment* is just a shorter syntax to redirect a constructor to another. However, they must have the same number and type of arguments otherwise it won't compile. Basically you're telling the compiler to call the constructor of `AESEncryption` (along with the params, if any) when the `factory` is invoked.

5.2 Extension methods

Starting from Dart 2.7 the team has added *extension methods*, a way to add functionalities to a library knowing nothing about its internal implementation. Let's take a look at the `String` class, which has a lot of methods:

```
// A string
var name = "Alberto";

// Some methods of the string class
name.toUpperCase();
name.toLowerCase();
name.trimLeft()
```

Earlier we worked hard to create a `Fraction` class and it would be really cool if we were able to integrate it with strings. Something like this:

```
final Fraction value = "1/3".toFraction();

value.reduce();
value.negate();
```

Given a string, we could add the `toFraction()` method that returns a `Fraction` whenever possible. We don't have access to the `String` class and we can **not** edit the file in which it's been declared, of course. **Extension methods** come to the rescue:

```
extension FractionExt on String {
    bool isFraction() => ...

    // Converts a string into a fraction
    Fraction toFraction() => Fraction.fromString(this);
}
```

The `this` identifier refers to the object on which you are calling the method. What's written as `this.contains("/")` is evaluated to `"2/5".contains("/")` is the example:

```
extension FractionExt on String {
    // code...

    void main() {
        var str = "2/5";

        if (str.isFraction()) {
            final frac = str.toFraction();
        }
    }
}
```

With extension methods we've just *extended* the `String` class with new functionalities without changing the definition of the class. Methods have been added "from the outside".

i Abstract classes, interfaces and mixins are for "internal use" in the sense that with them you touch the class directly from the inside. Extensions are for "external use" in the sense that you add methods from the outside without changing the internal

part of the class.

With extensions you can also define getters, setters, operators, static fields and/or methods. If you need to use some utility private functions inside the extension, just append an underscore (_) in front of the name. It's like with classes, where `void _method() {}` is private to its library.

! Extension methods also work with generic types. You could for example write `extension Test<T> on List<T> {}` where T is statically determined by the compiler. Generics and collections will be discussed in the next chapter.

5.2.1 Good practices

Generally extension methods should be put in a dedicated file. For example, the previous extension of fraction could go on `fraction_ext.dart` so that it can be imported as a library.

```
// === file fraction_ext.dart === //
extension FractionExt on String {
    // code...
}

// === file main.dart === //
import 'package:fraction_ext.dart';

void main() {
    final check = "hello".isFraction();
}
```

Short and trivial operations should be implemented with extensions in order to improve the usability, the readability. Don't you agree that...

```
final f = "1/2".isFraction();
```

... looks better than a static method call?

```
final f = Fraction.isFraction("1/2");
```

Keep in mind that extension methods cannot be used if you're treating a variable with `dynamic`. They support type inference and then the compiler must always exactly know the types.

```
// It compiles but gives a runtime exception
```

```
dynamic f = "1/2";
f.isFraction();
```

Before moving on, let's try to make a simple summary of all the features we've seen up to now in this chapter.

- **Internal use.** These techniques are meant for "internal use" because they can work directly inside the definition of the class.
 - **extends.** It's the typical OOP inheritance: you don't need to override every method, getter or setter. Use it when there is a "common" behavior to share along the hierarchy.
 - **implements.** The "interface" OOP concept: you need to override every method, getter or setter. Use it when you want to define an API for many types (they have no common behaviors to share).
 - **mixin.** A way to "copy/paste" methods into classes: they reduce code duplication and centralize the code. The imported methods are also shared along the hierarchy.
- **External use.** These techniques are meant for "external use" because they cannot modify the class from the inside.
 - **extension.** They add functionalities to a class without changing its internal definition. They're generally used to "improve" a class without touching its internals.

5.3 The Object class

Even if it's not explicitly written in the code, any Dart class descends from `class Object {}` which is at the root of any hierarchy. This is the same structure that some popular languages such as Java, C# and Delphi adopt. Every class can, and should, override the methods declared in `Object`.

- **String `toString()`.** This method is very important and you should always override it because a string representation of an object is quite handy, especially while debugging. If you have it defined you can better handle the string conversion:

```
final f = Fraction(1, 2);
final s = "My fraction is $f";
```

String interpolation automatically calls `toString()` which makes the code shorter and more readable.

Chapter 5. Inheritance and Exceptions

- `bool operator ==(SomeClass other)`. When you want to compare two objects you have to use the `==` operator. By default it doesn't really work as you'd expect; it returns true only if two variables point to the same object.

```
class Example {
    int a;
    Example(this.a);
}

void main() {
    final ex1 = Example(2);
    final ex2 = ex1;
    print(ex1 == ex2); //true
}
```

The console outputs true because `ex2` points to the same reference held by `ex1`; both variables point to the same object so they're equal. By default the `==` operator compares references and doesn't look at the object itself; it only cares about what's being pointed to.

```
void main() {
    final ex1 = Example(2);
    final ex2 = Example(2);
    print(ex1 == ex2); //false
}
```

Logically we'd think they were equal but the compiler tells us another story. `ex1` and `ex2` point to two different objects that are logically equal but practically different because each variable holds a different **reference**. You have to override the equality operator to get the desired behavior:

```
class Example {
    int a;
    Example(this.a);

    @override
    bool operator ==(Object other) {
        // 1.
        if (identical(this, other))
            return true;
```

```
// 2.  
if (other is Example) {  
    final example = other;  
  
    // 3.  
    return runtimeType == example.runtimeType &&  
        a == example.a;  
} else {  
    return false;  
}  
}  
  
// 4.  
@override  
int get hashCode => a.hashCode;  
}
```

That's a lot of boilerplate code but you don't have any other choices. For sure if you come from the Java world you might have dealt with this exact same thing thousands of times.

1. The function `identical()` is provided by the Dart code API and checks if two objects have the same reference.
2. If the type of the compared object is equal to the type of the current object, a smart cast happens in the body to proceed with the comparison. Otherwise, `false` is returned.
3. Check if the runtime types are the same and then make a one by one comparison for every instance variable.
4. See the next point.

Those two overrides allow a proper object comparison as you'd expect. Running this example would output `true` because we've taught `operator==` to look at the object itself and not only at the references.

```
void main() {  
    final ex1 = Example(2);  
    final ex2 = Example(2);  
    print(ex1 == ex2); // true -> that's what we wanted!  
}
```

- `int get hashCode`. When you override the equality operator, you must always remember



to also override `hashCode`. The hash code is useful when you call the `identical()` method and also when you use "hash maps" we're going to analyze in the next chapter.

If you know what hash tables are, you are aware that the keys must not clash in order to keep good performances. Dart's implementation of hash tables guarantees O(1) in insertion if hash codes of different objects return different values (no collisions).

5.3.1 Comparable<T>

Other than overriding `operator==` and `hashCode`, if you wanted to do a precise and complete comparison setup of a class, you should also implement `Comparable<T>`. This is how a complete comparison logic for a `Fraction` class would look like:

```
class Fraction implements Comparable<Fraction> {
    final int num;
    final int den;
    const Fraction(this.num, this.den);

    double toDouble() => num / den;

    @override
    int compareTo(Fraction other) {
        if (toDouble() < other.toDouble()) return -1;
        if (toDouble() > other.toDouble()) return 1;
        return 0;
    }

    @override
    bool operator==(Object other) { ... };

    @override
    int get hashCode { ... };
}
```



`Comparable<T>` exposes the `compareTo()` method which can be used for ordering and sorting. The returned `int` value must follow this convention:

- if this instance is naturally `<` than the other, return a negative number;
- if this instance is naturally `>` than the other, return a positive number;

- if the two instances are equal, return 0.

Returning 1 and -1 is just a convention; it can be any positive or negative number. In our case, we have done the following:

- a fraction is naturally $<$ than another if its double representation is smaller than the other. For this reason, we return -1 in case the num / den ratio of the current instance were $<$ than the other.

```
if (toDouble() < other.toDouble()) return -1;
```

- a fraction is naturally $>$ than another if its double representation is greater than the other. For this reason, we return 1 in case the num / den ratio of the current instance were $>$ than the other.

```
if (toDouble() > other.toDouble()) return 1;
```

- two fractions are equal if the rational number they represent is also equal. For this reason, we return 0 if none of the above statements passed.

```
return 0;
```

In general, comparing floating point numbers using $<$, \leq , $>$ and \geq is safe. Comparing floating point values with `operator==` is very error prone; two values that should be equal may not be due to arithmetic rounding errors. This is the reason why we use `return 0` as "fallback" rather than checking for equality of double values.

5.4 Exceptions

Dart code can throw exceptions to signal that an unexpected or erroneous behavior has happened during the execution. When you throw an exception you should really catch it otherwise your program will forcefully terminate with an error code.

```
class Fraction {
    int _numerator;
    int _denominator;

    Fraction(this._numerator, this._denominator) {
        if (_denominator == 0) {
            throw IntegerDivisionByZeroException();
        }
    }
}
```

```
}
```

The denominator of a fraction cannot be zero and in our example the constructor is the best place to throw an exception. It signals that something's gone wrong, in this case an invalid parameter. You can only throw **objects**.

- ➊ The `IntegerDivisionByZeroException()` class is provided by Dart and it should be used when... a division by zero happens! Despite you can throw any object, classes with meaningful names should be preferred.

```
class RandomClass() {}

void main() {
    // It compiles. It's perfectly fine
    throw RandomClass();
}
```

You could also throw a single string like `throw "Whoops";` because `String` is a Dart class.

There are a few interchangeable classes that can be thrown in exceptions. For sure, if you have a format error you're not going to throw a `TimeoutException`.

- `DeferredLoadException`
- `FormatException`
- `IntegerDivisionByZeroException`
- `IOException`
- `IsolateSpawnException`
- `TimeoutException`

You can throw everything except for `null`.

5.4.1 on and catch

If you don't catch an exception, your program forcefully terminates but you never want to be in such situation. It's always a good idea handling exceptions by catching them.

```
void main() {
  try {
    final f = Fraction(1, 0);
  } on IntegerDivisionByZeroException {
    print("Ouch! Division by zero!");
  }
}
```

This code is safe because the exception raised by the constructor of `Fraction` is caught by a `try` block. If you want to get an instance of the thrown exception, just add a `catch` statement:

```
void main() {
  try {
    final f = Fraction(1, 0);
  } on IntegerDivisionByZeroException catch (exc) {
    // use the exc object
    doSomething(exc);

    print("Ouch! Division by zero!");
  }
}
```

`exc` is an instance of the thrown object that can only be used within the scope of `on`. This approach can be useful when there's the need to print a stack trace or any useful debugging info about the exception.

```
void main() {
  try {
    final f = Fraction(1, 0);
  } on IntegerDivisionByZeroException {
    print("Division by zero!");
  } on FormatException {
    print("Invalid format!");
  }
}
```

You can handle code that could throw more than a single exception by specifying multiple `catch` clauses. The first clause that matches the thrown object will be picked to handle the error propagation. What if the above code threw an `IOException` or another object you haven't handled? The program would crash because there would be a not handled exception.

```

void main() {
  try {
    final f = Fraction(1, 0);
  } on IntegerDivisionByZeroException {
    print("Division by zero!");
  } on FormatException {
    print("Invalid format!");
  } catch (e) {
    // You arrive here if the thrown exception is neither
    // IntegerDivisionByZeroException or FormatException
    print("General error: $e");
  }
}
    
```

The final `catch` is a fallback that captures everything else that didn't match the above types. You're guaranteed that this code is safe because, worst case, the two `on` will be skipped and the execution will fall inside the final `catch`.

5.4.2 finally

The `try - catch` block protects your code against runtime exceptions that alter the normal execution flow. As you know, if no exceptions occur nothing happens because the `catch` block doesn't execute.

```

void main() {
  try {
    final f = Fraction(2, 3);
  } catch (e) {
    print("Error");
  }

  print("Finish");
}
    
```

This simple code only outputs `"Finish"` because no exceptions have been thrown. However, there is a way to force the execution of a part of the `try` statement.

```

void main() {
  try {
    final f = Fraction(2, 3);
    
```

```
    } catch (e) {
        print("Error");
    } finally {
        print("Always here");
    }

    print("Finish");
}
```

The body of a `finally` block is **always** executed, no matter if an exception occurs or not. The above code prints "Always here" and then "Finish" on a new line.

- The code inside `on` or `catch` is executed only if an exception is thrown and a valid handler for the object is provided. If no exceptions are thrown, nothing happens.
- The code inside a `finally` block is always executed regardless the fact that an exception object is thrown or not.

5.4.3 Good practices

The Dart API provides the `Exception` interface which is implemented by all core library exceptions. Production-quality code usually throws subtypes implementing one of the following classes.

- **Exception.** Implement this interface to create errors that will be caught when something goes wrong. For example, doing this is a good idea:

```
class FractionDivisionByZero implements Exception {
    final String message;
    const FractionDivisionByZero(this.message);

    @override
    String toString() => message;
}
```

We can throw the `class FractionDivisionByZero` specific object rather than a general purpose `IntegerDivisionByZeroException`. When you have to throw an object, implement `Exception` and choose a proper name that describes what's gone wrong.

- ➊ The official documentation ² suggests to not use `throw Exception("...");`. Yes, you could, but it's bad because it doesn't give a precise type that can be caught with the `on` keyword.

- **Error.** You should subclass this type for all those programmer-relates errors such as going out from the bounds of a list or having an assertion that evaluates to false. `Error` is thrown in case of an unexpected program flow and you should **NOT** catch it.

```
class Example {
    int x;
    Example(this.x) : assert(x != 0);
}
```

In debug mode, if `x` is 0 an `AssertionError` is thrown. You shouldn't catch it and that's perfectly fine because you, the developer, **must** see that kind of error: it signals you've coded something in the wrong way. The IDE will produce an alert when an object of type `Error` is being caught.

- ➋ In practice you're allowed to catch `Errors` but you shouldn't. This kind of exceptions signal that you, the programmer, failed something and you must see it because a fix is required!

`Error` is an alert for the developer telling him that something really bad happened, such as an index gone out of the bounds or a critical I/O issue.

The usage difference lies on the logical concept.

- You almost always want to `throw` an object that implements `Exception` because you are the owner of the code and you decide what's wrong. Anyone else going to use your code will catch your exception object in order to know that something went wrong.
- You want to `extend` (and not `implement`) `Error` if you're dealing with particular code that might cause I/O issues, lists out of bounds or other "particular" error that a programmer can do.

Put in other terms, exceptions are "logical errors" that can be fixed while errors are "language specific" mistakes that cause core problems in your code. Other recommendations taken from

²<https://api.dart.dev/stable/2.7.0/dart-core/Exception-class.html>

the official guidelines³ are:

1. In a `try` block you should use `on` to catch specific types and decide what to do according with the related error.

```
try {  
    // code...  
} catch (e) {  
    // handle...  
}
```

This code is really bad because it catches everything, including `Error` (or its subtypes) which should not be caught. What you've just seen is the "*Pokémon exception handling*" and it's not a joke! Even the official Dart doc mentions it.

- i** It actually makes sense: the Pokémons motto is "Gotta catch 'em all" because the goal of the game is capturing any Pokémons encountered during the journey. You don't want to be a Pokémons programmer who bravely catches any exception with no regards, don't you?

If you really need to "eat" as many exceptions as possible, use this version:

```
try {  
    // code...  
} on Exception catch (e) {  
    // handle...  
}
```

It's still not good but at least way better than before because `Error` is not captured and it's still free to propagate.

2. Use `on SomeException catch (e)` to implement a logic for any type of thrown object.
3. Don't "eat" exceptions. If you really need to do nothing when an exception occurs, print a message to the console or show a popup to notify your users that something's gone wrong.

```
try {  
    // Exception is 'eaten' because when caught, nothing happens  
} on Exception catch (e) {}
```

³<https://dart.dev/guides/language/effective-dart/usage#error-handling>

Chapter 5. Inheritance and Exceptions

4. If you implement `Exception`, append it at the end of your custom class name such as `FractionException` or `NumeratorNullException`. The same concept applies when you deal with `Error`.
5. Do not catch exceptions objects whose type is `Error`.
6. If you want to rethrow an exception, you should use `rethrow` rather than throwing the same object again with `throw`⁴.

```
try {  
  try {  
    throw FormatException();  
  } on Exception catch (e) {  
    print("$e");  
    rethrow;  
  }  
} catch (e2) {  
  print("$e2");  
}
```

```
try {  
  try {  
    throw FormatException();  
  } on Exception catch (e) {  
    print("$e");  
    throw e;  
  }  
} catch (e2) {  
  print("$e2");  
}
```

Both ways of rethrowing an exception are valid but you should prefer the usage of `rethrow` (on the left) because it preserves the original stack trace of the exception. `throw` instead resets the stack trace to the last thrown position.

⁴<https://dart.dev/guides/language/effective-dart/usage#do-use-rethrow-to-rethrow-a-caught-exception>

6 | Generics and Collections

6.1 Generic types

Nowadays a very common feature of languages is the support for parameterized programming, also known as *generic programming* in the Dart, Java, C# and Delphi world. The biggest advantages brought from this approach are type-safety and code reusability.

6.1.1 Introduction

The best way to show how generics can be incredibly useful is via example. In Flutter you'll use very often caches to store data coming from your device or from the internet. Pretend you had to store a complex object: the first idea popping in your mind could be:

```
class ComplexObjCache {  
    final _obj = ComplexObj();  
    // Constructor, getters, setters, methods...  
}
```

The next day you have the need for another cache, with the same structure, but it has to handle floating point numbers. Good, you already know what to do: copy-paste the previous class and change the types from `ComplexObj` to `double` using the IDE refactor tool.

```
class DoubleCache {  
    final _obj = 0.0;  
    // Constructor, getters, setters, methods...  
}
```

One week later you need the same cache also for Strings, integers, and other objects but things start to get complicated. You've copy-pasted classes because they have the exact same structure and methods but the only difference is the data type.

```
class ComplexObjCache { }
class ComplexObj2Cache { }
class ComplexObj3Cache { }
class StringCache { }
class IntegerCache { }
class DoubleCache { }
```

Massive code duplication is guaranteed to be a maintenance nightmare. With generics you can easily solve the problem using a *placeholder* type that will be evaluated at compile time. In general a single letter is enough, such as T or K, but there are no restrictions on the length (PIZZA would be valid for example).

```
class Cache<T> {
    final T _obj;
    // Constructor, getters, setters, methods...
}
```

Using the <T> notation you're telling the compiler that, at compile time, the letter T has to be substituted with the type given by the actual instance. In this way you have a single class allowed to work with multiple types and thus maintenance gets way easier.

- ❶ For example, in `final c = Cache<String>("");`; the letter T is substituted with `String` at compile time. Potential errors, such as bad type casts, produce a compilation error.

6.1.2 Type safety

Generics might remind you of `dynamic`, as it works with any type, and actually they could do the same things. For example, rather than `Cache<T>` you could have created a `Cache` with `dynamic` fields:

```
class Cache {
    dynamic _obj;
    dynamic get value => _obj;

    // Constructor, getters, setters, methods...
}
```

The biggest problem is that `dynamic` has **zero** type safety as it's intended to work with runtime casts. You'd have to deal with long series of `if` statements for each `dynamic` variable because

casts must be safe. The compiler and the IDE won't be able to help you with static analysis.

- i** Manually dealing with lots of casts and type checks is one of the worst maintenance nightmares, especially if you're in a complex architecture with tons of `dynamic` types. In addition, potential problems with types are spotted at runtime while with generics they're immediately caught by the compiler.

With generics, casts are not needed because the compiler ensures a "protection" against wrong types usages. The code is said to be *safe* because type errors are known at compile time and not at runtime.

```
class Cache<T> {  
    final T _obj;  
    T get value => _obj;  
    // Constructor, setters, methods...  
}  
  
void main() {  
    final cache = Cache<int>(20);  
    String value = cache.value; // Error!  
}
```

The compiler generates an error because you're trying to assign an integer value (`cache.value`) to a string type. You'll also get an alert from the IDE before compiling and the debugger will find this error immediately. In contrary:

```
// using 'class Cache {}', the non-generic one  
void main() {  
    final cache = Cache(20);  
    String value = cache.value; // Compiles, but it's wrong!  
}
```

Here you get no compilation errors because `dynamic` is evaluated at runtime. The IDE won't help you because it cannot predict the future and imagine that at runtime an `int` will be assigned to a `String`. Debugging might not be so easy and you're going to waste time.

- i** We've said *waste* because you could have saved that time with generics! The compiler would have been able to tell you in which line the error was but the usage

of `dynamic` defers everything at runtime, so no help at all.

As you may have guessed, using generics in place of `dynamic` is almost always the best choice you can make. When you try to write parameterized code, always consider generics first!

6.1.3 Usage

There's nothing new about hierarchies of one or more generic types; what we've already covered about classes and inheritances also applies here. They work in the same way as "regular" non-generic classes with the only difference that the type goes inside the diamonds `< >`.

```
abstract class Cache<T> {
    final T _obj;
    Cache(this._obj);

    T get value => _obj;
    void handle();
}

// 1.
class LocalCache<T> extends Cache<T> {
    LocalCache(T obj) : super(obj);
    void handle() {}
}

// 2.
class CloudCache<T, K> extends Cache<T> {
    CloudCache(T obj, K obj2) : super(obj);
    void handle() {}
}
```

There's the possibility to define a infinite number of generic types per class, they just need to be separated by commas in the diamond list (`< >`) with the letters inside. Generics also work with `extends` and `implements`.

1. A subclass must declare **at least** the same number of parameterized types defined by its superclass.
2. You cannot write `LocalCache<A> extends Cache<T> {}` because the compiler tries to pass

A to the superclass constructor which expects a T. The letters must match.

- You can declare how many parameters you want. In the above example T and K are separately treated even if they will be used to represent the same type. The usage of `CloudCache<int, int>` gives no problems because the compiler treats T and K as two different entities and it doesn't care about the type they represent.

There's the possibility to put constraints on the types if you don't want them be "anything". For example, if we wanted our caches to be used only with numbers, we could have done the following:

```
abstract class Cache<T extends num> { }

abstract class LocalCache<T extends num> extends Cache<T> { }
abstract class CloudCache<T extends num, K> extends Cache<T> { }
```

Remember that `int` and `double` are subclasses of `num`. With these changes, the generic type is allowed to be only a subtype of `num` and nothing else.

```
void main() {
    // OK. 'int' and 'double' are subclasses of
    // 'num' so this is allowed
    final local1 = LocalCache<int>(1);
    final local2 = LocalCache<double>(2.5);

    // NO. 'String' is not a subclass of 'num'
    // so this is NOT allowed
    final local3 = LocalCache<String>(3);
}
```

You could also use `LocalCache<num>(0)`; but superclasses of `num` are not allowed. In Dart methods can return generic types and/or have it as parameter, you just have to put the diamonds after the name as you'd do with classes.

```
// T used as return type and parameter
T printValue<T>(T val) {
    // ...
    return val;
}

// T user as parameter
```

```
void check<T>(T val) {  
    // ...  
    return val;  
}  
  
void main() {  
    // Diamond syntax uses the symbols '<' and '>'  
    final a = printValue<int>(1);  
    final b = printValue<String>("1");  
    check<double>(6.45);  
}
```



A non-generic class can have one or more generic methods inside:

```
class Example {  
    void doSomething<T>(T value) { ... }  
}
```

6.2 Collections

Dart implements the most common types of containers using generics so that you can take advantage of type safety without having to deal with specific implementations for each type. We've already seen all the benefits in the previous section so let's get started.

6.2.1 List

As you already know from chapter 2, the language doesn't have arrays but only lists. They're implemented with the `List<T>` generic container you've already seen many times up to now:

```
// The type of 'intList' is List<int>  
var intList = [2, 5, -8, 0, 1];  
// The type of 'stringList' is List<String>  
var stringList = ["a", "hello"];
```

Lists are 0-indexed and have many methods such as `add()`, `length`, `clear()` and so on; check out your IDE or the documentation ¹ for a complete list. Like with any other generic type, you have the "protection" of the compiler against bad types assignments:

¹<https://api.dart.dev/stable/2.7.0/dart-core>List-class.html>

```
final list = List<int>();
list.add("oops");
```

This is clearly a compilation error because you're trying to assign a string to a container that expects only integers. There is a very nice syntax shortcut allowing you to insert a series of values directly in the array:

```
// 'list1' contains [1, 2]
var list1 = [1, 2];
// 'list2' contains [-2, -1, 0, 1, 2]
var list2 = [-2, -1, 0, ...list1];
```

Given a list (`list1`), you can put every element inside another list using the **spread operator** (...). If you aren't sure about the non-nullability of the source, you can add a safety check with the **null-aware spread operator** (...?).

// this is null



```
List<int>? list1;

var list2 = [-2, -1, 0, ...list1]; // Error
var list3 = [-2, -1, 0, ...?list1]; // All good
```

There's the need to use ...? because elements are being added only if the source list is **not** null; if we had used ...`list1` we would have got an error. A list can also be initialized by specifying the type in the diamonds:

```
final list = const <int>[1, 3, 6, 7];
```

6.2.1.1 Collection statements

Starting from Dart 2.3, there's the possibility to use a very convenient syntax you might use some times in Flutter to add elements in a `ListView` (a scrollable container of UI items). You can put an `if` statement inside a list to decide whether it's the case to add or not an item.

```
final hasCoffee = true;

final jobs = [
  "Welder",
  "Race driver",
  "Journalist",
  if (hasCoffee) "Developer"
];
```

This list contains for sure the first 3 elements but "*Developer*" is going to be added only if `hasCoffee` is true. Putting an `if` statement with no parenthesis in a list decides whether the element has to be added or not.

```
const hasCoffee = true;

final jobs = const [
  "Welder",
  "Race driver",
  "Journalist",
  if (hasCoffee) "Developer"
];
```



If we declared `const hasCoffee` (rather than using `final`) then we could have also created a constant list of values. This is even better because the compiler could perform some optimizations at compile time. In a very similar way, you could also use a `for` loop:

```
final numbers = [
  0, 1, 2,
  for(var i = 3; i < 100; ++i) i
];
```

Like it happened with the `if`, the `for` statement adds a series of items to the list. It has no parenthesis or `return`: just write the values as they are. In this case, the list cannot be `const`.

6.2.1.2 Implementation

If you were to look at the online source code² you'd see that `List<E>` is defined as `abstract class` but objects can still be created in the "normal" way. You might be puzzled at this point: why can I instantiate an abstract class?

```
final howIsItPossible = List<int>();
```

As you know, an `abstract class` cannot be instantiated and that's always true but `List<E>` defines a series of factory constructors to do the "trick". We have already seen this pattern in 5.1.5. This is the actual implementation of `List<E>` in the Dart SDK.

```
abstract class List<E> implements EfficientLengthIterable<E> {
  // code...
```

²<https://github.com/dart-lang/sdk/blob/master/sdk/lib/core/list.dart>

```
    external factory List([int? length]);  
    // other code...  
}
```

Thanks to this, when you write `howIsItPossible = List<int>();` you're actually calling the factory constructor which returns a **concrete** implementation of `List<E>`. You cannot see it because of the `external` keyword, which loads the body of the constructor from the Dart VM, but the factory returns a concrete (non-abstract) class.

- ❶ The `external` keyword is used to interact with the Dart VM as it loads specific pieces of code. You'll **never** use it. It loads the body of a function directly from the internals of the SDK, which is almost always C++ code. So, instead of having a `factory` with a body...

```
factory List([int? length]) {  
    // Dart code here...  
}
```

... the definition is placed somewhere else. In other words, `external` is used to say "the body of this function is not here, in the Dart file, but it's located somewhere in the VM."

```
// Thanks to 'external,' the body is not directly loaded from here,  
// the .dart file, but it's taken from somewhere else  
external factory List([int? length]);
```

Unless you're a developer from the Dart team, this keyword is not for you.

Because of `factory List([int? length])` it **seems** you're instantiating an abstract class from your code but no, in reality you're calling a factory constructor returning a concrete instance. With this pattern, `List<T>` returns a default implementation which is generally good for most of the use cases. The subclasses returned by the factory can be:

- **Growable.** You can increase and decrease the length of the list with no restrictions. A growable list has no `length` parameter in the constructor.

```
// No int param in the parenthesis -> no length specified  
// so this list is growable  
final growable = List<int>();  
growable.length = 5;
```

- **Fixed.** You can **not** increase and decrease the length of the list with no restrictions. A fixed list has the length parameter in the constructor.

```
// int param in the parenthesis -> lenght is specified
// so this list is fixed
final growable = List<int>(3);
growable.length = 5; // Error!
```

In general creating a growable list with `List<int>()` is good but if you don't need to change the size of the container go for `List<int>(10)`. There are other useful factory constructors from this class:

- `filled()` It creates a list of the given length and initializes each position with the elements you want.

```
final example = List<int>.filled(5, 1, growable: true);
// Now example has this content: [1, 1, 1, 1, 1]
```

- `unmodifiable()` It returns a copy of the given list in which you cannot call `add`, `remove` or other methods that would modify its content.

```
var example = List<int>.unmodifiable([1,2,3]);
example.add(4); // Runtime error
```

- `generate()` Creates a list of the given length and fills each position according with the value returned by the generator. This example fills the array with the power of each number.

```
var example = List<int>.generate(5, (int i) => i*i);
// Now example has this content: [0, 1, 4, 9, 16]
```

Note that `List.unmodifiable()` creates a new **copy** of the list; if you just want to make an unmodifiable list without having to copy the contents, use `UnmodifiableListView<T>`. They both create two unmodifiable lists but in the second case, no copies are made (it's just a "wrapper" around a `List<T>` object).

6.2.2 Set

A `Set<T>` is a generic container in which there cannot be duplicate objects. Sets can be directly initialized using braces without having to create a new instance and then add elements with a loop.

```
final keys = {1, 2, 3, 3, 4, 5};
```

```
for(var key in keys)
    print(key);
```

The console will output `1 2 3 4 5` and not `1 2 3 3 4 5` because duplicates are not added (no exceptions are thrown). Elements can be inserted by using functions like `add()` or `addAll()`; for any possible method on `Set<T>` you should check the official documentation³.

- ➊ When creating an empty set, there's the need to use the diamonds to specify the type. Otherwise, avoid using type inference and annotate the type directly:

```
// 1. Direct type annotation
Set<int> emptySet = {};
// 2. Type inference with diamonds
final emptySet = <int>{};

// 3. This is a Map, not a set!!
final emptySet = {};
```

Only in cases 1 and 2 you're creating an empty set. In 3 the compiler considers the `final emptySet = {}` like if it were a map of `dynamic` key/value pair.

`Set<T>` supports the `...` and `...?` operators we've seen earlier with lists. You can also use `if` and `for` collection statements.

- Adding items in a `Set<T>` is pretty easy with `add()` but if you want to insert a series of values, you can also pass an array.

```
final example = <int>{};
example.addAll([5, 3, 7]);
```

- Use `bool contains(T value)`; to check whether an element is in the set or not.
- Remove elements with `remove()`.

If you use the factory `Set<T>.unmodifiable(Set<T>> other)` you get an instance of the same set with no possibility to add/remove values. There's no `UnmodifiableSetView<T>`.

³<https://api.dart.dev/stable/2.7.0/dart-core/Set-class.html>

6.2.2.1 Implementation

In Dart sets are `abstract class Set<E> extends EfficientLengthIterable<E>` and, like we've already seen with lists, they use factory constructors to return concrete instances. The implementation is easier to understand because there is no `external` keyword.

```
abstract class Set<E> extends EfficientLengthIterable<E> {
    // code...
    factory Set() = LinkedHashSet<E>;
    // code...
}
```

When we do something like `final s = Set<int>();` we're not instantiating the abstract class, which is impossible, but we're calling `factory Set()` which returns a concrete class. These lines are equivalent...

```
final set1 = Set<int>();
final set2 = LinkedHashSet<int>();
```

... because in Dart a `Set` is implemented by default as a `LinkedHashSet`. This class is a hash-based implementation of a `Set` and it keeps track of the order in which elements have been inserted.

6.2.3 Map

Also known as *dictionary*, a `Map<K,V>` is an unordered generic collection that stores key-value pairs. You can retrieve the object you're looking for by using the associated key.

```
final m = <int, String>{
    0: "A",
    1: "B",
    2: "C"
};
```

`Map.unmodifiable()` creates an unmodifiable `copy` of the map while `UnmodifiableMapView()` just takes a reference (no copies are made). There are two ways to insert new pairs in the map:

- Use the `putIfAbsent()` method to insert a new pair of values only if the key is not already in the list.

```
final example = <int, String>{
```

```

    0: "A",
    1: "B"
};

// The key '0' is already present, "C" not added
example.putIfAbsent(0, () => "C");
// The key '6' is not present, "C" successfully added
example.putIfAbsent(6, () => "C");

```

The first parameter is the desired key while the second one is a no-param function returning the value to be inserted.

- Use brackets [] to add an element at the given index without checking if the key is already in the collection.

```

final example = <int, String>{
    0: "A",
    1: "B"
};

// "A" has '0' as key and it's replaced with "C".
// Now the map contains {0: "C", 1: "B"}
example[0] = "C";
// The key '6' is not present, "C" gets added
example[6] = "C";

```

Maps doesn't allow duplicate keys. When we do `example[0] = "C"` there are no errors because the element with key 0 is updated with the new value.

Retrieving a value from a map is fairly simple: `var v = example[0];`. If you pass a key that's **not** in the map, a `null` reference is returned.

```

final example = <int, String>{
    0: "A",
    1: "B"
};

final ex1 = example[0]; // ex1 = "A"
final ex1 = example[8]; // ex1 = null

```

In general it'd be better using `bool containsKey(T key);` before accessing the item as it tells

whether the key is present or not in the map. The removal of an element simply happens with the `remove` method.

6.2.3.1 Implementation

You've already seen how lists and sets work and maps are no different; the core library defines an abstract class with a factory constructor that returns a concrete instance. It's the usual pattern adopted by the Dart API for collections.

```
abstract class Map<K, V> {  
  // code...  
  external factory Map();  
  // code...  
}
```

The Dart VM loads the body of the constructor from its internals thanks to the `external` keyword. If you could read the code, you'd see that the constructor returns a concrete instance of a class called `LinkedHashMap`. There are three types of maps you can use.

1. `LinkedHashMap<K, V>`. It's the default class returned by the factory constructor of `Map` so when you use `final map = <int, int>{}` you get an instance of `LinkedHashMap`. It's based on a hash-table, the insertion order is remembered and you can iterate over key/values.
2. `HashMap<K, V>`. It's based on a hash table and `null` can be a valid key. The insertion order is **not** remembered so if you iterate over key/value pairs you won't get them in the order in which they have been added.
3. `SplayTreeMap<K, V>`. It's based on a self-balancing BST and keys are compared with the `comparator` function you're asked to pass in the constructor. If the comparison function is not passed, the compiler assumes that keys implement the `Comparable<T>`⁴ interface.

The first implementation, the default one, guarantees the insertion order, the second one doesn't care about the order and the last keeps the keys sorted. There is not the best implementation because you have to choose one of them according with what you have to do; in general the default is good.

⁴<https://api.dart.dev/stable/2.7.0/dart-core/Comparable-class.html>

6.3 Good practices

You've seen that the language offers three important categories of containers and in general you should use them with their default implementations. A "default" map for example is generally `final myMap = <int, int>{};`; but if you need to keep the keys sorted, go for a `SplayTreeMap`. From the official documentation ⁵:

- Use the literal initialization syntax when you're good with the default implementation given by the language. Instead, for specific implementations, use regular constructors or factories.

```
final example = List<int>();    // Bad
final example = <int>[];        // Good
```

- When you iterate on a container prefer doing `for(item in list) {}` instead of using the `forEach()` method which adds verbosity to the code. The loop is clearer. However, if you have a function that can be referenced such as `print`, you can use `list.forEach(print);`.
- There's a function called `cast()` we've never discussed because it's not good to use. Don't do casts with `cast<T>()` because it adds verbosity and it actually doesn't do a neat work. Ditch it.

Each container offers a series of factories, such as `unmodifiable()` to return a collection with no add/remove operations (it's a read-only container). Refer to the docs ⁶ for a complete list of utilities methods.

6.3.1 operator== and hashCode

You already know from 5.3 "The Object class" how to properly override the equality operator AND the `hashCode` property. Sets and maps heavily use comparisons and the hash code of a given object so you really want to do a good override.

i Very shortly, imagine a *hash table* as a table with two columns: the key is on the left and the value is on the right. The key is needed to search values because, if present, you'll get access to the associated object: that's why maps are key/value pairs.

⁵<https://dart.dev/guides/language/effective-dart/usage#collections>

⁶<https://api.dart.dev/stable/2.7.0/dart-core/dart-core-library.html>

Common implementations of maps and sets are based on hash tables and the hash code determines how an item should be stored in the container. You should always override the equality operator and `hashCode` in your classes.

- `bool operator==(Object other)`. We've already seen how to override it in 5.3.
- `int get hashCode`. There are multiple ways to correctly override this getter but what's important is that a different integer is returned for every different value.

```
class Test {  
    final int a;  
    final int b;  
    final String c;  
    Test(this.a, this.b, this.c);  
  
    bool operator==(Object other) {...}  
  
    int get hashCode {  
        const prime = 31;  
        var result = 1;  
  
        result = prime * result + a.hashCode;  
        result = prime * result + b.hashCode;  
        return prime * result + c.hashCode;  
    }  
}
```

In general, it's a good practice taking every instance variable of the class and perform a series of multiplications with prime numbers. In this way, every time you create an instance of the same object, such as `Test(0, 1, "a")`; you'll always get the same hash code. Any other object, will return a different value.

If you had a class with a lot of member variables, there would be for sure a lot of boilerplate code. The `Equatable`⁷ package by Felix Angelov overrides `operator==` and `hashCode` in your class automatically so that you won't have to deal with multiplications and comparisons.

```
class Test extends Equatable {  
    final int a;  
    final int b;
```

⁷<https://pub.dev/packages/equatable>

```
final String c;
Test(this.a, this.b, this.c);

@Override
List<Object> get props => [a, b, c];
}
```

You just need to subclass `Equatable` and override the `props` getter passing it every `final` field of your class. The package does nothing special internally: it overrides `operator==` (with `identical()`) and `hashCode` (with a series of XORs similarly to what we did). It's much less code for you to write!

```
class Test extends SomeClass with EquatableMixin {
    final int a;
    final int b;
    final String c;
    Test(this.a, this.b, this.c);
}

@Override
List<Object> get props => [a, b, c];
}
```



Extending `Equatable` might not always be possible because, for example, your class might already have a superclass and Dart doesn't allow multiple inheritance. In this case, use a `mixin` which does the same work.

 If your class is **NOT** immutable, because not every instance field is `final`, do **NOT** override `operator==` and `hashCode`. Overriding `hashCode` with a mutable object could break hash-based collections. This is also written in the official Dart design guidelines ⁸.

If your class is mutable, do not define a custom equality logic because it could break hash-based collections; for the same reasons, do not use `Equatable` if your class is not immutable.

⁸<https://dart.dev/guides/language/effective-dart/design#avoid-defining-custom-equality-for-mutable-classes>

6.3.2 Transform methods

Collections give you a very nice way to filter data and act on them with a series of methods that can be chained. There are similarities in Java with *streams* and in C# with *LINQ* queries.

```
void main() {  
  // Generate a list of 20 items using a factory  
  final list = List<int>.generate(20, (i) => i);  
  
  // Return a new list of even numbers  
  final List<String> other = list  
    .where((int value) => value % 2 == 0) // 1.  
    .map((int value) => value.toString()) // 2.  
    .toList(); // 3.  
  
}
```

In this example we're creating a list containing numbers from 0 to 19 using the `generate` factory constructor. The interesting part is how we've built `other` so that it contains only strings representing even numbers.

1. The `where()` method iterates across the entire collection and returns a boolean expression. Here we analyze each element of the list, represented by `int value`, and we discard it in case it's not even. This method is a "filter" that adds values only if the boolean expression returns `true`. 
2. The `map()` method transforms a type into another. Since `other` must be a list of strings, we transform each filtered element (represented by `int value`) into a `String value`.
3. Now that we have a filtered list of transformed values, the terminal function returns an instance of a list.

Manually doing this kind of operation is actually verbose because you should create a function with temporary variables, loops and conditional statements. This syntax instead is elegant and very easy to understand.

-  While you traverse a collection with these methods, do **NOT** alter the content of the container itself! If you have a list of strings, don't change each item calling, for example, `toUpperCase()` while you're in a `where` condition.

The example is shown with a list but these methods are also available for sets and maps. There are two kind of operations: "intermediate" operations to process data and "terminal" operations to return values. Intermediates are meant to elaborate data and can be chained while terminals are called at the end to "group" the data.

- **Intermediates.** This is a category of functions that can be chained like you've seen above to create complex expressions. Most of them accept a function whose parameter is the element of the collection being accessed.

- `where()`: goes through the entire list and discards elements that evaluate the condition to `false`.
- `map()`: transforms the element from a type to another.
- `skip()`: skips the first *n* elements of the source collection.
- `followedBy()`: concatenates this container with another one, passed as parameter.



- **Terminals.** This is a category of function that can only be called at the end of the chain to return a value or an object.

- `toList()/toSet()/toMap()`: gathers the elaborated data through the "pipes" and returns an instance of a list/set/map.
- `every()`: returns a boolean indicating if every element of the collection satisfies the given condition.
- `contains()`: returns `true` or `false` whether the collection contains or not the object you're looking for.
- `reduce()`: reduces a collection to a single value which can be the result of operations in the elements of the container. You cannot use `reduce()` on empty collections. For example:

```
final list = <int>[1, 2, 3, 4, 5];
final sum = list.reduce((int a, int b) => a + b);

print(sum); // 15
```

The variable `sum` contains the sum of the elements in the list since `reduce((a,b) => c)` takes 2 elements of the source (`a, b`) and performs the given action on them (in this case, it sums the values).

Chapter 6. Generics and Collections

- `fold()`. It's very similar to `reduce()` but it asks for an initial value and the returned type doesn't have to be the same of the collection.

```
final list = <int>[1, 2, 3, 4, 5];
final sum = list.fold(0, (int a, int b) => a + b);

print(sum); // 15
```

Both `reduce()` and `fold()` can do the same things but the latter is more powerful. First of all, `fold()` can define a custom initial value for the operations:

```
final list = <int>[1, 2, 3, 4, 5];

final sum1 = list.fold(0, (int a, int b) => a + b);
final sum2 = list.fold(5, (int a, int b) => a + b);

print(sum1); // 15
print(sum2); // 20
```

With `fold()` you can perform operations on different data types while with `reduce()` you cannot. In this example, we're computing the sum of the lengths of strings in a collection.

```
final list = ['hello', 'Dart', '!'];

final value = list.fold(0, (int count, String item) => count + item.length);
print(value); // 10
```

`count` has the same type of the initial value (0 in this case, which is an `int`) and `item` represents an object in the collection. The returned value of the function must match the type of the initial value. You can't do the same in the other way:

```
final list = ['hello', 'Dart', '!';

// It doesn't compile
list.reduce((String a, String b) => a.length + b.length);
print(value);
```

This version doesn't work because `reduce()` expects the return type of the callback to be a `String`, the same type of the container. With `fold()` you don't have this constrain: it will always work. In reality, `reduce()` can be seen as a shortcut of the following:

```
final withReduce = list.reduce(someCallback);
```

```
final withFold = list.skip(1).fold(list.first, someCallback);
```

The two versions are equivalent but `withReduce` is just shorter. We strongly encourage you to use this fluent syntax when you have to work on collections rather than using temporary variables and/or conditional statements.

7 | Asynchronous programming

7.1 Introduction

Nowadays computers and mobile devices are very fast and users are well aware of this; they hate when the application "freezes" for a moment or if it doesn't always react immediately to inputs. There are however some situations in which the user must wait:

- database operations;
- usage of the internet connection, which might be slow and thus the entire process could take longer than expected;
- many I/O operations might slow down your app's performances due to the policies adopted by the OS.

In Flutter for example you often use an internet connection and, in the worst case, the user has to wait a few seconds. You *must* use asynchronous programming to show something like an animated progress bar while, at the same time, data are processed in the background.

i We'll talk about this in the Flutter part but the idea is that the app should never stop at a single long task. Asynchronous programming is made for executing time-consuming operations in the background so that, in the meanwhile, we can do something else.

Let's say you're working in a team and a colleague of yours sends you this function which is going to be used very often in your application.

```
int processData(int param1, double param2) {  
    var value = 0;
```

```
for(var i = 0; i < param1; ++i) {
    for (var j = 0; j < param1*param2; j++) {
        // a lot of work here...
    }
}

return httpGetRequest(value);
}
```

Suppose that the two nested loops may take up to 2 seconds to complete and the network request at the end adds other hundreds of milliseconds. For sure this function is slow as it returns the `int` after quite a lot of time (in the order of seconds).

```
void main() {
    final data = processData(1, 2.5);
    print(data);

    print("Welcome to... Dart!");
}
```

The `main()` function is going to be "blocked" for some seconds due to the long execution time of `processData`. The entire flow is stuck due to a bottleneck produced by a function call and the app itself looks like it's frozen.

7.2 Futures

A `Future<T>` represents a value or an error that will be available in the future. This generic class should be used whenever you're working with time-consuming functions returning a result after a notable amount of time. Here's what you can do to easily slim your execution flow:

```
Future<int> processData(int param1, double param2) {
    var value = 0;

    for(var i = 0; i < param1; ++i) {
        for (var j = 0; j < param1*param2; j++) {
            // a lot of work here...
        }
    }
}
```

```

final res = httpGetRequest(value);
return Future<int>.value(res);
}

```

It's almost identical the original code: we've just changed the type from `int` to `Future<int>` and the final statement, which uses a named constructor of `Future` to return a new instance. Of course you must be sure that the `Future<T>.value()` object is built with the proper type.

- i** The code didn't change so much but there's a huge difference in how the function is going to be called. In addition, every time you see `Future<T>` as return value, you immediately figure out the usage and thus you write your code consequently.

Since the function now returns a `Future<T>` we have to treat it differently:

```

// Types are explicit for sake of simplicity
void main() {
    Future<int> val = processData(1, 2.5);
    val.then((result) => print(result));
}

```

The `then()` callback gets called once the execution has finished, when the value is ready to be used. Thanks to a `Future<T>` you're able to execute the time-consuming task in the background and be notified of the completion via `then()`.

```

// Types are explicit for sake of simplicity
void main() {
    Future<int> val = processData(1, 2.5);
    val.then((result) => print(result))
        .catchError((e) => print(e.message));
}

```

Methods can be chained; catching potential exceptions thrown during the background execution happens via `catchError()`, which is the equivalent of a `try catch` block. Of course you can create more complex chains such as:

```

val.then((result) => anotherFunction1(result))
    .then((another) => anotherFunction2(another))
    .then((ending) => anotherFunction3(ending))

```

```
.catchError((e) => print(e.message));
```

There are not limits, you can always append a `then()` or a `catchError()`. You might have noticed that this approach is quite verbose and it's not so easy to read when many methods are chained.

- ➊ That's the reason why `async` and `await` must be your primary choice; they drastically reduce the verbosity making the code look almost identical to its synchronous counterpart.

In certain cases, you might want to wait for a series of `Future<T>`s to complete but you still don't want to block the execution. This is the perfect use case for `Future.wait<T>()`.

```
Future<int> one = exampleOne();
Future<int> two = exampleTwo();
Future<int> three = exampleThree();

Future.wait<int>([
  one,
  two,
  three
]).then(...).catchError(...);
```

The `wait()` method takes a list of `Future<T>`s, executes them and waits until everyone has finished. You can chain `then()` and/or `catchError()` because `wait()` returns a `Future<T>`. The API is very rich of useful named constructors you can use:

- `Future<T>.delayed()`

```
final future = Future<int>.delayed(const Duration(seconds: 1), ()=> 1);
```

Creates a `Future<T>` object that starts running after the given delay (in this case, it executes after 1 second).

- `Future<T>.error()`

```
final future = Future<double>.error("Fail");
```

Creates a `Future<T>` object that terminates with an error. Other than the message, you can also pass as second parameter the stack trace.

- `Future<T>.value()`

```
final future = Future<String>.value("Flutter Complete Reference");
```

Creates a `Future<T>` object that completes immediately returning the given value. Basically, this constructor is used to "wrap" a non-future value into a future value.

- `Future<T>.sync()`

```
final future = Future<void>.sync(() => print("Called immediately"));
```

Creates a `Future<T>` object that immediately calls the given callback. Generally, when calling `then()` you don't know when its body will be executed. In this case, you know that the callback is called immediately. This constructor is intended to be used when a `Future<T>` has to execute immediately but in practice, there are a very few usages (we will see one in 13.2.2 for example).

7.2.1 Comparison

In this section we're comparing a synchronous code snippet, which uses a "simple" `int`, and its asynchronous version, which uses a `Future<int>`, to emphasize the different behaviors.

1. **Non-future version** (synchronous code).

```
int processData(int param1, double param2) {
    // takes 4 or 5 seconds to execute...
}

void main() {
    final data = processData(31, 2.5);
    print("func result = $data");

    print("Future is bright");
}
```

Nothing difficult to understand up to here, you can easily predict what the console is going to output (10 is just there as example, it doesn't matter):

```
func result = 10;
Future is bright
```

Both `print()` statements are executed in sequence (as usual) but they appear after a few seconds. There is a visible delay which temporarily "freezes" the program because

`processData()` blocks the execution flow while performing calculations.

2. Future version (asynchronous code).

```
Future<int> processData(int param1, double param2) {  
    // function that takes 4 or 5 seconds to execute...  
}  
  
void main() {  
    final process = processData(1, 2.5);  
    process.then((data) => print("result = $data"));  
  
    print("Future is bright");  
}
```

The output is now different:

```
Future is bright  
result = 10; // <-- printed after 4 or 5 seconds
```

With the usage of a `Future<T>` object the execution flow doesn't get blocked anymore. The `then(...)` callback returns **immediately** so that other operations can take place; its body will be executed later once data are actually ready. If you had written...

```
final process = processData(1, 2.5);  
process.then((data) {  
    print("result = $data");  
    print("Future is bright");  
});
```

... then the console would have printed ...

```
result = 10;  
Future is bright
```

... as in the first example. The body of `then()` executes synchronously in our example so operations are executed in sequence.

Inside a `then()` callback you can execute asynchronous code as well but it's difficult to read, due to the verbosity of the code, and hard to understand, in case there were too much asynchrony in the flow.

i To put it very simply, when you use `then()` you're telling Dart: "*Continue doing your work, I don't want to wait for the operation to finish. When the result will be ready, notify me with the callback*".

If you have to deal with time-consuming operations, using a `Future<T>` is basically a **must** because blocking the execution flow is dangerous and wrong. Thanks to asynchronous code you keep your app always busy and **responsive**, which is a fundamental user experience factor.

7.2.2 `async` and `await`

The usage of `async` and `await` makes the code less verbose and consequently easier to understand. It's just syntactic sugar to avoid the usage of `then()` to write callbacks:

- Using `then`.

```
void main() {
    final process = processData(1, 2.5);
    process.then((data) => print("result = $data"));
}
```

- Using `async` and `await`.

```
void main() async {
    final data = await processData(1, 2.5);
    print("result = $data")
}
```

The above snippets are **equivalent** because the result is the same but the syntax is different. Let's start with three very important facts:

1. You can use `await` only in a function marked with `async`.
2. To define an asynchronous functions, put the `async` keyword before the body.
3. You're allowed to call `await` only on a `Future<T>`.

Our `main()` has the `async` modifier so that we're allowed to call `await`. Calling `await` on a `Future` moves the execution to the background and proceeds once the computation is done (which is exactly what `then()` does). To be clear, writing...

```
processData(1, 2.5).then((data) => print("result = $data"));
```

... is the same as ...

```
final data = await processData(1, 2.5);
print("result = $data");
```

... because the lines **after** the `await` keyword are executed only when the `Future<T>` completed (without blocking). In regard to the previous example, this code...

```
void main() async {
    final data = await processData(1, 2.5);
    print("result = $data");
    print("Future is bright");
}
```

... is equivalent to ...

```
void main() {
    final process = processData(1, 2.5);
    process.then((data) {
        print("result = $data");
        print("Future is bright");
    });
}
```

... but absolutely **NOT** equivalent to ...

```
void main() {
    final process = processData(1, 2.5);
    process.then((data) {
        print("result = $data");
    });
    print("Future is bright");
}
```

because **everything** after `await` is executed only when the `Future` is completed. You're guaranteed that functions are asynchronously executed and you won't block the normal execution flow. Exceptions are also easier to catch because...

```
void main() {
    processData(1, 2.5)
        .then((result) => print(result))
        .catchError((e) => print(e.message));
```

Chapter 7. Asynchronous programming

```
}
```

... gets simplified with the usage of `async` and `await`:

```
void main() async {
    try {
        final result = await processData(1, 2.5);
        print(result);
    } on Exception catch (e) {
        print(e.message);
    }
}
```

The second version is closer to what you're used to see in the traditional synchronous world. Furthermore there are no nested methods/callbacks and thus the code is way shorter and more readable.

7.2.3 Good practices

The first thing stated by the official ¹ usage guidelines is "*prefer `async/await` over using raw `futures`*". Since asynchronous code can be hard to read and debug, you should prefer `async` and `await` over a chain of `then()` and `catchError()`.

```
Future<String> example() async {
    try {
        final String data = await httpGetRequest();
        final String other = await anotherRequest(data);
        return other;
    } on Something catch (e) {
        print(e.message);
        return "fail";
    }
}
```

If it weren't for `await` it would look like "normal" synchronous code with no callbacks at all. It's neater and more readable if compared to the following:

```
Future<String> example() {
    return httpGetRequest().then((data) {
```

¹<https://dart.dev/guides/language/effective-dart/usage#asynchrony>

```
        anotherRequest(data).then((otherData) {
            return otherData;
        });
    }).catchError((e) {
        print(e.message);
        return "";
    });
}
```

It's not a matter of efficiency or performances because both examples are fine. The problem is about writing code with potentially many nested callbacks and functions that become impossible to read (the so called "*callback hell*").

 Again, thanks to `await` asynchronous code can be written in the same way as synchronous code. Other than leading to less boilerplate, it's also easier to read, maintain and understand!



Returning a `Future<T>` from a function can be done via named constructor `Future.value()` or, more easily, by making the function `async` and returning the plain value. The compiler will make an automatic conversion.

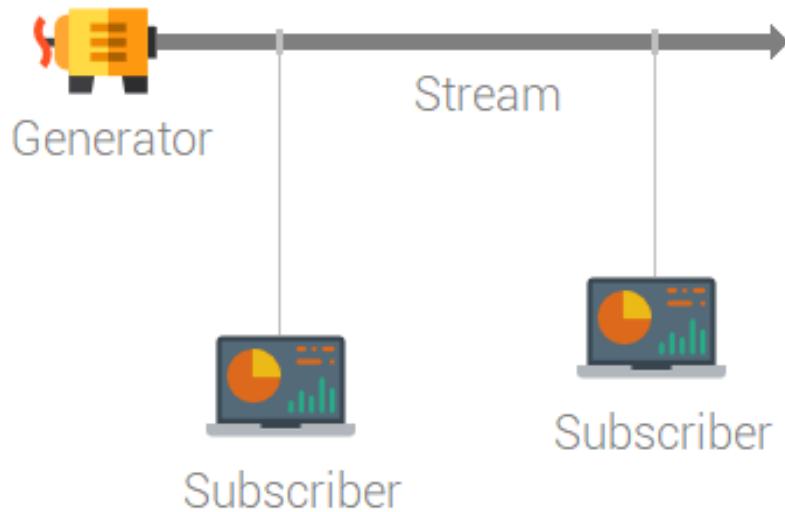
```
// Use the named constructor
Future<int> example() => Future<int>.value(3);

// Use async and the compiler wraps the value in a Future
Future<int> example() async => 3;
```

Both ways are valid but maybe you should prefer the second approach as it's a bit less verbose.

7.3 Streams

In Dart a *stream* is a sequence of asynchronous or synchronous events we can listen to. There's no need to check for updates because the stream notifies us automatically when there's a new event available.



This picture helps you to visualize how streams are intended to be used and who are the main actors involved. A *generator* is a source of information that lazily generates new data with a certain frequency; the *stream* is the pipe in which generated data flow.

- **Generator.** Creates new data and sends them over the stream.
- **Stream.** It's the place in which the generated data flow. You can start listening to a stream so that, when the generator emits new data, you will be notified.
- **Subscribers.** A subscriber is someone interested in the data travelling in the stream. If new data are sent over the stream by the generator, everyone listening (*subscribers*) will be notified.

A generator has to emit data only into a stream and nowhere else because a stream is the only reference for listeners to subscribe. There are two types of generators:

1. Asynchronous generators: they return a `Stream<T>` object. Because of this, you have to deal with an asynchronous flow of data that has to be handled by a subscriber with `await`.
2. Synchronous generators: they return an `Iterable<T>` object. Because of this, you have to deal with a synchronous flow of data that can be handled in a loop because data are sent in a sequential order.

A Flutter developer is used to work with `Stream<T>` because the framework has many asyn-

chronous generators. In practice, unless you're creating a package or a specific tool, you won't create generators too often but still you should at least be aware of how they generally work. Futures and streams are at the basics of Dart's asynchronous model.

7.3.1 Streams and generators

As a basic example, we're going to create an asynchronous generator producing 100 random numbers, one per second. In order to tell the compiler that this function is a generator, it has to be marked with the `async*` modifier.

```
Stream<int> randomNumbers() async* {           // 1.  
    final random = Random();  
  
    for(var i = 0; i < 100; ++i) {               // 2.  
        await Future.delayed(Duration(seconds: 1)); // 3.  
        yield random.nextInt(50) + 1;           // 4.  
    }  
}                                              // 5.
```

1. Since the function is a generator of asynchronous events (random numbers), the return type must be of type `Stream<T>`. The `async*` modifier allows the usage of `yield` to emit data.
2. The loop generates 100 random numbers.
3. The `Future.delayed(...)` named constructor creates a `Future` that returns after a certain delay given by the `Duration`² object. It's used to "sleep" the execution flow for a certain time without blocking.
4. The `yield` keyword pushes data on the stream. It is responsible of sending new events on the stream and it doesn't alter the loop (it continues to cycle regularly).
5. When a function has the `async*` modifier there **cannot** be a `return` statement. It would also be logically wrong because data are already sent over the stream by `yield` and thus you'd have nothing to return when the generator "turns off".

So generators are created with the `async*` modifier and events are emitted on the stream with the `yield` keyword. To make a comparison, here's the synchronous version of the generator which has a similar structure.

```
// contains the 'sleep' function  
import 'dart:io';
```

²See appendix A.3 to know more about `Duration`

```
Iterable<int> randomNumbers() sync* {
  final random = Random();

  for(var i = 0; i < 100; ++i) {
    sleep(Duration(seconds:1));
    yield random.nextInt(50) + 1;
  }
}
```

The `sync*` star modifier tells the compiler that this function is a synchronous generator. Due to its nature of being *synchronous* you **cannot** use futures and thus we must use `sleep()` instead of awaiting `Future.delayed()`.

i Asynchronous generators are meant to be used with asynchronous code. Synchronous generators are meant to be used with synchronous code. Intuitively, if your code needs to `await` something, you're going to need an asynchronous generator.

Both kind of generators start emitting data **on demand**, meaning that values are produced when a listener starts iterating on `Iterator<T>` or starts listening to `Stream<T>`. There can be more than a single `yield` statement in the same block and it would simply push many values, in sequence, on the stream:

```
Stream<int> randomNumbers() async* {
  final random = Random();

  for(var i = 0; i < 100; ++i) {
    await Future.delayed(Duration(seconds: 1));
    yield random.nextInt(50) + 1;
    yield random.nextInt(50) + 1;
    yield random.nextInt(50) + 1;
  }
}
```

This code emits 3 random number at each iteration so the stream is going to generate 300 values before completing. Streams can also be created using a series of useful named constructors for a "quick setup":

- `Stream<T>.periodic()`

```
final random = Random();

final stream = Stream<int>.periodic(
    const Duration(seconds: 2),
    (count) => random.nextInt(10)
);
```

Creates a new a stream that repeatedly emits events at the given Duration interval. The argument of the anonymous function starts at 0 and then increments by 1 for each event emitted (it's an "event counter").

- `Stream<T>.value()`

```
final stream = Stream<String>.value("Hello");
```

Creates a new stream that emits a single event before completing.

- `Stream<T>.error()`

```
Future<void> something(Stream<int> source) async {
    try {
        await for (final event in source) { ... }
    } on SomeException catch (e) {
        print("An error occurred: $e");
    }
}

// Pass the error object
something(Stream<int>.error("Whoops"));
```

Creates a new stream that emits a single error event before completing; the behavior is very similar to `Stream<T>.value()`.

- `Stream<T>.fromIterable()`

```
final stream = Stream<double>.fromIterable(const <double>[
    1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9
]);
```

Creates a new single-subscription stream that only emits the values in the list.

- `Stream<T>.fromFuture()`

```
final stream = Stream<double>.fromFuture(
    Future<double>.value(15.10)
);
```

Creates a new single-subscription stream from the given `Future<T>` object. In particular, when the `Future<T>` completes, 2 events are emitted: one with the data (or the error) and another to signal the stream has terminated (the "done" event).

- `Stream<T>.empty()`

```
final stream = Stream<double>.empty();
```

This kind of stream does nothing: it just immediately sends a "done" event in order to signal the termination.

You should really check out the `Stream<T>` online documentation for a complete reference of all the methods you can call on it. The most "popular" ones for example are:

- `drain(...)`: it discards all the events emitted by the stream but signals when it's either done or an error occurred;
- `map(...)`: transforms the events of the current stream into events of another type;
- `skip(int count)`: skips the first `count` event on the stream;

7.3.2 Subscribers

The generator is now ready to periodically emit new random numbers and thus we can subscribe to get notified for new values.

```
Stream<int> randomNumbers() async* {
    // see code above...
}

void main() async { // 1.
    final stream = randomNumbers(); // 2.

    await for (var value in stream) { // 3.
        print(value);
    }

    print("Async stream!"); // 4.
}
```

```
}
```

1. Since we're dealing with an asynchronous stream, there's the need to mark the function with `async` because we're going `await` soon.
2. We subscribe to the stream by simply getting a reference to it. This is the moment in which the generator starts emitting data because someone has just started listening (on-demand initialization).
3. When dealing with streams, the `await for` loop is able to "catch" values sent over the stream by a `yield` in the generator. It works exactly like a regular `for` loop.
4. The string is printed when the loop terminates so it will appear at the end.

To be precise, the example is *almost* good because generators are generally put inside classes and exposed with a getter, which returns a `Stream<T>` instance, or with a `listen()` method which does a manual subscription. Let's give a look at the synchronous version of the generator:

```
Iterable<int> randomNumbers() sync* {
    // see code above...
}

void main() {
    final stream = randomNumbers();

    for(var value in stream) {
        print(value);
    }

    print("Sync stream!");
}
```

Apart from a plain `for` instead of an `await for` the code is identical. The `List<T>` and `Set<T>` classes are both `Iterable<T>`, exactly like most types in the collection library. Exceptions are caught in the usual way:

```
void main() {
    final stream = randomNumbers();

    try {
        await for(var value in stream) {
            print(value);
        }
    } catch (e) {
        print("Error: $e");
    }
}
```

```
    }  
} on Something catch (e) {  
    print("Whoops :(");  
}  
}
```

Other than *data events* a stream can also send *error events*, which may occur because an exception has been thrown in the generator. Here's another example of a stream continuously sending data at a given interval:

```
Stream<int> counterStream([int maxCount = 10000]) async* {
    final delay = const Duration(seconds: 1);
    var count = 0;

    while (true) {
        if (count == maxCount) {
            break;
        }
        await Future.delayed(delay);
        yield ++count;
    }
}

void main() async {
    await for(var c in counterStream) {
        print(c);
    }
}
```

Each second a new number is printed to the console until `maxCount` is reached; if you wanted the loop to last forever, just remove the `if` condition. For an even more detailed coverage about streams, check out the official Flutter YouTube channel along with the documentation which is full of details and examples:

1. Stream usage: <https://dart.dev/tutorials/languagestreams>
 2. Generators: <https://dart.dev/articles/libraries/creating-streams>
 3. Stream docs: <https://api.dart.dev/stable/2.9.2/dart-async/Stream-class.html>

In Flutter the `StreamBuilder<T>` object is used to subscribe to a stream and we're going to use

it quite often, especially in Part III of the book.

7.3.3 Differences

Key differences for asynchronous and synchronous streams are summarized in this short table:

Asynchronous	Synchronous
Returns a <code>Stream<T></code>	Returns an <code>Iterable<T></code>
Mark function with <code>async*</code>	Mark function with <code>sync*</code>
Can use <code>await</code>	Cannot use <code>await</code>
Subscribers have to use <code>await for</code>	Subscribers have to use <code>for</code>

In both cases `yield` is used to send data on the stream and there must not be a `return` statement in the function. You might be wondering: "If I wanted to write a generator, should I make it synchronous or asynchronous?"

i As we've already said, you rarely need to write a generator if you work with Flutter, unless you're doing something specific. In most of the cases you will `subscribe` to streams so you won't have to decide anything because the generator is exposed by the library.

A reasonable answer to the question would be "*it depends*" because there are cases and cases. As a general guideline we can say that a good decisional aspect is whether you have to use `Future<T>` or not.

- If you have to deal with `Future<T>`s because of network usages or I/O operations for example, you are forced to use an asynchronous generator otherwise you can't use `await`.
- When you don't have to deal with asynchrony and you have the need of sending a series of sequential data, go for a synchronous stream.

Sometimes, especially when using the `flutter_bloc` library we will cover in chapter 11, it can be

Chapter 7. Asynchronous programming

useful splitting the logic of a `Stream<T>` into multiple pieces. This is the case where `yield*` is required:

```
Stream<int> numberGenerator(bool even) async* {
    if (even) {
        yield 0;
        yield* evenNumbersUpToTen();
        yield 0;
    } else {
        yield -1;
        yield* oddNumbersUpToTen();
        yield -1;
    }
}

Stream<int> evenNumbersUpToTen() async* { ... }
Stream<int> oddNumbersUpToTen() async* { ... }
```

Basically `yield*` is used to "pause" the execution and start emitting values from the other stream; once finished, the source stream is "restarted" so that it can regularly send its values again. To be more clear, here's an example of what happens when `numberGenerator(true)` is called:

- The value 0 is emitted with `yield`, the "normal" way of sending data on the stream.
- Because of `yield*`, `numberGenerator` pauses and starts emitting values generated from the other stream (`evenNumbersUpToTen`).
- Once `evenNumbersUpToTen` has completed, `numberGenerator` resumes and executes the next `yield` statement.

In practice `yield*` is used to say "stop here, emit values from the other stream and when it's completed you're free to restart your regular flow". This pattern is used when a stream should internally use another stream to split the logic in multiple functions to make the code more readable.

7.3.4 Using a controller

The examples we've shown so far aren't very useful actually. In particular, the creation of an "in place" stream hasn't much use cases a part from examples and demos. You're already familiar with this simple setup:

```
Stream<String> someStream() async* { ... }
```

```
void main() async {
    final stream = someStream();
}
```

The stream is started as soon as we do the `stream = someStream();` assignment. This is the manual way of creating streams but it doesn't scale well on larger applications. Especially in Flutter, you'll find out that `StreamController<T>` is a more convenient way to work with streams. We're going to create a more complex stream that periodically produces random numbers.

```
/// Exposes a stream that continuously generates random numbers
class RandomStream {
    /// The maximum random number to be generated
    final int maxValue;
    static final _random = Random();

    Timer? _timer;
    late int _currentCount;
    late StreamController<int> _controller;

    /// Handles a stream that continuously generates random numbers. Use
    /// [maxValue] to set the maximum random value to be generated.
    RandomStream({this.maxValue = 100}) {
        _currentCount = 0;
        _controller = StreamController<int>(
            onListen: _startStream,
            onResume: _startStream,
            onPause: _stopTimer,
            onCancel: _stopTimer
        );
    }

    /// A reference to the random number stream
    Stream<int> get stream => _controller.stream;

    // other methods coming soon...
}
```

Chapter 7. Asynchronous programming

Notice how we've used triple slashes (///) to document the code ³. The `Timer` class comes from the `dart:async` package: it's a count-down timer that can be configured to fire once or repeatedly. It counts down from the given duration up to 0 and then triggers the callback. It has two constructors:

- `Timer(Duration duration, void callback())`
Executes once the callback after the given duration.
- `Timer.periodic(Duration duration, void callback(Timer timer))`
The callback is invoked repeatedly with `duration` intervals.

A timer can be stopped with `cancel()`. We're using it to push each second new random numbers in the stream, handled by `StreamController<T>`. This class is basically a wrapper around a stream with many facilities to easily send data, errors and done events.

- `onListen`: this callback is called when the stream is listened to (new subscription made).
- `onCancel`: this callback is called when the stream is canceled (subscription canceled).
- `onPause`: this callback is called when the stream is paused (subscription paused).
- `onResume`: this callback is called when the stream is resumed (subscription resumed).

A `StreamController<T>` is a very handy tool to easily manage a `Stream<T>`. Thanks to `get stream` we expose to the outside a reference to the stream so that listeners can subscribe and receive events. This is how we've defined the callbacks of the controller:

```
void _startStream() {
    _timer = Timer.periodic(const Duration(seconds: 1), _runStream);
    _currentCount = 0;
}

void _stopTimer() {
    _timer?.cancel();
    _controller.close();
}

void _runStream(Timer timer) {
    _currentCount++;
    _controller.add(_random.nextInt(maxValue));
```

³More on documenting code in 23.1.2

```
    if (_currentCount == maxValue) {
        _stopTimer();
    }
}
```

We have declared `Timer? _timer` as a nullable variable because we cannot immediately initialize the timer in the constructor. Doing so would be an error because events would start being emitted on the stream from the beginning, even if there are **no** listeners!

```
RandomStream({this.maxValue = 100}) {
    _currentCount = 0;
    _controller = StreamController<int>(...);

    // WRONG! In this way, the timer is started and thus events are
    // emitted on the stream immediately (even if no one is listening)
    _timer = Timer.periodic(...);
}
```

We safely use the `?.` operator to access the nullable variable. Inside `_startStream()` we actually initialize the timer so that it pushes new random values every 1 second. The actual processing is done inside `_runStream()`:

```
// New value added to the stream. Listeners will be notified
_controller.add(_random.nextInt(maxValue));

// When the maximum value is reached, we need to stop both the
// timer AND close the controller to stop the stream.
if (_currentCount == maxValue) {
    _stopTimer();
}
```

We can now play with our `RandomStream` class. In this example, we're subscribing to the stream using `listen()` and then we cancel the subscription after a certain delay. You'll see that random numbers are printed to the console only 3 times in total.

```
void main() async {
    final stream = RandomStream().stream;
    await Future.delayed(const Duration(seconds: 2));

    // The timer inside our 'RandomStream' is started
    final subscription = stream.listen((int random) {
```

```
        print(random);
    });

    await Future.delayed(const Duration(milliseconds: 3200));
    subscription.cancel();
}
```

After 2 seconds, we subscribe to the stream using `listen()` but numbers are printed only three times because, 3 seconds later, we cancel the subscription. As you can see, `StreamController<T>` is more complex to use but more powerful and scalable: it's the preferred way to work with streams in Dart and Flutter.

7.4 Isolates

Many popular programming languages such as Java and C# have a very wide API to work with multiple threads and parallel computation. They can handle complex multithreading scenes thanks to the various primitives they support. Dart however has none of the following:

- there is no way to start multiple threads for heavy background computation;
- there is no equivalent, for example, of thread-safe types such as `AtomicInteger`;
- there are no mutexes, semaphores or other classes to prevent data races and all those problems arisen from multithreaded programming.

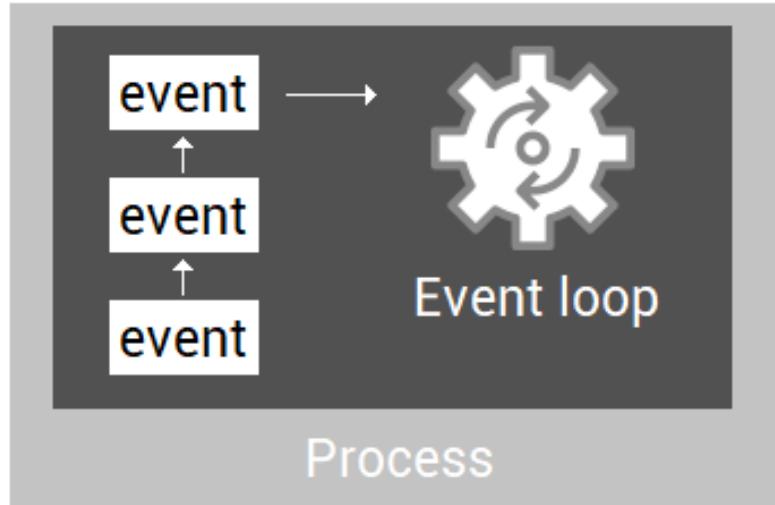
The Dart code (and thus Flutter applications) is run inside an `isolate` which has its own private area of memory and an event loop. An isolate can be seen as a special thread in which an event loop processes the instructions. If you aren't familiar with these concepts, we will break down for you what is going on:



Any program runs in a process which can be made up of one or more threads. Some programming languages (picture on the left) allow you to manually create multiple threads to execute long running tasks in the "background" not to block the UI.

- ➊ All the threads living on a process share the **same** memory. You need to be aware of this because writing the same data, at the same time, in the same memory area can lead to problematic situations known as *data races*.

In Dart, a process is made up of one or more **isolates** containing an event loop. Differently from classic threads, each isolate allocates its own memory area so there are no data sharing issues. In other words, the key difference is that threads **do** share the same memory while isolates **don't**. Thanks to this fact, Dart needs no data synchronization primitives since problems like data races can never happen by default. If we made a zoom on an isolate, it would look like this:



The white *event* rectangles can be anything from I/O disk operations, HTTP requests, actions triggered by a finger tap in the Flutter framework and so on. The gear on the right is the **event loop**, a sort of machinery that continuously executes events. You might be asking yourself: if there's only a single thread, how is asynchronous code executed? Let's take a look at those 2 simple examples:

1. Let's say that somewhere in our Dart program (or Flutter app) there are two methods which get called in sequence. They are synchronous, because inside they use no asynchronous code (no `Stream<T>`s or `Future<T>`s).

```
// this is called first
var json = myModel.readFromDisk();

// and this is called after the above
final result = computeIntegerValue();
```

The event loop processes incoming events in order one by one so first it executes the I/O operation and then the computation. Here's a visual representation of the situation:



The second event is processed only when the first is finished. If there were no events available, the event loop would be in "idle" waiting for new work to do. The event loop is the "engine" that actually executes the Dart code you've written.

- Let's now see another example in which a `Future<T>` is involved in order to understand how asynchronous code is processed. The same strategy is also applied when it comes to streams.

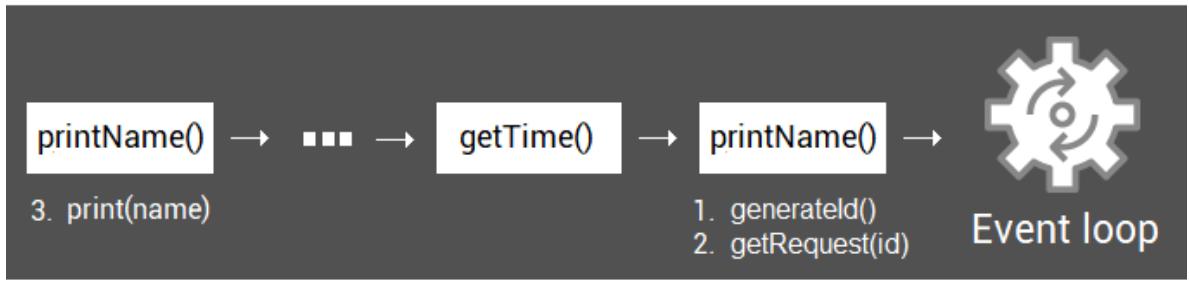
```
void printName() async {
    final int id = generateId();
    final String name = await HttpModel.getRequest(id);

    print(name);
}
```

As you already know, what comes after `await` is executed only when the `Future<T>` has terminated. In this case, the value will be printed only when the HTTP request is finished. Pretend to have this code:

```
printName();
final time = getTime();
```

The first event to be executed is `printName()` but since it internally calls `await`, what comes after (the `print(name)` statement) is separated and added later as a new event in the queue! This is the actual sequence that will be processed:



In practice, asynchronous calls are divided in multiple events: the synchronous part and the callbacks. What comes after an `await` is not executed immediately because there's the need to wait for the `Future<T>` to finish. In order to not waste time, the callback is divided from the event, "remembered" and added in the queue again later (when the `Future<T>` finished).

```
// This part is executed immediately; it's the rightmost rectangle on
// the image
final int id = generateId();
final String name = await HttpModel.getRequest(id);

// This callback is executed later; it's the leftmost rectangle on the
// image. This part "separated" and added later in the event loop again
// to complete the execution
print(name);
```



Splitting function calls is fundamental because it avoids events on the queue to wait for futures to finish. If `printName()` were executed entirely, the event loop would have been blocked until the `Future<T>` completed and other events would have to wait.

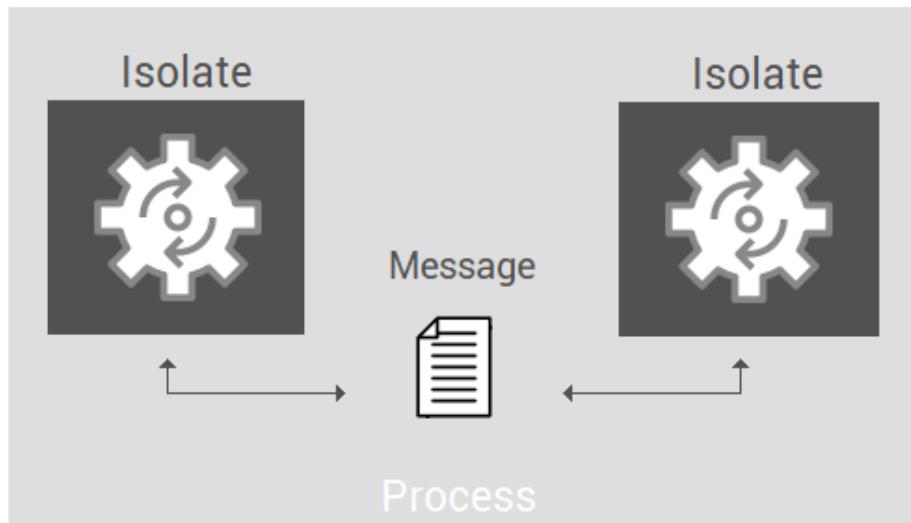
The event loop should always be busy but it shouldn't execute long-lived events otherwise others will be blocked. In addition, other than events fired by your app there are also other kind of actions to be performed such as garbage collection. To sum it up, here's a comparison with other programming languages:

- **Java or C#.** You can create multiple threads to run time-consuming work in the background. Threads share memory, which can be dangerous, but you have a rich API with mutexes, atomic types and so on to keep consistency in your program.
- **Dart.** There's only a single thread with its own memory. You cannot create multiple threads. The event loop processes anything sequentially as soon as possible. In order to

not waste time, asynchronous calls are split so that callbacks are executed in a second moment in order to not block the loop.

7.4.1 Multiple isolates and Flutter

A single Dart application can have more than a single isolate; you can create them by using `Isolate.spawn()` from the "`dart:isolate`" library. Isolates have their own event loop and memory area, there are no dependencies or shared components at all. The only way they have to communicate is via *messages*.



Each isolate has a port from which messages enter and exit; they are respectively represented by `ReceivePort` and `SendPort`. A message is regularly processed by the event loop as any other action but this really is the only way to communicate.

i There's the possibility to also spawn new isolates in Flutter but you'd have to use `Future<T> compute(...)` rather than `Isolate.spawn`.

Working with isolates is quite low level and it's something you generally don't do on a regular basis. Using `async/await` is almost always enough. To make a practical example, if you had a Flutter app with a really time-expensive data computing your frame rate might drop under

Chapter 7. Asynchronous programming

60fps. This is the case for a new isolate:

```
// Very computational-heavy task
int sumOfPrimes(int limit) {...}

// Function to be called in Flutter
Future<int> heavyCalculations() {
    return compute<int, int>(sumOfPrimes, 50000);
}
```

The `compute()` method requires the function to be executed (which cannot be an anonymous function) and the parameters it needs (if any). If you had the need for multiple input parameters, simply wrap them into a model class and pass it as a dependency, like this:

```
// Model class
class PrimeParams {
    final int limit;
    final double another;
    const PrimeParams(this.limit, this.another);
}

// Use the model as parameter
int sumOfPrimes(PrimeParams data) {
    final limit = data.limit;
    final another = data.another;
    ...
}

// Function to be called in Flutter
Future<int> heavyCalculations() {
    final params = PrimeParams(50000, 10.5);
    return compute<PrimeParams, int>(sumOfPrimes, params);
}
```

In `compute<Q,R>` the parameters are defined as follows: `Q` is the type of the parameter needed by the function and `R` is the return type. In chapter 16 we will discuss when it's convenient using separated isolates in Flutter applications to optimize performances.

8 | Coding principles with Dart

This chapter is a big "good practice" section as it contains some well-known suggestions from the OOP world. We'd love to also talk about design patterns, TDD, clean code and much more but these contents go beyond the scope of this book. Many people have written books and articles on these topics, we recommend you read up on these for more in-depth details.

- Design patterns are a series reusable solutions to common, well-known problems. The original concept came by a group of four people, called *Gang of four*, but nowadays new patterns come out in parallel with the evolution of languages.
 - Look for any recent book or resource that includes the widest range of patterns. They apply to any programming language; the programming language in which they're explained is not so relevant.
- TDD, abbreviation of **T**est **D**riven **D**evelopment, is a programming style strongly centered on code testing. You have to first write the tests, cover every possible case and only after this process you can start coding.
 - <https://resocoder.com/flutter-clean-architecture-tdd>
- DDD, abbreviation of **D**omain **D**riven **D**esign, is a programming style which focuses on code maintainability and separation of concerns. We recommend to follow Reso Coder's DDD course which gives a step-by-step explanation about DDD using Dart and Flutter.
 - <https://resocoder.com/flutter-firebase-ddd-course>
- Dart has a very wide, user-friendly documentation in which you can find examples for almost any topic. It's a wide growing resource that tells you how to properly write Dart code through good practices and articles.
 - <https://dart.dev/guides/language/effective-dart>
 - <https://dart.dev/tutorials>

That said, you can of course completely skip this part since it has no core Dart or Flutter concepts but we encourage you to at least know what SOLID and DI are about. These concepts are valid regardless the programming language in which they're applied.

8.1 SOLID principles

The term *SOLID* should actually be written as *S.O.L.I.D.* because it's an acronym for 5 design principles, one for each letter, which help the programmer writing maintainable and flexible code.

8.1.1 Single Responsibility Principle

Very intuitively, this principle (abbreviated with SRP) states that a class should only have a single responsibility so that it could change for one reason and no more. In other words, you should create classes dealing with a single duty so that they're easier to maintain and harder to break.

```
class Shapes {
  List<String> cache = List<>();

  // Calculations
  double squareArea(double l) { /* ... */ }
  double circleArea(double r) { /* ... */ }
  double triangleArea(double b, double h) { /* ... */ }

  // Paint to the screen
  void paintSquare(Canvas c) { /* ... */ }
  void paintCircle(Canvas c) { /* ... */ }
  void paintTriangle(Canvas c) { /* ... */ }

  // GET requests
  String wikiArticle(String figure) { /* ... */ }
  void _cacheElements(String text) { /* ... */ }
}
```

This class totally destroys the SRP as it handles internet requests, painting and calculations all in one place. You'll have to make changes very often to `Shape` because it has many duties, all in one place; maintenance for this class is not going to be pleasant. What about this?

```
// Calculations and logic
abstract class Shape {
    double area();
}

class Square extends Shape {}
class Circle extends Shape {}
class Rectangle extends Shape {}

// UI painting
class ShapePainter {}

// Networking
class ShapesOnline {}
```

There are 3 separated classes focusing on a single task to accomplish: they are easier to read, test, maintain and understand. With this approach the attention of the developer is focused on a certain area of interest (such as mathematical calculations on `Shape`) rather than on a messy collection of methods, each with different purposes.

8.1.2 Open closed principle

The open closed principle states that in a good architecture you should be able to add new behaviors without modifying the existing source code. This concept is notoriously described with the sentence "*software entities should be open for extensions but closed for modifications*". Look at this example:

```
class Rectangle {
    final double width;
    final double height;
    Rectangle(this.width, this.height);
}

class Circle {
    final double radius;
    Rectangle(this.radius);

    double get PI => 3.1415;
}
```

```

class AreaCalculator {
    double calculate(Object shape) {
        if (shape is Rectangle) {
            // Smart cast
            return r.width * r.height;
        } else {
            final c = shape as Circle;
            return c.radius * c.radius * c.PI;
        }
    }
}
    
```

Both `Rectangle` and `Circle` respect the SRP as they only have a single responsibility (which is representing a single geometrical shape). The problem is inside `AreaCalculator` because if we added other shapes, we would have to edit the code to add more `if` conditions.

```

class Rectangle {...}
class Circle {...}
class Triangle {...}
class Rhombus {...}
class Trapezoid {...}

class AreaCalculator {
    double calculate(Object shape) {
        if (shape is Rectangle) {
            // code for Rectangle...
        } else if (shape is Circle) {
            // code for Circle...
        } else if (shape is Triangle) {
            // code for Triangle...
        } else if (shape is Rhombus) {
            // code for Rhombus...
        } else {
            //code for Trapezoid...
        }
    }
}
    
```

Having added 3 new classes, the `double calculate(...)` must be changed because it requires

more `if` conditions to handle proper type casts. In general, every time that a new shape is added or removed, this method has to be maintained due to the presence of type casts. We can do better!

```
// Use it as an interface
abstract class Area {
    double computeArea();
}

// Every class calculates the area by itself
class Rectangle implements Area {}
class Circle implements Area {}
class Triangle implements Area {}
class Rhombus implements Area {}
class Trapezoid implements Area {}

class AreaCalculator {
    double calculate(Area shape) {
        return shape.computeArea();
    }
}
```

Thanks to the interface, now we have the possibility to add or remove as many classes as we want **without** changing `AreaCalculator`. For example, if we added `class Square implements Area` it would automatically be "compatible" with the `double calculate(...)` method.

i The gist of this principle is: depend on abstractions and not on implementations. Thanks to `abstract` classes you work with abstractions and not with the concrete implementations: your code doesn't rely on "predefined" entities.

8.1.3 Liskov Substitution Principle

The Liskov Substitution Principle states that subclasses should be replaceable with superclasses without altering the logical correctness of the program. In practical terms, it means that a subtype must guarantee the "usage conditions" of its supertype **plus** something more it wants to add. Look at this example:

```
class Rectangle {  
    double width;  
    double height;  
    Rectangle(this.width, this.height);  
}  
  
class Square extends Rectangle {  
    Square(double length): super(length, length);  
}
```

We have a big logic problem here. A square must have 4 sides with the same length but the rectangle doesn't have this restriction. We're able to do this:

```
void main() {  
    Rectangle fail = Square(3);  
  
    fail.width = 4;  
    fail.height = 8;  
}
```

At this point we have a square with 2 sides of length 4 and 2 sides of length 8... which is absolutely wrong! Sides on a square must be all equal but our hierarchy is logically flawed. The LSP is broken because this architecture does **NOT** guarantee that the subclass will maintain the logic correctness of the code.

! This example also shows that inheriting from abstract classes or interfaces, rather than concrete classes, is a very good practice. Prefer composition (with interfaces) over inheritance.

To solve this problem, simply make `Rectangle` and `Square` two independent classes. Breaking LSP does not occur if you depend from interfaces: they don't provide any logic implementation as it's deferred to the actual classes.

8.1.4 Interface Segregation Principle

This principle states that a client doesn't have to be forced to implement a behavior it doesn't need. What turns out from this is: you should create small interfaces with minimal methods. Generally it's better having 8 interfaces with 1 method instead of 1 interface with 8 methods.

```
// Interfaces
abstract class Worker {
    void work();
    void sleep();
}

class Human implements Worker {
    void work() => print("I do a lot of work");
    void sleep() => print("I need 10 hours per night..."); 
}

class Robot implements Worker {
    void work() => print("I always work");
    void sleep() {} // ?? 
}
```

Robots don't need to sleep and thus the method is actually useless, but it still needs to be there otherwise the code won't compile. To solve this, let's just split `Worker` into multiple interfaces:

```
// Interfaces
abstract class Worker {
    void work();
}
abstract class Sleeper {
    void sleep();
}

class Human implements Worker, Sleeper {
    void work() => print("I do a lot of work");
    void sleep() => print("I need 10 hours per night..."); 
}

class Robot implements Worker {
    void work() => print("I always work");
}
```

This is definitely better because there are no useless methods and we're free to decide which behaviors should the classes implement.

8.1.5 Dependency Inversion Principle

This is very important and useful: DIP states that we should code against abstractions and not implementations. Extending an abstract class or implement an interface is good but descending from a concrete classed with no abstract methods is bad.

```
// Use this as interface
abstract class EncryptionAlgorithm {
    String encrypt(); // <-- abstraction
}

class AlgoAES implements EncryptionAlgorithm {}
class AlgoRSA implements EncryptionAlgorithm {}
class AlgoSHA implements EncryptionAlgorithm {}
```

Dependency injection (DI) is a very famous way to implement the DIP. Depending on abstractions gives the freedom to be independent from the implementation and we've already dealt with this topic. Look at this example:

```
class FileManager {
    void secureFile(EncryptionAlgorithm algo) {
        algo.encrypt();
    }
}
```

The `FileManager` class knows **nothing** about how `algo` works, it's just aware that the `encrypt()` method secures a file. This is essential for maintenance because we can call the method as we want:

```
final fm = FileManager(...);

fm.secureFile(AlgoAES());
fm.secureFile(AlgoRSA());
```

If we added another encryption algorithm, it would be automatically compatible with `secureFile` as it is a subtype of `EncryptionAlgorithm`. In this example, we're respecting the 5 SOLID principles all together.



8.2 Dependency Injection

Two classes are said to be "coupled" if at least one of them depends on the other. Class A depends on class B when you can't compile class A without the presence of class B. This can be very dangerous, let's see why.

```
class PaymentValidator {
    final Date date;
    final String cardNumber;
    const PaymentValidator(this.date, this.cardNumber);

    // Uses the MasterCard payment circuit
    void validatePayment(int amount) { ... }
}

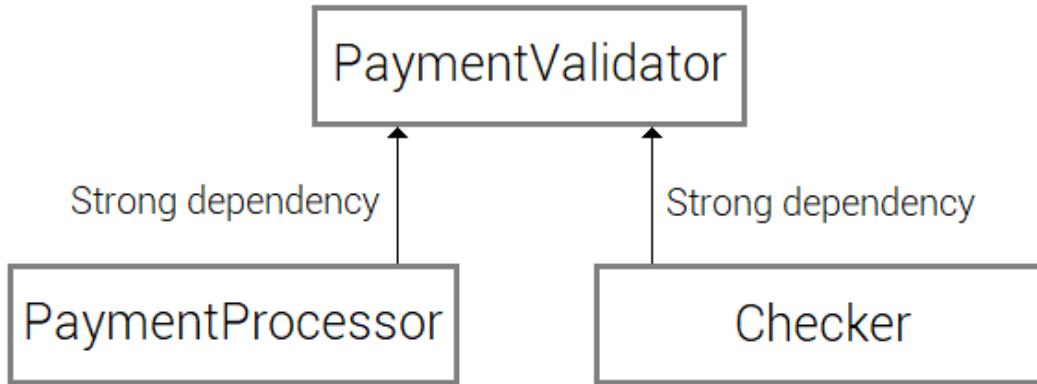
class PaymentProcessor {
    late final _validator;
    PaymentProcessor(String cardNumber) {
        _validator = PaymentValidator(DateTime.now(), cardNumber);
    }

    Date get expiryDate => _validator.date;
    void pay(int amount) =>
        _validator.validatePayment(amount);
}

abstract class Checker {
    PaymentValidator mastercardCheck();
}

class CheckerOne extends Checker { /*... code ... */}
class CheckerTwo extends Checker { /*... code ... */}
```

Both `Checker` and `PaymentProcessor` have a strong dependency on `PaymentValidator` because it's essential in order to compile. Subclasses, of course, inherit the dependency too.



Let's say you've written this code at work. One day, your project manager tells you to ditch Mastercard and replace it with PayPal. You'll quickly get a stomach ache as soon as you realize that, from an apparently small change, the whole architecture has to be refactored.

1. Paypal just requires an email but your Mastercard implementation requires date and card number. You're forced to entirely change `PaymentValidator` but by consequence you also need to update both `PaymentProcessor` and `Checker` as they're strong dependencies.

```
class PaymentValidator {  
    final String _email;  
    const PaymentValidator(this._email);  
  
    void validatePayment(int amount) { ... }  
}  
  
class PaymentProcessor {  
    late final PaymentValidator _validator;  
  
    PaymentProcessor(String email) :  
        _validator = PaymentValidator(email);  
  
    void pay(int amount) =>  
        _validator.validatePayment(amount);  
}
```

There's been a "cascade" effect because changes made to a single class had consequences to other classes as well.

2. The above changes break another part of the code: the abstract class `Checker` also depends on `PaymentValidator` so there's the need to fix the code.

```
abstract class Checker {  
    // earlier it was called 'mastercardCheck()'  
    PaymentValidator paypalCheck();  
}
```

This change has consequences on `any` subclass of `Checker` which has to be updated. Dependencies on superclasses are inherited by its children and thus the coupling propagates.

3. There are no ways to solve this problem other than manually updating every single subclass of `Checker`. Your IDE will come to the rescue with a refactor tool but maintenance is a pain anyway.

As you've just seen, an apparently small and quick change on a class propagated to an entire hierarchy and other big components of our project. All of this happened because classes are strongly coupled and they depend on implementations rather than abstractions.

8.2.1 Constructor injection

Using dependency injection and abstractions rather than implementations, the above problems fade away. Dependencies passed from the outside create a *weak* coupling which is safer than a *strong* one as it relies on abstractions.

```
abstract class PaymentValidator {  
    const PaymentValidator();  
    void validatePayment(int amount);  
}  
  
class MasterCard implements PaymentValidator {  
    // Define date, card number and the constructor  
    const MasterCard();  
    void validatePayment(int amount) {...}  
}  
  
class PayPal implements PaymentValidator {  
    // Define an email and the constructor
```

```

const PayPal();
void validatePayment(int amount) {...}
}

```

The `PaymentProcessor` class is still going to have a `PaymentValidator` dependency but it's **weak** because it's just an interface. Using "constructor injection" we pass from the outside a concrete implementation, which can later be replaced with anything else.

```

class PaymentProcessor {
    final PaymentValidator _validator;
    const PaymentProcessor(this._validator);

    void pay(int amount) =>
        _validator.validatePayment(amount);
}

// And then we can freely use PayPal or MasterCard
void main() {
    final p1 = const PaymentProcessor(MasterCard());
    final p2 = const PaymentProcessor(PayPal());
}

```

In this case, we're passing an instance of a concrete class via constructor and that's **fundamental**. `PaymentProcessor` knows nothing about the implementation details of the validator object, it just knows he has to call `validatePayment(int)`. We can also use `const` constructors now!

- This code is very flexible and maintainable. If your boss told you to add support for the Visa circuit as well, you would simply have to create a new subtype of `PaymentValidator`.

```

class Visa implements PaymentValidator {
    const Visa();
    void validatePayment(int amount) {...}
}

```

No changes are required to the existing code and you at the same time you're still embracing S.O.L.I.D. principles. The architecture is robust!

- `PaymentProcessor` now doesn't care anymore about Mastercard, Paypal or whatever because they're given from the outside. Internally he weakly depends on an abstraction which just gives an abstraction: the implementation is passed via constructor.
- You have a series of classes, one per payment method, that are super easy to test. You

could make a "mock" class for unit tests just like a regular validator type:

```
class TestValidator implements PaymentValidator {  
  const TestValidator();  
  void validatePayment(int amount) {...}  
}
```

Last thing we need to refactor is the `Checker` class as it has to return an abstraction rather than an implementation.

```
abstract class Checker {  
  PaymentValidator paymentCheck();  
}  
  
class CheckerOne extends Checker {...}  
class CheckerTwo extends Checker {...}
```

Since `PaymentValidator` is `abstract`, any class along the hierarchy inherits a *weak* dependency which is safe.

8.2.2 Method injection

Constructor injection is used when you class really needs an external dependency to work. When you have an "optional" dependency not strictly required from your class, you can use method injection.

```
abstract class CheckProcessor {  
  const CheckProcessor();  
  bool isActive();  
}  
  
class MastercardCheck implements CheckProcessor {  
  final MasterCardApi _api;  
  const MastercardCheck(this._api);  
  
  bool isActive() async => await _api.isOnline();  
}  
  
class PaypalCheck implements CheckProcessor {  
  final PaypalApi _api;
```

```

const PaypalCheck(this._api);

bool isActive() async => await _api.available();
}

```

These classes connect to the internet, perform some GET requests and return `true` or `false` whether the service provider is online or not. Let's say this feature is not essential in our architecture but it's nice to have it. It might be used but it's not certain.

```

class PaymentProcessor {
    final PaymentValidator _validator;
    const PaymentProcessor(this._validator);

    void pay(int amount) => ...

    bool isProcessorActive(CheckProcessor check) =>
        return check.isActive();
}

```

In this way, if we wanted to check the availability of the service we could do this:

```

void main() {
    final api = MasterCardApi(...);
    final processor = MasterCard(api);
    final checker = MastercardCheck();

    final payment = PaymentProcessor(processor);
    final isOnline = payment.isProcessorActive(checker);
}

```

Note the difference: while the processor is fundamental, and thus it's passed via constructor, the connection checker made with `isProcessorActive` is not always required. So in general:

- constructor injection is for essential dependencies that your class is **always** going to use;
- method injection is for optional dependencies that you class **might** use.

Actually both type of injection use the same concept, which is depending on abstractions and passing implementations from the outside, but they differ in order of "importance". Dependencies passed via constructor are fundamental while the ones passed via method are just useful but not essential.

PART II

THE FLUTTER FRAMEWORK

"Programs must be written for people to read, and only incidentally for machines to execute."

ABELSON AND SUSSMAN

9 | Basics of Flutter

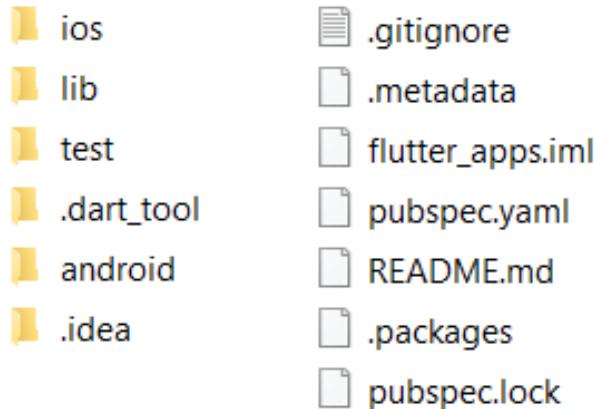
9.1 Structure and tools

Android Studio (AS), along with the official plugin, is Google's first-choice IDE which offers a very pleasant development experience. Alternatively, Flutter apps can also be created using Visual Studio Code (VS Code), Emacs or any text editor along with the Flutter command line tool.

- ❶ The official Flutter documentation gives you a step-by-step install guide ¹ for Windows, macOS and Linux. Please follow it carefully to properly setup your environment. We have used Android Studio 4.0.1, which is the latest version at the time of writing this book.

Any new Flutter project, whether it's created with Android Studio or VS Code, requires a series of files and directories for both you and the IDE. Most of them can safely be ignored because you'll spend basically all of your time inside `lib/` and `test/`.

¹<https://flutter.dev/docs/get-started/install>



This is what we get on Windows using Android Studio but depending on the IDE you're using, there might be some different configuration files. Regardless the operating system and the IDE, for sure there will **always** be at least:

- **android/** and **ios/**: These folders contain platform-specific code for each OS and they're automatically managed by the IDE and the compiler. The structure is exactly the same you'd get with a new Android project on Android Studio or iOS project on XCode.
- **lib/**: This folder is essential: it contains the Dart source code of your Flutter app. You're going to spend a countless amount of hours in here.
- **test/**: Unit tests, widget tests and integration tests all go in this folder. Chapter 16 is a in-depth guide on how to properly test your Flutter apps.
- **pubspec.yaml**: This file is fundamental as it defines a Dart package and lists dependencies of your Flutter app.
- **README.md**: It's the typical markdown file you can find in any git repository. It's used for your git repository and at the same time as "home page" at <https://pub.dev> in case you wanted to publish a package.

All the other files or folders we haven't mentioned in the above list are automatically managed by the IDE (or the compiler) so you shouldn't care about them.

9.1.1 Folder structure

Before starting your coding journey, it'd be a good idea to have a solid background in folder structure and organization. The official Flutter documentation doesn't give any guideline about this since you're free to do what you prefer. We have some suggestions for you:

- **lib/**. Your app's source code goes here. Grouping files in proper folders is essential if you don't want to get lost in your own architecture so, before coding, mind the structure.

- routes/
- models/
- widgets/
- main.dart
- routes.dart

You could mind your project's folder structure starting from this simple skeleton. `routes/` contains your app's pages, `models/` is for the "business logic" and `widgets/` is for reusable UI widgets.

- localizations/
- routes/
- widgets/
- models/
 - * blocs/
 - * providers/
 - * repositories/
 - * ...
- main.dart
- routes.dart

If you plan to make your apps available in multiple languages, consider grouping all the localization logic inside `localizations/`. A complete coverage about localization techniques in Flutter will be discussed in detail in chapter 13. Structure folders with many sub folders.

- **test/**. Flutter has a powerful automated testing suite; we recommend splitting test files according with their use case.
 - unit/
 - widget/
 - integration/

We will cover testing in depth in chapter 16.

There would be the possibility to throw all your files inside a single folder (for example `lib/`) with no structure but... no! In a medium-large app architecture, maintenance is going to be painful because there's no logical organization.

9.1.2 The `pubspec.yaml` file

This file is very important and it deserves to be properly described. It gives you control on: dependencies used by Flutter, resources/assets of your app and the versioning system for production binaries.

i YAML is a data-serialization language commonly used for configuration files. It exposes a series of settings in a human-readable way; it has no punctuation as it relies on indentation and line breaks.

Indentation and line breaks are very important because there are no semicolons or commas as separators. We're only giving an overview of the most important attributes but of course the official documentation ² will give you a full reference.

- **version**. Any package is required to specify a version number which increments at any release; in chapter 24 we'll see how to write a Flutter package that can be uploaded to <https://pub.dev>. Here you see a library with its version number:

²<https://dart.dev/tools/pub/pubspec>

flutter_bloc 6.0.1

Published Jul 22, 2020 ·  [bloclibrary.dev](#)

When you're releasing an app for the Google Play store or the Apple App store, this number is used to assign a version value to the product. For example if you had...

```
version: 1.1.0+5
```

... it would mean that your app's version name would be 1.1.0 and the build number would be 4. In the Android world, inside `build.gradle`, the field `versionName` would be 1.1.0 and `versionCode` would be 5.

- **sdk.** This section contains the constraints indicating which SDK versions your app supports. The Dart team recommends to always include a lower and an upper bound but you could simply use "`>= 2.7`" and it'd be valid anyway.

```
environment:
```

```
  sdk: ">=2.7.0 <3.0.0"
```

With the above range you can use everything coming from version 2.7 onward so Dart extensions for example (introduced in 2.7) are supported.

- **uses-material-design.** Ensures that your Flutter app is able to use icons from the Google Material design ³ project. They are pretty common in the Google world, especially in Android as they're the default icons being used in many apps.



Having `uses-material-design: true` icons are already available, you've nothing to download or setup because they're bundled in the Flutter SDK. Icons are actually vectorial images so they resize without quality loss.

³<https://material.io/>

- **dependencies.** This is probably the most important label because it declares any package the app is going to depend on. You just need to go to <https://pub.dev>, look for a package and add a new line.

```
dependencies:  
  flutter:  
    sdk: flutter  
  
  http: ^0.12.2  
  provider: ^4.3.2+2  
  flutter_svg: ^0.18.1
```

On the left there's the name of the package while on the right there's the version being downloaded from the repository. They have been added in this very simple way:

1. open <https://pub.dev>;
2. search for "http" or any other meaningful keyword;
3. choose a package from the list and click on the **Installing** tab;
4. copy/paste the given installation string, in our case "**http: ^0.12.2**"

- **assets.** This label specifies the paths to static resources your app will use such as images, SVG vectorials, audio/video files or simple text. For example, you could create a folder called `images/` and put everything in there without having to list files one by one.

```
flutter:  
  assets:  
    - images/  
    - files/text/myFile.txt  
    - audio/
```

In the second line we have imported a text file giving the exact location. The root of the project is the directory in which you have the `pubspec.yaml` file. When you declare any kind of asset, the starting point for the path is the root.

- **fonts.** By convention this label is put at the bottom of the file, after **assets**. You can download font files from <https://fonts.google.com> and import them directly in your app.

```
flutter:  
  assets:  
    - images/
```

```
fonts:
  - family: Roboto
    fonts:
      - asset: fonts/Roboto-Regular.ttf
      - asset: fonts/Roboto-Italic.ttf
        style: italic
  - family: RobotoMono
    fonts:
      - asset: fonts/Righteous-Regular.ttf
        weight: 400
```

Once you've downloaded the `.ttf` files from Google Font, create a folder called (by convention) `fonts/` and put the files in there. There's nothing more to do because Flutter will take care of automatically loading them.

– https://pub.dev/packages/google_fonts

Starting from January 2020 there's an official Flutter package called `google_fonts` which retrieves fonts from <https://fonts.google.com/> and caches them. This is ideal for development: no need to place font assets in the `font/` folder because they will be automatically downloaded and cached.

```
Text(
  'This is Google Fonts',
  // Download 'pacific' and cache it
  style: GoogleFonts.pacific(),
),
```

However, you could download font files anyway and include them as assets because it's faster and more secure. You might be in trouble if the user opened your app for the first time with no internet connection. The Google Fonts package will prioritize pre-bundled files over http fetching, so you could do the following:

1. While developing, use http font fetching from the internet, which is very convenient.
2. Before publishing the app, go to <https://fonts.google.com/>, download the font files you need and move them to the `font/` folder in your Flutter project.
3. Open the `pubspec.yaml` file and add `font/` under `assets` so that `google_fonts` can automatically load font files from there.

```
flutter:  
  assets:  
    - images/  
    - fonts/
```

There is no need to have the `fonts` section because files are already included as assets.

In this way `google_fonts` will load font assets at startup rather than at the first usage. However, if a given font is required and it's not in the assets, it will be downloaded and cached automatically.

In summary, we recommend the usage of `google_fonts` for development but you should provide font files as assets so that they can be loaded at startup (rather than at runtime, via HTTP request).

9.1.3 Hot Reload

If you know the basics of HTML, you're aware of the fact that any edit to an `.html` file can be seen immediately clicking the refresh button of the browser. It's literally a matter of seconds because you just need to save the file and press F5. Flutter works in the same way!



Android Studio



VS Code

Thanks to the **hot reload** feature, you can refresh the UI in your emulator (or physical device) while writing Dart code. There is no need to make a build every time and wait for gradle/Xcode to complete. It's like if you pressed F5 in your browser to refresh the HTML source file.

- ❶ You have to *Run* the app in debug mode for the first time but then you can press

the yellow lightning which is the *Hot reload* button. You'll see the UI immediately updated and fully functional in sync with the latest code you've written.

Hot reload is blazing fast as it takes less than a second to refresh the UI. It increases a lot the productivity because changes are immediately applied and ready to be tested, no need to wait for build processes. Hot reload works in most of the cases but in certain circumstances you have to stop and re-run the app entirely:

- when you make changes to the `initState()` method (more in it in the next chapter)
- when you change the definition of a `class` into an `enum` and vice versa,
- when you make changes to `static` fields in classes,
- when you make changes to code inside `void main() {}`.

In debug mode Flutter uses the JIT compilation model that, in combination with the Dart Virtual Machine, allows fast injection of the source code and quick incremental rebuilds. In other words, we can say that "Flutter's *hot reload* is super fast!".

i While creating Flutter apps, it's common having an Android or iOS simulator on the right of the screen and your favorite IDE in the remaining space. This is the fastest way to write code and see the results immediately with the hot reload feature.

9.1.4 Linter rules

After hours of coding, you might forget to give a generic class the type and thus the compiler automatically assigns `dynamic`. It's all good because compilation successfully executes but the code is not type safe and thus you're not following the good practices.

i A linter is a very helpful tool that reads the source code and spots syntax errors, suspicious constructs, styling errors and much more. By default Dart's linter is very permissive and it marks something as error only when really needed.

Making the linter more severe is very productive as it can discover problems and potential bugs even before executing the code. In order to do this there's the need to create a file called `analysis_options.yaml` in the same folder as the `pubspec`.

```
analyzer:  
  strong-mode:  
    implicit-casts: false  
    implicit-dynamic: false  
linter:  
  rules:  
    - avoid_unused_constructor_parameters  
    - await_only_futures  
    - directives_ordering  
    - empty_constructor_bodies  
    - empty_statements  
    - hash_and_equals  
    - implementation_imports  
    - null_closures  
    - package_api_docs  
    - slash_for_doc_comments  
    - test_types_in_equals  
    - throw_in_finally  
    - type_init_formals
```

Visit the official Dart documentation ⁴ to get a complete list of any linter rule. We strongly encourage you to create an `analysis_options.yaml` for every Flutter app or Dart project you create. There is also the possibility to change the default behavior of the linter:

```
analyzer:  
  errors:  
    include_file_not_found: error  
    dead_code: warning
```

For example, by default when a given include file could not be found a warning is emitted. If you want this issue to be more important, it can be treated as an error by overriding its severity to one of these levels:

- **error**: causes static analysis to fail;
- **warning**: static analysis doesn't fail unless warnings are treated as errors by the analyzer;
- **info**: just an message info which doesn't make static analysis fail;
- **ignore**: ignores the given rule.

⁴<https://dart-lang.github.io/linter/lints/>

Basically in `errors` you can redefine the severity of warnings and errors as you want. In general the default setup is fine as it is, you don't need to override rules and in particular try to avoid using `ignore`. Visit the official documentation⁵ to get a complete list of any overrideable property.

- ➊ The `analysis_options.yaml` file summarizes the Dart's good practices guidelines so that you don't have to remember everything. The IDE is able to read this file and emit visual messages for you.

Very simply put, having proper rules set on `analysis_options.yaml` is like having something that guides you to follow Dart's best practices. Go to the *Resources* page of our website to download a good template we recommend you to use.

9.1.5 Tree shaking and constants

Using `import "package:flutter/foundation.dart"` might be very useful while developing and debugging Flutter apps. It exposes three constant boolean values the developer can use to execute a series of instructions according with the build mode:

- **Debug mode.**

```
if (kDebugMode) {  
    // code to be executed when running the app in debug mode...  
}
```

- **Profile mode.**

```
if (kProfileMode) {  
    // code to be executed when running the app in profile mode...  
}
```

- **Release mode.**

```
if (kReleaseMode) {  
    // code to be executed when running the app in release mode...  
}
```

When building a Flutter app (in any mode), "tree shaking" is automatically performed. It's basically the compiler removing dead code depending on variables being constant or not plus other factors. For example, look at this piece of code:

⁵<https://pub.dev/documentation/analyzer/latest/analyzer/analyzer-library.html>

```
String get name {
    if (kDebugMode) {
        return "Demo";
    } else {
        return _real();
    }
}
```

The *Run* button of Android Studio and VS Code builds the app in debug mode so the above code will always return "Demo". The other statement (`return _real();`) is automatically removed by the compiler because it will never be reached. According with the build mode, after the compilation the same piece of code can look like this:

Debug mode

```
String get name {
    return "Demo";
}
```

Profile mode

```
String get name {
    return _real();
}
```

Release mode

```
String get name {
    return _real();
}
```

You should really use these constants while developing your apps as they're very useful. There's also no need to manage them because the compiler will automatically remove the unused parts (dead code is automatically discarded). Tree shaking works with any constant value:

```
const isGood = true;

if (isGood) {
    print("Good!");
} else {
    print("Bad!");
}
```

The compiler will remove the `else` branch because it's considered to be dead code.

9.2 Widgets and State

In Flutter everything that appears on the screen is called "*widget*" because, technically speaking, it's a descendant of the `Widget` class. When you create user interfaces in Flutter you make a composition of widgets by nesting them one inside the other.

- i** If you talk about widgets you refer to buttons, text fields, animations, containers and even the UI page itself. Anything appearing on the screen or interacting with it is a widget. Widgets everywhere!

When you nest widgets one inside the other you create a hierarchy called "widget tree" in which there are parents and children. In a fresh new Flutter project, the IDE prepares a sample application in `main.dart` having this minimal structure:

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

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

class MyApp extends StatelessWidget {
    @override
    Widget build(BuildContext context) {...}
}
```

As you know, any Dart program must have a `void main() {}` entry point and Flutter is no exception; the `runApp()` method takes an instance of a `Widget` and makes it the root of the widget tree. At the beginning you get a tree with a single leaf (the root itself):

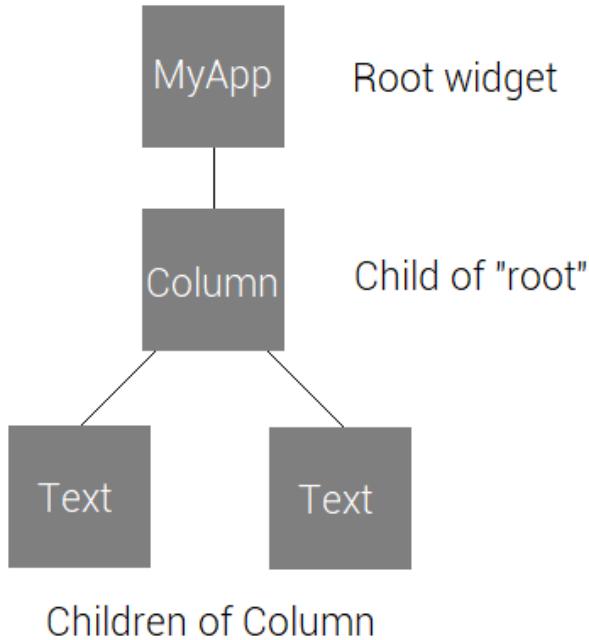


In the next section we will see that a class in Flutter becomes a widget when it inherits from `StatelessWidget` or `StatefulWidget`. For now, note that the `runApp()` method has made the

class called `MyApp` the root of the tree. Let's add more contents to see how the widget tree expands.

```
void main() => runApp(const MyApp());  
  
class MyApp extends StatelessWidget {  
  const MyApp(); // Good idea!  
  
  @override  
  Widget build(BuildContext context) {  
    return Column(  
      children: <Widget>[  
        Text("Hello"),  
        Text("Flutter"),  
      ]  
    );  
  }  
}
```

The method `Widget build(BuildContext context)` adds new leaves to the widget tree in order to place new graphical items in the UI. Widgets are nested one inside the other using named parameters in the constructors to make the reading of the code very expressive. This is the new situation:



The addition of widgets makes the tree bigger and places new items on the screen. The `context` parameter in `build(BuildContext context)` gives important information about the position of the leaf in the tree. In particular:

- A `BuildContext` instance is used by Flutter to know details about the widget when the tree is being traversed;
- We will see that calling `SomeWidget.of(context)` returns the closest widget in the tree whose type is `SomeWidget`;
- Each widget has its own `BuildContext` instance which becomes the parent context of the widget(s) returned by its `build` method.

Other than passing an instance of `BuildContext` to widgets' constructors you won't do anything else with it. It's intended to be used by Flutter to get info about the widget tree; the developer hardly never is asked to use it directly.

- ❶** The interpolation of classes, nested with named constructors, embraces the so called *declarative* UI design which is the typical Flutter coding style.

9.2.1 Basic widgets

Flutter has a countless amount of widgets that can be found both in the core library or online at <https://pub.dev>. We're immediately going to list the most important ones but you'll discover many others while reading the book.

💡 We strongly encourage you to visit the online catalog as it showcases the most important UI widgets for both material (Android) and cupertino (iOS) styles. The Flutter team is very active and the catalog is improved/expanded very often.

9.2.1.1 Text

You wouldn't be surprised to find out that the `Text` widget is used to display a piece of text on the screen. It's highly customizable as you can change the color, the font using font assets or the Google Font package and much more.

```
const Text(  
    "Text on the screen",  
    style: TextStyle(  
        color: Colors.amber,  
        fontSize: 16,  
        wordSpacing: 3,  
    ),  
) ;
```

It just requires a string as a parameter, which is the text being displayed on the UI, and the styling is made with the `TextStyle()`⁶ class. It defines many properties of the text itself and it's also the place in which the `google_fonts` package can be used.

```
Text(  
    "Text on the screen",  
    style: GoogleFonts.lato(  
        textStyle: const TextStyle(  
            color: Colors.amber,  
            fontSize: 16,  
            wordSpacing: 3,  
        ),  
) ,
```

⁶<https://api.flutter.dev/flutter/painting/TextStyle-class.html>

```
    ),  
);
```

9.2.1.2 Row

This widget places one or more children in the horizontal axis with the given space constraints. It's used very often when you need to have multiple items aligned side by side. There is no `const` constructor for `Row` but you can assign it, when possible, to the `children` value.

```
Row(  
    mainAxisAlignment: MainAxisAlignment.center,  
    children: const [  
        Text("Hello"),  
        Text("Flutter!"),  
        Text("!!"),  
    ],  
,
```

You can easily figure out that there will be three `Text` widgets side by side at the center of the screen. By default, a row tries to totally cover the available horizontal space; you can make sure it shrinks to fit the width of its content using:

```
Row(  
    mainAxisSize: MainAxisSize.min,  
,
```

Widgets in rows can be placed in different ways according to the value of `mainAxisAlignment`. The default behavior is `start` but of course it can be changed passing different values to the constructor.

- **center**. Places the items at the center of the row.



- **start**. Places the items at the beginning of the row.



- **end.** Places the items at the end of the row.



- **spaceAround.** Places the items with equal distance between each other and the margins.



- **spaceBetween.** Places the items with an evenly space between them.



9.2.1.3 Column

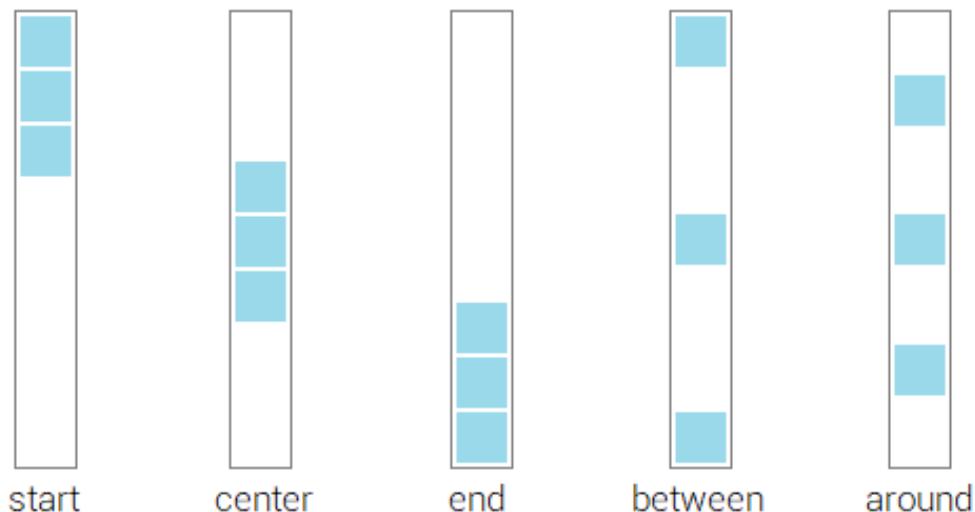
This widget places one or more children in the vertical axis with the given space constraints. A **Column** is the opposite of a **Row** as it has the same purpose but it works in the opposite direction (vertical rather than horizontal).

```
Column(
  mainAxisSize: MainAxisSize.center,
  children: const [
    Text("Hello"),
    Text("Flutter!"),
    Text("!!!"),
  ],
),
```

It's identical to a `Row` but here items are placed one above the other because a `Column` works in the vertical axis. It tries to totally cover the available vertical space; you can make sure that it shrinks to fit the height of its content using

```
Column(
  mainAxisSize: MainAxisSize.min,
),
```

You can place widgets in different ways just by passing a new alignment to the `mainAxisAlignment` parameter of the constructor. It's completely identical to a `Row` because alignments work in the same way.



A column does **NOT** have a scroll behavior so if there's not enough space, you'll get an overflow error at runtime.

9.2.1.4 ListView

A `ListView` is basically a `Column` with scrolling behavior as it places one or more children in the vertical axis, in sequence. This widget is very widely used because it provides the possibility to scroll contents when they are bigger than the screen size.

```
ListView(
  children: const [
    Text("Hello"),
    Text("Flutter!"),
    Text("!!!"),
  ],
),
```

Widgets are aligned to the top and the scrolling direction is vertical by default but of course you can change it. With `scrollDirection` you can decide whether the list has to scroll in the horizontal or vertical axis.

```
ListView(
  scrollDirection: Axis.horizontal,
),
```

When the content of the list is known in advance, children are simply declared inside a list as you've seen above. The `ListView.builder(...)` named constructor is very useful when the list has to be built based on an existing collection.

```
// Somewhere in the code there's a list of 100 integers
final myList = List<int>.generate(100, (i) => i);

// The 'builder' named constructor builds a list of widgets
// by taking the 'myList' list as data source.
ListView.builder(
  itemCount: myList.length,
  itemBuilder: (context, index) {
    return Text("${myList[index]}"),
  },
),
```

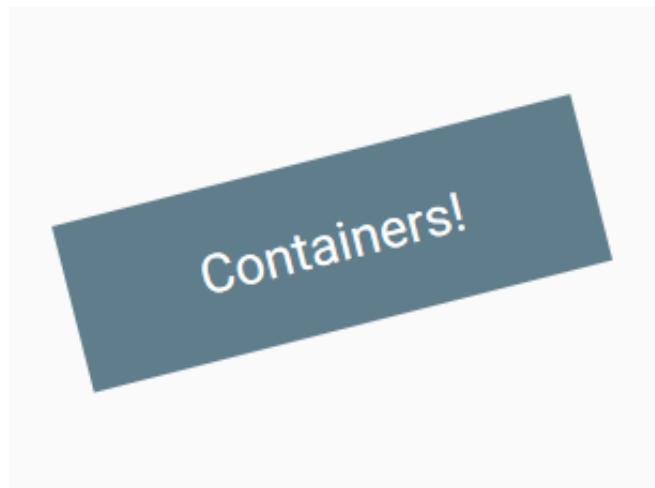
The official Flutter documentation ⁷ suggests to use the `builder(...)` named constructor when the data source is a long list because it **efficiently** manages the children. So, rather than manually

⁷<https://flutter.dev/docs/cookbook/lists/long-lists>

filling a long `ListView` with a for loop, use its `builder()` which is more efficient.

9.2.1.5 Container

This widget is the equivalent of a `<div></div>` tag in the HTML world; it's a general purpose container you can use to customize painting, positioning, sizing and much more. A `Container` is very widely used since it accomplishes many use cases such as making rounded borders or working with shapes.



It might seem a complex result to achieve but it's actually very easy because a container is made exactly for this kind of purposes.

```
Widget build(BuildContext context) =>
    Container(
        height: 80,
        width: 260,
        color: Colors.blueGrey,
        alignment: Alignment.center,
        transform: Matrix4.rotationZ(-0.25),
        child: const Text(
            "Containers!",
            style: TextStyle(
                color: Colors.white,
                fontSize: 25
            )
        )
    )
```

```
 );
```

The rotation is obtained thanks to `transform: Matrix4.rotationZ(-0.25)`, which defines how to place an object in the 3D space. This is often used in animations but you'll have to wait until chapter 14 to read more. In order to style a `Container` you have to use a `BoxDecoration` class:

```
Container(  
    child: const Center(...),  
    width: 100,  
    height: 100,  
    decoration: const BoxDecoration(  
        shape: BoxShape.circle,  
        boxShadow: [  
            BoxShadow(  
                color: Colors.grey,  
                spreadRadius: 5,  
                blurRadius: 7,  
                offset: Offset(0, 3),  
            ),  
        ],  
        gradient: LinearGradient(  
            begin: Alignment.topCenter,  
            end: Alignment.bottomCenter,  
            colors: [  
                Color.fromARGB(...),  
                Color.fromARGB(...)  
            ],  
        )  
    ),  
);
```

In this example we've created a circle with a shadow behind and a linear gradient as background. The `BoxShadow` class is very similar to the CSS `box-shadow` property, exactly like `LinearGradient` which can of course interpolate more than two colors. It isn't the only type of gradient you can use:

- `LinearGradient`: a progressive transition of two or more colors along a straight line;
- `RadialGradient`: a progressive transition of two or more colors radiating around a central

```
point;
```

- **SweepGradient**: a progressive transition of two or more colors with a circular sweep on a central point.

You can also have rounded borders with `borderRadius: BorderRadius.circular(30.0)` or a simple plain background color with the `color:` property. Be sure to check out the official documentation about `BoxDecoration`⁸ to see how you can fully customize a `Container`.

9.2.1.6 Stack and Positioned

Thanks to the `Stack` widget you can overlap widgets and freely position them on the screen using `Positioned`. Even if children are placed outside the bounds of the UI, no overflow errors will appear because a `Stack` doesn't constrain the bounds of width and height.

```
Stack(
  children: [
    Container(
      width: 40,
      height: 40,
      decoration: const BoxDecoration(
        color: Colors.red
      )
    ),
    const Text("Hello"),
  ]
)
```

With this simple example, the UI is created with a red box and the `Text` widget is painted **in front** of the `Container`. The order in which you place the widgets really matters because children at the bottom of the list are placed, relatively, in front of the ones at the top. The foreground widget goes at the end of the list.

```
Stack(
  children: [
    const Text("Hello"),
    Container(

```

⁸<https://api.flutter.dev/flutter/painting/BoxDecoration-class.html>

```
        width: 40,  
        height: 40,  
        decoration: const BoxDecoration(  
            color: Colors.red  
)  
,  
    ]  
)
```

In this case the red box would be placed in front of the `Text` widget because, in order, it comes after and thus "Hello" is not visible because covered by the `Container`. You could however decide to move the text at a specific position of the screen:

```
Stack(  
    children: const [  
        Positioned(  
            top: 40,  
            left: 65  
            child: Text("Hello"),  
)  
,  
    ]  
)
```

You could also have used a negative offset such as `left: -15` to position the text outside the bounds of the visible area.

9.2.2 Stateless and Stateful widgets

A class becomes a Flutter widget when it subclasses `StatelessWidget` or `StatefulWidget` and overrides the `Widget build(...);` abstract method. That's it: the main task of a widget is laying out other widgets on the tree using the `build()` method.

- ❶ You already know from the previous examples how widgets are laid out inside `build()` to compose the UI. They're nested one inside the other.

Before creating a widget the developer must decide whether the `state` will change during the time or not. If the state changes at some point, then it means that something has happened such as:

- the user has tapped on a button and thus something in the UI must change;
- the device has been rotated and the UI must be repainted;
- there's a new event on a stream and a widget depending on it is notified (and thus a rebuild happens in order to reflect the changes brought by the stream).

In other words, you have to ask yourself if the widget is immutable or if it's "dynamic", in the sense that something might change during the time. The decision translates into Dart code by extending one of these two classes.

- **Stateless widget.** Use this kind of widget when you need to create a piece of UI that is not going to change over the time. It's a "standalone" block that doesn't depend on external events or sources; it just relies on its constructor and the internal data.

```
class MyName extends StatelessWidget {  
    // Notice the constant constructor  
    const MyName();  
  
    @override  
    Widget build(BuildContext context) {  
        return Row(  
            mainAxisAlignment: MainAxisAlignment.spaceAround,  
            children: const [  
                Icon(Icons.person),  
                Text("Flutter developer"),  
            ]  
        );  
    }  
}
```

This is a perfect example of a `StatelessWidget` because the contents will always be the same; no external dependencies or streams are going to change the text or the icon. Once created, the widget is "static" because it will never change: it's a "solid block".

```
class MyName extends StatelessWidget {  
    final String name;  
    const MyName({  
        // use the annotation '@required' if your Dart version does not  
        // support nnbd  
        required this.name  
    })
```

```
});  
  
@override  
Widget build(BuildContext context) => const Text(name);  
}
```

By convention Flutter widgets have named optional parameters in the constructor; in case they were required, use the `required` keyword. Since this class is **immutable** it's a very good idea marking the instance variables as `final` so that a `const` constructor can be declared.

- ➊ Even if the class takes a string from the outside, via constructor, it still doesn't change over the time. The given name will always be the same and thus the widget will never change/rebuild, so a stateless solution is fine.

If you're working with Dart 2.9 or lower versions, instead of the `required` keyword you're going to use the `@required` annotation.

- **Stateful widget.** Use this kind of widget when you need to create a piece of UI that is going to change over the time. In this case the UI is going to dynamically change due to external events such as the received response of an HTTP request or the callback triggered by a button tap.

```
class Counter extends StatefulWidget {  
    // Don't forget the constant constructor!  
    const Counter();  
  
    @override  
    _CounterState createState() => _CounterState();  
}  
  
// Notice the underscore: the state is a package-private class  
class _CounterState extends State<Counter> {  
    int _counter = 0;  
  
    @override  
    Widget build(BuildContext context) {  
        return Column(
```

```
        children: [
            Text("$_counter"),
            IconButton(
                child: Icon(Icons.add),
                onPressed: () {
                    setState(() => _counter++);
                }
            ),
        ],
    );
}
```

Buttons and UI design will be introduced in the next section, they aren't key point now. In this widget the main actor is the `IconButton`: once tapped by the user, the `onPressed` callback is triggered and the `_counter` variable is incremented.

1. `Counter` is the widget itself and thus it's inserted in the widget tree; `_CounterState` is the mutable state of the `Counter` widget. When Flutter rebuilds the widget tree to refresh the UI, the `build(...)` method of `State<T>` is called.
2. This is the standard pattern for the creation of a stateful widget and you should really follow it. Both Android Studio and VS Code can automatically create the boilerplate for you.
3. Subclasses of `State<T>` gain access to the `setState(...)` method which rebuilds the widget (it's like a refreshing tool).
4. Member instances, such as `_counter`, survive to rebuilds. Only what's inside the `build()` method is refreshed.

When you tap on the button, the `onPressed` callback is activated: `setState(...)` executes its body and then Flutter rebuilds the widget. In our example, `int _counter` gets incremented by 1 and then the widget is refreshed so that `Text` can display the new updated value.

- ❶** The variable `_counter` belongs to the state `State<Counter>` object and for this reason it's not reset when the widget is rebuilt. Remember that the state "survives" when a build happens.

Counter is the **widget** (what's inserted on the tree) while `_CounterState` is the **state**. The state survives to rebuilds but its `build()` method doesn't. If you're not convinced yet, we're going to tell you that this snippet works as intended.

```
class _CounterState extends State<Counter> {
    int _counter = 0;

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

The state survives to rebuilds so `_counter` is not reinitialized to zero, it keeps the count. Only what's inside the `build` method is refreshed. If you did this...

```
class _CounterState extends State<Counter> {
    @override
    Widget build(BuildContext context) {
        int _counter = 0;

        return Column(...);
    }
}
```

... your counter would always be zero! It's still correctly incremented by `setState()` which nicely does `_counter++` but then the widget is rebuilt and the first line does `_counter = 0` which sets it back to zero.

If you used the constructor of a `StatefulWidget` to set some data, the associated `State<T>` class can get a reference to them by simply using the `widget` getter. Again, try to use `const` as much as possible.

```
class WidgetDemo extends StatefulWidget {
    final int id;
    const WidgetDemo(this.id);

    @override
    _WidgetDemoState createState() => _WidgetDemoState();
}
```

```
class _WidgetDemoState extends State<WidgetDemo> {
  @override
  Widget build(BuildContext context) {
    return Text("The given id is ${widget.id}");
  }
}
```

In the above example, `WidgetDemo` is allowed to have a `const` constructor because once inserted in the widget tree, it will **never** change. What really changes is its state, represented by `_WidgetDemoState`, which in fact cannot have a constant constructor.

9.2.2.1 Good practices

First of all, there's the need to say that there are **NO** performance differences between a stateful widget and a stateless widget. You don't have to think that a stateless widget is an optimized version of a stateful one or vice versa.

- ➊ Actually a `StatelessWidget` can be seen as a `StatefulWidget` without the `setState()` method. When creating a stateful widget, the state is clearly visible because it's a separated private class:

```
// Widget
class Counter extends StatefulWidget { ... }
// Widget's state
class _CounterState extends State<Counter> { ... }
```

A stateless widget is just syntactic sugar for those cases in which you don't need to create a custom state. A `StatelessWidget` has a state too but you can't see it because it's not meant to be manually changed.

You could use `StatefulWidget`s all day all night without having problems but it wouldn't make sense. If the state doesn't change, go for a `StatelessWidget` which is less boilerplate code and it exposes less methods. Here's a guideline to help you deciding which one should be used:

- When every instance variable of your widget can be marked with the `final` modifier, use a stateless widget with a `const` constructor.

```
class PersonWidget extends StatelessWidget {
  final String name;
```

```
final String age;
const PersonWidget({
    required this.name,
    required this.age
});

@Override
Widget build(BuildContext context) { ... }
}
```

This is an **immutable** class because it has `final` variables and a constant constructor: once instantiated, the widget will never change. This is a "static block", widget that doesn't change over the time.

- When your widget has some variables that cannot be `final` because they might change over the time, use a stateful widget. It might happen when you have to lazily initialize some values or you're waiting for an asynchronous request.

```
class Counter extends StatefulWidget {
    const Counter();

    @override
    _CounterState createState() => _CounterState();
}

class _CounterState extends State<Counter> {
    int _counter = 0;

    @override
    Widget build(BuildContext context) { ... }
}
```

In this case `_counter` cannot be `final` because the `build()` method is going to alter it. When an instance variable can be changed over the time, by consequence the state of the widget will also change. In this case, a `StatefulWidget` is the right choice.

- In all those cases where a widget is something "static" that doesn't depend on anything external, consider making it stateless as you don't need to change its state.

```
class AuthorsWidget extends StatelessWidget {
    const AuthorsWidget();
```

```
  @override
  Widget build(BuildContext context) {
    return Row(
      children: [
        Text("Alberto Miola"),
        Text("Felix Angelov"),
        Text("Rémi Rousselet"),
        Text("Matej Rešetár"),
      ]
    );
  }
}
```

This widget doesn't need to change its state nor it depends on external data. It's just a single reusable "block".

To sum it up, a `StatelessWidget` is good when you have to make independent "reusable" widgets or when you don't need to change the state of your widget. In all the other cases, consider using a `StatefulWidget`.

9.2.3 Keys

You might have noticed that any widget provided by Flutter has the optional `key` parameter. Very simply, it's used to uniquely identify a widget in the tree, like when a primary key is assigned to a column of a relational database. There are mainly four types of keys (they're all sub-types of `Key`):

- `ValueKey<T>`. Suppose you created a shopping list using a `ListView` and a series of `Text` widgets (with no duplicated strings). A `key` can be assigned in this way:

```
final itemKey = ValueKey<String>("item-id-0025");

// and then on the build method...
Text(
  itemText,
  key: itemKey,
)
```

Use a `ValueKey` when you have an object represented by an unique and constant value.

In this case, the `String` doesn't change and the list doesn't have duplicates so it's a good choice. It is the default type of key returned by `Key`:

```
abstract class Key {  
  const factory Key(String value) = ValueKey<String>;  
}
```

- `ObjectKey`. Suppose you had a list of complex objects, such as `List<Task>` where `Task` is internally made up of other classes.

```
final list = [  
  Task(  
    owner: const OwnerData(...),  
    date: "...",  
    duration: const Duration(...),  
  )  
]
```

When you're not guaranteed that a single field is unique but a combination of multiple values is, go for an `ObjectKey`. In this case, we're sure that each `Task` is unique but some might have the same `date` for example. However, we know that there cannot be 2 tasks with the same owner/date/duration combination so the object itself is unique.

- `UniqueKey`. This key is only equal to itself: there's only one across the entire app. Use an `UniqueKey` when there are no constant unique values (so no `ValueKey`) and no single combinations of values (so no `ObjectKey`).
- `GlobalKey`. You will see it in action in chapter 19 as it's also used to work with input validation on form fields. Generally, global keys are useful to keep in sync the state of multiple widgets.

In practice, a `ValueKey` is used when a single value can **uniquely** represent an object (like an id). `ObjectKey` is good when there isn't a single unique value but a combination of properties (such as name, surname, birthday and fiscal code) can be unique. In Flutter, we will see you'll need a `GlobalKey`. In any other case, go for a `UniqueKey`.

UniqueKey



ValueKey<T>



ObjectKey



The bar at the bottom represents how many fields combined together represent an unique entity. In general keys can safely be ignored because there are only a few cases in which they're useful (in fact `key` is optional). Here's when you might have the need to uniquely identify a widget with a Key:

1. In chapter 16 you'll see that a key can be useful while testing to easily identify a widget on the tree. Being it unique, Flutter can quickly reach the widget on the tree and obtain a reference to it.
2. Imagine you had two tabs having, in both pages, a scrollable list. You want to store the scroll position even when tabs are swiped so that the user doesn't have to start scrolling from the top every time.

```
// Tab layout is covered in chapter 21
TabBarView(
    controller: tabController,
    children: [
        ListView.builder(
            key: const PageStorageKey<String>('list1'),
            itemBuilder: (context, index) {...},
        ),
        ListView.builder(
            key: const PageStorageKey<String>('list2'),
```

```
        itemBuilder: (context, index) {...},  
    ),  
],  
)
```

A `PageStorageKey<T>` (subclass of `ValueKey<T>`) is used to remember the scrolling position of a list when the page of a tab is changed. If you didn't use a `PageStorageKey`, the scroll position of the lists will be reset to 0 every time that a tab is changed (the position is not remembered by default).

Keys are also useful when you want to swipe to dismiss an item from a list: you'll see an example later in 19.2.1. Of course, you might decide to define an unique key for any widget you create but it would be useless.

9.3 Rebuilds and optimization

The framework traverses the widget tree very often. The `build()` method is called, for sure, the first time the UI is rendered. It will be called more than once during your app's lifetime but you can't predict how many times because lots of factors can trigger a re-build:

- calling `setState`,
- rotating the screen of the device,
- awaiting the result of a future,
- listening to incoming stream events.

When we say "the framework does many rebuilds" we mean that the method `build()` of a specific widget is called more than once. Because of the structure of the widget tree, every children will be rebuilt as well because there must be consistency along the hierarchy.

i Flutter is very efficient at traversing the widget tree and rebuilding the leaves. However, if you write bad code your app might suffer of performance issues or it won't always run at 60 fps (on average).

Even if Flutter is very fast, you don't have to abuse of its efficiency because your goal should always be: "*allow rebuilds of widgets **only** when it's really needed*". Let's see what can be done in order to write good code that doesn't waste time and memory.

9.3.1 const constructor

You already know something from 4.3.1 "const constructors" and now it's time to see why they're so useful in Flutter. Let's say you had this simple widget:

```
class ExampleWidget extends StatelessWidget {  
    const ExampleWidget();  
  
    @override  
    Widget build(BuildContext context) {...}  
}
```

Since there's a constant constructor defined for this class, you're allowed to create a constant list of widgets. Of course, it would have been the same if you used a `Row` or a `Column`:

```
ListView(  
    children: const [  
        ExampleWidget(),  
        ExampleWidget(),  
        ExampleWidget(),  
        ExampleWidget(),  
    ]  
,);
```

If you mark a list with `const` by consequence every object inside it will also be constant. Flutter builds constant widgets **one time** only. Using `const` constructors on widgets is like caching them: once created, they will never be re-built again.

i It really makes sense! If the class is allowed to have a `const` constructor, then it's *immutable*. It will never change over the time so Flutter doesn't have to rebuild it more than once. A constant constructor on a big subtree can save a lot of computational time.

Try to use `const` constructors as much as possible because the `build` method of a constant widgets is executed only **once** (at the time of the creation). Any subsequent re-build will simply ignore every widget whose constructor have been marked with `const`.

```
class ExampleWidget extends StatelessWidget {  
    // No constant constructor
```

```
@override
Widget build(BuildContext context) { .}
}
```

In this case there isn't a constant constructor and thus the widget cannot be inserted in the tree using `const ExampleWidget()`. Stateful widgets can have a `const` constructor too of course:

```
class Example extends StatefulWidget {
  const Example();

  @override
  _ExampleState createState() => _ExampleState();
}
```

Try to use constant constructors as much as possible but don't get obsessed with them because they can't be created in every situation. In certain cases, `const` constructors can cache very large subtrees and save much computational time!

constant version

```
ListView(
  children: const [
    ExampleWidget(),
    ExampleWidget(),
    ExampleWidget(),
    // ... + other 7 entries
  ],
);
```

not-constant version

```
ListView(
  children: [
    ExampleWidget(),
    ExampleWidget(),
    ExampleWidget(),
    // ... + other 7 entries
  ],
);
```

The visual difference is minimal but the computational difference is big. If `ExampleWidget` had a very complex `build()` method, the performance gap would be even bigger. Without `const`, the entire list is unnecessarily rebuilt many times.

9.3.2 Prefer widget composition over functions

It's common knowledge that code duplication is bad and so you'll create very often reusable widgets. For example, many apps have a "footer" which includes icons and a bit of text about the copyrights.

```
class FooterWidget extends StatelessWidget {
    const FooterWidget();

    @override
    Widget build(BuildContext context) {
        return Column(
            mainAxisAlignment: MainAxisAlignment.min,
            children: [
                Row(
                    mainAxisAlignment: MainAxisAlignment.spaceAround,
                    children: const [
                        Icon(Icons.email),
                        Icon(Icons.tablet_mac),
                    ],
                ),
                const Text("Developed by X"),
            ],
        );
    }
}
```

We've decided to make `FooterWidget` stateless because it's a reusable block of code that doesn't change its state and it's not influenced by external events. It lives on its own and it can be reused in many different pages to show a footer at the bottom:

- there is a constant constructor because the class has no mutable variables;
- in `Column` we can't use `children: const [...]` because `Row` does not define a constant constructor. Nevertheless, we can manually put `const` in the single child inside, where possible.

That's how you should write widgets and the same concept applies if it were a stateful one, no differences. Instead, what you absolutely **DON'T** have to do is this:

```
Widget footerWidget(BuildContext context) =>
    Column(
        mainAxisAlignment: MainAxisAlignment.min,
        children: [
            Row(
                mainAxisAlignment: MainAxisAlignment.center,
```

```
        children: const [
            Icon(Icons.email),
            Icon(Icons.tablet_mac),
            Icon(Icons.tune)
        ],
    ),
    const Text("Developed by X"),
]
);
}
```

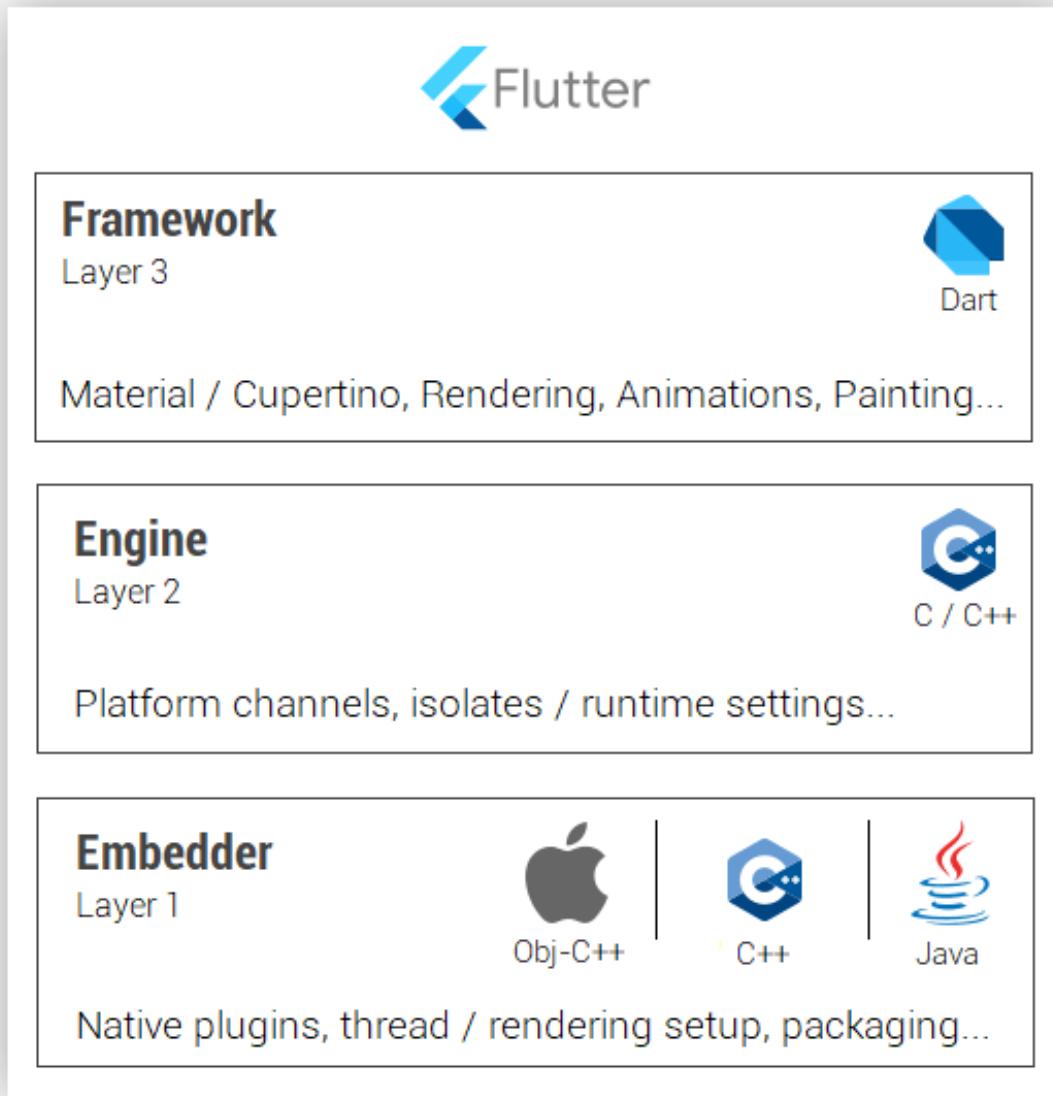
It's a function returning the `Column` widget with its children: you should absolutely **NEVER** prefer functions over widgets because:

- Functions of course doesn't have `const` constructors.
- Flutter is forced to rebuild widgets returned by a function **every time** because it knows nothing about them (no `BuildContext` is provided).
- Classes are leaves of the widget tree but functions aren't and thus there's no `BuildContext` available.

Widgets can be cached thanks to `const` constructors; functions can **NOT** be cached and thus they're executed every time. You should (or actually... must!) always rely on reusable widgets rather than functions.

9.4 Architecture

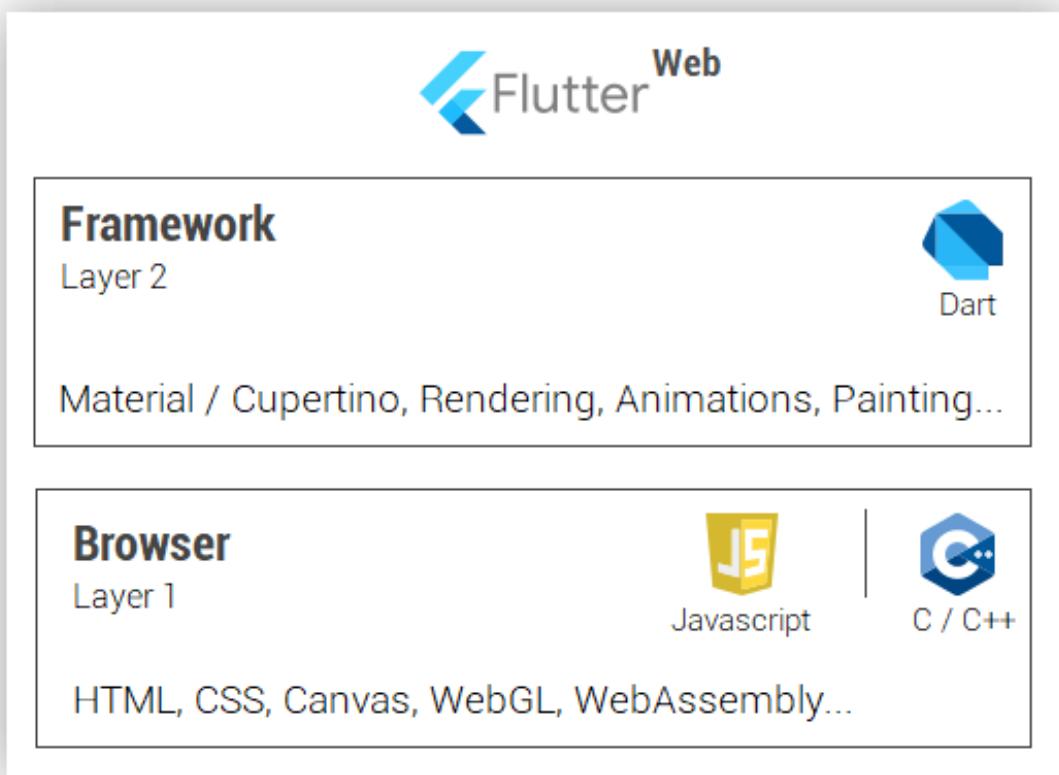
In this section we're giving a general overview of the architecture of the framework digging a bit more into the details. Flutter is divided into three layers (it's said to be a **layered system**) where each depends on the one below. Layers are made up of libraries written in different languages.



The **embedder** is written in different languages according with the platform in which Flutter has to run: Objective C++ for iOS / macOS, Java / C++ for Android and C++ for Linux / Windows. It's a native application that takes care of "hosting" your Flutter contents on the OS. When the app is started, the embedder provides a valid entrypoint, obtains threads for UI and rendering, starts the Flutter engine and much more.

- i** The embedder is at the lowest layer and it directly interacts with the operating system providing entry points for access to services. The developer mostly works on the third layer and sometimes on the second, but never on the first.

The **engine** is the heart of Flutter, it's mostly written in C++ and it's always packaged in the binary produced by the `flutter build` tool. It's a portable runtime for hosting Flutter application which includes core libraries for network I/O, file, animations and graphics. The engine is exposed to the developer via `import "dart:ui"`, which basically wraps C++ sources into Dart classes. For the web world, the situation is different:



The C++ engine is designed to work with the operating system but for the web Flutter has to

deal with a browser. For this reason, the approach has to be different. Dart can be compiled to JavaScript thanks to the highly-optimized **dart2js** compiler so, by consequence, Flutter apps can be ported as well. There are 2 ways to deploy an application for the web:

1. **HTML** mode. Flutter uses HTML, CSS, JavaScript and Canvas.
2. **WebGL** mode. Flutter uses CanvasKit, which is Skia compiled to WebAssembly.

For the web, there's no need for the Dart runtime because your Flutter app is compiled to JavaScript as we've already said. The produced code is already minified and it can be deployed to any server. At the time of publishing this book (September 2020), web support for Flutter is only available in the beta channel.

i In case you didn't know, **WebAssembly** is recognized ⁹ by the W3C as the 4th language to natively run on browsers along with HTML, CSS, and JavaScript. WebAssembly can be both AOT and JIT compiled.

9.4.1 Element and RenderObject

You've already seen that, to build the UI, the developer has to create a **widget tree** by nesting widgets one inside the other. In reality, Flutter doesn't only rely on widgets because internally there are two other kinds of trees maintained in parallel ¹⁰. Through this section, we're assuming that **SomeText** is just a simple widget showing some text.

```
class MyWidget extends StatelessWidget {  
    const MyWidget();  
  
    @override  
    Widget build(BuildContext context) {  
        return Container(  
            decoration: BoxDecoration(),  
            child: SomeText(  
                text: "Hello"  
            ),  
    );  
}
```

⁹<https://www.w3.org/TR/wasm-core-1/>

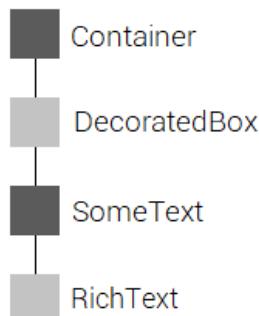
¹⁰See "The Layer Cake" by Frederik Schweiger on Medium

```
    }  
}
```

When it's time to render, Flutter calls the `build()` method of the widget which might introduce some new widgets, in case of nesting. In our case, the widget tree will contain `Container`, `SomeText` plus some more you actually don't see. In fact, if you looked at the definition of a `Container`...

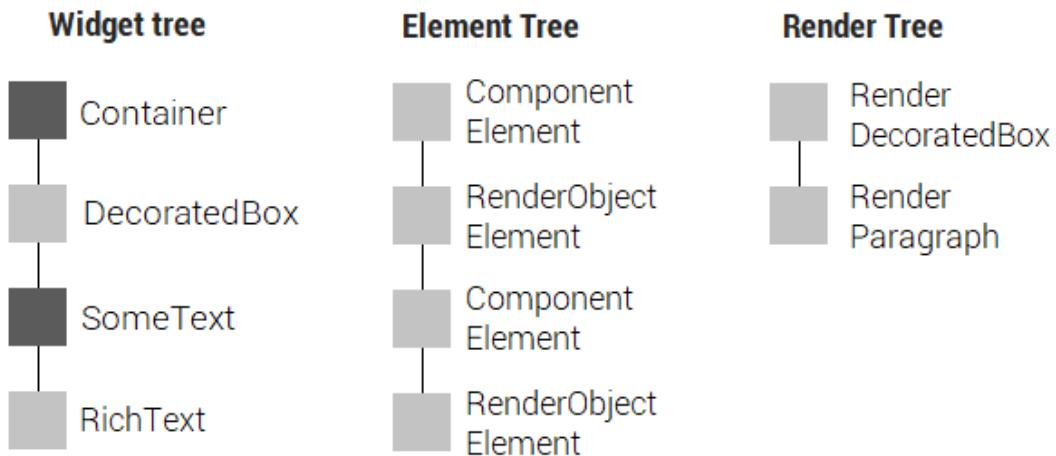
```
if (decoration != null)  
    current = DecoratedBox(decoration: decoration, child: current);
```

... you'd notice that an instance of `DecoratedBox` is added under the hood if a `decoration` is given. For this reason, if you made a DevTools¹¹ inspection you'd see more children than you actually inserted. It's because widgets might insert other widgets inside but you just don't see it; the tree actually looks like this:



Some boxes are in grey to visualize the fact they haven't been added by you. Along with the widget tree, Flutter also builds in parallel the `element` tree and the `render` tree. They are created calling respectively `createElement()` and `createRenderObject()` on the widget being traversed. Note that `createElement()` is always called on widgets but `createRenderObject()` is only called on elements whose type is `RenderObjectElement`. So yes, at the end Flutter works with 3 trees.

¹¹More on it in chapter 16



An **Element** can hold a reference to a widget and the respective **RenderObject**. There are a lot of new things you've never seen up to now so let's carefully analyze the trees to understand how Flutter really works.

- **Render tree.** A **RenderObject** contains all the logic to render the corresponding widget and it's expensive to create. They take care of the layout, the constraints, hit testing and painting. The framework keeps them in memory as much as possible, changing their properties whenever there's a chance. They can be of many types:

- **RenderFlex**
- **RenderParagraph**
- **RenderBox** ...

During the build phase, the framework updates or creates a new type of **RenderObject** only when a **RenderObjectElement** is encountered in the element tree.

- **Element tree.** An **Element** is the link between a **Widget** and its respective **RenderObject** so it holds references inside. **Elements** are very good at comparing items and looking for changes but they don't perform rendering. They can be of two types:

- **ComponentElement.** An element that contains other elements. It's associated to a widget that can nest other widgets inside.

```
abstract class ComponentElement extends Element { ... }
```

- `RenderObjectElement`. An element that takes part in painting, layout and hit testing phases.

```
abstract class RenderObjectElement extends Element { ... }
```

The element tree is basically a series of `ComponentElement` or `RenderObjectElement`, depending on the widget they refer to. In our example, a `Container` is a `ComponentElement` because it can host other widgets inside.

- **Widget tree.** It's made up of classes extending `StatelessWidget` or `StatefulWidget`. They're used by the developer to build the UI and are not expensive to be created (much less than a `RenderObject`).

Whenever the widget tree is changed (by a state management library for example), Flutter uses the element tree to make a comparison between the new widget tree and the render tree. An `Element` is a "middle way" between a `Widget` and a `RenderObject` used to make quick comparisons needed to keep the trees updated.

1. A `Widget` is "light" and it's instantiated quickly so frequent rebuilds aren't a problem at all. Widgets are **all** immutable and that's why the state of a `StatefulWidget` is implemented in another separated class. A stateful widget itself is immutable but the state it returns can mutate.

```
class Example extends StatelessWidget {
    const Example();

    @override
    _ExampleState createState() => _ExampleState();
}

class _ExampleState extends State<Example> {
    @override
    Widget build(BuildContext context) { ... }
}
```

The widget itself (`Example`) is immutable and so its mutable state (`_ExampleState`) is implemented in another class. A `StatelessWidget` is immutable as well.

2. A `RenderObject` is relatively "expensive" and it takes time to instantiate so it's recreated only when really needed. Most of the times they're internally modified (reusability is the key).

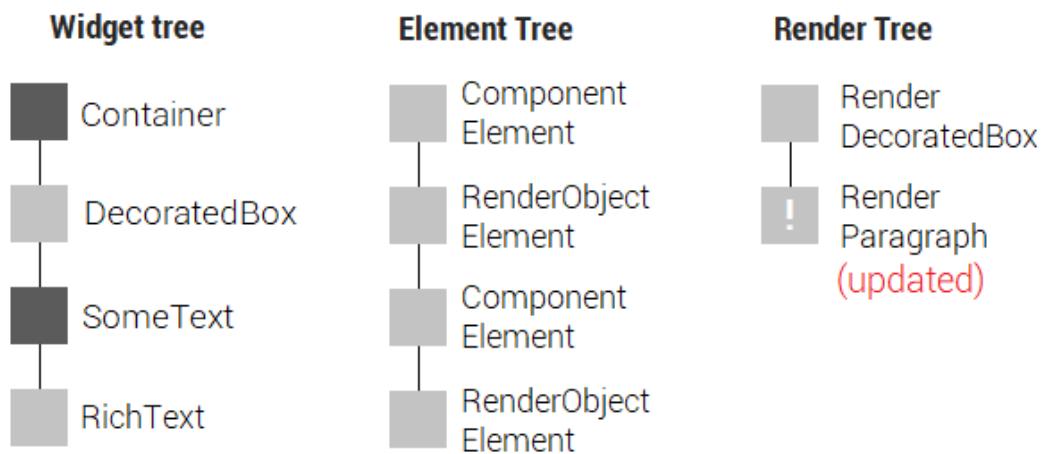
For each rebuild, Flutter traverses the entire tree looking for changes on widgets. If the type of the `Widget` changed, then it'd be removed and replaced **together** with its associated `Element` and `RenderObject`. All the 3 subtrees would also be recreated. If the `Widget` were of the same type and just some properties changed, the `Element` would stay untouched and the `RenderObject` would be updated (and **not** recreated). Let's see an example:

```
Widget build(BuildContext context) {
    return Container(
        decoration: BoxDecoration(),
        child: SomeText(
            text: "Hello"
        ),
    );
}
```

This is what we had earlier. Of course, on the first build the 3 trees are entirely created but from now on, the framework will try to recreate the render tree as less as possible. Let's say our state management library changed the text of `SomeText`.

```
Widget build(BuildContext context) {
    return Container(
        decoration: BoxDecoration(),
        child: SomeText(
            text: "Hello world!"
        ),
    );
}
```

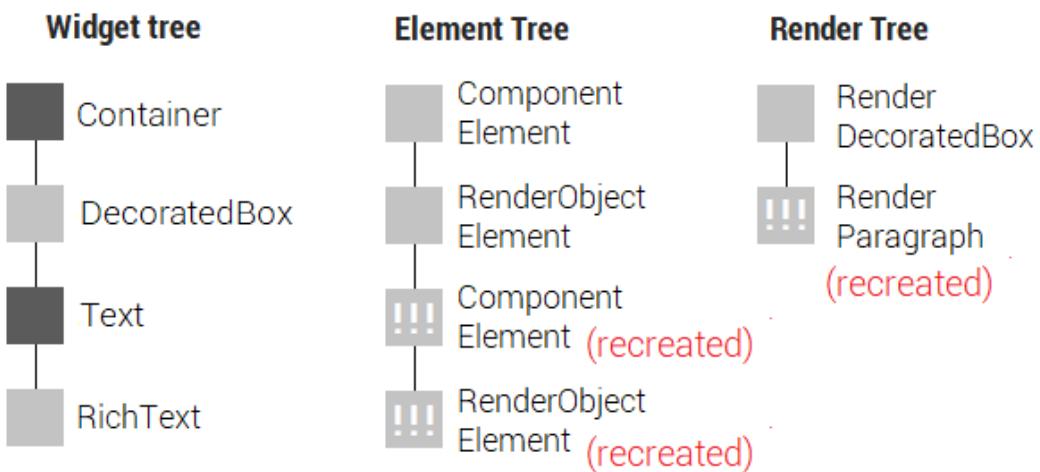
When a rebuild happens, thanks to the element tree, Flutter notices that the type is still the same (`SomeText`) but an internal property (`text`) has changed. By consequence, the associated `RenderObject` just needs an update, which is cheap.



This process is very fast because the `RenderObject` is not recreated but it's just modified. Widgets and elements are also quick to update so this is a good situation. Let's now say that our library replaces `SomeText` with Flutter's `Text` widget.

```
Widget build(BuildContext context) {
  return Container(
    decoration: BoxDecoration(),
    child: Text("Hello world!"),
  );
}
```

While traversing the tree, the framework notices again the change thanks to the element tree. In particular, this time the type of the widget is completely different so there's the need to rebuild the entire subtrees (widgets, elements and renders).



The associated `RenderObject` is not updated: it has to be entirely **recreated** because the widget has a different type and thus there's no way to reuse the old instance. In summary, Flutter relies on 3 trees to efficiently handle the rendering and tries to reuse `RenderObjects` as much as possible. Thanks to `Elements`, the framework knows when something has changed on `Widgets`.

i The `BuildContext` parameter you see in any `build()` method basically represents the `Element` associated to the widgets. In reality, `BuildContext` objects are `Element` objects. The Flutter team created `BuildContext` to avoid the direct interaction with `Element`, which should be used by the framework and not by you.

The render tree is the one that actually takes care of painting elements to the UI. The widget tree is manually built by you, the developer. The element tree is maintained by the framework to decide whether it's time to update or recreate a `RenderObject`.

9.4.2 Foreign Function Interface

Thanks to the `dart:ffi` library, also known as **Foreign Function Interface**, your Dart code can directly bind to native APIs written in C. FFI is very fast because there's no serialization required to pass data since calls are made to dynamically or statically linked libraries. Here's a an example:

```
// demo.h
void print_demo() {};
```

```
// demo.c
#include <stdio.h>
#include "demo.h"

void print_demo() {
    printf("Dart FFI demo!");
}

int main() {
    print_demo();
    return 0;
}
```

We're going to call `void print_demo()` written in C inside a Dart app thanks to FFI. To keep the example simple, we assume that every file is in the same folder and the following Dart code is all inside `main.dart`. Let's start with the fundamentals:

```
import "dart:ffi" as FFI;

// Signature of the function in C
typedef print_demo_c = FFI.Void Function();
// Signature of the function in Dart
typedef PrintDemo = void Function();
```

The first `typedef` uses FFI to represent the signature of the C function we're going to call. It's basically used to represent the C function into its Dart counterpart, identified by `PrintDemo`. Of course, you have to declare two `typedef` whose signatures match.

```
import "dart:ffi" as FFI;

typedef print_demo_c = FFI.Void Function();
typedef PrintDemo = void Function();

void main() {
    // Open the library
    final path = "demo_lib.dll"; // On Windows
    final lib = FFI.DynamicLibrary.open(path);

    // Create a "link" from C to Dart
```

```

final PrintDemo demo = lib
    .lookup<FFI.NativeFunction<print_demo_c>>('print_demo')
    .asFunction();

// Call the function
demo();
}

```

In general, when working with FFI you always have to create two `typedef`: one for the "C side" and the other for the "Dart side". When building the C code, various files are created but you're only interested in the one with the following extension: `.dll` on Windows, `.so` on Linux and `.dylib` on macOS. On Windows, be sure that your compiler properly exports to the DLL the functions Dart has to use.

```

int sum(int a, int b) {
    return a + b;
}

```

The above code can easily be used by Dart in the same way we did earlier in the `demo` function. Inside `dart:ffi` you'll find many types representing the C primitive ones, such as `Int32`, `Double`, `UInt32`, `Handle` and much more.

```

typedef sum_c = FFI.Int32 Function(FFI.Int32 a, FFI.Int32 b);
typedef Sum = int Function(int a, int b);

```

Check out the official documentation ¹² for some nice examples on how to interact with `structs`, strings and SQLite databases.

9.4.3 Method channels

Available only for mobile and desktop, *method channels* allow Dart to call platform-specific code of your hosting app. Data are serialized from Dart and then deserialized in Java, Kotlin, Swift or Objective-C. Look how easy it is:

```

const channel = MethodChannel("person");
final name = await channel.invokeMethod<String>("getPersonName");
print(name); // 'name' is a regular Dart string

```

As example, let's say the above code is going to call the `getPersonName(): String` function declared in a native Android app written in Kotlin. There's a similar setup to do in the native

¹²<https://api.dart.dev/stable/2.9.2/dart-ffi/dart-ffi-library.html>

part as well but it's very simple to understand:

```
// Initialization
val channel = MethodChannel(flutterView, "person")
channel.setMethodCallHandler {call, result ->
    when (call.method) {
        "getPersonName" -> result.success(getPersonName())
        else -> result.notImplemented()
    }
}

// This function is defined somewhere
fun getPersonName(): String {
    return "Alberto"
}
```

In both cases, the `MethodChannel` instance has to be created with the same name (`"person"`) otherwise the "link" between Dart and Kotlin won't work. The name of the function matches the actual name on the native side just for convenience but it's not required. With `invokeMethod<T>()` you can also pass parameters in case the function were asking for some. For example, if you called this in Dart...

```
const channel = MethodChannel("random");
final random = await channel.invokeMethod<int>("getRandom", 60);
```

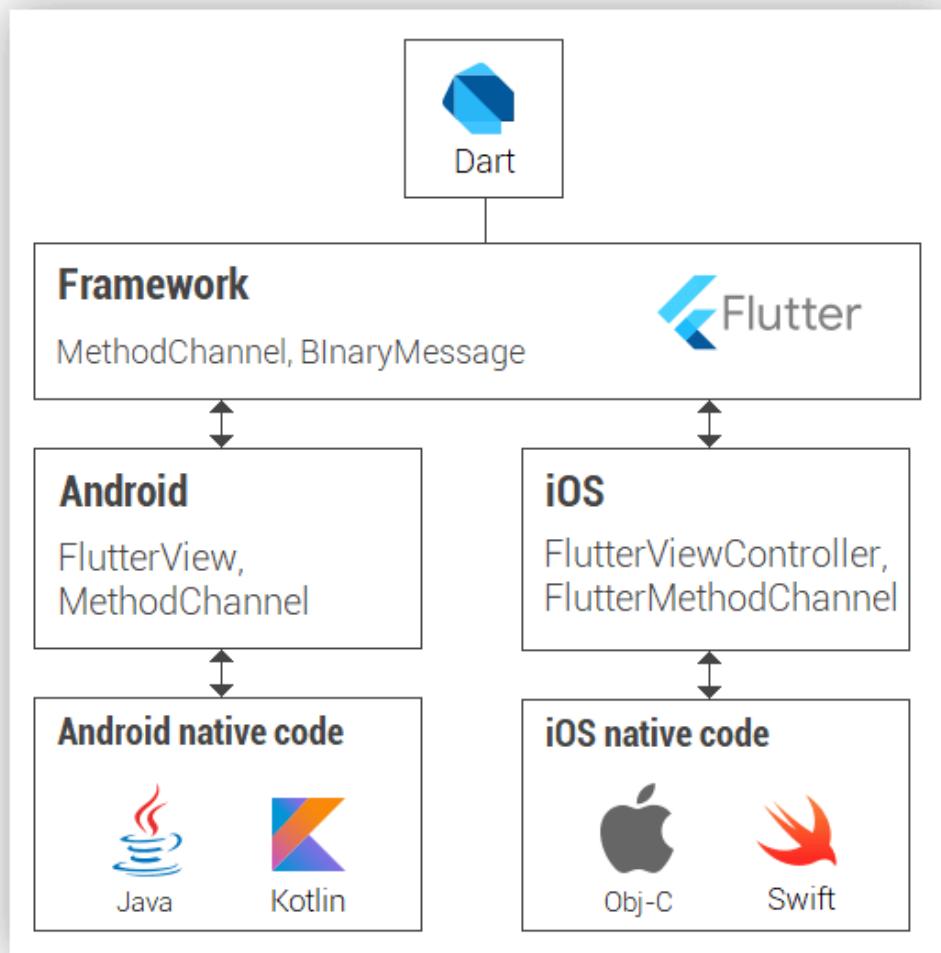
... it would mean that you're expecting a method on the native language called `getRandom` asking for a single integer parameter. This example instead is written in Swift but the logic is always the same (just a different syntax):

```
// Initialization
let chl = FlutterMethodChannel(name: "random", binaryMessenger: flutterView)
chl.setMethodCallHandler {
    (call: FlutterMethodCall, result: FlutterResult) -> Void in
    switch (call.method) {
        case "getRandom": result(getRandom(call.arguments as! Int))
        default: result(FlutterMethodNotImplemented)
    }
}

// This function is defined somewhere
```

```
func getRandom(value: Int) -> Int {  
    return Int.random(in: 0...value);  
}
```

Thanks to `call.arguments` you access the argument passed via method channel which could be, for example, a primitive type or a map. A `MethodChannel` is a common interface for both Dart and the other native language that allows Flutter to send/receive messages. This is a scheme of how method channels are implemented:



In the native code, method channels must be called in the main thread and not in a background

one (in Android, the "main" thread is actually called **UI** thread). To sum up, the communication flow works like this:

1. Flutter sends a message to the iOS or Android part of the app using a method channel;
2. the underlying system listens on the method channel and so the message is received;
3. one or more platform-specific APIs are called, using the native programming language;
4. a response is sent back to the client (Flutter) which processes the result.

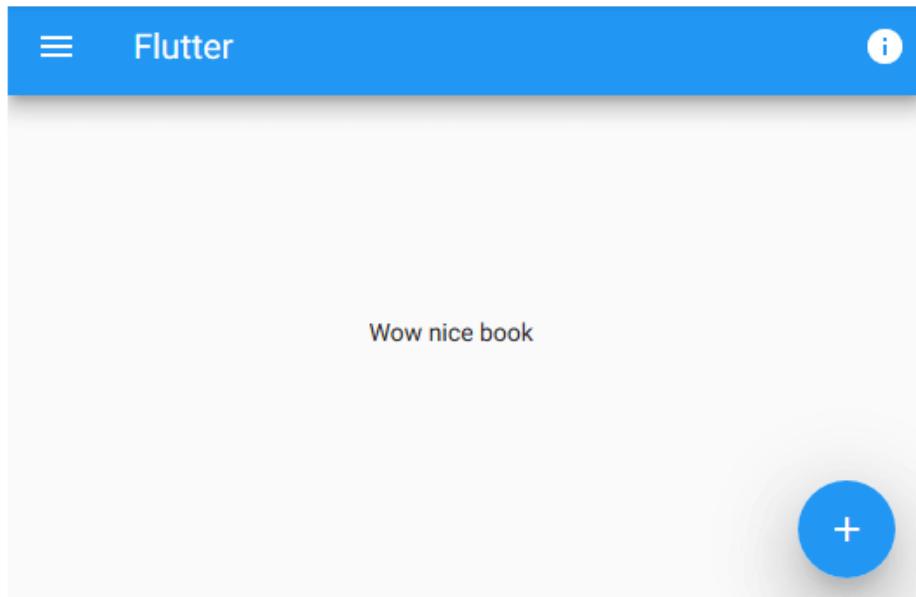
You can't do the same with FFI because there are no libraries to be linked and data need serialization/deserialization: method channels work differently and require a native implementation as well.

10 | Building UIs in Flutter

Flutter allows the developer to completely customize the UI: you have control over each single pixel of the screen. Layouts can be created from scratch but for common use-cases there are a series of built-in widgets that are going to save hours of work.

10.1 Material

Flutter gives you a series of pre-built components to create apps embracing the typical Android design, also known as *Material Design*. It's very likely you've already seen this kind of UI appearance somewhere, in landscape mode:



This is a classic example of material design with a *Floating Action Button* (FAB) on the bottom right and an app bar at the top. There are two possible ways to create the above material layout:

- **Not recommended.** Create the entire layout from scratch using stateless and stateful widgets. Actually there'd be quite a lot of work to do because you'll have to deal with screen dimensions, positioning, buttons and so on.
- **Recommended.** Import the *material.dart* package and use the `MaterialApp()` widget provided by Flutter. It represents the "skeleton" of a UI following the material design guidelines¹; it's very convenient:

```
Widget build(BuildContext context) {  
    return MaterialApp(  
        home: Scaffold(  
            appBar: AppBar(  
                title: const Text("Flutter"),  
                actions: const [  
                    Padding(  
                        padding: EdgeInsets.only(right: 20),  
                        child: Icon(Icons.info),  
                    )  
                ]  
            ),  
            drawer: const Drawer(),  
            body: const Center(  
                child: Text("Wow nice book"),  
            ),  
            floatingActionButton: FloatingActionButton(  
                onPressed: () {},  
                child: const Icon(Icons.add),  
            ),  
        ),  
    );  
}
```

The big advantage is that you don't have to write thousands of lines of code trying to

¹<https://material.io/design/guidelines-overview/>

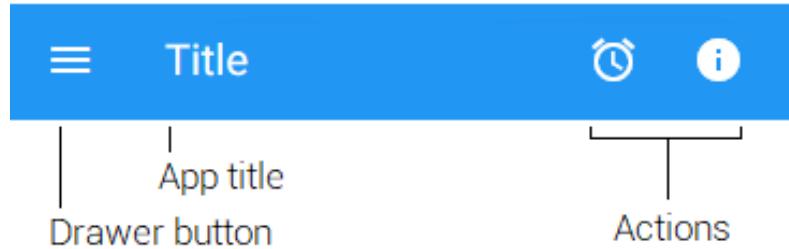
emulate the material design aspect. Flutter already gives you everything you need and with `MaterialApp` you're guaranteed to create a beautiful material app.

The constructor of `class MaterialApp(...)` has many interesting parameters we will cover later in the book, such as the ones for setup pages navigation and localization. We're now giving you a showcase of the most relevant material widgets.

10.1.1 Scaffold

As you've seen in the preceding code snippet, `class Scaffold(...)` implements the basic material design layout structure for you. Other than providing the typical Android "look-and-feel" it gives the possibility to handle many other widgets:

- **AppBar.** It's always placed at the top of the screen and it's the Java/Kotlin equivalent of the `Toolbar` class. If the `Scaffold` had a drawer, an hamburger button would automatically be added to handle the opening/closing of the menu.



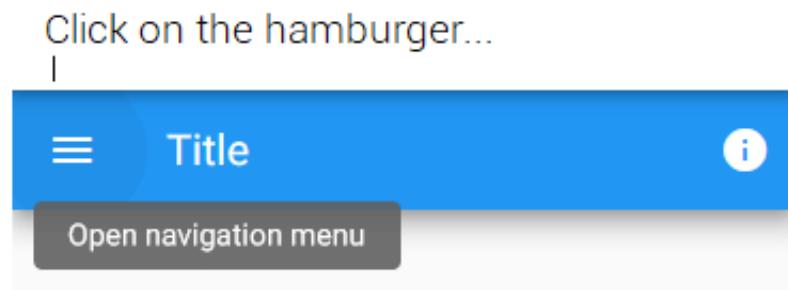
While navigating back and forth between your app's pages (or *routes*, in the Flutter world) the `AppBar` automatically adds a "back" button, the typical left arrow.

```
Scaffold(  
    appBar: AppBar(  
        // Set this to false if you don't want the  
        // back button to automatically appear next  
        // to the title while navigating among pages  
        automaticallyImplyLeading: false,  
        // Title  
        title: const Text("App Bar without Back Button"),  
    )  
)
```

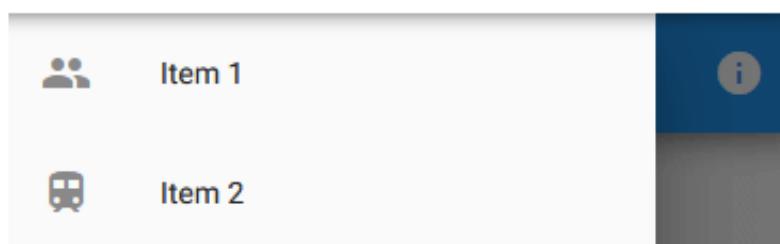
Buttons on the right are called *action buttons* and they can be set by passing a list of widgets to the `actions` named parameter. In general, actions are clickable icons that visually represent what's the purpose of that button.

```
Scaffold(  
    appBar: AppBar(  
        actions: [  
            IconButton(  
                icon: const Icon(Icons.info),  
                onPressed: () {...}  
            ),  
        ]  
    )  
)
```

- **Drawer.** A drawer is a container that horizontally slides from a side of the screen to show a series of items. In general it's used to display a combination of icons and texts that route the user to specific pages of the app.



... and this container appears!



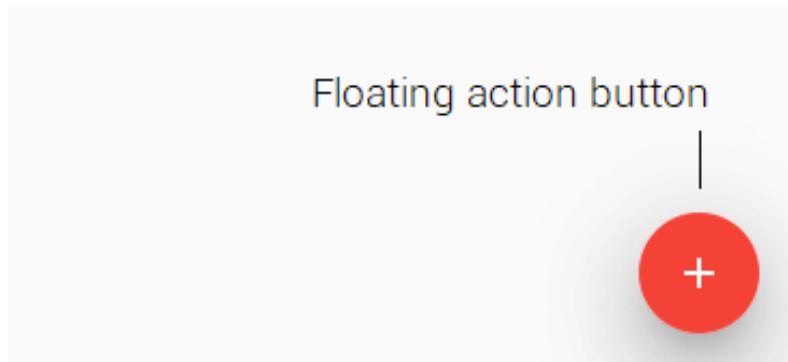
By default a drawer slides from the left to the right but you can also create an `endDrawer` which slides in the opposite direction, from right to left. It's still the same `Drawer()` class but it's assigned to another constructor named parameter.

```
Scaffold(  
    // The 'classic' left to right drawer  
    drawer: Drawer(  
        child: ListView(  
            ListTile(  
                leading: const Icon(Icons.people),  
                title: const Text("Item 1"),  
                onTap: () {},  
            ),  
            ListTile(  
                leading: const Icon(Icons.train),  
                title: const Text("Item 2"),  
                onTap: () {},  
            )  
        ),  
        // The same as before but this slides  
        // from the right to the left  
        endDrawer: Drawer()  
    )
```

- **Floating Action Button.** Also known as *FAB*, it is a special rounded button with elevation that usually appears on the bottom-right corner of the screen. By using a `floatingActionButtonLocation` you can decide the position of the widget:

```
Scaffold(  
    floatingActionButton: FloatingActionButton(  
        child: const Icon(Icons.add),  
        backgroundColor: Colors.red,  
        onPressed: () {...},  
    ),  
)
```

This snippet adds a FAB to the default position of the screen (bottom-right):



Do you want to have it at the center rather than on the right? There are many positions available you can use:

```
floatingActionButton: FloatingActionButton(
    child: const Icon(Icons.add),
    backgroundColor: Colors.red,
    onPressed: () {},
),
floatingActionButtonLocation:
    FloatingActionButtonLocation.centerFloat,
```

A **Scaffold** can only have a single FAB.

The **Scaffold** widget is for sure very important as it's the base building block for material user interfaces and probably you'll use it very often. In the third part of the book you'll see many examples highlighting the strength of this widget.²

10.1.2 Material widgets

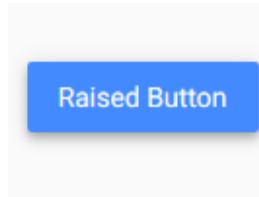
Flutter provides a very big collection of widgets that follow the official material guidelines. The Flutter team constantly improves them and adds new ones as time goes by. We're not going to make a full list of every kind of widget because you'll encounter them in the practical examples of Part III.

²<https://api.flutter.dev/flutter/material/Scaffold-class.html>

10.1.2.1 Buttons

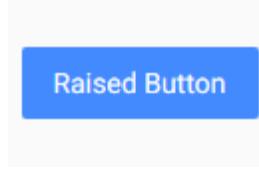
Buttons are fundamental in any kind of app because they're the most intuitive way to tell the user that, when pressed, something is going to happen. Flutter has many material widgets created to follow material guidelines for buttons.

- **RaisedButton.**



This is a typical Android button with a rectangular shape and a default elevation (the shadow behind). When hovered the elevation increases so that you can visually see the interaction with the button.

- **FlatButton.**



It's very similar to a `RaisedButton` with the difference that here there is no elevation and nothing happens visually when you tap on it. It's like a "static" version of a `RaisedButton`.

- **ButtonBar.**



It's an horizontal container holding a series of button. It can be useful in those cases where you have a dialog and you want to show two buttons like NO and YES.

```
ButtonBar(  
    alignment: MainAxisAlignment.center,
```

```
children: [
    FlatButton(
        onPressed: () {},
        child: const Text("No")
    ),
    RaisedButton(
        onPressed: () {},
        child: const Text("YES")
    ),
],
);
```

- **IconButton.**

You've already seen an example in the `AppBar` as they make icons clickable; it's a button whose content is a material `Icon` rather than a plain string.

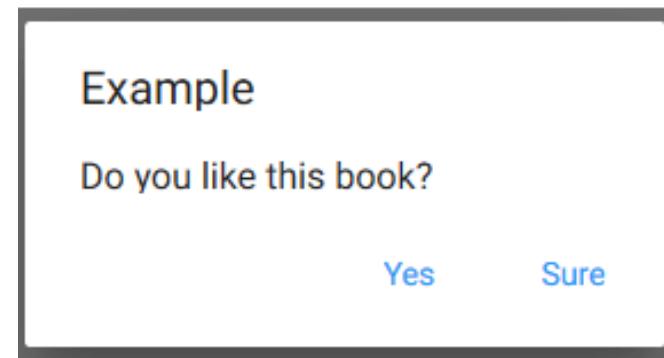
10.1.2.2 Dialogs

The most common type of dialog is the one that asks you for the confirmation or rejection of an action; this kind of widget is well-known as `AlertDialog`. You have to use the `showDialog(...)` method to make it appear on the screen.

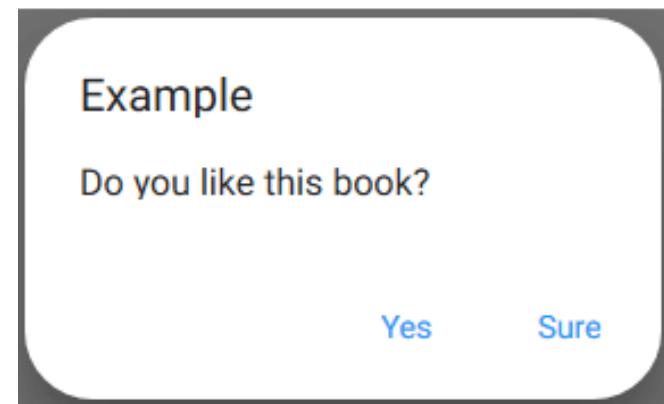
```
showDialog<void>(
    context: context,
    builder: (BuildContext context) {
        return AlertDialog(
            title: const Text("Example"),
            content: const Text("Do you like this book?"),
            actions: [
                FlatButton(
                    child: const Text("Yes"),
                    onPressed: () {},
                ),
                FlatButton(
                    child: const Text("Sure"),
                    onPressed: () {},
                )
            ]
        );
    }
);
```

```
        ]  
    );  
}  
);
```

By default a dialog can be closed just by tapping outside of the white area of the box. If you don't want this behavior, just set `barrierDismissible: false` and the dialog will disappear only if a button is pressed.



Of course you can fully customize the dialog as much as you want because the `content` parameter can be any widget such as an image. You could use icons or raised buttons instead of the flat ones. You could even change the borders of the dialog.



It's just a matter of using `RoundedRectangleBorder` and Flutter will take care of everything

else.

```
showDialog<void>(
    context: context,
    builder: (BuildContext context) {
        return AlertDialog(
            title: const Text("Example"),
            content: const Text("Do you like this book?"),
            actions: [...],
            shape: RoundedRectangleBorder(
                borderRadius: BorderRadius.circular(30),
            )
        );
    }
);
```

If you want the dialog to look like a circle, use instead `CircleBorder` as border shape. In order to close a dialog there's the need to use `class Navigator {}` which will be fully covered in chapter 12.

```
AlertDialog(
    actions: [
        FlatButton(
            child: const Text("Close"),
            onPressed: () =>
                Navigator.pop(context),
        ),
    ],
);
```

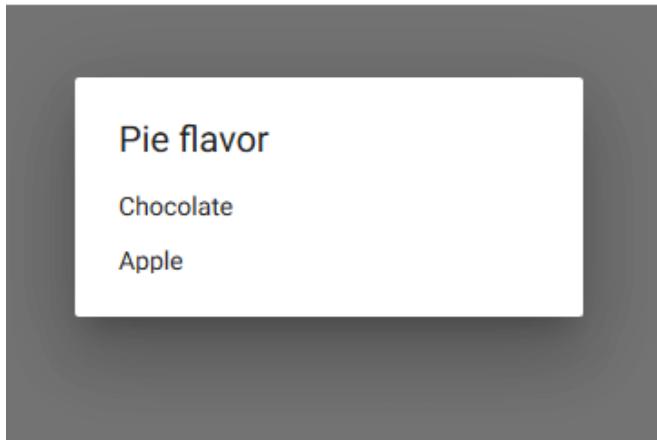
There are also other kinds of dialogs you could use in your applications:

- **SimpleDialog.** This is a simple dialog in which the user can choose between a series of options. The selected value will be asynchronously returned by `T showDialog<T>()`.

```
final type = await showDialog<String>(
    context: context,
    builder: (BuildContext context) {
        return SimpleDialog(
            title: const Text("Cake flavor?"),
            children: [
```

```
SimpleDialogOption(  
    onPressed: () =>  
        Navigator.pop(context, "chocolate"),  
    child: const Text("Chocolate"),  
,  
SimpleDialogOption(  
    onPressed: () =>  
        Navigator.pop(context, "apple"),  
    child: const Text("Apple"),  
,  
]  
);  
}  
);
```

Very shortly, the `Navigator` class is used to navigate between *routes* (your app's pages) and close alert dialogs (which are *routes* too). The `pop()` method removes the currently visible route from the screen and thus the alert disappears.



As always, this is fully customisable because instead of a boring plain text you could have put any widget such as images or a coloured text with a fancy font.

- **showBottomSheet**. This method animates a dialog that slides from the bottom of the

screen up to a certain height, which is determined by the size of the contained widget. In general it's a good idea having a `Container` as "base" child which has an easy setup for height, shape and colors.

```
showBottomSheet<String>(
    context: context,
    builder: (BuildContext context) {
        return Container(
            color: Colors.blueAccent,
            height: 40,
            child: const Center(
                child: Text(
                    "BottomSheet",
                    style: TextStyle(
                        color: Colors.white,
                )
            )
        ),
    );
}
);
```

The result of this code is a blue stripe at the bottom of the screen which slides up and shows its content. Calling `Navigator.pop(context)`; closes the dialog which slides down until it disappears.

10.2 Cupertino

The `CupertinoApp` widget is the Apple counterpart of `MaterialApp` as it focuses on the typical iOS design. It's a series of pre-built components that allow you to create applications that follow the typical iOS UI style.

Cupertino App

Cupertino theme!

(CupertinoPageScaffold)

The same recommendations we made for *material* also apply here. You could create an iOS theme from scratch but it would require a lot of time and testing; it's not worth the effort because you can use Flutter's *cupertino* components.

```
Widget build(BuildContext context) {  
    return CupertinoApp(  
        home: const CupertinoPageScaffold(  
            navigationBar: CupertinoNavigationBar(  
                middle: Text("Cupertino App"),  
            ),  
            child: Center(  
                child: Text("Cupertino theme!"),  
            ),  
        ),  
    );  
}
```

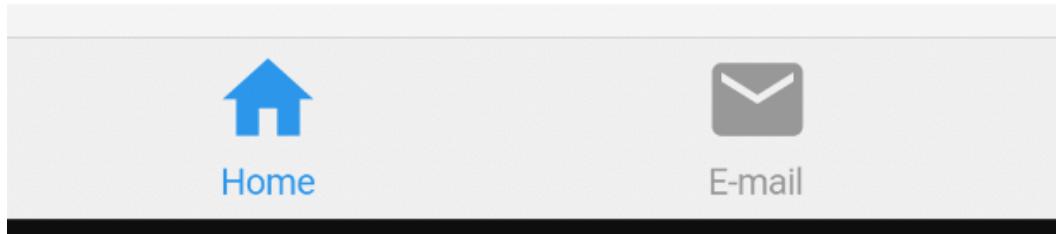
The constructor of `class CupertinoApp()` has many interesting parameters we will cover later in the book, such as the ones to setup page navigation and localization. We're now giving you a showcase of the most relevant *cupertino* widgets.

10.2.1 CupertinoPageScaffold

There are two main kind of scaffolds in the *cupertino* library: the `CupertinoPageScaffold`, which has a navigation bar at the top, and the `CupertinoTabScaffold`, which uses tabs to display contents.

```
CupertinoApp(  
    home: const CupertinoPageScaffold(  
        // It's a "plain" bar with the title and some icons  
        navigationBar: CupertinoNavigationBar(  
            middle: Text("Page title"),  
            trailing: Icon(CupertinoIcons.info),  
        ),  
        child: Center(  
            child: Text("Body of the app")  
        )  
    ),  
);
```

The `CupertinoTabBar` widget instead implements the tab navigation pattern in a typical iOS style. It allows multiple pages to be shown in a single view by tapping on the icons at the bottom but there's no swipe gesture enabled. The `onTap` callback is triggered when the user taps on an icon:



The only required widget is the icon but the text that lies underneath can be omitted if it's not needed. Visit the official documentation ³ to see any custom parameter that can be set for this widget.

```
CupertinoTabScaffold(  
    ...  
);
```

³<https://api.flutter.dev/flutter/cupertino/CupertinoTabBar-class.html>

```
tabBar: CupertinoTabBar(  
    onTap: (index) {...},  
    activeColor: Colors.blue,  
    items: const [  
        BottomNavigationBarItem(  
            icon: Icon(Icons.home),  
            title: Text("Home"),  
        ),  
        BottomNavigationBarItem(  
            icon: Icon(Icons.email),  
            title: Text("E-mail"),  
        ),  
    ],  
,  
    tabBuilder: (context, index) {...}  
,  
,
```

Once icons have been defined via `items`, the `tabBuilder` parameter defines which pages have to be shown when an item is tapped. When the tab becomes inactive, its content is automatically cached in the widget tree for better reusability in subsequent calls.

```
CupertinoTabScaffold(  
    tabBar: CupertinoTabBar(...)  
    tabBuilder: (context, index) =>  
        CupertinoTabView(  
            builder: (context) {  
                switch (index) {  
                    case 0:  
                        return const PageOneWidget();  
                    case 1:  
                    default:  
                        return const PageTwoWidget();  
                }  
            },  
        ),  
,  
,
```

In general, `CupertinoTabScaffold` should be your first choice when a tabbed layout is required for your UI; if it's not the case, go for `CupertinoPageScaffold` which is basically a "plain" iOS

page.

10.2.2 Cupertino widgets

Flutter provides a very big collection of widgets that follow the iOS design guidelines. At the time of writing this book, the *cupertino* library has less widgets than *material* but the Flutter team has stated in their roadmap ⁴ that the collection is going to grow over the time.

- **CupertinoAlertDialog.**



An iOS alert dialog that notifies the user and requires an action, defined by buttons. Typically an instance of `CupertinoAlertDialog` is passed as child widget to `showDialog()`, which displays the dialog.

```
showDialog<void>(
    context: context,
    builder: (context) {
        return CupertinoAlertDialog(
            title: const Text("Cupertino Alert"),
            content: const Text("iOS alert dialog"),
            actions: <Widget>[
                CupertinoButton(
                    child: const Text("Ok"),
                    onPressed: () => Navigator.pop(context),
                )
            ]
        );
    }
)
```

⁴<https://github.com/flutter/flutter/wiki/Roadmap>

```
        ],
    );
}
),
```

Generally iOS dialog buttons are red, in case of a deletion action, or in old blue, in case of a default option. Rather than using a `CupertinoButton`, a `CupertinoDialogAction` would be better:

```
CupertinoAlertDialog(
    actions: <Widget>[
        CupertinoDialogAction(
            isDefaultAction: true,
            child: const Text("Ignore"),
            onPressed: () {...},
        ),
        CupertinoDialogAction(
            isDestructiveAction: true,
            child: const Text("Delete"),
            onPressed: () {...},
        ),
    ],
);

```

`isDestructiveAction: true` makes the text red while `isDefaultAction: true` makes it blue.

- **CupertinoButton.**



iOS Button

It's a typical flat iOS button which has no background color by default but of course it can be set via `color` property. When tapped, the `onPressed` callback is triggered.

```
CupertinoButton(
    child: const Text("iOS Button",
    style: TextStyle(
```

```
        color: CupertinoColors.white  
    ),  
),  
color: CupertinoColors.activeBlue,  
onPressed: () {...},  
) ,
```

The *cupertino* library has less widgets than *material* because an iOS button style has less "variants" than the Android counterpart but still it's fully customizable. Be sure to check the *cupertino* catalog⁵ in the official Flutter documentation.

10.3 Building layouts

10.3.1 Platform support

You have just seen that the framework has a lot of useful pre-built components you can use to create beautiful UIs. Before starting the development you have to think about what the end user needs and how many platforms you have to support at the same time.



Mobile



Web



Desktop

At the time of writing this book, Flutter is at production quality only for mobile devices. Web support is in beta while desktop support is still in early alpha. Nevertheless, it's just a matter of time because in the future Flutter will target **any** platform. For this reason, your apps should adopt to various screen sizes and input types.

10.3.1.1 Single OS

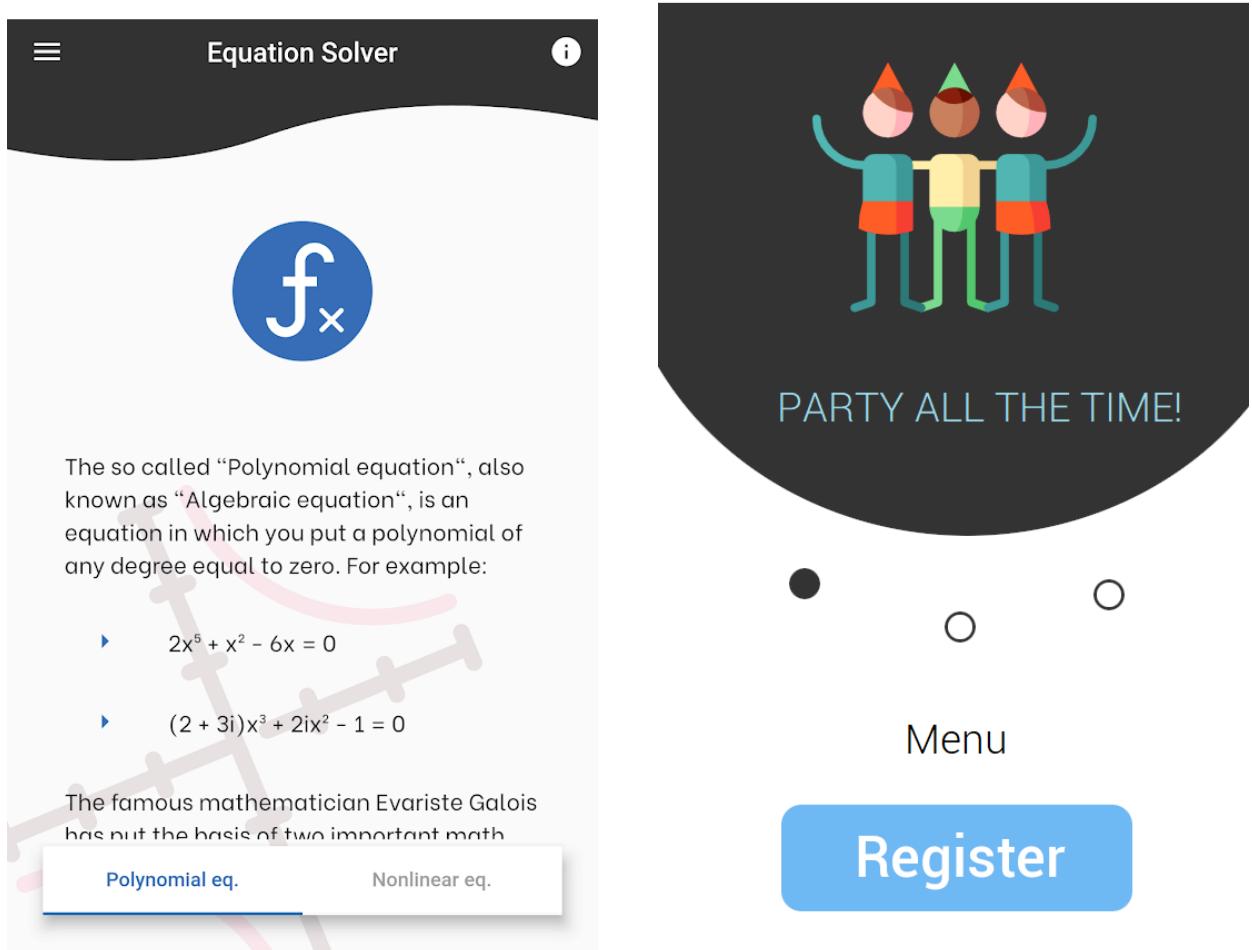
The simplest case is the one in which you have to create an application that is going to run exclusively on Android or iOS, not in both. If the structure of the app is fairly "standard", *Material* or *Cupertino* widgets are enough to do the work. To be more precise, saying "standard" in the Android world refers to the fact of having at least:

- a horizontal bar at the top with the title and maybe a series of buttons;

⁵<https://flutter.dev/docs/development/ui/widgets/cupertino>

- a menu sliding from the right/left of the screen (a `Drawer`);
- probably a FAB or a layout with swipeable tabs.

A `Scaffold` helps you to easily create a layout with the above characteristics but if you want to have a completely different structure or something very particular, don't use it. Look at these two examples:



Both are Android apps and you can immediately see that the one on the left is very close to a "traditional" material app: it has a drawer, and app bar, icons and a swipeable tab on the bottom.

- i** The example on the right would be impossible to realize with a `Scaffold` because it has a completely different structure from the one that a `Scaffold` proposes, so in this case you'd have to create everything from scratch.

The thing is: don't use the `material` library when your app's design has nothing to do with material guidelines. If the layout is very particular, it'd be better if you created it from scratch and it's perfectly doable in Flutter. Don't force yourself using Flutter's material/cupertino libraries because they aren't always the right choice for the UI you're going to implement.

- i** Of course the same recommendations also apply to the `cupertino` library. If your iOS app is going to look quite different to what is a "standard" iOS design, go for a complete custom UI and don't rely on `cupertino`.

10.3.1.2 Multiple OSes

This is the hardest case and probably the most common because, in general, companies want their app to be available for both Android and iOS. Here's where cross-platform tools, such as Flutter, come to the rescue. As always, they can become your enemy if misused but we have some recommendations for you.

- i** In the following paragraphs, let's pretend that you're asked to create an app for a restaurant with the possibility to make a reservation, see the menu and a gallery of images.

UI consistency is very important: the app should look exactly the same in any OS it's running on. Surely your customer will ask not only for the same design, but also for identical functionalities. This is what we suggest:

- **DO NOT** use `material` and `cupertino` to create two different versions for the same app. For example, doing this is absolutely wrong:

```
// Contains 'TargetPlatform'  
import 'dart:io' show Platform;  
  
void main() {
```

```
if (Platform.isAndroid) {
    runApp(const AndroidVersion());
} else {
    runApp(const iOSVersion());
}

// Uses MaterialApp
class AndroidVersion extends StatelessWidget {}

// Uses CupertinoApp
class iOSVersion extends StatelessWidget {}
```

With this approach, you're forced to keep 2 separate versions of the same app and all the advantages of cross-platform development are gone. This is the same as having two separated native projects in Java/Kotlin and Objective-C/Swift!

- **DO** create an nice UI for both operating systems so that you can write the code only once. Even if this might seem obvious, it's still worth saying rather then taking it for granted. Flutter is made for this purpose.

```
import 'dart:io' show Platform;

void main() => runApp(RestaurantApp());

class RestaurantApp extends StatelessWidget {
    const RestaurantApp();

    // Depending on the platform, return a different logo
    String _logoName() {
        if (Platform.isIOS) {
            return "Welcome iOS user!";
        }
        return "Welcome Android user!";
    }

    Widget build(BuildContext context) {...}
}
```

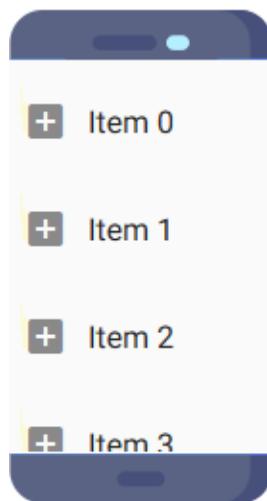
In this example, the same **single** codebase works for both Android and iOS. Certain settings

may differ according to the OS type but it's fine since there's still a single project being maintained.

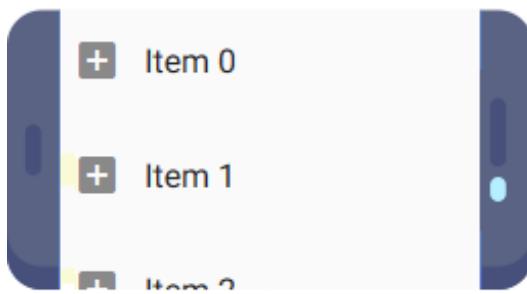
You should try as much as possible to not depend on OS-specific settings or configurations; always try, when possible, to make your architecture working fine regardless the OS on which it runs. Writing robust cross-platform applications is not easy but Flutter gives you a great boost in the correct direction!

10.3.2 Responsive UIs

High quality apps are *responsive* because they automatically adapt to the size of the screen by rearranging the UI in order to properly fill all the available space. Just think of an orientation changing in your mobile phone for example: the horizontal space increases/decreases and your UI should adapt by consequence.



Lists are a common UI element and in fact `ListView` is a very popular Flutter widget. In the above picture you can see a scrollable list with a series of items. It looks good as it is but if we rotated the screen the situation would change:



Now there is a lot of free space on the right and there are only two fully visible items; the user has to scroll a lot more than before. In such cases, a good responsive UI rearranges its contents in order to cover all the available space.

i This problem is not only tied to the screen rotation case. If your app runs on both mobile phones and tablets (which is very likely) there are big size differences on the screens and your UI should be flexible enough to look well in any case.

More in general, if you plan to run your Flutter app on mobile devices, desktop and web there will be huge screen size differences. You really need to take this into account and create the UI by consequence.

10.3.2.1 LayoutBuilder

Considering the above example of a simple list with items, the code is pretty simple at the moment. The `ListView` is always used in the vertical direction without taking into account the orientation of the screen. As we've seen, this is not a responsive usage at all:

```
Scaffold(  
    body: ListView.builder(  
        itemCount: 50,  
        itemBuilder: (context, id) {  
            return ListTile(  
                leading: const Icon(Icons.add_box),  
                title: Text("Item $id"),  
            );  
        }  
    )
```

```
)  
,
```

The `LayoutBuilder` widget gives information about the constraints of the parent such as the width and the height. Really consider using this class to make your apps responsive because it can be used to decide how to arrange the UI according to the available space.

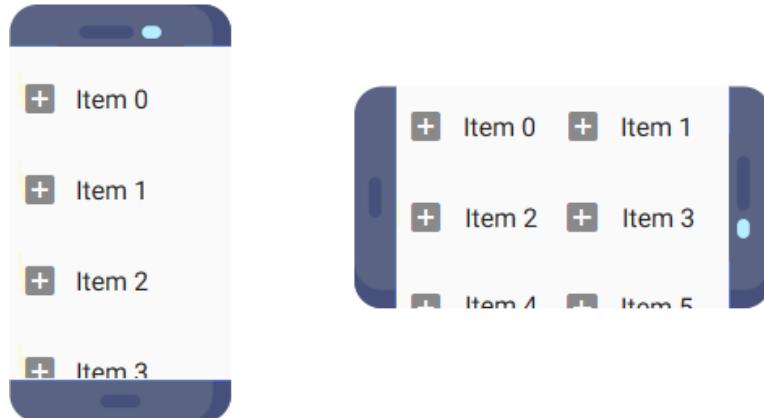
```
Scaffold(  
    body: LayoutBuilder(  
        builder: (BuildContext context, BoxConstraints sizes) {  
            if (sizes.maxWidth < 500) {  
                return const ListData();  
            }  
            return const GridData();  
        }  
    )  
)
```

The `BoxConstraints` class gives a series of information about the size of the parent widget. In this case, we're using it to decide the following: if the width is lower than 500 a list is good, otherwise it's better rearranging in a grid to fill the space in a better way.

```
class ListData extends StatelessWidget {  
    const ListData();  
  
    @override  
    Widget build(BuildContext context) {  
        return ListView.builder(  
            itemCount: 100,  
            itemBuilder: (context, id) {  
                return ListTile(  
                    leading: const Icon(Icons.add_box),  
                    title: Text("Item $id"),  
                );  
            }  
        );  
    }  
  
    class GridData extends StatelessWidget {
```

```
const GridData();  
  
@override  
Widget build(BuildContext context) {  
    return GridView.count(  
        crossAxisCount: 2,  
        children: List.generate(100, (index) {  
            return Center(  
                child: ListTile(  
                    leading: const Icon(Icons.add_box),  
                    title: Text("Item $index"),  
                );  
        });  
    );  
}  
}
```

The `Grid` widget automatically places elements in a grid and the number of columns is determined by the value passed to `crossAxisCount`. This code is said to be **responsive** because when the width of the screen changes, thanks to `LayoutBuilder`, the UI is rearranged accordingly.



If you opened your app in a tablet, which has a very wide screen, you'll already see the grid instead of the list. `LayoutBuilder` is good for screen rotations and much more; you can (and should) use it to adapt the UI to the dimensions of many devices such as mobile phones, tablets

and desktop.

10.3.2.2 MediaQuery

The `MediaQuery` class is a sort of more powerful version of `LayoutBuilder` because it's always available and it gives you more control over various settings of the screen. It just requires a context:

```
@override
Widget build(BuildContext context) {
    // We're using 'double.nan' but it could have been any other value
    final width = MediaQuery.of(context)?.size.width ?? double.nan;
    return Text("\$width");
}
```

With `size` you also have access to height, padding, distances and much more. You can handle the nullable value returned by `of()` with a default value (like we did) or with an null check (an `if` statement). For example, you might want to know which is the current orientation of the device:

```
final orientation = MediaQuery.of(context)?.orientation;

// Using a null check rather than providing a default value
if ((orientation != null) && (orientation == Orientation.portrait)) {...}
```

10.3.2.3 Good practices

High quality applications are responsive and so we strongly encourage you to not test your app only on a mobile device locked in portrait mode. Do a lot of tests with different screen sizes in both portrait and landscape mode. Other than this, here are a few tips you can try:

- If you have to make your app responsive, avoid using `MediaQuery.of(context)` to calculate the spaces and the dimensions. It holds a lot of metadata about the physical screen but it knows nothing about the widget itself.
- Use `LayoutBuilder` to make responsive layouts as it provides dimensions about the containing widget and **NOT** about the screen itself, like `MediaQuery` does. This example might give a better idea of the difference between the two approaches.

```
// 1.
Widget build(BuildContext context) {
    final width = MediaQuery.of(context)?.size.width ?? 0;
```

```
        return Text("$width");
    }

// 2.

Widget build(BuildContext context) {
    return LayoutBuilder(
        builder: (context, constraints) {
            return Text("${constraints.maxWidth}");
        }
    );
}
```

To get the most out of this example, put this `build` method in the root widget of your tree. You'll see that both cases 1 and 2 will print the same size because:

- `MediaQuery` returns the width of the screen
- `LayoutBuilder` returns the width of the parent widget but, being it the root, it takes the entire size of the screen.

Now try to run this example which is a little different:

```
// 1.

Widget build(BuildContext context) {
    final width = MediaQuery.of(context)?.size.width ?? 0;
    return Padding(
        padding: EdgeInsets.all(15),
        child: Text("$width"),
    );
}

// 2.

Widget build(BuildContext context) {
    return Padding(
        padding: EdgeInsets.all(15),
        child: LayoutBuilder(
            builder: (context, constraints) {
                return Text("${constraints.maxWidth}");
            }
        )
    );
}
```

```
    );  
}
```

Here there's the important difference between the two widgets. `MediaQuery` still returns the same value, because the width of the screen hasn't changed, but `LayoutBuilder` returns a **different** size which is 30 units smaller than before.

- `LayoutBuilder` takes into account the fact that there is a padding and the returned dimension is `screenSize - paddingAmount`, which is the available space.
- `MediaQuery` will **always** return the same value because the device's screen width didn't change. It **doesn't** consider the padding.

You really should not use `MediaQuery` as it's just the "measures" of the device; use `LayoutBuilder` instead which calculates the actual remaining space by considering the dimensions of other widgets that contain it.

As we've already seen, use `MediaQuery.orientation` if you only need to know whether the device is in landscape mode or not. Be aware that there is also the `OrientationBuilder` widget:

```
OrientationBuilder(  
    builder: (context, orientation) {  
        if (orientation == Orientation.portrait) {  
            //work in portrait mode  
        } else {  
            //work in landscape mode  
        }  
    }  
);
```

Pay attention to the fact that `OrientationBuilder` depends on the parent widget's orientation, which is **not** the device orientation. For example, if your device were in portrait mode and you opened the keyboard to fill a form, the height might become smaller than the width and thus `OrientationBuilder` would return `landscape`.

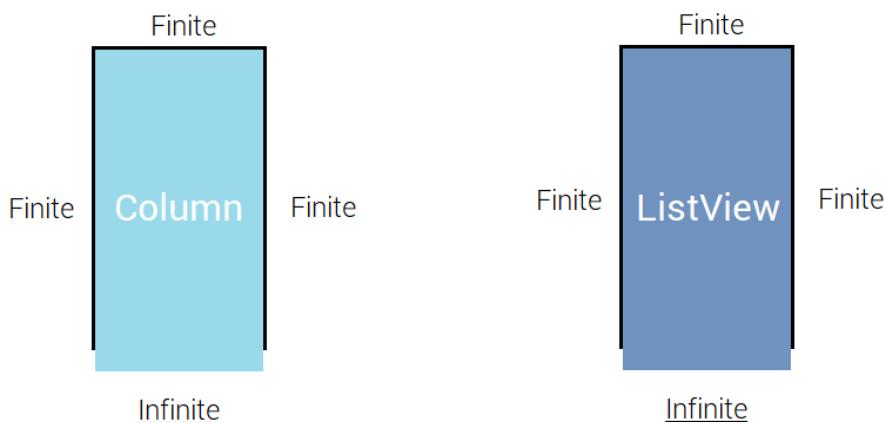
1. Use `MediaQuery` to get the current orientation of the device. In practice it's the physical position (horizontal or vertical) of the mobile phone or tablet,
2. Use `OrientationBuilder` to get the current orientation according to the parent widget's orientation. It's not based on the orientation of your physical device but it relies on the dimensions of the containing widget (whether the height is smaller or greater than the height).

10.3.3 Scrolling and constraints

Both `ListView`s and `Columns` are very popular but you have to pay attention to how they treat the contents in the vertical axis. This code seems to work as intended but it will throw a runtime exception because the height of the column is *infinite*.

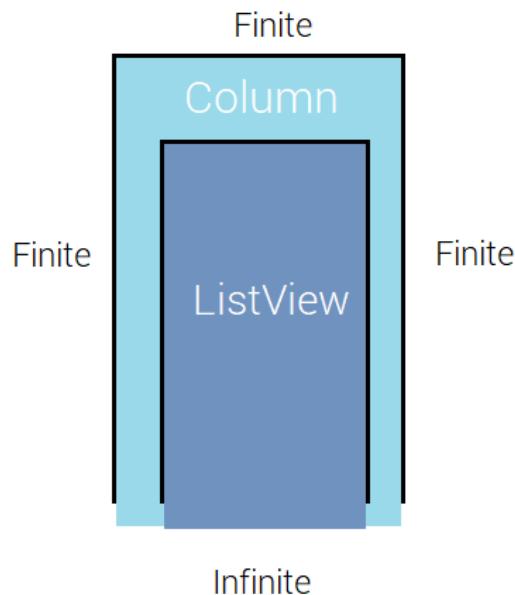
```
Column(  
    children: [  
        const Text("My name"),  
        const Text("My surname"),  
  
        ListView(  
            children: const [  
                Text("Skill 1"), Text("Skill 2"),  
            ]  
        )  
    ]  
) ;
```

You want your name and surname to stay always at the top but the list of your endless skills must scroll. The above solution is not doable because a `Column` expands to fill the entire available space and a `ListView` does the same. They both don't have a defined height. Here's how the boundaries look like:



They are two widgets without a specific value for the height because they always try to fill the entire available space and so determining a value in advance is not possible. If you nested them

like this...



... you would get an error because both try to expand to cover the entire space but there are **no** parent widgets with a fixed height. The solution is to always be sure they're inside a widget which sets a finite height and there are a few ways to do this:

1. You can use the `Expanded` widget which expands to fill the remaining available space in a `Column` or in a `Row` by giving a full set of dimensions.

```
Column(  
    children: [  
        const Text("My name"),  
        const Text("My surname"),  
  
        Expanded(  
            child: ListView(  
                children: const [  
                    Text("Skill 1"),  
                    Text("Skill 2"),  
                ]  
            ),  
        ),  
    ],
```

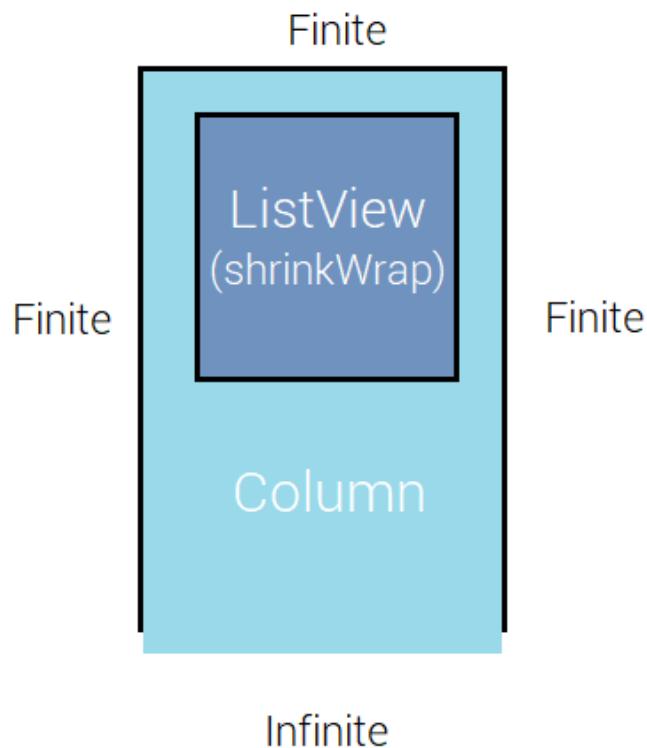
```
        ),  
    ]  
);
```

In this way `ListView` works properly because `Expanded` doesn't return an infinite height: it expands to fill exactly the remaining space and calculates a finite height. This widget can also be used with `Columns` and `Rows`.

- ➊ A `Row` has the same "problem" but in the other direction (the horizontal axis). Every consideration made for columns is also valid for rows with the only difference that the orientation is on the x-axis rather than the y-axis.
- 2. There's an interesting attribute of `ListView` that changes the behavior of the widget so that it has a fixed height and the problem of the lower bound disappears.

```
Column(  
    children: [  
        Text("My name"),  
        Text("My surname"),  
  
        ListView(  
            shrinkWrap: true,  
            children: [  
                Text("Skill 1"),  
                Text("Skill 2"),  
            ]  
        )  
    ]  
);
```

By setting `shrinkWrap: true` the list occupies only the space it needs and it does **not** expand to fill all the available space. In this way it has a well-defined height because it's calculated according to the dimensions of the children.



However, since `Column` doesn't handle overflows with scroll bars, if the list is too long and the screen cannot contain it entirely you'll see anyway the runtime overflow exception. This isn't really a "safe" solution.

3. The `Expanded` widget is generally the most convenient to use but you can put the list inside any kind of widget with a well-defined height such as a `Container` or a `SizedBox`.

```
SizedBox.fromSize(  
    size: const Size(100, 100),  
    child: ListView(...),  
);
```

You could also make it so that the container automatically fills the entire width and force its height to be a fixed value. It's like an `Expanded` on which you can control the height and/or the width.

```
Container(
```

```
        constraints: const BoxConstraints.expand(
            height: 200
        ),
        child: ListView(...),
    );
}
```

There are lots of possibilities, including the combined usage of a `LayoutBuilder` with a `Container` but it might get too complicated. Try to keep it simple by using `Expanded` or another single widget that automatically handles the sizes.

- ➊ In general, you get this kind of "infinite constraint" issues with rows, columns and lists as they're very frequently used. Very often, the simple solution is called `Expanded` but if you're looking for a more sophisticated approach, consider using `SizedBoxes` or `Containers`.

10.3.4 Using themes

If you want to share font styles, colors and other UI appearance settings throughout an app, use the `ThemeData` class, in case of a `MaterialApp`, or a `CupertinoThemeData`, for the `CupertinoApp` widget. For example, if you used this kind of setup...

```
MaterialApp(
    theme: ThemeData(
        fontFamily: "Times New Roman",
    ),
)
```

... the default `fontFamily` property of your app's widget will be `"Times New Roman"`. This is very convenient because, for example, any `Text` widget will inherit that specific font family and you wouldn't have to do this all the time:

```
const Text("Something",
    style: TextStyle(
        // Useless because "Times New Roman" is already inherited thanks
        // to 'ThemeData'
        fontFamily: "Times New Roman",
    )
)
```

With the usage of `ThemeData` changes are automatically reflected on `any` children so the mainte-

nance is a lot easier. Do you want to use a different font family such as "**Georgia**"? Just setup the new value in `ThemeData` and automatically the changes will be reflected anywhere else. It's a centralized place in which you can style widgets:

```
MaterialApp(  
    theme: ThemeData(  
        buttonColor: Colors.red, // Color of a RaisedButton  
        focusColor: Colors.white, // Color when a widget is focused  
        selectedRowColor: Colors.orange,  
  
        primaryColor: Colors.green,  
        accentColor: Colors.red,  
    ),  
)
```

Generally a `ThemeData` should always declare a `primaryColor`, which defines the color of the most common UI widgets (scaffolds, tab bars, textfields focuses...), and a `primaryAccent`, which defines the color for foreground widgets (FABs, list overscroll glow...). If you want to implement a dark or light theme for your app, consider using the following named constructors:

```
// Predefined set of colors to implement dark and light themes  
theme: ThemeData.dark()  
theme: ThemeData.light(),
```

You can get a reference to the theme properties by calling `Theme.of(context)` anywhere on the widget tree. There is also the possibility to override only a specific set of properties for a given theme in order to preserve the other settings:

```
MaterialApp(  
    theme: ThemeData.dark().copyWith(  
        primaryColor: Colors.grey  
    ),  
)
```

This is a `dark()` theme with the only difference that `primaryColor` has been changed to be grey (all the other parameters are still the same). You could also decide to override theme settings for certain parts of the tree rather than applying a global theme like we've done in the above example. This is possible thanks to the `Theme` widget:

```
// main.dart  
MaterialApp(
```

```
    theme: ThemeData.dark());
}

// light_footer.dart
Widget build(BuildContext context) {
    return Theme(
        data: ThemeData.light(),
        child: const MyFooter(),
    );
}
```

In this example, the entire app has a `dark()` theme but the `MyFooter` widget and **all** of its children will use the `light()` theme instead. In other words `Theme` is used to override the current theme with a new one for the entire subtree. In fact:

- calling `Theme.of(context)` inside `MyFooter` returns a reference to the light theme;
- calling `Theme.of(context)` outside `MyFooter` returns a reference to the dark theme.

Note that `Theme.of()` doesn't return a nullable value. In the next chapter we will show you how to change your app's theme from dark to light (and vice versa) with ease using the `HydratedBloc` from the *flutter_bloc* package.

11 | State management

Up to this point, you know what is the *state* in a Flutter app because we've exhaustively treated this topic in chapter *9.2.2 Stateless and Stateful widgets*. What you don't know yet is how to properly handle the changes of the state.

- ❶ We're going to analyze in detail *provider* and *flutter_bloc* but there are many other state-management libraries out there such as *Redux*, *MobX* or *Scoped model*. You can find more details in the official documentation ¹.

Knowing how to properly handle the state of a Flutter app is fundamental: a well-structured code is easy to read and maintain. In addition, you'll almost always create production apps with interaction from the user (or external sources) and thus the state is going to change a lot of times. In this chapter we're going to create this simple app:



Very easily, it does nothing more than incrementing and decrementing the counter in the middle when you press respectively **+1** or **-1**, which are `FlatButton`. We're going to implement the app using 3 different state management strategies:

¹<https://flutter.dev/docs/development/data-and-backend/state-mgmt/options>

1. Updating the UI using `setState`,
2. Passing the state around the widget tree, with the help of the `Provider` widget,
3. Alternatives to `setState()`, implemented with the help of the `BlocBuilder` widget.

You'll end up having seen the same app built in 3 different ways, one for each strategy, so that you can analyze their mechanisms and see the differences in how they work.

11.1 Updating the UI

It's the simplest way of handling the state of a widget but you should really avoid this approach because it mixes UI logic with business logic. You're also going to see very soon that a proper usage of `setState(...)` requires too much boilerplate code.

i Please note that we haven't said you should avoid using `StatefulWidget`s but you should avoid the usage of `setState`: you'll see soon why. As you already know, a stateful widget is fundamental when a stateless one can't be created due to the lack of immutability of the class itself.

Before explaining the reasons why directly using `setState` is bad, let's see again how it works in a traditional counter app.

```
// 1.  
class DemoPage extends StatefulWidget {  
    // 2.  
    const DemoPage();  
  
    // 3.  
    @override  
    _DemoPageState createState() => _DemoPageState();  
}  
  
class _DemoPageState extends State<DemoPage> {  
    // 4.  
    int _counter = 0;  
  
    // 5.
```

```
void _increment() {
    setState(() => _counter++);
}

void _decrement() {
    setState(() => _counter--);
}

@Override
Widget build(BuildContext context) {...}

}
```

This is the typical setup of a widget whose state is managed with `setState`.

1. You have to follow this pattern: there's the need for a class that extends `StatefulWidget` as it's going to be put in the widget tree. The other class is private as it represents and handles the state of the widget.

```
MaterialApp(
    // don't forget the const constructor!
    home: const DemoPage(),
);
```

2. There's the possibility to define a `const` constructor for `DemoPage` because it's not going to change over the time. What is going to change is the `state` of the widget, represented by `_DemoPageState` (hence the class `_DemoPageState` can't have a constant constructor).
3. The creation of the widget's persisting state which will "survive" to rebuilds. You're going to work a lot with this class as it exposes the `setState(...)` method.
4. The counter which will be displayed in a `Text` widget.
5. Two functions that increment and decrement the counter; they both call `setState` so that the widget and its children get rebuilt in order to refresh the UI.

The `build` method is very easy to understand because the UI is minimal, you'll get immediately what's going on. Note the usage of `const` in front of `Text`, when possible, which "caches" both `Text` and `TextStyle`.

```
Row(
    mainAxisAlignment: MainAxisAlignment.spaceAround,
```

```
children: [
    FlatButton(
        child: const Text("+1",
            style: TextStyle(
                color: Colors.green,
                fontSize: 25
            ),
        ),
        onPressed: _increment,
    ),
    Text("$_counter",
        style: const TextStyle(
            fontSize: 30,
        ),
    ),
    FlatButton(
        child: const Text("-1",
            style: TextStyle(
                color: Colors.red,
                fontSize: 25
            ),
        ),
        onPressed: _decrement,
    ),
],
)
```

As you already know, when `setState` is called its callback is executed and then the widget is rebuilt. Since the state persists, the increment of the variable is "remembered" and so the `Text` widget will display the new updated value.

11.1.1 Considerations

First of all there are **ABSOLUTELY NO** reasons to say that this approach causes performance issues because we've used a `StatefulWidget` instead of a `StatelessWidget`. The problem is that `setState` has to be used together with `InheritedWidget` otherwise there will be uncontrolled rebuilds.



In the case of a widget with no children like A, when `setState` is called a rebuild happens only for A. In the above image, the black box represents a rebuilt widget. Performance issues start to get real when the widget being rebuilt has one or more children. Look at this example:

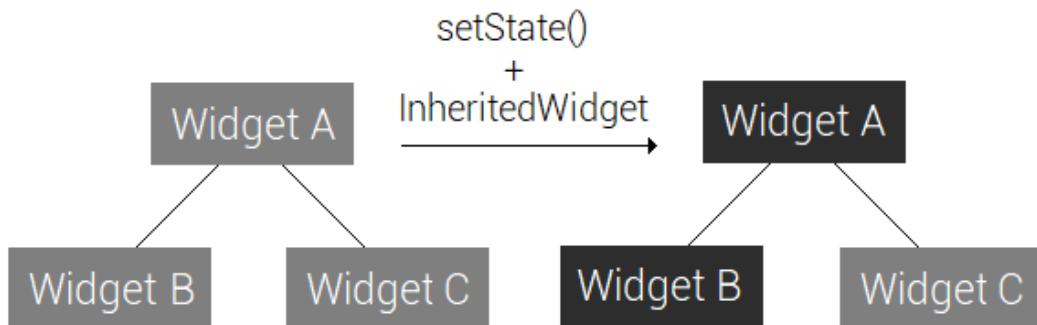
```
class _WidgetAState extends State<WidgetA> {
    int _value = 0;

    @override
    Widget build(BuildContext context) {
        return Column(
            children: [
                const WidgetB("$_value"),
                const WidgetC(),
                RaisedButton(
                    child: const Text("Update"),
                    onPressed: () => setState(() {
                        _value += 10;
                    }),
                ),
            ],
        );
    }
}
```

When tapping on the button, `setState` will **always** trigger a rebuild for the current widget **and** all of its children. In the above example, both `WidgetB` and `WidgetC` are rebuilt even if only `WidgetB` should, as it's the only which has a dependency from `WidgetA`.



As you can see from the image, children of A are always rebuilt even if they don't have variables in common or any other kind of dependency. Calls to `setState()` rebuild the entire subtree, even if it's not really needed. It would be better if Flutter rebuilt only the widgets that really need to be updated, like this:



If you used `setState` in combination with a particular class, called `InheritedWidget`, you would be able to make optimized rebuilds that doesn't waste resources. When used together, you get the possibility to rebuild **only** those widgets that really need to be updated leaving others untouched.

- i** The big problem is the usage of a stateless widget combined with an `InheritedWidget`: it produces a lot of boilerplate which is very hard to understand and maintain. You **NEVER** want to deal with it as it's not needed; there are many libraries (such as `provider`) that do all this tedious work for you!

If you are curious to understand the details of `InheritedWidget`, as always we recommend you to visit the official documentation ² which also has a video about it. We're not covering it in the book because there's no point in doing it since nowadays `provider` is the default choice, which actually is just syntactic sugar for `InheritedWidget`.

1. Stateless and stateful widgets are both efficient and good, you don't have to think that one is less performant than the other. If you want to manage the state using a `StatefulWidget` you should really use `setState` and `InheritedWidget` together.
2. Passing data down the tree and controlling the rebuilds with `InheritedWidget` is complicated and produces a lot of boilerplate code; don't do it. Prefer the usage of a library such as `provider` which does everything for you with less code (and it's also way more readable).
3. With `setState` you're mixing UI logic and business logic. For example, `_DemoPageState` is like a huge hammer dropping from the sky and totally destroying the single responsibility principle glass. It does too many things:
 - it handles the UI logic, which is responsible of drawing widgets;
 - it handles the business logic, which takes care of `_counter`;
 - it handles the state of the app, which is managed by `setState`
4. The usage of `setState` is too "basic" as it just tells Flutter to rebuild the widget and all its children; for a more subtle control there would be the need to also use `InheritedWidget`.

Inheriting from `State<T>` gives access to the `initState` method; it's called only `once` in the moment in which the state is created. Since the state persists until the widget is disposed, you're guaranteed that `void initState()` will run only once during the lifecycle of the class.

```
class _DemoPageState extends State<DemoPage> {

    @override
    void initState() {
```

²<https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html>

Chapter 11. State management

```
super.initState();
// put the code here...
}

@Override
Widget build(BuildContext context) {...}

}
```

You should override `initState` when the widget has to be configured before being built or if there's the need to call methods that must be executed only once at the creation of the state. Actually `initState()` can be seen as if it were the constructor of a widget and `dispose()` the destructor.

```
class _DemoPageState extends State<DemoPage> {

    @override
    void dispose() {
        // your code here...
        super.dispose();
    }

    @override
    Widget build(BuildContext context) {}

}
```

It's executed only once when the state is destroyed and it should be used when there's the need for a clean up of resources used by the widget. If you had the need to declare a variable that cannot be immediately initialized, and you don't want it to be nullable, you'd use `late`.

```
late String value;

@Override
void initState() {
    super.initState();
    value = "Init me";
}
```

If that value is going to be assigned only once inside `initState`, consider making it `late final`.

With the arrival of NNBD, variables could also be directly initialized in this way:

```
late String value = "Init me";
```

No need to use `initState()` at all but that's just because you're doing an assignment: you can't do this when it comes to calling functions, for example.

11.1.2 Good practices

We want to point out again the fact that using stateful widgets is absolutely fine: the problem lies in the usage of `setState` with no `InheritedWidget` associated. Other than giving too many responsibilities to the widget, it doesn't give you control on rebuilds of the children widgets and this can be a big performance issue. Other than this:

- Do the initialization of the widget inside the `initState` method so that you're guaranteed that the phase will be executed only once. In case of assignments, consider using `late final` and initialize variables directly to reduce the amount of boilerplate code.
- The official documentation³ says that `setState` should only update values, like assigning new values to variables, it shouldn't compute anything. For example you should do this...

```
void _increment(int value) async {
    setState(() {
        _counter += value;
    });

    await writeToFile(_counter);
}
```

... rather than calling the function inside the state updater:

```
void _increment(int value) {
    setState(() async {
        _counter += value;
        await writeToFile(_counter);
    });
}
```

Furthermore if the callback function is an instance of a `Future` you'll get a runtime exception. Regardless, this is bad anyway because writing data to a file should not happen in a class that deals with the UI!

³<https://api.flutter.dev/flutter/widgets/State/setState.html>

- Do not call `initState` after `dispose` since it leads to undefined behavior. `StatelessWidget` has the `bool get mounted` property which tells you whether the widget is created or disposed.

Use stateful widgets with no worries but avoid using `setState` for state management; prefer using a library like `provider` or `flutter_bloc`. Avoid the "raw" usage of `setState`, because it rebuilds the `entire` subtree, and the `setState() + InheritedWidget` combination as well, as it produces a huge amount of boilerplate code.

11.2 Passing the state with Provider

The `provider`⁴ package has been created by Rémi Rousselet and it's available in the official package repository. Be sure to properly install it by opening the `pubspec` and adding the dependency.

```
dependencies:
  provider: ^4.3.2
```

The Flutter SDK includes a simple class called `ChangeNotifier` which provides change notification to its listeners. In practical terms, if you use this class as a `mixin` you get the possibility to send an "alert" that something has changed to the subscribed widgets

- Provider uses Flutter's `ChangeNotifier` to create a class that encapsulates the state and, when something changes, the interested widgets are notified and rebuilt. As the name suggests, it's a *notifier* that alerts *listeners* about changes.

We're still going to create the same app which increases and decreases the counter but in the "provider-way". First of all there's the need to create a class that takes care of the business logic and makes the state to persist; we're creating a file named `counter_model.dart` with this content:

```
// The mixin is needed because it contains 'notifyListeners()'
class CounterModel with ChangeNotifier {
  int _counter = 0;

  void increment() {
```

⁴<https://pub.dev/packages/provider>

```
        _counter++;
        notifyListeners();
    }

    void decrement() {
        _counter--;
        notifyListeners();
    }

    int get currentCount => _counter;
}
```

If you go back to the previous section, you'll see that this code is almost identical to what's inside `_DemoPageState` with the difference that we're calling `notifyListeners()` instead of `setState(...)`.

- The `void notifyListeners()` method, contained in the `ChangeNotifier` mixin, is used to send a "signal" to the interested widgets that something has changed and a rebuild is needed to update the data.

```
// Stateful widget
setState(() => _counter++);

// ChangeNotifier mixin
_counter++;
notifyListeners();
```

Logically they do the same things: first the variable is increased and then the UI is rebuilt because there's a signal that something has changed.

- With this approach we've moved the logic of the app from the UI widget to a separated class (`CounterModel`) and it's a big step forward. We're now respecting the Single Responsibility Principle.

At this point all the logic lies inside a class called `CounterModel` and the next move is "linking" the model with the UI part. There are two steps to do:

1. Use the `ChangeNotifierProvider` widget to create an instance of the class (which is mixed with `ChangeNotifier`) so that the entire subtree will be able to use it.

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

```

    }

class MyApp extends StatelessWidget {
    const MyApp();

    @override
    Widget build(BuildContext context) {
        return ChangeNotifierProvider(
            create: (context) => CounterModel(),
            child: const DemoPage(),
        );
    }
}

```

Thanks to `ChangeNotifierProvider`, the `DemoPage` widget and **every** children of it will be able to get the instance of `CounterModel`. The provider package has the goal to expose an object to the subtree of a given widget.

- Obtain the value from the above widgets using `Provider.of<T>(context)` and use it to read and/or update the state of the widget.

```

class DemoPage extends StatelessWidget {
    const DemoPage();

    @override
    Widget build(BuildContext context) {
        // The type of 'counter' is CounterModel
        final counter = Provider.of<CounterModel>(context);

        return Scaffold(
            body: Center(
                child: Row(
                    mainAxisAlignment: ...,
                    children: [
                        // FlatButtons and Text widget
                    ]
                )
            );
    }
}

```

```
}
```

The method `of<CounterModel>(context)` returns the instance of the given type that's been created and exposed from above the tree. `ChangeNotifierProvider` can be seen like a "cache" that stores a class and serves it to the children when they ask for it. This is the new body of the `Row`:

```
FlatButton(
    child: const Text(
        "+1",
        style: TextStyle(
            color: Colors.green,
            fontSize: 25
        ),
    ),
    onPressed: () => counter.increment(),
),
Text(
    "${counter.currentCount}",
    style: const TextStyle(
        fontSize: 30,
    ),
),
FlatButton(
    child: const Text(
        "-1",
        style: TextStyle(
            color: Colors.red,
            fontSize: 25
        ),
    ),
    onPressed: () => counter.decrement(),
),
```

If you pressed on `+1`, the `increment()` method would increase by 1 the counter and then it'd call `notifyListeners()` which triggers a rebuild of the widget. A rebuild takes place because `DemoPage` is a `listener` (it's a child of `ChangeNotifierProvider`) and thus it's listening to changes.

The `Text` widget shows the newly updated count thanks to `currentCount`, which is simply a getter

that returns the current value of the counter. To sum up, with this approach you have to:

1. create a model class which uses `ChangeNotifier` as a `mixin` and then call `notifyListeners` whenever the UI has to be updated;
2. use `ChangeNotifierProvider` to create an instance of the model that can be exposed and watched by the children;
3. in the `build` method use `Provider.of<T>(context)`, where `T` is the type that you're looking for, to get the instance.

11.2.1 Considerations

You've just seen how the state can be handled without having to use `setState`. Thanks to `provider` there's the possibility to use a combination of stateless widgets and notifier classes to achieve the same result. Nevertheless `ChangeNotifierProvider` isn't the only important feature of the package.

11.2.1.1 Provider class

In the examples we've shown that `class Provider<T>` has the static method `of<T>()` which obtains, from above the tree, the instance of the given type `T`. Other than notifying children about UI updates, it's also a very useful way to cache classes and expose them.

```
class Something {
    final description = "something is better than nothing";
    final descriptionCache = {...}
}

class ExamplePage extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return Provider<Something>(
            create: (context) => Something(),
            child: ChildWidget(),
        );
    }
}

class ChildWidget extends StatelessWidget {
```

```
  @override
  Widget build(BuildContext context) {
    final value = Provider.of<Something>(context);

    return Text(" ${value.description}");
  }
}
```

Notice that we've used `Provider<T>` rather than `ChangeNotifierProvider<T>` and thus children won't be able to listen to updates. How can this be useful?

- You might want to put a provider at a certain point of the widget tree and use it as a "cache" which holds data in memory. For example, you'll see in chapter 12 that *provider* is very convenient when it comes to share data between multiple pages.

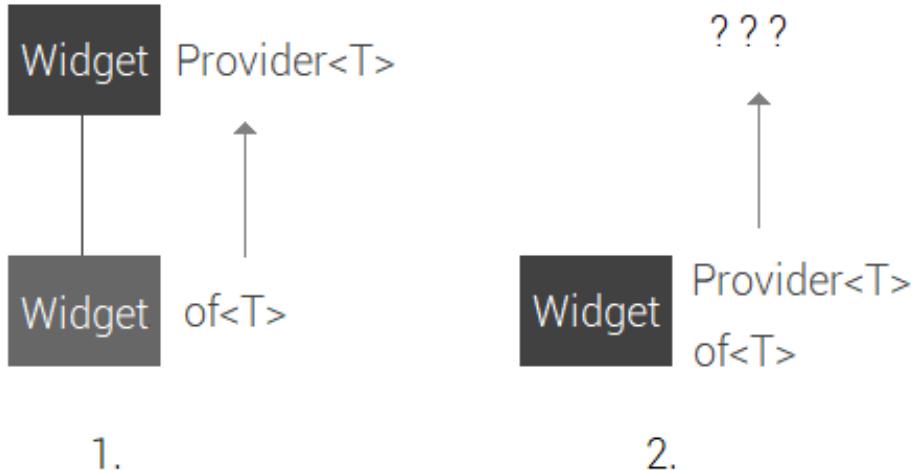
```
return Provider<DataCache>(
  create: (context) => DataCache(),
  child: PageWithTabs(),
);
```

The cache is served to the children pages with provider and it won't be destroyed because it's one level above in the tree. You can make a simple call to `Provider.of<DataCache>()` instead of creating complicated ways to pass data between widgets.

- The usage of *provider* is like an automatic usage of `initState` and `dispose` so you have less code to write and less lifecycle logic to implement.
- Your app's UI logic is separated from the business logic.

11.2.1.2 Consumer class

This widget does nothing more than automatically calling `Provider.of<T>(context)` in a new widget to give you the instance of the class. It also solves a problematic situation in which you cannot use `Provider<T>` inside `build`:



1. The static method `of<T>(...)` looks for the instance `T` starting from one level **above** the current leaf. No problems here.
2. When `of<T>(...)` is being called in the same widget that exposes a class via `Provider<T>`, an exception occurs because there's no matching provider above.

```
class ExamplePage extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return Provider<Something>(  
            create: (context) => Something(),  
            child: Text(  
                "${Provider.of<Something>(context).description}"  
            ),  
        );  
    }  
}
```

Both `Provider<T>` and `of<T>(...)` are in the same widget and thus there will be a runtime failure.

In certain cases however there might be the need to call `of<T>(...)` at the same level of the *provider* but the classic approach doesn't work, as you've just seen. There are two solutions:

- Wrap the code requiring to access the provider in a child widget so that it gets a descendant context from its parent. This is not bad because you can also define a `const` constructor.

```
// Widget with provider
Provider<Something>(
    create: (context) => Something(),
    child: const ChildWidget(),
);

// Another widget
class ChildWidget extends StatelessWidget {
    const ChildWidget();

    @override
    Widget build(BuildContext context) {
        final description = Provider
            .of<Something>(context)
            .description;

        return Text(description);
    }
}
```

- Use the `Consumer<T>` class which automatically obtains the value for you. In this case there's no need to create additional widgets, it will automatically take care of taking the instance for you.

```
class ExamplePage extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return Provider<Something>(
            create: (context) => Something(),
            child: Consumer<Something>(
                builder: (_, value, __) {
                    return Text("${value.description}");
                }
            );
    }
}
```

}

You're allowed to use `Consumer<T>` as a direct child of a `Provider<T>` as it will automatically take care of properly returning the object.

Both ways are absolutely fine but probably the first one would be better because of the possibility to define a `const` constructor.

11.2.2 Good practices

Whenever you call `Provider.of<T>(context)` the associated widget is rebuilt unless you pass `listen: false` to the method. For example, if you're using `Provider` as a cache just to hold data, you probably don't need to always trigger a rebuild.

```
// Equivalent to of<DataCache>(context) because the
// 'listen' parameter is set to true by default
Provider.of<DataCache>(context, listen: true)

// A later value change won't trigger a rebuild
Provider.of<DataCache>(context, listen: false);
```

Remember to set `listen: false` when you're just working with the data and UI should not listen to changes. It may also be useful in such cases where a provider is called from outside the widget tree:

```
void _action(BuildContext context) {
    final p = Provider.of<MyObject>(context, listen: false);
    ...

}

@Override
Widget build(BuildContext context) {
    return RaisedButton(
        child: const Text("Tap me"),
        onPressed: () => _action(context);
    );
}
```

The method `_action` is not inside `build` and thus `Provider.of<T>()` is called outside of the widget tree. You'll get a runtime exception if `listen` is `true`. Nevertheless, in general `Consumer<T>` should be preferred because it could optimize your code. Consider this example:

```
class Test extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final value = Provider.of<Info>(context);

    return Center(
      child: Padding(
        padding: EdgeInsets.all(15),
        child: Text("${value.text}");
      ),
    );
  }
}
```

As usual, `Provider.of<T>()` is used to get an instance of the object from above the tree. Note that only the `Text` widget depends on provider because `Center` and `Padding` don't care about `value`. However, with this code, not only `Text` will be updated but also `Center` and `Padding`.

- ➊ Being in a stateful or stateless widget doesn't make any difference, the concept is the same. What's important for performance is trying to rebuild only what really needs to be updated.

In this case we want that, when `Info` changes, only the `Text` widget gets rebuilt while `Center` and `Padding` don't. In other words, we want to **optimize** the notifier so that only a few widgets will be rebuilt and not everything. Here's the solution:

```
Widget build(BuildContext context) {
  return Center(
    child: Padding(
      padding: EdgeInsets.all(15),
      child: Consumer<Info>(
        builder: (_, value, __) => Text("${value.text}");
      )
    ),
  );
}
```

This is much better because only `Text` will be rebuilt while `Center` and `Padding` will stay

untouched. With this example we want to suggest you to avoid using `of<T>()` and prefer `Consumer<T>` for a more granular rebuild.

```
MultiProvider(
  providers: [
    Provider<HttpCache>(create: (_) => HttpCache()),
    Provider<LocalCache>(create: (_) => LocalCache()),
    ChangeNotifierProvider<Charts>(create: (_) => Charts()),
  ],
  child: const AwesomeApp(),
)
```

Very likely you'll have to use more providers at once and, if they're at the same level, consider using `MultiProvider`. It is a nice way to group multiple providers without having to nest them one by one.

 Don't put any kind of provider "too high" in the tree if it's not needed because it pollutes the scope. Place in at a reasonable position, ideally immediately before the first widget that needs it.

With *provider* there's also the possibility to expose values returned by a `Future<T>` or explicitly write the rebuild constraints for listeners.

- So far we've only told you how to expose non-future values but *provider* has a very rich collection of classes.

```
FutureProvider<T>(
  create: (_) async => _makingHttpRequest(),
  catchError: (context, error) => _inCaseOfError(),
  child: const MyWidgets(),
),
```

This class listens to a `Future<T>` and exposes its result to the children, as it happens with any other type of provider. Passing a `Future<T>` that could emit errors without providing the `catchError` callback is considered an error.

```
FutureProvider<T>(
  create: (_) async => _makingHttpRequest(),
  catchError: (_, __) => _inCaseOfError(),
```

```

    lazy: false,
    child: const MyWidgets(),
),

```

The default behavior is `lazy: true` so that `create` is called only the first time the value is read. If you set `lazy: false` the future is fetched immediately and not lazily.

- A `Selector<T>` is the equivalent of a `Consumer<T>` which can explicitly set the rebuild constraints. If your widget rebuilds too often even with a `Consumer<T>`, consider using a `Selector` which is an even finer optimization system:

```

Selector<PersonData, String>(
    selector: (context, person) => person.name,
    builder: (context, name, _) {
        return Text(name);
    }
),

```

By default a `Selector<T>` does a deep comparison of the value obtained by the `Provider<T>` and the one returned by the `selector` callback. If they are different, a rebuild happens. In the above example, `person.name` is compared with the value obtained by the provider to decide whether a rebuild has to take place or not.

```

Selector<PersonData, String>(
    selector: (context, person) => person.name,
    shouldRebuild: (previous, next) {
        return (previous != next) &&
            (person.age > 18);
    }
    builder: (context, name, _) {
        return Text(name);
    }
),

```

Using the `shouldRebuild` callback you can implement a custom logic to decide when the children should rebuild; it overrides the default deep comparison behavior. Define this behavior when simple object comparison is not enough.

When using `Selector<T>` be sure to use classes that override `operator==` or collections. Using mutable classes, the comparison of the values might not work as you'd expect. Another thing to mention is the usage of `listen: false` when you try to get an instance of an object. Rémi

Rousselet suggests:

1. Don't think that `listen: false` always boosts your app's performances by default: yes, it could, but it's an implementation detail which depends on your architecture.
2. When inside the widget tree, in general leaving the default value (`listen: true`) is good. You might decide to use `listen: false` in a second moment but it might be a maintenance problem in the future if you don't document this behavior properly. Watch out to not introduce breaking changes in your codebase. In general, avoid using `listen: false` when you're inside the `build` method.
3. Use `listen: false` when a provider is called anywhere outside the `build` method of a widget. It's actually fact and rather than a good practice because not doing so causes a runtime exception!

Starting from version 4.1.0 there are some shortcuts you can use to reduce the boilerplate code required for common actions. The library introduced two new `extension` methods you should prefer over the classic verbose way:

```
// Default provider (with listen: true)
final before = Provider.of<T>(context);
final after = context.watch<T>();

// Non-listening provider (with listen: false)
final before = Provider.of<T>(context, listen: false);
final after = context.read<T>();
```

They all work in the same way, no differences, but the new version is shorter and more readable. Note that calling `read<T>()` from outside the widget tree is fine but if it's used inside the `build` method an exception is thrown (see the point 2 of the above Remi's suggestions). If you're looking for a recap:

1. When inside the `build` method (also when using a `ChangeNotifier`):
 - use `Provider.of<T>(context)` or
 - use `context.watch<T>()` (identical to above) or
 - use `Consumer<T>` (for optimization).

You should prefer using `watch<T>` over `of<T>` as it's less code to write and more modern. `Consumer<T>` can do performance optimizations so consider using it.

2. When inside the `build` method and you don't want to listen for changes or you need to listen only for certain values:

- use `Selector<A, S>` or
- use `context.select((A a) => S)` (identical to above)

With `Selector` you can optimize even more than `Consumer<T>` as it allows you to select the exact values to listen or specify rebuild conditions.

3. When you are outside of the `build` method:

- use `Provider.of<T>(context, listen: false)` or
- use `context.read<T>()` (identical to above)

Avoid using `of<T>`, prefer instead the usage of `context.read<T>` which is shorter and more modern as it uses Dart's extension methods.

Note that you can access data via provider inside `initState()` even if there isn't a visible `BuildContext` variable being passed as parameter. As such, you could do this:

```
void initState() {  
    super.initState();  
    myValue = context.read<Something>().value;  
  
    // Or also, using the 'old' syntax:  
    // myValue = Provider.of<Something>(context, listen: false).value;  
}
```

Using `watch<T>()` (which is the equivalent of calling `of<T>()` without `listen: false`) will cause an exception.

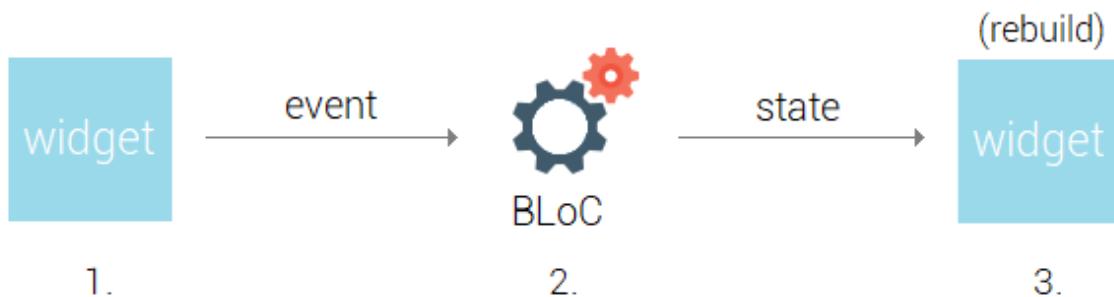
11.3 Alternative to `setState`: BLoC pattern

The *Flutter Bloc*⁵ package has been created by Felix Angelov and it's of course available in the official package repository. It's an implementation of the BLoC state management pattern that was presented at the Google I/O event back in 2018.

```
dependencies:  
  flutter_bloc: ^6.0.5
```

⁵https://pub.dev/packages/flutter_bloc

We're going to create the same app which increases and decreases the counter but in the "bloc-way". The term "BLoC" is an acronym which stands for *Business Logic Components* and it relies on asynchronous streams.



The gist of a bloc is the conversion of an event into a state. There are a series of steps involved:

1. A widget sends an event to the *bloc*, which is a class implementing a certain logic;
2. The bloc is notified because a new event has arrived on the stream. It processes the requests and then produces an output: the new state for the widget;
3. The listening widget receives the new state given by the bloc and rebuilds.

Let's map these theoretical steps to our concrete example. When the user presses on **+1** or **-1**, the flat button sends an **increment** or **decrement** event to the stream. The bloc does the calculation and then it produces a new state for the **Text** widget (a new **int**, the updated counter), which is rebuilt.



The first thing to do is the creation of the bloc, the class taking care of the logic we're going to implement. This is very good because we're separating the UI logic from the business logic in favor of the Single Responsibility Principle.

```
// 1.  
enum CounterEvent { increment, decrement }  
  
// 2.  
class CounterBloc extends Bloc<CounterEvent, int> {  
    // 3.  
    CounterBloc() : super(0);  
  
    // 4.  
    @override  
    Stream<int> mapEventToState(CounterEvent event) async* {  
        switch (event) {  
            // 'state' is a getter defined inside Bloc<E,S> which  
            // represents the current state of the bloc  
            case CounterEvent.increment:  
                yield ++state;  
                break;  
            case CounterEvent.decrement:  
                yield --state;  
                break;  
        }  
    }  
}
```

Chapter 11. State management

```
    }
}
```

This is the content of a file named `counter_bloc.dart`.

1. Events are represented by an `enum` and the two items represent, very intuitively, the increment or decrement of the counter. In Part III we will see that for more complex cases, using classes is better than enums.
2. The `CounterBloc` class is the bloc itself which takes a stream of events and produces new updates for the state. The `int` is produced by the bloc and it represents the new state of the `Text` widget, which will be rebuilt.
3. This is the initial value of the bloc, which is required. From version 5.0.0 onward, you have to pass the value via `super` as we've done above. Versions 4.0.1 and earlier had to override `initialState` instead:

```
// v4.0.1 and lower
class CounterBloc extends Bloc<CounterEvent, int> {
    // No calls to 'super'

    @override
    int get initialState => 0;

    @override
    Stream<int> mapEventToState(CounterEvent event) async* {...}
}
```

Basically from version 5 there's been a breaking change that replaced the `initialState` getter with a call to `super`. There's no logical difference since the purpose of both is giving the bloc an initial state (it's just a syntactic difference).

4. This method is the heart of the bloc as it listens for incoming events and produces new states. The syntax is very nice and self-explanatory because you can immediately guess what comes in and what goes out.

```
// Something of type 'CounterEvent' comes in from the stream and
// it's going to be converted by the bloc into an integer
class CounterBloc extends Bloc<CounterEvent, int> {}
```

The variable `state` is given by the bloc, it's initialized by overriding `initialState` and in this case it's an integer. If the incoming event were of type `increase` a new state (containing

the old value increased by 1) would be emitted.

The business logic has been properly isolated inside a single class, which is `CounterBloc`, and now it's time to link the UI with the bloc. We have to make it so that widgets can send events and receive new states.

```
void main() => runApp(const MyApp());  
  
class MyApp extends StatelessWidget {  
  const MyApp();  
  
  @override  
  Widget build(BuildContext context) {  
    // Yes, provider is used internally by the bloc library  
    // to expose instance of blocs to the children widgets  
    return BlocProvider<CounterBloc>(  
      create: (context) => CounterBloc(),  
      child: const DemoPage(),  
    );  
  }  
}  
}
```

The `flutter_bloc` package uses `provider` internally so you might find this familiar. You may have guessed that `class BlocProvider<T>` exposes a particular instance of a bloc to the children so that they can send events and listen to updates.

```
class DemoPage extends StatelessWidget {  
  const DemoPage();  
  
  @override  
  Widget build(BuildContext context) {  
    // For older versions of Dart that don't use extension  
    // methods, simply go for:  
    // final counterBloc = BlocProvider.of<CounterBloc>(context);  
    final counterBloc = context.bloc<CounterBloc>();  
  
    return Scaffold(  
      body: Center(  
        child: Row(  
          mainAxisAlignment: MainAxisAlignment.spaceAround,
```

```
        children: [
            // FlatButtons and Text widget
        ]
    )
);
}
}
```

The content of *counter* is simply a reference of the CounterBloc instance that has been passed from above the widget tree. This variable will be used by widgets to dispatch new events and receive state updates.

```
FlatButton(
    child: const Text(
        "+1",
        style: TextStyle(
            color: Colors.green,
            fontSize: 25
        ),
    ),
    onPressed: () => counterBloc.add(CounterEvent.increment),
),
BlocBuilder<CounterEvent, int>(
    builder: (context, count) =>
        Text(
            "$count",
            style: const TextStyle(
                fontSize: 30,
            ),
        ),
),
FlatButton(
    child: const Text(
        "-1",
        style: TextStyle(
            color: Colors.red,
            fontSize: 25
        ),
),
)
```

```
    onPressed: () => counterBloc.add(CounterEvent.decrement),  
) ,
```

The `BlocBuilder<T, K>` widget is the one that rebuilds the UI in response to state changes sent by the bloc. Thanks to `final counterBloc` we're able to dispatch events very easily:

```
// +1  
onPressed: () => counterBloc.add(CounterEvent.increment),  
  
// -1  
onPressed: () => counterBloc.add(CounterEvent.decrement),
```

The new state emitted by the bloc is captured by `BlocBuilder<CounterBloc, int>` which rebuilds **only** what's inside the `builder` callback. The variable `int count` contains the new counter value (the new state) which has been processed by the *bloc*.

- ❶ Widgets just send rebuild requests (events) and wait for the arrival of a new state from the bloc. The UI knows nothing about the logic behind it as it's entirely deferred to the bloc itself; this is a complete separation of business logic from design logic. Awesome!

11.3.1 Considerations

In this simple example we had to deal with increasing and decreasing a counter; nothing complex to represent in both ways (events and states) `enum` and `int` were enough. However things can be more complicated than this and there could be the need for a better representation of the input and outputs of the bloc.

- ❶ Enumerations are good when the type of events are very simple to represent and they have no data to carry. Consider using classes when events need to have some info since they can be easily stored and accessed via setters and getters.

In most of the cases, both input events and output states are represented by classes following a particular structure (it's a convention, not a strict rule). Taking again our counter app example, if we decided to use classes rather than enumerations for events the code would be the following:

Chapter 11. State management

```
abstract class CounterEvent extends Equatable {
    const CounterEvent();

    @override
    List<Object> get props => [];
}

class Increment extends CounterEvent {
    const Increment();
}

class Decrement extends CounterEvent {
    const Decrement();
}
```

An important note is that classes **must** be immutable because the new altered state is going to be changed exclusively by the bloc. In the same way, if the output state were represented by a **class** rather than an **enum** the structure would be identical. The advantage here is that classes can carry extra-data (as instance variables), if needed.

```
class CounterState extends Equatable {
    final int count;

    const CounterState(this.count);

    @override
    List<Object> get props => [count];
}
```

Logically this is nothing different from what you've seen in the original example; the difference is that we're using classes instead of an enumerations. In this cases a hierarchy is useless because a simple **enum** suffices but if we had data or a logic to carry with the event, classes would be useful.

 Using **Equatable** is not required but it's very convenient as it reduces the amount of boilerplate code required for proper object comparisons. If you don't want to use it, simply go for a classic overriding of **operator==** and **hashCode**.

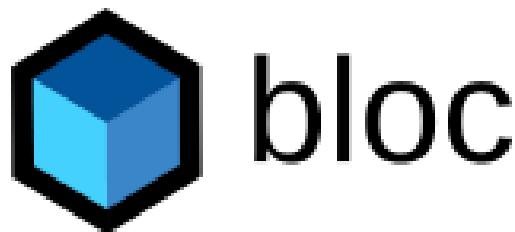
The management of the bloc itself would be a bit different too. There's the need to recognize the

type of incoming events (with a series of `ifs`) and dispatch a proper response.

```
class CounterBloc extends Bloc<CounterEvent, CounterState> {
    CounterBloc() : super(const CounterState(0));

    @override
    Stream<CounterState> mapEventToState(CounterEvent event) async* {
        if (event is Increment) {
            final newCount = state.count + 1;
            yield CounterState(newCount);
        } else if (event is Decrement) {
            final newCount = state.count - 1;
            yield CounterState(newCount);
        }
    }
}
```

Finally, `BlocBuilder<CounterBloc, CounterState>` widget will simply require to reference `newCount` to get the newly updated counter. You'll find most of the online examples and documentation relying on classes rather than enumerations so we recommend you to get familiar with them.



The official *flutter_bloc* documentation⁶ contains a lot of step-by-step examples with images and many side note explanations to make sure that you won't get lost.

⁶<https://bloclibrary.dev>

11.3.1.1 BlocListener class

This widget is useful in such cases where a callback has to be triggered whenever the bloc outputs a new state. For example, we could make it so that when the user presses on **+1** or **-1**, a snack bar with a message appears in the UI notifying the alteration of the counter.

i There's an awesome package called *Flushbar*⁷ which allows you to fully customize snack bars in your apps. It doesn't require a **Scaffold** or any other particular setup: we're using it in this example.

We're referring to the original example where the bloc takes a **CounterEvent** enumeration as input and emits an **int**.

```
class DemoPage extends StatelessWidget {
    const DemoPage();

    @override
    Widget build(BuildContext context) {
        return BlocListener<CounterBloc, int>(
            listener: (context, state) {
                Flushbar(
                    message: "The counter has been altered!",
                    duration: Duration(seconds: 1),
                )..show(context);
            }
            child: const ButtonsAndText(),
        );
    }

    // Clickable buttons and the Text widget with the counter
    // have been moved here.

    class ButtonsAndText extends StatelessWidget {
        const ButtonsAndText();

        @override
```

⁷<https://pub.dev/packages/flushbar>

```
Widget build(BuildContext context) {
    final counterBloc = context.bloc<CounterBloc>();

    return Scaffold(
        body: Center(
            child: Row(
                mainAxisAlignment:
                    MainAxisAlignment.spaceAround,
                children: [...]
            )
        )
    );
}
```

Thanks to `class BlocListener<B,S>()` we're able to listen to the stream and "catch" the responses (the states) that have been emitted by the *bloc*. In other words, this widget is a listener that listens for new states and lets you do something in response.

```
return BlocListener<CounterBloc, int>(
    condition: (previous, current) {
        // return true/false to decide if listening or not
    },
    listener: (context, state) {
        // code...
    },
    child: const ButtonsAndText(),
);
```

The `listenWhen` callback is **not** required because `true` is the default value. `condition` returns a boolean value which decides whether the listener callback will be called or not. It exposes the current and the previous values of the state so that they can be compared, if needed.

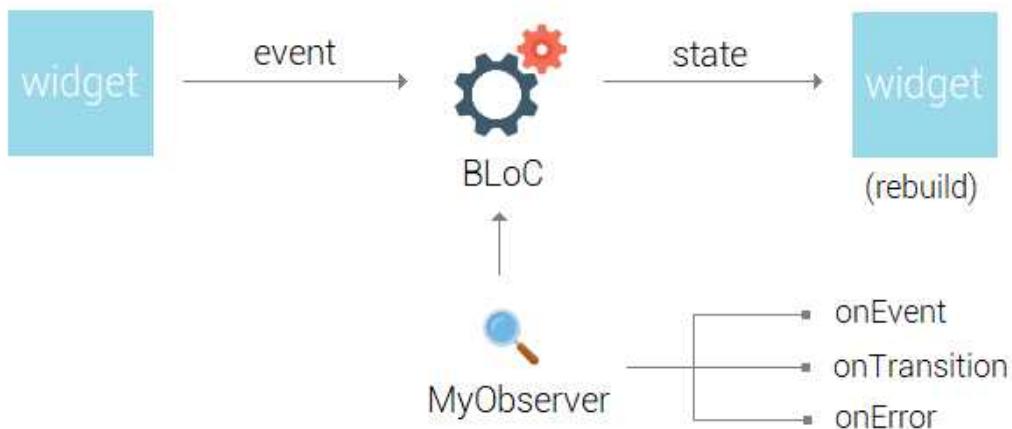
```
return BlocListener<CounterBloc, int>(
    listenWhen: (previous, current) {
        return (previous != current) && (current > 5);
    },
    listener: (context, state) {
        _showBar(context);
    },
);
```

```
        child: const ButtonsAndText(),
    );
}
```

In this example, the `listener` callback will be triggered only if the newly emitted value of the counter is greater than five.

11.3.2 BlocObserver class

If you're developing an app with multiple blocs and thus a lot of events flowing on the streams, it might be useful being able to log the activities to the console. This can be very helpful while debugging and it's not hard to do at all because the `BlocObserver` class comes to the rescue.



An observer is an object that observes what's going on in the bloc and gives you the possibility to log various things. It's very useful while debugging your app; you can log the data flow in the console for example or make some kind of analytics. An observer does **NOT** interact with the logic of the bloc because it's just a passive observer.

```
class MyObserver extends BlocObserver {
    @override // 1.
    void onEvent(Bloc bloc, Object event) {
        super.onEvent(bloc, event);
        print(event);
    }

    @override // 2.
    void onTransition(Bloc bloc, Transition transition) {
        super.onTransition(bloc, transition);
        print(transition);
    }

    void onError(Bloc bloc, Object error) {
        super.onError(bloc, error);
        print(error);
    }
}
```

```
void onTransition(Bloc bloc, Transition transition) {
    super.onTransition(bloc, transition);
    print(transition);
}

@Override // 3.
void onError(Bloc bloc, Object error, StackTrace stacktrace) {
    super.onError(bloc, error, stacktrace);
    print('$error | $stacktrace');
}
```

A bloc observer is nothing more than a subtype of `class BlocObserver` which overrides a series of methods. There's no need to implement them all, only define the ones that you need:

- override `onEvent` if you want to log information about incoming events;
- override `onTransition` if you want to log information about a transition. A *transition* is made of the current state, the next state and the input event;
- override `onError` if you want to log errors.

The only other thing that you have to do is initialize the observer and you're done. In the case of a Flutter app, the best place is before the initialization.

```
void main() {
    Bloc.observer = MyObserver();
    runApp(const MyApp());
}
```

You don't need to pass any list of blocs in use or whatever else because *Bloc* does everything automatically. Every bloc of your app will be observed by the observer which will trigger the events that you've overridden.

11.3.3 Persisting the state with HydratedBloc

The `hydrated_bloc` package is an extension of the `flutter_bloc` library which automatically stores states so that they can be restored even if the app is closed and opened again later. We're going to create a bloc that switches the theme of the app between light and dark automatically storing the state. In this way, the user's preferred theme is remembered and used the next time the app is opened.

- **Events.** The events being fired are only two, indicating whether the current theme should be light or dark. We haven't used an `enum` because in the future you could want to add more themes, maybe created by you, which might require a complex setup.

```
abstract class ThemeEvent extends Equatable {
  const ThemeEvent();

  @override
  List<Object> get props => [];

}

class DarkTheme extends ThemeEvent {
  const DarkTheme();
}

class LightTheme extends ThemeEvent {
  const LightTheme();
}
```

- **States.** The states can simply be of type `ThemeData` since it's exactly what we need. There are no extra data to return, so no need for a class hierarchy.

So far it's the same usual setup for a bloc where both events and states have been defined. The bloc itself is also very similar to the usual one but this time we're extending `HydratedBloc`. It's basically a Bloc with two more methods to override:

```
class ThemeBloc extends HydratedBloc<ThemeEvent, ThemeData> {
  static final _lightTheme = ThemeData.light();
  static final _darkTheme = ThemeData.dark();

  @override
  Stream<ThemeData> mapEventToState(ThemeEvent event) async* {
    if (event is DarkTheme) {
      yield _lightTheme;
    }

    if (event is LightTheme) {
      yield _darkTheme;
    }
  }
}
```

```
// other overrides coming soon...
}
```

We've decided to cache the states just to avoid copy/pasting the named constructor many times as you'll see in a moment. That's absolutely not required, you could have gone for a classic `yield ThemeData.dark()` for example. Here's the other interesting part of the bloc:

```
class ThemeBloc extends HydratedBloc<ThemeEvent, ThemeData> {
    ThemeData() : super(ThemeData.light());

    @override
    Stream<ThemeData> mapEventToState(ThemeEvent event) async* {...}

    @override
    ThemeData? fromJson(Map<String, dynamic> source) {
        try {
            if (source['light'] as bool) {
                return ThemeData.light();
            }

            return ThemeData.dark();
        } catch (_) {
            return null;
        }
    }

    @override
    Map<String, bool>? toJson(ThemeData themeData) {
        try {
            return {
                'light': state != ThemeData.light()
            };
        } catch (_) {
            return null;
        }
    }
}
```

Exactly as it happens with *flutter_bloc*, the constructor makes a call to `super` to pass the initial state of the bloc. However, there's an internal check on the base class constructor that loads the persisted state in case it existed. In other words, the above example works like this:

- If there isn't a state stored on the disk, `ThemeData.light()` is picked as initial state thanks to `super(ThemeData.light());`.
- If there is a state stored on the disk, the value passed to `super()` is ignored and the stored state is picked as initial.

Version 4.0.1 and earlier of *hydrated_bloc* had to override the `initialState` getter in order to set an initial state. If we made this example using any version prior to 5.0.0 we'd have to initialize the bloc in this way:

```
// hydrated_bloc 4.0.1 and earlier versions
ThemeData get initialState =>
    super.initialState ?? ThemeData.light();
```

It checks whether there's or not a state stored on the disk: if not, the default `ThemeData.light()` value is passed. In version 5.0.0 this check is automatically performed when calling `super()`. The other two important methods you're asked to override are:

- `fromJson`: it's called when trying to read the state from the internal storage. Be sure to return `null` in case of exceptions or if there are no cached states.
- `toJson`: it's called on each state change to store on the device, in this case, the user's preferred theme. If `null` is returned, no caching happens.

Last but not least, in order to properly initialize the bloc storage you need to make these two calls before the `runApp()` method. This is also the perfect place to change the directory in which `HydratedBloc` should store the data (by default it picks the device's temporary directory):

```
void main() {
    WidgetsFlutterBinding.ensureInitialized();
    HydratedBloc.storage = await HydratedStorage.build(
        storageDirectory: await getApplicationDocumentsDirectory()
    );

    runApp(const App());
}
```

The convenient `path_provider` package gives you a cross-platform access to various locations of the filesystem in which the app is running. In the end, an `HydratedBloc` is a bloc (they work in

the same way) with the addition of automatic state storage management.

11.3.4 Undo and redo with ReplayBloc

Undo and redo operations are quite common actions to find in a program. To keep consistency with the examples we made, imagine this flow in a classic counter app:

1. At startup, the counter is 0.
2. Tap on **+1**: the counter is now 1.
3. Tap on **+1**: the counter is now 2.
4. Tap the "undo" button: the counter is now 1 because the last action has been canceled.

Basically, "undo" means canceling the latest operation performed on the bloc. On the other side, "redo" means repeating the latest action performed on the bloc. Thanks to `ReplayBloc` you can very easily add automatic undo and redo support with almost no efforts.

```
// states.dart
class CounterEvent extends ReplayEvent {
    const CounterEvent();
}

class Increment extends CounterEvent {
    const Increment();
}

class Decrement extends CounterEvent {
    const Decrement();
}
```

Rather than extending `Equatable`, this time we need to subclass `ReplayEvent` in order to enable undo and redo operations on a bloc. Other than this, nothing new to do: we're already ready to create the Bloc for our counter:

```
class CounterBloc extends ReplayBloc<CounterEvent, int> {
    CounterBloc() : super(0);

    @override
    Stream<int> mapEventToState(CounterEvent event) async* {
        if (event is Increment) {
```

```
        yield state + 1;
    }

    if (event is Decrement) {
        yield state - 1;
    }
}
```

The state, for sake of simplicity, is just an `int`. There's really nothing new: it's a "regular" bloc which overrides `mapEventToState` as usual. The difference lies in the usage because, other than being able to send new events on the bloc with `add()`, there are also the two additional methods: `undo()` and `redo()`.

```
// Get a reference to the bloc
final counterBloc = context.bloc<CounterBloc>();

// Then in the 'build' function, add events or call undo()/redo()
FlatButton(
    child: const Text("+1"),
    onPressed: () => counterBloc.add(const Increment()),
),
FlatButton(
    child: const Text("-1"),
    onPressed: () => counterBloc.add(const Decrement()),
),
FlatButton(
    child: const Text("Undo"),
    onPressed: () => counterBloc.undo(),
),
FlatButton(
    child: const Text("Redo"),
    onPressed: () => counterBloc.redo(),
).
```

If you want to add undo/redo support to a bloc, just remember to extend `ReplayEvent` in your classes representing the event. All the other setup is just a regular bloc. You can even add undo/redo support for existing kinds of blocs, such as an `HydratedBloc` for example:

```
class HydratedCounterBloc extends HydratedBloc<CounterEvent, int>
```

```
with ReplayBlocMixin {
  CounterBloc() : super(0);

  @override
  Stream<int> mapEventToState(CounterEvent event) async* { ... }

  @override
  int fromJson(Map<String, dynamic> json) { ... }

  @override
  Map<String, int> toJson(int state) { ... }
}
```

With `ReplayBlocMixin` any bloc gains access to undo/redo operations, even "regular" blocs you've created extending `Bloc<E,S>`. You just need to add the mixin; there's nothing to add in the definition of the class.

```
// Get a reference to the bloc
final hydratedCounterBloc = context.bloc<HydratedCounterBloc>();

// Then in the 'build' function, add events or call undo()/redo()
hydratedCounterBloc.add(const Increment());
hydratedCounterBloc.add(const Decrement());
hydratedCounterBloc.undo();
hydratedCounterBloc.redo();
```

This example shows how powerful mixins can be: they add functionalities to a class without changing its internal definition.

11.3.5 The internals of Bloc: Cubit

Starting from version 6.0.0 of `flutter_bloc` the internals of the library got reworked and this is the reason why `initialState` has been removed and replaced with `super()`. There's a new class called **Cubit** which is the new core of **Bloc**:

- Cubit is absolutely **NOT** a replacement of Bloc. Bloc uses Cubit internally!
- There's no point in asking which one is better. If you're using Bloc, then you're also using Cubit because it's the "engine" that runs your blocs.
- A Cubit is a "lightweight" version of a Bloc.

Keep in mind that Cubit has a simple and concise API which is great for managing simple states. However, if you plan to completely replace cubit with Bloc for complex state management cases, you might get in trouble. Here's a comparison that might help you making a mental map:

- *Bloc* is a very powerful solution which is great for both simple and complex state management. It can really be used in any Flutter project with no restrictions.
- *Cubit* has a very simple and concise API and it should be used when the state to be managed is simple. It's a "subset" of the Bloc library.

We're going to show you how a `Cubit<T>` can be used to easily increment and decrement a counter. Here's a comparison on how the traditional "Counter app" would look like if you used *Cubit* or *Bloc* to manage the state. Let's start with the state management code:

- **Cubit.**

```
// counter_cubit.dart
class CounterCubit extends Cubit<int> {
    CounterCubit() : super(0);

    void increment() => emit(state + 1);
    void decrement() => emit(state - 1);
}
```

- **Bloc.**

```
// event.dart
abstract class CounterEvent extends Equatable {...}

// state.dart
class CounterState extends Equatable {...}

// count_bloc.dart
class CounterBloc extends Bloc<CounterEvent, CounterState> {
    CounterBloc() : super(const CounterState(0));

    @override
    Stream<CounterState> mapEventToState(CounterEvent e) async* {...}
}
```

As you can see, the `CounterCubit` version is simpler and with much less code to write. Note that cubits don't have states and events: they simply emit new states using methods. The usage of a

cubit in the widget tree should be very familiar to you:

- **Cubit**

```
BlocProvider<CounterCubit>(  
    create: (_) => CounterCubit(),  
    child: CounterPage(),  
) ,
```

- **Bloc**

```
BlocProvider<CounterBloc>(  
    create: (_) => CounterBloc(),  
    child: CounterPage(),  
) ,
```

Once the provider has been placed right above the widgets you need, you can easily use a "builder widget" to listen for updates. It's no different from the structure you've been used to see up to now and all *flutter_bloc* widgets are compatible with both *Bloc* and *Cubit* instances.

- **Cubit**

```
// Cubits are used inside 'BlocBuilder's  
BlocBuilder<CounterCubit, int>(  
    builder: (_, counter) {  
        return Text("$counter");  
    },  
) ,
```

- **Bloc**

```
BlocBuilder<CounterBloc, CounterState>(  
    builder: (context, state) {  
        return Text("${state.count}");  
    },  
) ,
```

If we compared the amount of code, it's clear that blocs require more boilerplate than cubits. In this simple counter app, going for Cubit is fine since the state to manage is very easy (but still, Bloc would be fine anyway). We want to point out again that cubit **shouldn't** be the default choice for any project: for advanced state management you'd better use Bloc.

i In certain cases, you might not be sure that the state will always be simple to manage. For example, it might happen that your project manager could tell you to add more features and things might get complicated. If you're stuck in this situation, go for Bloc which has you covered in any situation.

There are also `HydratedCubit<T>` and `ReplayCubit<T>` which do the same thing as the "bloc counterpart" with the same name. In the following example, we show how to persist and restore the state of a `Cubit<T>` of a traditional "counter app".

```
class CounterCubit extends HydratedCubit<int> {
    CounterCubit() : super(0);

    void increment() => emit(state + 1);
    void decrement() => emit(state - 1);

    @override
    int fromJson(Map<String, dynamic> json) =>
        json['value'] as int;

    @override
    Map<String, int> toJson(int state) =>
        {'value': state};
}
```

It works exactly like a `HydratedBloc<T>` with the only difference that only cubits are allowed. Of course, you need to initialize the storage before starting the application.

```
void main() async {
    WidgetsFlutterBinding.ensureInitialized();
    HydratedBloc.storage = await HydratedStorage.build();
    runApp(MyApp());
}
```

11.3.6 Good practices

If you decide to use `flutter_bloc` for your projects you'll probably find yourself in the situation where it's required to have a `BlocBuilder<B,S>` is inside a `BlocListener<B,S>`. It's perfectly fine:

```
BlocListener<BlocA, StateA>(
    listener: (context, state) {
        // listen to state changes and do something,
        // like showing a snackbar or a dialog
    }
    child: Column(
        children: [
            const Text("Something"),
            BlocBuilder<BlocA, StateA>(
                builder: (context, state) {
                    return UseTheState(state);
                }
            )
        ]
    )
)
```

No performance problems here but there's quite a lot of code to write and many types to define. There are two main ways to improve the readability of this code:

- Use the `BlocConsumer<B,S>` widget which allows you to use a listener and a builder all together. It's like nesting a `BlocBuilder<B,S>` inside a `BlocListener<B,S>` but with less boilerplate code.

```
BlocConsumer<BlocA, StateA>
    // This is the 'BlocListener' part
    listener: (context, state) {
        // listen to state changes and do something,
        // like showing a snackbar or a dialog
    }
    // This is the 'BlocBuilder' part
    builder: (context, state) {
        return Column(
            children: [
                const Text("Something"),
                UseTheState(state),
            ]
        )
    }
}
```

)

It's much easier to read and for sure you have less code to write because both `listener` and `builder` expose the main parameters of the respective widgets. You should use consumer only when you need to rebuild the UI **and** listen to state changes; if this is not the case, use `BlocBuilder` and `BlocListener` alone.

- Move the content of the child in a separated widget so that you can also try to make, if possible, a `const` constructor. It's what we've done earlier while showing how to implement a Snackbar for the counter app:

```
BlocListener<BlocA, StateA>(
  listener: (context, state) {
    // listen to state changes and do something,
    // like showing a snackbar or a dialog
  }
  child: const SubWidget(),
)
```

And then move the `Column` with its contents inside a dedicated widget:

```
class SubWidget extends StatelessWidget {
  const SubWidget();

  @override
  Widget build(BuildContext context) {
    // Column with text and BlocBuilder
  }
}
```

Always remember to make constant constructors whenever possible!

If your app has a complex logic which requires more than a single bloc in the same widget, consider using a `MultiBlocProvider`. The name should sound familiar because in practice it's just a `MultiProvider` which exposes a series of blocs to the children:

```
MultiBlocProvider(
  providers: [
    BlocProvider<Bloc1>(
      create: (context) => Bloc1(),
    ),
    BlocProvider<Bloc2>(

```

```
        create: (context) => Bloc2(),
    ),
    BlocProvider<Bloc3>(
        create: (context) => Bloc3(),
    ),
),
];
);
```

Be aware that *flutter_bloc* uses *provider* under the hood but you cannot mix the two libraries together, simply because your code won't compile. Prefer using the above approach rather than nesting a series of `BlocProviders` because the code might become very hard to read.

11.4 Good practices for state management

We want to repeat (for the last time!) that using `StatefulWidget`s is perfectly fine and sometimes fundamental, for example when the class has non-final instance variables like `int counter = 0;`. A stateful widget isn't more efficient than a stateless one and vice versa. What is bad is the usage of `setState` without `InheritedWidget`:

- It breaks the SRP because a widget has to take care of the UI and the logic together, which can also become messy very quickly in large apps.
- In a complex architecture the usage of `setState` alone can be problematic because it doesn't give the possibility to optimize children rebuilds. It should be integrated with `InheritedWidget` but there would be a lot of work to do in terms of development, testing and maintenance.
- If you really have to use it because you're doing maintenance of an app made by someone else, be sure that it doesn't compute anything. Don't call functions that calculate/compute something; just do simple assignments.

The official Flutter documentation⁸ maintains a list of active state management libraries; among them you can find *provider* and *flutter_bloc* that we've covered in the previous sections.

- i** You can decide to manage the state of your app with whatever package you want, whether it be a provider, bloc, Redux or MobX. The decision is up to you because they all do their job very well.

⁸<https://flutter.dev/docs/development/data-and-backend/state-mgmt/options>



It's not possible to say which one is the best or if one works better than the other. For sure they solve the same problem but they use different approaches and different logics so it's difficult to make a fair comparison.

If you want to have another complete example about the `provider` package, check the official documentation ⁹ which uses `ChangeNotifier` and `ChangeNotifierProvider`. In the end, here's a quick recap of the best practices about state management:

- **provider**

- Prefer the usage `Consumer<T>` which rebuilds only the widgets that you want. If you want to have even more control for better optimization, use `Selector` which is able to specifically tell when a rebuild should happen.
- Avoid the usage of `Provider.of<T>(context)` inside `build` as it will rebuild any children, even the ones that don't need to be updated.
- Consider using a `MultiProvider()` when you have more than a single provider at the same level; it makes the code more readable.
- Prefer using `context.watch<T>()` and the other extension methods on `BuildContext` since it's more concise and modern. There's no performance difference if compared to `Provider.of<T>()` but it's just a shorter syntax.
- In general, prefer using extension methods since they're shorter and more modern.

- **flutter_bloc**

⁹<https://flutter.dev/docs/development/data-and-backend/state-mgmt/simple>

- Prefer using `context.bloc<T>()` which is less verbose than using the non-extension version `BlocProvider.of<T>(context)`
- Put inside `BlocBuilder<T>()` only the widgets that actually need a rebuild in response to a state change. Don't place into it widgets that don't need to listen to changes, even if they are `const` because they "pollute" the scope of the bloc.
- If you have the need to both listen and build for a bloc, use `BlocConsumer<B,S>` instead of nesting widgets; it reduces the amount of boilerplate code.
- Consider using `ReplayBloc` or `ReplayBlocMixin` to add undo/redo actions to your blocs.

12 | Routes and navigation

12.1 Basics of navigation and routing

Organizing information across several screens is one of the most important building blocks of any architecture. A very common example is the one in which your app's first page is a login form and, if the user provides a correct combination of username and password, a welcome page appears.

- ➊ In the Flutter world, your app's pages are called *routes*, or *screens*, and they are the equivalent of *Activities* in Android or *ViewControllers* in iOS.

The most common way to create a page involves the usage of a stateless or stateful widget with a `Scaffold` (or a `CupertinoPageScaffold`). This chapter is going to show how to properly structure and manage the pages of your app. Before getting started, we want to suggest you a possible folder structure for your app:

```
- lib/
  - routes/
  - widgets/
  - main.dart
  - routes.dart
```

Very intuitively, `routes/` is going to contain all those UI widgets representing a route of the app. Inside `widgets/` we recommend putting all those reusable widgets that support the creation of your app's pages. Look at this simple widget:

```
class FooterName extends StatelessWidget {
  const FooterName();
```

```
@override  
Widget build(BuildContext context) {  
    return Text("Your name",  
        style: const TextStyle(...)  
    );  
}  
}
```

Placing this code in `widgets/footer_name.dart` is very convenient because you know it's an "utility" piece of UI that will be reused across multiple pages as a footer. It can be integrated with `const FooterName()` and a simple change on `footer_name.dart` is automatically reflected everywhere.

12.1.1 Creation of routes

We're going to create a simple application with two screens: `HomePage`, the first route which appears when the app starts, and `RandomPage`, a route that displays a random number at the center of the screen. The layout of the pages will be created with a convenient `Scaffold` from the *material* library.

```
// Located in routes/home_page.dart  
class HomePage extends StatelessWidget {  
    const HomePage();  
  
    @override  
    Widget build(BuildContext context) {  
        return Scaffold(  
            body: Center(  
                child: RaisedButton(  
                    onPressed: () {},  
                    child: const Text("Random"),  
                ),  
            ),  
        );  
    }  
  
// Located in routes/random_page.dart
```

```

class RandomPage extends StatelessWidget {
  const RandomPage();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Text("${Random().nextInt(20)}"),
      ),
    );
}

```

Any other page of the app will go in the `routes/` folder which might contain other folders to better organize the files. When there are many files, creating a good hierarchy of directories and sub-directories is fundamental in order to not get lost in your own architecture.

12.1.2 The main.dart file

The general structure of a `main.dart` file might look like the following. The usage of `material` instead of `cupertino` is not relevant, it just means we're using `MaterialApp()` in place of `CupertinoApp()`.

```

import 'package:flutter/material.dart';

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

// 1.
class RandomApp extends StatelessWidget {
  const RandomApp();

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      onGenerateTitle: (context) => "Random App",

      initialRoute: RouteGenerator.homePage,           // 2.
      onGenerateRoute: RouteGenerator.generateRoute,   // 3.
    );
}

```

```
// Hides the debug stripe on the top-right corner
// which might be annoying to see!
debugShowCheckedModeBanner: false,
);
}
}
```

This is how you should structure your `main.dart` file. Of course you can rearrange it as you prefer but generally it should only contain the code that initializes the app and nothing more.

1. This is the widget located at the root of the tree which has the task to setup the various routes and redirect the user to the first page.
2. This is a string. We're going to explain what is `RouteGenerator` in a moment; it basically tells Flutter which is the first route that has to be loaded.
3. This is a method reference. The navigation between pages in Flutter works like if you were using a browser. Each page has a name, which is like an "url" and you show a page by "navigating" to that address.

If your pages need to share a cache or you have the need for one (or more) notifiers, it's good practice wrapping the `MaterialApp` inside a `MultiProvider` which nicely gathers a series of providers.

```
@override
Widget build(BuildContext context) {
    return MultiProvider(
        providers: [
            Provider<DataCache>(create: (_) => DataCache()),
            ChangeNotifierProvider<Something>(
                create: (_) => Something()
            ),
        ],
        child: MaterialApp(...)
    );
}
```

12.1.3 The routes.dart file

Having a single class that handles the entire routing of your app is very nice as it fully embraces the single responsibility principle. By consequence, when you'll have to deal with routes you'll

Chapter 12. Routes and navigation

always open `routes.dart` because every route-related logic is only there.

- i** This is a "centralized" control of your routes which keeps the code clean. Give this file whichever name you want but we suggest it should contain the word "`route`" so that you'll recognize its contents immediately.

The following snippet shows the contents of `routes.dart`. The example is using the material library, so we're using `return MaterialPageRoute()`, but if you were in the cupertino world you'd go for `return CupertinoPageRoute()`.

```
// 1.
class RouteGenerator {
  // 2.
  static const String homePage = '/';
  static const String randomPage = '/random';

  // 3.
  RouteGenerator._() {}

  // 3.
  static Route<dynamic> generateRoute(RouteSettings settings) {
    // 4.
    switch (settings.name) {
      case homePage:
        // .5
        return MaterialPageRoute(
          builder: (_) => const HomePage(),
        );

      case randomPage:
        return MaterialPageRoute(
          builder: (_) => const RandomPage(),
        );
    }

    default:
      throw FormatException("Route not found");
  }
}
```

```

        }
    }

// 5.
class RouteException implements Exception {
    final String message;
    const RouteException( this.message);
}

```

There **must** be a route named '`/`' which has to map to the first page that's being shown when your app starts. It's a requirement, not just a good practice.

1. Actually this class is just a "wrapper" for a single `static` function because declaring global functions is possible, but it's not a good idea. `generateRoute()` is the main actor.
2. Each page of the app is uniquely identified by a string; it's the same thing you're used to see on the internet where web pages are identified by URLs. In this case:
 - The `DemoPage` route is associated with the '`/`' path
 - The `RandomPage` route is associated with the '`/random`' path

Do you want to press on a button and show the `RandomPage` widget? It's very easy, you just need to write...

```
Navigator.of(context)?.pushNamed(RouteGenerator.randomPage);
```

... and the new screen appears. The `pushNamed()` method takes a path, which is linked to a page, and navigates to it; in this case it looks for '`/random`' and shows the widget that's been assigned to it.

3. The `settings` parameter carries some info gathered by the `Navigator.of` method such as the name of the route. This part is very important because it's where you map an URI (the path) to the route/screen.
4. The `MaterialPageRoute<T>` class replaces one screen with another by using an Android slide transition. The equivalent class for the cupertino package is `CupertinoPageRoute<T>` which does the same job but with an iOS slide transition.

You've just seen how to create a series of "**named routes**" (routes at which you assign a name). For example, the route '`/random`' is said to be the *named route* of the `RandomPage` widget.

```
// Inside 'MaterialApp()' of main.dart
initialRoute: RouteGenerator.homePage,
```

```
onGenerateRoute: RouteGenerator.generateRoute,
```

`initialRoute` takes the path of the first page that Flutter has to load when the app is opened. `onGenerateRoute` maps an URI (a string like `'/random'`) to a widget (a route like `RandomPage`) so that the `Navigator` class is able to show the pages.

12.1.4 Navigating between pages

At this point, we have our routes properly set up inside `routes.dart` and we're ready to move from one page to the other. A quick recap of the job we've done so far:

- creation of two routes called `DemoPage` and `RandomPage`;
- setup of the routes in the `MaterialApp()` of `main.dart`;
- The creation of the `RouteGenerator` class which maps an URI to a widget so that the `Navigator` class can open routes.

The first page of our example app is `HomePage` which has to open another route when the button is pressed. All we need to do is calling `pushNamed()` with the name of the target route that has to appear.

```
// The 'HomePage' widget's build method
return Scaffold(
    body: Center(
        child: RaisedButton(
            onPressed: () =>
                Navigator.of(context)?.pushNamed(
                    RouteGenerator.randomPage
                )
            child: const Text("Random"),
        ),
    ),
);
```

This code makes the `RandomPage` route appear with a sliding animation from the bottom to the top of the screen. If you want to come back to the `HomePage` route just press the back button or make a simple call to `Navigator.of(context)?.pop()`.

```
// The 'RandomPage' widget's build method
final rand = Random().nextInt(20);
```

```
return Scaffold(  
    body: Center(  
        child: Text("$rand"),  
    ),  
) ;
```

12.1.5 Good practices

Both `MaterialApp` and `CupertinoApp` are needed to configure the routes that will be used by the `Navigator`, you've just seen it. Actually, we've only described the good way to implement routes management but in reality there are more ways to achieve the same result:

- **Good.** Encapsulate the routing management in a single class of a dedicated file. Other than respecting the SRP, you'll be able to easily handle a lot of routes with ease in a single, centralized place.

```
MaterialApp(  
    initialRoute: RouteGenerator.homePage,  
    onGenerateRoute: RouteGenerator.generateRoute,  
) ;
```

Nothing new, this is what we've implemented in the previous section. The route management is completely separated from the rest of the app thanks to `class RouteGenerator`.

- **Bad.** Do the routing management directly inside the `MaterialApp` widget with no logic separation between various areas. This example may look good, but if there were many routes there would probably be maintenance problems.

```
MaterialApp(  
    initialRoute: '/',  
    routes: <String, WidgetBuilder>{  
        '/': (BuildContext context) => const HomePage(),  
        '/random': (BuildContext context) => const RandomPage(),  
    },  
) ;
```

You can use the `routes` parameter and assign it a `Map<String, WidgetBuilder>` which maps a path to a route. This is what we did inside `generateRoute` with the exception

that here the routing logic is being injected directly in the widget. In this case, there's no separation between UI logic and routing logic.

```
MaterialApp(  
    initialRoute: '/',
    routes: <String, WidgetBuilder>{
        '/': (BuildContext context) => MaterialPageRoute(  
            builder: (context) => const HomePage(),
        ),
        '/random': (BuildContext context) => MaterialPageRoute(  
            builder: (context) => const HomePage(),
        ),
    },
);
```

Wrapping screens inside `MaterialPageRoute()` automatically adds the transition animation while navigating and a `BuildContext` to use if needed. The biggest problem is that builder methods can be complicated or require more logic; if you had many routes your code would become messy and hard to maintain.

- **Bad.** Do the same thing but use `home` instead of `initialRoute`.

```
MaterialApp(  
    home: const HomePage(),
    routes: <String, WidgetBuilder>{
        // define routes...
    },
);
```

Do **NOT** define `home` and `initialRoute` together because it'd be a conflict that might end up in an unexpected behavior.

While the `RouteGenerator` class approach is harder to understand and involves multiple steps, it's the best to use as it also scales well. Do not mix different routing management approaches together, you might get lost in your own code!

```
MaterialApp(  
    // This is very error-prone!
    initialRoute: RouteGenerator.homePage,  
    onGenerateRoute: RouteGenerator.generateRoute,
```

```
routes: <String, WidgetBuilder>{
  '/': (BuildContext context) => MaterialPageRoute(
    builder: (context) => const HomePage(),
  ),
  '/random': (BuildContext context) => MaterialPageRoute(
    builder: (context) => const HomePage(),
  ),
},
);
```

Can you guess what's going on? How can you know which parameter will be chosen by the framework to handle the routing? There is an order actually¹:

1. Flutter looks first if the `home` parameter is set;
2. if `home` is null, then the `routes` parameter is chosen;
3. if `routes` is null, then the `onGenerateRoute` callback is chosen;
4. if none of the above resolved to a valid call, the `onUnknownRoute` callback is called.

You don't need to remember this sequence at all. In order to not make confusion avoid the setup of the routes inside `MaterialApp` or `CupertinoApp` and walk in the SRP-way by creating a separated class, like `RouteGenerator`.

12.1.6 Navigator 2.0

The `Navigator` class existed since the early days of Flutter and we're describing how to use it in this chapter. The Flutter team will deliver an update on the routing system with the following changes (some of them are **breaking**):

- `pop()` doesn't return a value anymore;
- the `isInitialRoute` property of `RouteSetting` is deprecated;
- the presence of a new widget called `Router`.

`Router` is a new widget you can use to open and close pages of an app. It wraps a `Navigator` and configures its current list of pages based on the current app state. Let's see what you'll have to do in order to migrate the code to the new navigation system.

¹<https://api.flutter.dev/flutter/material/MaterialApp-class.html>

- If you rely on `RouteSettings.isInitialRoute`, replace it with the new callback introduced in both `MaterialApp` and `CupertinoApp`. Of course, the `RouteGenerator` class has to be changed accordingly as well:

```
// Before
RouteGenerator {
    static Route<dynamic> generateRoute(RouteSettings settings) {
        if (settings.isInitialRoute) {
            return A();
        }
        return B();
    }
}

MaterialApp(
    onGenerateRoute: RouteGenerator.generateRoute,
)

// Migration
RouteGenerator {
    static Route<dynamic> generateRoute(RouteSettings settings) {
        return B();
    }

    static List<Route> generateInitialRoutes(String name) {
        return <Route>[ A() ];
    }
}

MaterialApp(
    onGenerateInitialRoutes: RouteGenerator.generateInitialRoutes,
    onGenerateRoute: RouteGenerator.generateRoute,
)
)
```

Basically, you don't have to perform a check inside `generateRoute` anymore because it's done internally by the framework when you assign values to `onGenerateInitialRoutes`.

- In the next section, we will see that `pop()` can return a value. You'll learn that working

with `pop()` to exchange data is not good and should be avoided. Furthermore, with the new update you'll be forced to use `canPop()` and then call `pop()` separately.

```
// Before
if (Navigator.pop(context)) {
    print("Can pop");
} else {
    print("Cannot pop");
}

// Migration
if (Navigator.canPop(context)) {
    print("Can pop");
} else {
    print("Cannot pop");
}
Navigator.pop(context);
```

The navigator pops the route anyway, so in the new version we call `pop()` at the end.

Keep in mind that `Navigator` won't be removed in favor of `Router`. Be sure to have a look at the documentation when the navigator 2.0 upgrade will come out.

12.2 Passing data between pages and widgets

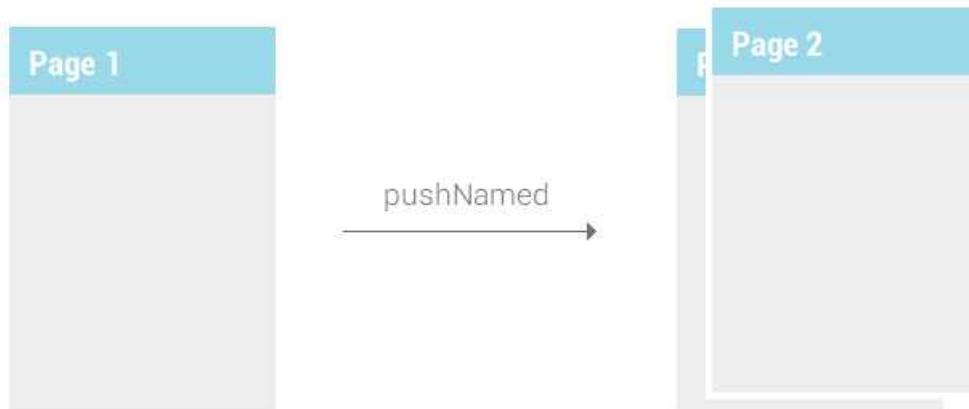
Sharing data between two or more pages is a very common need: different widgets located in various parts of the tree might require some data sharing. Common use cases can be:

1. in a layout with many tabs, you need to pass data from a tab to another;
2. you have a route (a page) which has to send a primitive type or a complex object to another route when navigating to it;
3. in the same page, two widgets need to exchange data.

All of these tasks can be easily accomplished with the `provider` package as it's a perfect option for data sharing among widgets. There would also be the possibility to pass data using `Navigator` but we will see why it's discouraged.

12.2.1 The Navigator class

The `Navigator` class allows you to move from a route to another going back and forth. The pages you navigate to are "overlapped" and the framework keeps track of them using a stack. This picture gives you an idea of how it works:



When you navigate to a new screen with `pushNamed("/route_name")`, the `Navigator` adds a new route to its internal stack. By consequence, the route at the top of the stack is the one you're seeing on the screen. If you want to go back to the previous page, call `pop()`:



This method removes the route at the top of the stack so that the widget below becomes the

new visible screen. Going forth and back between pages is just a matter of *pushing* and *popping* routes in a stack data structure managed by the `Navigator`.

- `Navigator.of(context)? .pushNamed("/route_name")`
- `Navigator.of(context)? .pop()`

But there's also another equivalent way to use them:

- `Navigator.pushNamed(context, "/route_name")`
- `Navigator.pop(context)`

Maybe it'd be better using the `of(context)` version as it's a more common pattern also used by other widgets, but it returns a nullable type. If you don't want to use named routes, you can use "*plain routes*" that work without strings:

- **Named route.**

```
Navigator.of(context)? .pushNamed("/route_name");
```

- **Plain route**

```
Navigator.of(context)? .push(  
    MaterialPageRoute(  
        builder: (context) => YourScreen()  
    )  
);
```

If you aren't using named routes, you have to manually specify the destination every time (= code duplication) and mix UI logic with navigation logic. Again, the `RouteGenerator` approach we've covered at the beginning of the chapter is what we recommend!

12.2.2 Passing data with Navigator

We're creating a simple to-do app; when an item of the list is tapped, a new route appears with a description of the selected to-do. The first thing to do is the creation of a `Todo` type which represents an entry of the list.

```
class Todo {  
    final String title;  
    final String description;
```

Chapter 12. Routes and navigation

```
    const Todo(this.title, this.description);
}
```

In the first page, called `TodosPage`, we're creating a list of `Todos` that are displayed using a `ListView`. When a to-do is tapped, a new route called `InfoPage` is opened to show the description of the selected item.

```
class TodosPage extends StatelessWidget {

    // Some todos to show on the screen.
    final List<Todo> todos = [
        Todo("Item 1", "First to-do of the list"),
        Todo("Item 2", "Second to-do of the list"),
        Todo("Item 3", "Third to-do of the list"),
    ];

    const TodosPage();

    void _itemPressed(BuildContext context, Todo item) =>
        Navigator.of(context)?.push(
            MaterialPageRoute(
                builder: (context) => InfoPage(todo: item),
            ),
        );
}

@Override
Widget build(BuildContext context) {...}
}
```

The to-do list is directly given by the widget just to keep the example simple and focus on the problem, which lies in `_itemPressed`. As you can see named routes aren't being used and we'll explain why in a moment.

```
// build method of 'TodosPage'
return Scaffold(
    body: ListView.builder(
        itemCount: todos.length,
        itemBuilder: (context, index) {
            return ListTile(
                title: Text(todos[index].title),
```

```

        onTap: () => _itemPressed(context,
            todos[index].description
        ),
    );
},
),
);

```

Finally, we need a route that displays the description of the selected widget such as `InfoPage`.

```

class InfoPage extends StatelessWidget {
    final Todo item;
    const InfoPage(this.item);

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            body: Center(
                child: Text("${item.description}")
            ),
        );
    }
}

```

The biggest problem of this approach is that `Navigator` cannot take a named route with the `RouteGenerator` strategy we're used to adopt. You have to inject `MaterialPageRoute` directly inside the method of the navigator and this can be a problem:

- First of all, the single responsibility principle is not respected because the navigation logic is spread across multiple files. Widgets now have to take care about navigation management and, if any, custom transitions between pages.
- With this approach you can send data to `InfoPage` by writing some boilerplate code. What if you had the need to return data back from `InfoPage` to `TodosPage`? There is a solution:

```

// In the 'TodosPage' widget, send data to 'InfoPage'
void _itemPressed(BuildContext context, Todo item) async {
    // Result contains the data returned by 'pop' in InfoPage
    final result = await Navigator.of(context)?._push(
        MaterialPageRoute(
            builder: (context) => InfoPage(todo: item),

```

```
        ),
    );
}

// In the 'InfoPage' widget, send data back to 'InfoPage'
void _returnSomething() {
    Navigator.pop(context, "Pasta pizza maccheroni");
}
```

We've changed the body of the method because the `push` function returns a `Future<T>` which completes only when `pop` has been called somewhere. In practice the `push` method returns a value when `pop` is called somewhere in the tree; in this case the string is stored inside `final result`.

To sum up, passing data with `Navigator` is complicated and badly mixes many logics inside your widgets. Keeping track of the data flow across various `push` and `pop` calls can quickly get confusing.

12.2.3 Passing data with provider

Sharing data among routes with `Navigator` can be troublesome and you've just seen why. Using *provider* you can get rid of all those problems and easily pass objects between widgets without caring about the navigation.

1. The first thing to do is the creation of a model class, which is a class representing a to-do item, like we've done earlier.

```
// Still the same todo.dart file
class Todo {
    final String title;
    final String description;

    const Todo(this.title, this.description);
}
```

2. Rather than having the to-do list in a widget, we move it to a dedicated class that's going to be used as a cache. Thanks to *provider* the various pages will be able to access the list whenever they want without having to move data back and forth.

```
// todo_cache.dart
class TodoCache {
```

```
var _index = -1;

final List<Todo> _todos = [
    Todo("Item 1", "First to-do of the list"),
    Todo("Item 2", "Second to-do of the list"),
    Todo("Item 3", "Third to-do of the list"),
];

void addItem(String title, String descr) {
    _todos.add(Todo(title, descr));
}

int get index => _index;
set index(int value) {
    if ((value >= 0) && (value < _todos.length)) {
        _index = value;
    } else {
        _index = -1;
    }
}

UnmodifiableListView<Todo> get list =>
    UnmodifiableListView<Todo>(_todos);
}
```

We know this is a very simple logic: you could also implement the deletion of an item, the lookup on the list and much more. For the scope of our example that's enough.

3. Using a `Provider<TodoCache>` we can expose the list to the children (the routes) so that they're able to share the data. We've decided to place a provider right above the widget containing our pages and not inside `runApp()` just to not pollute the scope. It actually wouldn't make any difference but `MyApp` doesn't require data sharing so including it in the provider would be useless.

```
// todo_cache.dart
void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
    @override
```

Chapter 12. Routes and navigation

```
Widget build(BuildContext context) {
    return Provider<TodoCache>(
        create: (_) => TodoCache(),
        child: const TodosPage(),
    );
}
```

Nothing new up to here because that's the typical setup for a *provider*. The `TodosPage` doesn't change much; the only difference from before is on how the `Navigator` is called. We're about to do dependency injection thanks to *provider* and thus a huge step forward to respect the SRP.

```
class TodosPage extends StatelessWidget {
    const TodosPage();

    void _open(BuildContext context) =>
        Navigator.of(context)?.pushNamed(RouteGenerator.infoPage);

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            body: Consumer<TodoCache>(
                builder: (context, cache, _) {
                    return ListView.builder(
                        itemCount: cache.list.length,
                        itemBuilder: (context, index) {...}
                    );
                }
            ),
        );
    }
}
```

We can now use named routes with the `RouteGenerator` approach because data are not coupled to the navigation logic anymore. In this way we have separated data sharing from navigation logic and thus there's no need to work with `push` and `pop` to send/receive objects.

```
// 'itemBuilder' of ListView.builder in TodosPage
itemBuilder: (context, index) {
    return ListTile(
```

```
        title: Text("${cache[index].title}"),
        onTap: () {
            cache.index = index;
            _open(context);
        }
    );
}
```

Tapping an item of the list calls `_open` which pushes a new route to the screen. Thanks to `cache.index = index;` we can remember which item has been selected so that other widgets will be able to get a reference to the to-do just by calling:

```
final selectedTodo = cache.list[cache.index];
```

For example, if you tapped on the second item of the list `cache.index` would be 1 because the selected item is the second to-do of the list. Our `InfoPage` route doesn't need to get the index via constructor anymore because the provider has anything we need:

```
class InfoPage extends StatelessWidget {
    const InfoPage();

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            body: Center(
                child: Consumer<TodoCache>(
                    builder: (context, cache, _) {
                        final item = cache.list[cache.index];
                        return Text("${item.description}");
                    }
                ),
            ),
        );
    }
}
```

If you wanted to improve this code even more, you could remove the `index` variable and use another strategy to select the correct to-do item in the list. Apart from this, what you should learn from this chapter is that dependency injection (with *provider*) is the way to go for data sharing among routes.

- the routing logic can be isolated in a `RouteGenerator` class or whatever you want to name it;
- the business logic can be isolated in a dedicated class (`TodoCache` for example);
- the UI is not mixed with the logic; thanks to provider you inject the dependencies in the widgets but there's a very loose coupling between components.

Passing data using `push` and `pop` directly is bad: you cannot use named routes, routing logic mixes with business logic and keeping track of the data flow quickly becomes cumbersome.

12.3 Other routing techniques

As you've seen throughout this chapter, a `BuildContext` is **essential** to navigate among routes. In some cases, you could have the need to move to a new page without having a context available: in this case, you can use keys.

```
class RouteGenerator {
  RouteGenerator._();

  // Expose a key to use a navigator without a context
  static final Key = GlobalKey<NavigatorState>();

  static Route<dynamic> generateRoute(RouteSettings settings) {...}
}
```

We've added a new `static` field to our famous `RouteGenerator` class which will be used to call `pushNamed` without a context. It needs to be linked to the material or cupertino root widget in order to work as expected:

```
MaterialApp(
  navigatorKey: RouteGenerator.key,
)

CupertinoApp(
  navigatorKey: RouteGenerator.key,
)
```

Now you can use `RouteGenerator.key.currentState?.pushNamed()` to navigate among routes as usual. The difference is that no `BuildContext` is required but keep in mind the following:

- Try to use `Navigator.of(context)?pushNamed()` as much as possible. It should be your default choice.
- Remember that global keys are relatively expensive to use.
- Use a `GlobalKey<NavigatorState>()` only if really needed. In chapter 22 we will see a case in which a navigation key is essential, but still it's a single case.

Flutter's navigation tools are easy to use but they aren't perfect; data sharing between routes can become quite cumbersome and thus relying on *provider* is essential. Here's a list of the most relevant packages for an alternative routes management (with a link to their repository):

1. Flutter modular.

https://pub.dev/packages/flutter_modular

```
class AppModule extends MainModule {
    // Injectable classes such as "provider" or "flutter_bloc"
    @override
    List<Bind> get binds => [];

    // Your app's routes
    @override
    List<Router> get routers => [
        Router("/", child: (_, __) => HomePage()),
        Router("/info", child: (_, __) => InfoPage()),
    ];

    // The widget that contains MaterialApp or CupertinoApp
    @override
    Widget get bootstrap => MyRootWidget();
}
```

2. Fluro.

<https://pub.dev/packages/fluro>

```
final router = Router();

// Define a route
var loginRoute = Handler(
```

```

        handlerFunc: (context, params) {
            return LoginScreen();
        }
    );

// Assign it to the router
void defineRoutes(Router router) {
    router.define("/login", handler: loginRoute);
}

// Navigate
router.navigateTo(context, "/login",
    transition: TransitionType.fadeIn
);

```

3. Sailor.

<https://pub.dev/packages/sailor>

```

// Routing class
class Routes {
    static final sailor = Sailor();

    static void createRoutes() {
        sailor.addRoute(SailorRoute(
            name: "/loginPage",
            builder: (context, args, params) {
                return LoginPage();
            },
        )));
    }
}

// Routes initialization
void main() async {
    Routes.createRoutes();
    runApp(const MyApp());
}

```

```
// Registration of Sailor
Widget build(BuildContext context) {
  return MaterialApp(
    navigatorKey: Routes.sailor.navigatorKey,
    onGenerateRoute: Routes.sailor.generator(),
  );
}

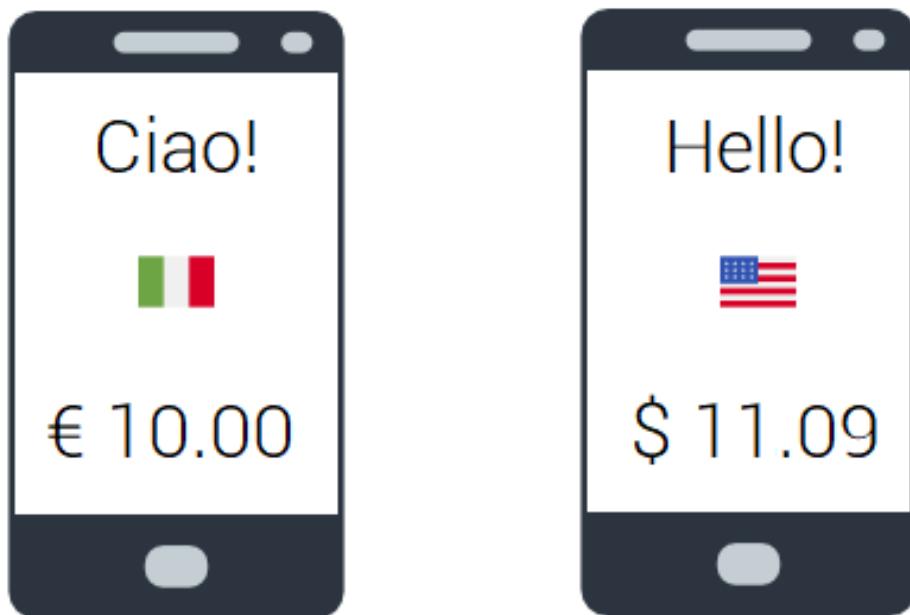
// No context needed for navigation
Routes.sailor.navigate("/loginPage");
```

They're not necessarily better or worse than Flutter's Navigator approach; it's just a series of alternatives you might find interesting.

13 | Localization and internationalization

13.1 Introduction

Other than being responsive, high quality apps are also well **localized**: it means that they automatically adapt the contents in order to appeal to a geographically specific target device. While this might sound quite complicated to understand, this example will get you to the point:



The above picture shows how the **same** application behaves in two different mobile phones of

people coming from different parts of the world. The app is said to be "**localized**" because, according with the locale in which it's run, it adopts to the user's culture and geographical area.

- On the left, the app is loaded in a mobile device whose locale is set to "Italy". For this reason it shows the price in euro, the italian flag and the text is translated in the native language.
- On the right, the app is loaded in a mobile device whose locale is set to America. For this reason it shows the price in dollars, the american flag and the text is translated in the native language.

In other words, your app is localized when certain graphical elements of the UI automatically change according with the culture and the geographical area of the device. In general, the most relevant aspects to take into account are:

- translating sentences in the proper language;
- showing prices in the proper currency (euro, dollar, Sterling and so on);
- choosing between 24h or 12h time format (18.30 or 8.30pm)
- time-based implementations should take into account time zones.

Making a fully localized app is not so easy because there are many aspects to consider; the bigger your app is, the more you'll have to localize. However, Flutter comes to the rescue with the *intl*¹ package which includes many localization facilities.

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

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

class DemoApp extends StatelessWidget {
    const DemoApp();

    @override
    Widget build(BuildContext context) {
        // Same exact thing with 'CupertinoApp'
        return MaterialApp(
```

¹<https://pub.dev/packages/intl>

```

        localizationsDelegates: [
            GlobalMaterialLocalizations.delegate,
            GlobalCupertinoLocalizations.delegate,
            GlobalWidgetsLocalizations.delegate,
        ],
    );
}
}

```

The minimal setup required for `intl` to work is the declaration of a few localization delegates in `MaterialApp()` or `CupertinoApp()` at the root of your widget tree; they are used by `intl` to automatically gather information about the device's locale. The package provides a series of facilities:

- The `simpleCurrency` method gets currency information according with the device's locale:

```

// The device in which this example is run is italian and thus the
// locale resolves to "it". Since 'localeOf()' is nullable, we
// provide a default value so that "en" becomes the fallback.
var localeFormat = NumberFormat.simpleCurrency(
    locale: Localizations.localeOf(context)?.toLanguageTag() ?? "en",
);

debugPrint("${localeFormat.currencySymbol}"); // € (the 'euro' symbol)
debugPrint("${localeFormat.currencyName}"); // EUR

```

If you want to get data about a specific currency just hard-code the language tag with, for example, `simpleCurrency("en_US")` which will return `"$"` or `"USD"`.

- Thanks to `DateFormat` there are many ways to format a `DateTime` object according with the current locale. Visit the official documentation ² to get a detailed explanation of the various possible combinations

```

var today = DateTime.now();

// Using the default locale obtained by intl ("en")
final d1 = DateFormat("yyyy-MM-dd - kk:mm").format(today);
// d1 = "2020-05-31 - 10:58"
final d2 = DateFormat("EEE, M/d/yyyy").format(today);

```

²<https://pub.dev/documentation/intl/latest/index.html>

```
// d2 = "Sun, 5/31/2020"  
  
// Using a given locale which overrides the default one  
final d2 = DateFormat("EEE, M/d/yyyy", "it").format(today);  
// d2 = "dom, 5/31/2020"
```

While "Sun" stands for "Sunday", in the second example "dom" stands for "domenica" which is the italian translation of sunday.

- A very useful feature of this package is the automatic string translation for material and cupertino widgets. For example if you had a search button in your app, rather than hard coding the textual value...

```
RaisedButton(  
    child: const Text("Search"), // <-- hard coded string  
    onPressed: () {}  
)
```

... you can use `intl` which automatically translates the strings according with the device's locale:

```
RaisedButton(  
    // "Search" for the English/American locale  
    // "Cerca" for the Italian locale  
    // "Rechercher" for the French locale  
    // and so on...  
    child: Text(MaterialLocalizations.of(context).searchFieldLabel),  
    onPressed: () {}  
)
```

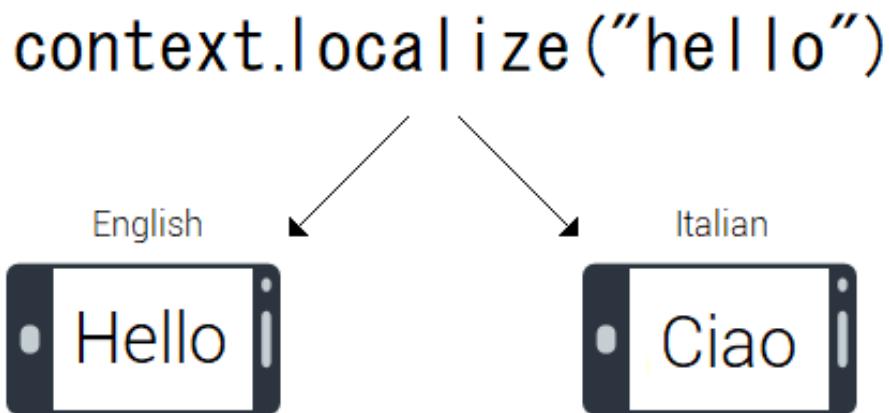
The `CupertinoLocalizations` class works in the same exact way but it's meant to be used with cupertino components.

When localizing an app, it's also fundamental approaching to **internationalization** which is the process of translating strings into different languages according with the current locale. You can do it by using `MaterialLocalizations` and `CupertinoLocalizations` but they're limited to translate a series or pre-built widgets.

💡 Localization (L10n) and internationalization (i18n) are **not** the same thing, even if sometimes they're erroneously used as if they were synonyms. Internationalization, date time formatting, currency formatting and much more more are all localization

techniques.

What you should do is being able to completely translate each string of your app, including those that don't belong to Flutter's UI libraries. In the following two sections we're going to show how to completely internationalize Flutter applications so that they automatically translate according with the device's locale.



When the app is opened, the user doesn't have to choose the preferred language because the device automatically recognizes it (thanks to [class Locale](#)). Before starting, keep in mind that:

- Both techniques you're going to see are good: don't think that, in terms of performance, one is better than the other. If you're not sure which one you should use, try both approaches to determine which is the one that suits more to your needs.
- Manual internationalization is faster to implement and probably easier to use but it just translates strings. **Basic** approach.
- Internationalization with *intl* requires various configurations to setup but it's more than plain string translation (subtle control for plurals, nouns, first/third person and so on). **Advanced** approach.

13.2 Manual internationalization

This kind of internationalization is *manual* because it doesn't rely on automated tools or external files; you have to manage the various translations by yourself. In terms of efficiency there are no problems at all but probably, in larger apps, maintenance might be tedious if there are a lot of languages to support.

```
lib/
  localization/
    routes/
      main.dart
      routes.dart
```

We recommend the creation of a folder called `localization` which is going to contain localization-specific files, such as the ones we're going to create now to internationalize the app.

13.2.1 AppLocalization

Create a new file called `app_localization.dart` which is going to host the `AppLocalization` class. It contains a database with the translations of each string for all locales that the app is going to support. We have decided to use English (`"en"`) as default language.

```
// app_localization.dart
class AppLocalization {
  final Locale locale;
  const AppLocalization(this.locale);

  static AppLocalization? of(BuildContext ctx) =>
    Localizations.of<AppLocalization>(ctx, AppLocalization);

  static Map<String, Map<String, String>> _db = {
    "en": {
      "hello": "Hello!",
      "hello_world": "Hello world",
    },
    "it": {
      "hello": "Ciao!",
      "hello_world": "Ciao mondo",
    },
  };
}
```

```
"es": {  
    "hello": "Hola!",  
    "hello_world": "Hola Mundo",  
}  
};  
}
```

The database is implemented with maps because when it comes to data retrieval they are very fast ($O(1)$ time complexity); making it **static** is a good idea because data has to be initialized only once. In this example, our app is going to translate strings for the following locales:

- **en** - English;
- **it** - Italian;
- **es** - Spanish

The static method `of()` takes care of extracting the proper locale according with the context. In fact, thanks to it you're able to do the following...

```
final locale = AppLocalization.of(context)?.locale.languageCode ?? "en";
```

... where `final locale` is a string containing the language code of the device's locale such as "`en`", "`it`" or "`es`". Having a `final Locale` instance variable is not required but it might be good to have if you wanted to implement some locale-specific features. Feel free to remove it if you won't need it.

- i** In the translations database (`_db`), the various locale keys must all match otherwise the lookup strategy won't work. You cannot use different keys like this:

```
"en": {  
    "hello": "Hello!",  
},  
"it": {  
    // NO, should be 'hello'  
    "ciao": "Ciao!",  
},
```

To keep the example simple we've decided to use strings. However, a better approach would be the usage of an `enum` for keys in order to avoid working with strings, which might introduce typing errors.

```
enum _LocKeys {
    hello,
    hello_world
}

class AppLocalization {
    static Map<String, Map<_LocKeys, String>> _db = {
        "en": {
            _LocKeys.hello: "Hello!",
        },
        "it": {
            _LocKeys.hello: "Ciao!",
        },
        "es": {
            _LocKeys.hello: "Hola!",
        },
    }
}
```

Less strings to maintain! Make sure to make `_LocKeys` private to hide access to it from the outside (hiding implementation details is always a good practice).

Still in the same file, create an extension method which allows calling a `localize` method directly on a `BuildContext` object, which is very convenient. The extension method can access `AppLocalization`'s private members because they live in the same file.

```
extension LocalizationExt on BuildContext {
    String localize(String value) {
        // Getting the device's locale, which can be for example
        // "en", "it", "es" or anything else
        final code = AppLocalization.of(this)?.locale.languageCode ?? "en";
        final database = AppLocalization._localizedValues;

        // Checks whether the current app locale is supported
        if (database.containsKey(code)) {
            return database[code]?[value] ?? "-";
        } else {
            // Default to English if the locale is not supported
        }
    }
}
```

```
        return database["en"]?[value] ?? "-";
    }
}
}

class AppLocalization {...}
```

In our example, if the locale were `"en"`, `"it"` or `"es"` the condition would evaluate to `true` because the database contains translation strings for those locales. If the device's locale were `"fr"` (French) for example, which is not supported...

```
return database["en"]?[value] ?? "-";
```

... the `if` statement would go to the second branch which is a fallback to the English locale. In other words, if your app doesn't support the device's locale, English is chosen as default. At this point you're able to retrieve internationalized strings like this:

```
Text(context.localize("hello"));
```

Very short and concise!

13.2.2 Localization delegate

Create another file called `localization/localization_delegate.dart` which is going to contain the localization delegate for our `class AppLocalization`. A delegate is a class that produces collections of localized values according with the current locale (which is the device's locale).

```
class AppLocalizationDelegate
extends LocalizationsDelegate<AppLocalization> {
    const AppLocalizationDelegate();

    @override
    bool isSupported(Locale locale) =>
        ["en", "it", "es"].contains(locale.languageCode);

    @override
    Future<AppLocalization> load(Locale locale) =>
        SynchronousFuture<AppLocalization>(
            AppLocalization(locale)
        );
}
```

```
    @override
    bool shouldReload(LocalizationsDelegate<AppLocalization> d) => false;
}
```

This is the standard pattern when creating localization delegates; the only part you should take care of is the first overridden method as it tells which locales your app supports.

1. `isSupported`: the returned boolean indicates whether the device's locale is supported by the application or not. Since our database has support for English, Italian and Spanish, the array contains respectively `["en", "it", "es"]`.
2. `load`: loads resources for the given locale and the object generated with this method can be referenced later by using `Localizations.of<T>()`. You have noticed the usage of a weird class here:

```
class SynchronousFuture<T> implements Future<T> { ... }
```

This is a `Future<T>` in which the callback defined inside `then` runs immediately (so it doesn't wait for something at all). This case is probably the only one in which it can be used; in general **avoid** working with a `SynchronousFuture<T>`.

3. `shouldReload`: returns `true` or `false` whether the resources for this delegate should be loaded again with `load` or not. It's almost never required so returning `false` by default is fine.

The last setup takes place in the `MaterialApp` or `CupertinoApp` at the root of your widget tree. There's the need to "install" the localization delegate we've just created.

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

class MyApp extends StatelessWidget {
  const MyApp();

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      initialRoute: RouteGenerator.homePage,
      onGenerateRoute: RouteGenerator.generateRoute,

      localizationsDelegates: [
        const AppLocalizationDelegate(),
        // Global localization delegates provide localized
```

```
// strings and other values for the Material and
// Cupertino libraries. They support 70+ locales!
GlobalMaterialLocalizations.delegate,
GlobalCupertinoLocalizations.delegate,
GlobalWidgetsLocalizations.delegate,
],
supportedLocales: const [
Locale.fromSubtags(languageCode: "en"),
Locale.fromSubtags(languageCode: "it"),
Locale.fromSubtags(languageCode: "es"),
],
);
}
}
```

You have to include `AppLocalizationDelegate` in the `localizationsDelegates` list along with the other common delegates so that the correct locale can be automatically inferred. Also be sure that the same type of locales are supported in both the delegate and the widget:

```
// Inside 'AppLocalizationDelegate'
bool isSupported(Locale locale) =>
["en", "it", "es"].contains(locale.languageCode);

// Inside 'MaterialApp'
supportedLocales: const [
Locale.fromSubtags(languageCode: "en"),
Locale.fromSubtags(languageCode: "it"),
Locale.fromSubtags(languageCode: "es"),
]
```

In both cases `"en"` (English), `"it"` (Italian) and `"es"` (Spanish) are supported locales and thus there's consistency (which is correct). Be sure to keep these two properties "in sync" to get a proper behavior.

13.2.3 Backward compatibility

Dart 2.6 and earlier version don't support extension method and thus you have to change your strategy. It's nothing difficult at all as there's only the need to move the logic inside

AppLocalization.

```
class AppLocalization {
    final Locale locale;
    const AppLocalization(this.locale);

    // Rather than being in the extension method, 'localize()' is directly
    // inside the AppLocalization class.
    String localize(String value) {
        final code = locale.languageCode;

        if (_db.containsKey(code)) {
            return _db[code][value] ?? "-";
        } else {
            return _db["en"][value] ?? "-";
        }
    }
}
```

It's basically the same thing we did in the extension method but now the code is a bit more verbose than earlier:

```
// Dart 2.6 - It also works up to Dart 2.9 because nnbd isn't supported yet
Text(AppLocalization.of(context).localize("hello"))
```

Of course, if you want, you can also use this approach in Dart 2.10 or higher versions. However, you'd have to deal with nullable types (and thus every time you'd have to perform null-checks or provide default values).

```
// Dart 2.10 - Null checks or default values are required because the
// returned value is of type 'String'?
Text(AppLocalization.of(context)?.localize("hello") ?? "-")

// Dart 2.10 - No checks to do because they're done inside the extension
Text(context.localize("hello"))
```

13.3 Internationalizing using intl

The internationalization process with `intl` is similar to the one we've just seen since they share the same identical setup but the translations database is managed by dedicated files rather than a simple `Map<K, V>`.

```
dependencies:  
  flutter:  
    sdk: flutter  
  flutter_localizations:  
    sdk: flutter  
  
  intl: ^0.16.1  
  intl_translation: ^0.17.10
```

Add the required dependencies and consider using the following template for the creation of a folder structure. It's very similar to the one in the previous section; the only additions are in `localization/` since we're going to deal with potentially a lot of files.

```
lib/  
  localization/  
    arb_files/  
    dart_files/  
  routes/  
  main.dart  
  routes.dart
```

13.3.1 AppLocalization

Create the usual `app_localization.dart` file which is going to host `AppLocalization`. It's the translation database but differently from the manual approach, rather than using a `Map` we're interacting with command line tools.

```
class AppLocalization {  
  final Locale locale;  
  const AppLocalization(this.locale);  
  
  static AppLocalization? of(BuildContext context) =>  
    Localizations.of<AppLocalization>(context, AppLocalization);
```

```
static Future<AppLocalization> load(Locale locale) async {
    final String localeName = Intl.canonicalizedLocale(
        locale.languageCode
    );

    // It produces an error now but it's fine; it will
    // disappear as soon as we use code generation to
    // create internationalization utilities
    await initializeMessages(localeName);

    // Setup intl to work with the device's locale
    Intl.defaultLocale = localeName;

    return AppLocalization(locale);
}

String get helloWorld => Intl.message(
    "Hello world!",
    name: "helloWorld",
);

String get hello => Intl.message(
    "Hello",
    name: "hello",
);
}
```

Strings that need to be internationalized and/or localized are defined as getters returning the result of `Intl.message`, which is made up of two basic (required) parameters:

1. the string itself, such as `"Hello world!"`;
2. the **exact** name of the getter which returns the string to be localized. To be clear, this is fine:

```
String get helloWorld => Intl.message(
    "Hello world!",
    name: "helloWorld",
);
```

But this is **not** fine because the `name` parameter doesn't exactly match the getter name:

```
String get helloWorld => Intl.message(  
    "Hello world!",  
    name: "hello_world", // NO, must be "helloWorld"  
) ;
```

Getters defined inside `AppLocalization` contain the "default" value of the string and in order to internationalize them, we need to use a code generator.

```
flutter pub run intl_translation:extract_to_arb  
--output-dir=lib/localization/arb_files  
lib/localization/app_localization.dart
```

Open the command line from your environment or, in Android Studio, just click the *Terminal* label at the bottom and type the above command. It will generate, in `localization/arb_files`, an `.arb` file which is used to manage translations of strings in various languages.

i ARB files are nothing more than JSON files whose keys start with "`@@`". They can be edited manually by you with a simple text editor but the more languages and strings you're going to support, the harder they become to handle.

- Localizely
- BabelEdit
- Crowdin

The above list is a series of localization services that help you (or your team) with the management of ARB files. We recommend you to look at them because manual management gets exponentially harder for each new language you decide to support.

The tool generates `intl_messages.arb`: it's a JSON format map with one entry for each `Intl.message()` function defined in `AppLocalization`. It's a template you can use to create one ARB file per locale:

```
localization/  
  arb_files/  
    intl_messages.arb  
    intl_en.arb  
    intl_it.arb
```

```
  intl_es.arb  
  dart_files/
```

Each file contains the various translations of the strings you defined in `AppLocalization`. You should not touch the keys of the JSON map otherwise the automated tool will fail; just change the string values. Here's an example:

```
// A piece of 'intl_en.arb'  
"hello": "Hello",  
"@hello": { ... }  
  
// A piece of 'intl_it.arb'  
"hello": "Ciao",  
"@hello": { ... }  
  
// A piece of 'intl_es.arb'  
"hello": "Hola",  
"@hello": { ... }
```

As you can see, the "translations database" is made up of the various ARB files. In order to use them in your Flutter app, run this command which converts those JSON maps into .dart files.

```
flutter pub run intl_translation:generate_from_arb  
--output-dir=lib/localization/dart_files  
--no-use-deferred-loading lib/localization/app_localization.dart  
lib/localization/arb_files/intl_en.arb  
lib/localization/arb_files/intl_it.arb  
lib/localization/arb_files/intl_es.arb
```

The tool has generated a series of files inside your `dart_files/` which are the ones Flutter is going to look for when it comes to internationalizing strings. Notice that code generation created the `initializeMessages` method inside `AppLocalization` so that you can now successfully compile.

```
static Future<AppLocalization> load(Locale locale) async {  
  ...  
  
  // No errors anymore  
  await initializeMessages(localeName);  
  ...
```

```
}
```

13.3.2 Localization delegate

The creation and setup of the delegate "links" your custom localization logic (`AppLocalization`) to the app itself (`MaterialApp` or `CupertinoApp`). It's very similar to the manual approach:

```
class AppLocalizationDelegate
    extends LocalizationsDelegate<AppLocalization> {
    const AppLocalizationDelegate();

    @override
    bool isSupported(Locale locale)
        => ["en", "it", "es"].contains(locale.languageCode);

    @override
    Future<AppLocalization> load(Locale locale)
        => AppLocalization.load(locale)

    @override
    bool shouldReload(LocalizationsDelegate<AppLocalization> d) => false;
}
```

And again, remember to setup the delegate and the supported locales list in your app's root widget:

```
return MaterialApp(
    localizationsDelegates: [
        const AppLocalizationDelegate(),
        GlobalMaterialLocalizations.delegate,
        GlobalCupertinoLocalizations.delegate,
        GlobalWidgetsLocalizations.delegate,
    ],
    supportedLocales: const [
        Locale.fromSubtags(languageCode: "en"),
        Locale.fromSubtags(languageCode: "it"),
        Locale.fromSubtags(languageCode: "es"),
    ],
);
```

Strings can be obtained in the same way we have seen in the previous section. The difference is that, instead of passing a `String` or an `enum`, you're using a getter.

```
// Good but there are null-checks to do
Text(AppLocalization.of(context)?.helloWorld ?? "-");
```

As we had already suggested in the previous section, you should prefer the usage of extension methods. You'd have to use `context.localize.helloWorld` which is shorter, more readable and doesn't involve nullable types.

13.3.3 Plurals and data interpolations

The `intl` internationalization strategy is pretty handy when it comes to advanced strings localization. For example consider having an application that, according with today's date, has to show a countdown label:

```
final days = getDaysLeft();
final value = "$days days remaining.;"
```

It's fine but there's a problem with the plural when the days left counter is 1 or 0. The output might be one of the following:

- "2 days remaining."
- "1 days remaining." (the noun should be singular)
- "0 days remaining." (doesn't make much sense)

We should fix the code in order to get proper singular/plural nouns and, in case of the counter being zero, show a different message. It's doable with a bit of boilerplate code but strings are still **not** internationalized:

```
final days = getDaysLeft();
var value = "$days days remaining.;

if (days == 1) {
    value = "$days day remaining.";
}

if (days == 0) {
    value = "No days left.";
}
```

Now it's better but strings are not internationalized and thus we should work on localization, which is even more boilerplate code to add. `intl` offers a very elegant solution to deal with plurals and arguments:

```
class AppLocalization {  
    final Locale locale;  
    const AppLocalization(this.locale);  
    // other code...  
  
    String remainingDays(int days) =>  
        Intl.plural(days,  
            zero: "No days remaining",  
            one: "One day left",  
            other: "$days days remaining",  
            name: "remainingDays",  
            args: [days],  
        );  
}
```

According with the number of days left, `intl` shows a different string and it can also interpolate values. We've used a function because getters cannot declare input parameters:

- `zero`: string returned when `days` is 0;
- `one`: string returned when `days` is 1;
- `other`: string returned when `days` is ≥ 2 ;

As usual, launch the command line tools and translate the strings into the generated ARB files. Once you've got the `dart` files with the localization, you'll be able to call this from your widgets:

```
final days = getDaysLeft();  
final value = AppLocalization.of(context)?.remainingDays(days) ?? 0;  
  
/*  
 * days = 0 / "No days remaining"  
 * days = 1 / "One day left"  
 * days = 2 / "2 days remaining"  
 * days = 3 / "3 days remaining" ...  
 */
```

With a single line of code you're able to not only localize a string, but also interpolate data (a number in this case), handling plurals (*days* or *day*) and showing variants for the same string. Note that `args` parameter must contain, in order, the arguments that are being interpolated in the strings.

13.4 Considerations

Both internationalization ways are fine but one might be better than the other under certain circumstances. They can also be implemented with similar steps; the difference lies on the data source from which they extract the various strings translations:

Manual

1. Create `class AppLocalization` and also an extension method, which is very convenient.
2. Fill the database (it's a `Map<K, V>`).
3. Create the localization delegate.

Using intl

1. Create `class AppLocalization` and also an extension method, which is very convenient.
2. Create `.arb` files by copy/pasting the generated template.
3. Translate strings and convert from `.arb` to `.dart`.
4. Create the localization delegate.

When the application starts, the device's locale is recognized and automatically passed to the delegate so that the user isn't asked to select its language. Only for iOS, there's an extra step to do in order to get localization working properly:

- open with XCode the project under the `ios/` folder;
- under the Runner folder, open the `Info.plist` file;
- select **Information Property List**, click *Add item* and then choose *Localization*
- add one entry per locale, such as English, Italian, Spanish etc. Keep consistency with the list you've declare in `supportedLocales`

While both localization approaches are perfectly fine, there are cases in which one suits better than the other. Take the following considerations just as general guidelines:

- **Manual**

- Good when there are a few languages to support, like two or three, and a relatively small amount of terms/sentences to localize. To be more precise, this approach is good as long as you're able to manually maintain the strings database in a reasonable amount of time.
- Very easy setup.
- No generation tools or external files involved.

- **intl**

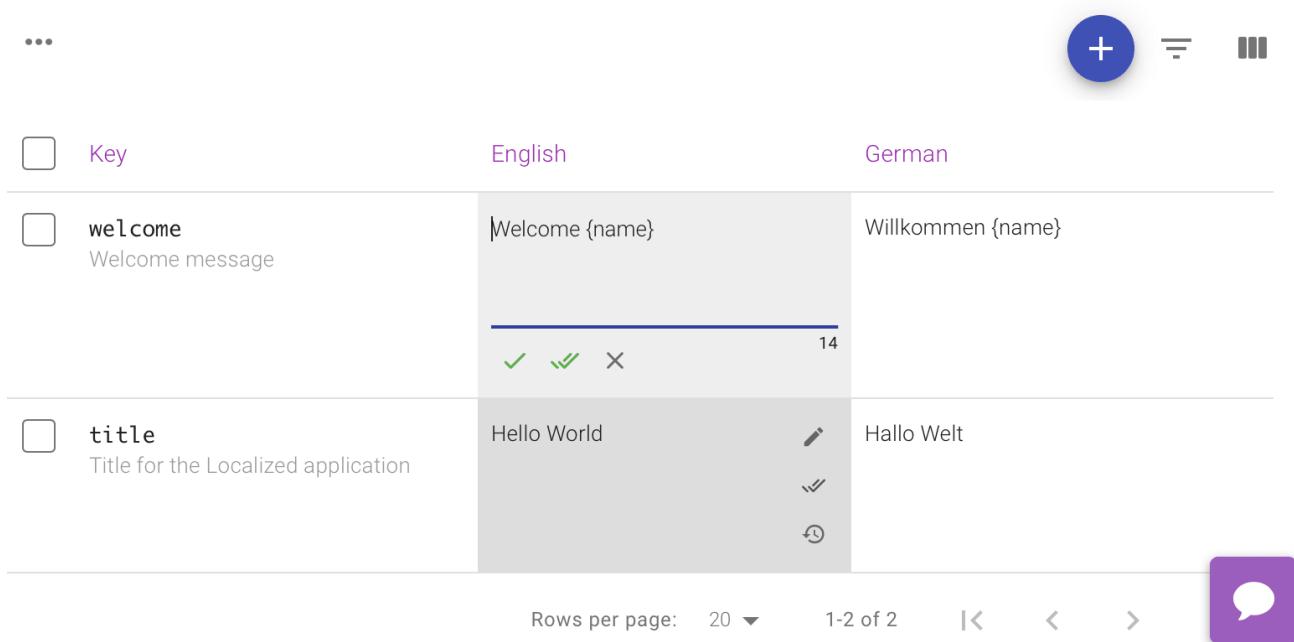
- Good when there is a wide range of languages to support, like four or more. It supports plurals management, strings interpolation and also other localization utilities such as date format and currencies.
- Easy setup but an efficient ARB files management requires an external tool.
- Relies on code generation tools and external ARB files.

We can say that `intl` is very powerful and scalable but such strength isn't always required for smaller applications. If you're working on a team, using ARB files is essential as they can be shared and managed by a GitHub repository or external tools.



LocalizeLy is a service that allows you to easily work with strings localization using ARB files. It's especially powerful when you're working on a team and/or your apps have to support many languages. They offer a free plan for open source projects publicly hosted on GitHub; take a look at their website³ for more info.

³<https://localize.ly>



The screenshot shows the Localizeley website interface for managing ARB files. At the top right are three circular buttons: a blue one with a plus sign, a white one with a minus sign, and a grey one with three horizontal bars. Below these are two tabs: "Key" and "String". The "Key" tab is selected, showing a table with two rows.

Key	English	German
welcome Welcome message	Welcome {name}	Willkommen {name}
title Title for the Localized application	Hello World	Hallo Welt

Each row has a checkbox at the top left. The first row also has a "Welcome message" subtitle below it. The English column contains placeholder text {name}. The German column contains the translated text. Each row has a small toolbar at the bottom right with a green checkmark, a green checkmark with a dot, and a red X. The German row also has a blue pencil icon, a blue double arrow icon, and a blue circular refresh icon. At the bottom of the table are buttons for "Rows per page: 20" (with a dropdown arrow), "1-2 of 2", and navigation arrows. To the right is a purple speech bubble icon.

By installing the Flutter Intl plugin for Android Studio or VS Code you get the possibility to integrate your ARB files with *Localizeley*. They can be uploaded to the website and later downloaded with the newly localized strings so that you only need to compile them into .dart files.

14 | Animations

When using Flutter, the developer is able to control each single pixel of the UI so that he can fully customize the design of the app. Animations are no exception as the framework gives you the possibility to create something from scratch or using the built-in widgets. There are three types of "implementation difficulty" when it comes to making animations:

1. **Implicit animations:** it's a series of pre-built widgets integrated in the Flutter SDK and ready for you to use. They are very convenient: sometimes they don't even need a setup because default settings are just fine.
2. **Animation library:** inheritable classes or utility widgets that help you building custom animations. They give you more control on the animation itself and the possibility to manually improve performances.
3. **Custom animations:** animations made from scratch by using specific classes such as `Matrix4` (4D matrix) or `Transform`. You're creating "low level" animations using trigonometry by directly controlling the 3D space.

We are covering all of those animation techniques, which have been sorted from the easiest to the hardest. Flutter helps the developer even in the most complicated scenarios; nevertheless, building complex animations is not so straightforward as it requires some maths and experience.

14.1 Implicit animations

Thanks to the animation facilities shipped with the SDK you can add animated widgets and create visual effects in your UI with a minimal effort. Flutter offers a set of pre-built widgets, called *implicitly animated widgets*, that manage animations automatically: in most of the cases, you've almost nothing to do.

i Not every widgets have their animated counterpart but the Flutter team might implement them in the future. Generally, the animated version of a widget is called *AnimatedFoo* where *Foo* is the name of a "standard" UI component such as a `Container`.

The developer has almost nothing to do in order to animate a widget: it's just a matter of assigning one or two parameters. Look at this simple container whose dimensions are determined by the `counter` variable held by a provider:

```
// "CounterModel" is exposed by a ChangeNotifierProvider
// so the dimensions vary at any change of the variable.
Consumer<CounterModel>(
    builder: (_, counter, __) {
        return Container(
            height: counter,
            width: counter,
            color: Colors.lightBlue,
            alignment: Alignment.center,
        );
    }
)
```

When the value of `counter` changes the container resizes as well but with no animation (you see an "immediate" change). For a better user experience, you can animate the size change by simply renaming the widget and setting the duration.

```
Consumer<CounterModel>(
    builder: (_, counter, __) {
        return AnimatedContainer(
            height: counter,
            width: counter,
            color: Colors.lightBlue,
            alignment: Alignment.center,
            duration: const Duration(seconds: 2),
        );
    }
)
```

That's it, changes are automatically animated for you. We've just renamed a widget and the animation is ready. With a bit more of effort it's also possible, for example, changing the background

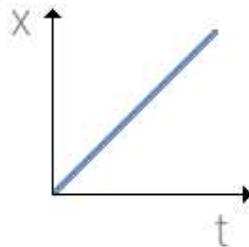
color or the Container and setting a different animation type.

```
color: counter % 2 == 0 ? Colors.lightBlue : Colors.lime,  
alignment: counter < 500 ? Alignment.center : Alignment.topCenter,  
curve: Curves.easeInOut,
```

The default curve is `Curves.linear` but thanks to the `curve` parameter you can change the animation type. Check the official documentation ¹ to see a complete list of curves that Flutter currently supports.

- i** A *curve* is a mathematical function that expresses how values change over the time. For example, a `linear` curve is a type of animation which executes at constant speed for all of its duration.

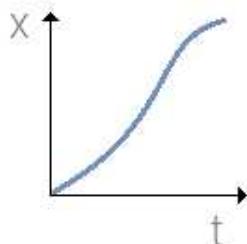
Linear curve



As time goes by (horizontal axis) the values changed by the animation (vertical axis) follow a linear trend. That's why in the first animated example the size "linearly" changes.

¹<https://api.flutter.dev/flutter/animation/Curves-class.html>

Ease in-out curve



The animation described by a `easeInOut` starts slowly, speeds up and then finishes slowly again. The documentation has a very nice page with a detailed description for each curve and short video that shows how the animation looks like.

We're now going to showcase the entire list of implicitly-animated widgets at the time of writing this book. In any case, you'll always have the possibility to set the duration and the type of curve.

- `AnimatedAlign`. Animated version of the `Align` widget that transitions the position of a child widget from one area of the screen to another. Here's an example:

```
// Somewhere 'value' has been declared as 'int'  
AnimatedAlign(  
    alignment: value % 2 == 0 ? Alignment.center :  
        Alignment.topCenter,  
    duration: const Duration(seconds: 1),  
)
```

- `AnimatedOpacity`. Animated version of the `Opacity` widget that transitions the opacity of a child widget from one value to another. The `opacity` parameter can range from 0 to 1 where 0 makes the widget invisible.

```
// Somewhere 'value' has been declared as 'int'  
AnimatedOpacity (  
    opacity: value,  
    duration: const Duration(seconds: 2),  
    curve: Curves.easeInOut,
```

```
)
```

In terms of performances, this widget is relatively expensive (as it requires an intermediate buffer) so use it carefully (or avoid using it at all).

- `AnimatedDefaultTextStyle`. Animated version of the `DefaultTextStyle` widget that transitions the style of a `Text` widget from the current setup to a different one.

```
// Somewhere 'color' and 'fontSize' have been declared and they
// can change over the time
AnimatedDefaultTextStyle(
    duration: const Duration(milliseconds: 500),
    curve: Curves.elasticInOut,
    style: TextStyle(
        fontSize: fontSize,
        color: color,
        fontWeight: FontWeight.bold,
    ),
    child: const Text("Pasta"),
)
```

The animation is applied to `any` children whose type is `Text` so the same effect works on different instances. For example, you could display multiple lines of text in a `Column` and every children will be equally animated:

```
AnimatedDefaultTextStyle(
    duration: const Duration(seconds: 2),
    style: TextStyle(...),
    child: Column(
        children: [
            const Text("I'm animated"),
            const Text("I'm animated as well"),
        ]
    ),
)
```

The unchanged properties such as `textAlign`, `softWrap`, `textOverflow`, and `maxLines` are not animated.

- `AnimatedPadding`. Animated version of the `Padding` widget that transitions the padding between widgets from one value to another.

```
AnimatedPadding(  
    // 'value' has been declared somewhere and it may change  
    duration: const Duration(minutes: 1),  
    padding: EdgeInsets.only(top: value),  
    child: const Text("Padding!"),  
)
```

- `AnimatedPhysicalModel`. Animated version of the `PhysicalModel` widget that transitions the elevation, the color and the border radius of the widget. The shape is not animated.

```
// Somewhere 'value' has been declared as 'int'  
AnimatedPhysicalModel (  
    duration: const Duration(seconds: 2),  
    borderRadius: BorderRadius.circular(value),  
    elevation: value,  
    shape: BoxShape.rectangle,  
    color: Colors.lightBlue,  
    shadowColor: Colors.black45,  
    child: const Text("Hello"),  
) ,
```

This widget requires you to define, at least, all those parameters. If you want to also add a transition effect for the background color of the shape, you have to set `animateColor: true`.

```
// Somewhere 'canChange' has been declared as 'boolean'  
AnimatedPhysicalModel (  
    // code...  
    animateColor: true,  
    color: canChange ? Colors.lime : Colors.black,  
    child: const Something(),  
) ,
```

- `AnimatedPositioned`. Animated version of the `Positioned` widget that transitions the position from a point of the screen to another. It works only when inside a `Stack`.

```
// Somewhere 'value' has been declared as 'int'  
Stack(  
    children: [  
        AnimatedPositioned(  
            duration: const Duration(seconds: 1),  
            top: value,
```

```
        right: 30,  
        child: const Text("Hello"),  
    ),  
],  
) ,
```

Pay attention: with this widget, the size of the child would change at the end of the animation. If the child must not change its dimensions, consider using a `SlideTransition` we're going to cover later.

- `AnimatedCrossFade`. This widget cross-fades between two children and animates while the transition (with the eventual resize) is happening. For example, when an item goes out of stock you want to replace the purchase button with some text saying "**SOLD OUT!**".

```
// Somewhere 'canBuy' has been declared as 'bool'  
AnimatedCrossFade (   
    duration: const Duration(seconds: 1),  
    crossFadeState: canBuy ? CrossFadeState.showFirst :  
                           CrossFadeState.showSecond,  
    firstChild: RaisedButton(  
        child: const Text("Buy item"),  
        onPressed: () {...}  
    ),  
    secondChild: const Text("SOLD OUT!"),  
),
```

`firstChild` appears if `CrossFadeState` is `showFirst` otherwise `secondChild` is shown. Changing this value animates the transition between the two widgets.

- **AnimatedSize**. Animated version of the `Size` widget that transitions the size whenever its child's size changes. In other words, this widget automatically animates the resizing when its child changes dimensions.

```
// Somewhere 'value' has been declared as 'int'  
AnimatedSize(  
    duration: const Duration(seconds: 1),  
    curve: Curves.easeIn,  
    child: Container(  
        width: 100,  
        height: value  
)
```

) ,

Whenever the `Container` changes the height, the `AnimatedSize` widget will automatically change its height as well.

- `AnimatedIcon`. Animated version of the `IconData` widget which animates the transition of an icon from a shape to another.

```
AnimatedIcon(  
    icon: AnimatedIcons.menu_arrow,  
    progress: controller,  
)
```

In the next section you'll understand how `AnimatedController controller` works. The `menu_arrow` constant is basically an hamburger icon which can change into a back arrow with a nice rotation:



Visit the official documentation for the `AnimatedIcon` class ² to see a complete list of all the available animated icons.

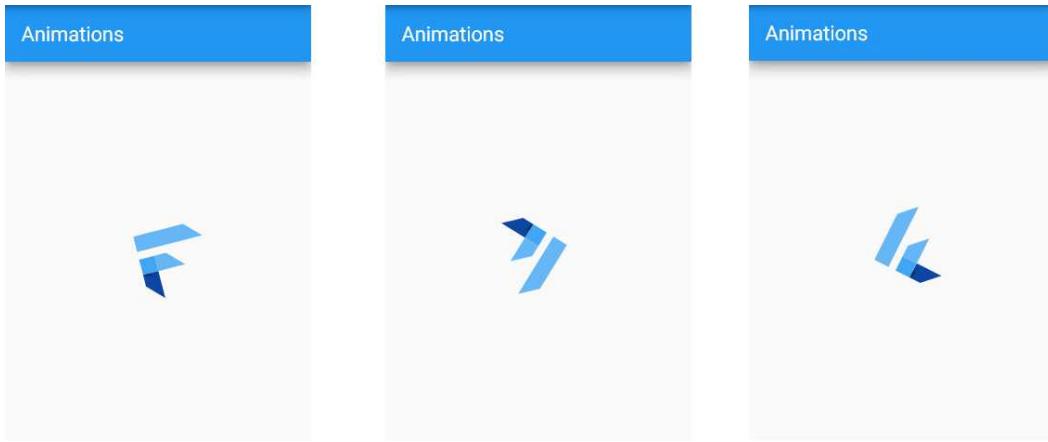
14.2 The animation library

When you're looking for a more advanced type of animation, very likely implicit animations won't be enough anymore. They're customizable... but not so much. In this case you need to create the animation by yourself working with controllers and values. From now on, you should be familiar with some basic trigonometry.

14.2.1 AnimatedWidget

The creation of an animation involves generally two steps. We're going to show them by creating a simple app which has a rotating Flutter logo at the center of the screen. Here's some captures of the running animation to get the idea:

²<https://api.flutter.dev/flutter/material/AnimatedIcons-class.html>



We're going to use a widget called `FlutterLogo()` which simply shows a high quality Flutter logo. It's like a vectorial image and thus it can be resized without losing quality. Let's create the rotating animation:

1. Our goal is taking a `class FlutterLogo()` and animating it with an endless rotation. An `AnimatedWidget` has the purpose of taking any widget and providing facilities to animate it so we're going to use it:

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

// 'AnimatedWidget' can animate any widget
class RotatingLogo extends AnimatedWidget {
    final AnimationController _controller;
    const RotatingLogo({
        required AnimationController controller
    }) : _controller = controller, super(listenable: controller);

    // (a).
    static const _fullRotation = 2 * math.pi;

    @override
    Widget build(BuildContext context) {
        // (b).
        return Transform.rotate(
            // (c).
            angle: _controller.value * _fullRotation,
            child: const FlutterLogo(
```

```

        size: 80,
    ),
);
}
}
}

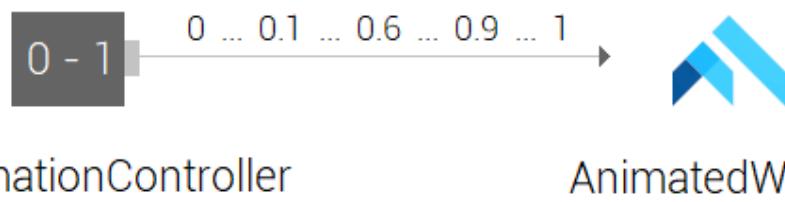
```

When working with animations, `const` constructors are even more important because they tell Flutter to **not** rebuild certain widgets every frame. The framework has a very high refresh rate (up to 120 fps) so unneeded rebuilds might waste resources.

- (a) In order to achieve a full rotation, values for `angle` must range from 0 to 360 degrees. Since Flutter works with radians, there's the need to make a conversion.
- (b) The `Transform.rotate()` named constructor returns a rotated widget with an angle of given radians. That's why we had to use the `math.pi` constant.
- (c) The `listenable` object is passed via constructor following the good dependency injection practices. You'll see soon that `_controller.value` is constantly updated every frame and it's what gives "life" to the widget making it rotate.

`Transform.rotate()` itself doesn't give motion to the widget, it just sets a specific inclination and nothing more. The rotation animation is obtained because `_controller.value` constantly changes and the widget is rebuilt so that, at each frame, you see the widget with a different angle.

2. Now it's time to put the animation on the widget tree and give it the data source for motion. The `AnimationController` class generates data with a specific frequency and a given duration.



By default, a controller emits values from 0 to 1 with a linear interval. Our animated widget receives them and gets rebuilt. The `_controller` variable holds a reference to the animation controller and `value` represents values that periodically arrive.

Chapter 14. Animations

```
angle: _controller.value * _fullRotation,
```

The product between the emitted values and the angle (in radians) represents the rotation of the logo at a certain frame. Here's how to link a controller to the widget:

```
class FLSpinner extends StatefulWidget {
  const FLSpinner();

  @override
  _FLSpinnerState createState() => _FLSpinnerState();
}

class _FLSpinnerState extends State<FLSpinner>
  with TickerProviderStateMixin {
  // (a).
  late final AnimationController _controller;

  @override
  void initState() {
    super.initState();
    _controller = AnimationController(
      duration: const Duration(seconds: 3),
      // (b).
      vsync: this,
    )..repeat(); // (c).
  }

  @override
  void dispose() {
    _controller.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return RotatingLogo(controller: _controller);
  }
}
```

A `StatefulWidget` is essential here because there's the need to dispose the controller when the widget is "destroyed"; in this way the animation won't waste resources. Always remember to initialize and clean the animation instances via `initState` and `dispose`.

- (a) The `AnimationController` class³ continuously generates a series of values, for a given duration, whenever the hardware is ready for a new frame. By default it produces values from 0.0 to 1.0 for the entire duration.
- (b) The `AnimationController` class generally generates 60 new values **per second** but it may vary according with the refresh rate of the device. Having multiple animations running even if they aren't visible might cause a performance drop.
- (c) The `repeat()` method executes the animation forever. In our example, it means that there will be an infinite chain of animations lasting for the given duration (3 seconds). The class gives you the flexibility to choose between different behaviors:
 - `forward`: starts the animation forward, in the sense that values vary from start to the end. ($0 > 1$)
 - `reverse`: starts the animation reverse, in the sense that values vary from the end to the start. ($1 > 0$)
 - `repeat`: starts the animation in forward direction and then restarts it (in the same direction). ($0 > 1, 0 > 1, 0 > 1\dots$)

If you used `forward()` instead of `repeat()` you'd have seen the logo spinning around only once and not forever.

```
AnimationController(  
  vsync: this,  
)
```

The `vsync` parameter of the controller prevents offscreen animations from consuming unnecessary resources⁴. Initializing it with `this` gives the stateful widget the possibility of handling the animation resources automatically.

```
... with TickerProviderStateMixin {}
```

There's the need to add this mixin to your class because it allows a `StatefulWidget` to be assignable to a `vsync`, which is an instance of a `TickerProvider` (a class internally used by animations).

³<https://api.flutter.dev/flutter/animation/AnimationController-class.html>

⁴<https://flutter.dev/docs/development/ui/animations/tutorial>

The setup for the animation is now ready. We've worked on two parts: the widget being animated, `RotatingLogo`, and the surrounding widget, `FLSpinner`, which is responsible for periodically generating new values with a controller.

```
MaterialApp(  
    home: Scaffold(  
        appBar: AppBar(  
            title: const Text("Animations"),  
        ),  
        body: const Center(  
            child: FLSpinner(),  
        ),  
    ),  
);
```

The `FLSpinner` widget internally has a controller which generates, for the given duration, a series of values very frequently (generally 60 per second). These values are passed each time to the `Transform` object which is being repainted and thus the animation runs smoothly on the UI.

14.2.2 AnimationBuilder

The `AnimationBuilder` class is simply an `AnimatedWidget` which automatically optimizes the animations to avoid unnecessary rebuilds. Understanding how animations work is very important because you don't want them to slow down your app due to the lack of optimization in your Dart code.

```
class RotatingLogo extends AnimatedWidget {  
    // Constructor, getter and other code omitted for brevity  
  
    @override  
    Widget build(BuildContext context) {  
        return Transform.rotate(  
            angle: _controller.value * _fullRotation,  
            child: const FlutterLogo(  
                size: 80,  
            ),  
        );  
    }  
}
```

As you already know, thanks to a constant constructor the `FlutterLogo` widget is built only once. What gets rebuilt every 1/60 of second (potentially 1/120) is only the `Transform` widget because it has to actually make the animation.

```
// 'Container' has no constant constructor so it's not cached
Transform.rotate(
  angle: _controller.value * _fullRotation,
  child: Container(...),
),
```

Pretend you had the same situation but rather than having a simple `FlutterLogo` to rotate there would be a `Container` with many non-constant children. It happens quite often when working with rows and columns as well because they have no `const` constructor.

- ❶ If children of `Container` were not constant as well the situation might be even worse. The framework could potentially rebuild every 1/60 second a big part of a subtree: it might not be efficient at all.

In such cases, where you cannot have a `const` constructor, there are two possible solutions that should really be taken into account:

1. Manually cache the portion of the tree that doesn't depend on the animation so that the framework won't rebuild it every time.

```
late final Widget _cachedTree = _cachedTree();

@Override
Widget build(BuildContext context) {
  return Transform.rotate(
    angle: _controller.value * _fullRotation,
    child: _cachedTree,
  );
}

Widget _cachedTree() {
  return Container(
    child: Row(
      children: [
```

```

        WidgetNonConst1(),
        WidgetNonConst2(),
    ]
)
);
}

```

This is the equivalent of using a constant constructor; the initialization of the `final` variable is executed only once so subsequent rebuilds will take effect only on `Transform`.

2. Make the code more readable with less boilerplate using `AnimatedBuilder`, which internally uses an `AnimatedWidget` and does the caching for you.

```

@Override
Widget build(BuildContext context) {
    return AnimatedBuilder(
        animation: someController,
        builder: (context, child) {
            return Transform.rotate(
                angle: _controller.value * _fullRotation,
                child: child,
            );
        }
        child: Container(...),
    );
}

```

The `child` parameter holds the part of the tree being cached (so that it gets built only **once**). The `builder` part instead contains the animation logic and the `child` parameter is a reference to the cached subtree.

Which widget should you use between `AnimatedBuilder` and `AnimatedWidget`? The one that you like more because they both do the same thing! Look at how Flutter internally implements `class AnimatedBuilder`:

```

class AnimatedBuilder extends AnimatedWidget {
    final TransitionBuilder builder;
    final Widget child;

    const AnimatedBuilder({
        Key? key,

```

```
        required Listenable animation,
        required this.builder,
        this.child,
    ) : super(key: key, listenable: animation);

    @override
    Widget build(BuildContext context) {
        return builder(context, child);
    }
}
```

This is the **exact same** thing we've done in the first option of the above list: the `child` parameter is stored in a `final` instance so that it doesn't get rebuilt every frame. It's doing "manual caching". This is a very important optimization you have to keep in mind when creating custom animations.

- ➊ Don't ask yourself which widget is the best because they really do the same thing, use the one that you like more. Maybe `AnimatedBuilder` could be a bit better because it leads to less boilerplate and increases the readability of the code.

14.2.2.1 Curves

Any animation is linear by default, meaning that values are emitted at a constant speed like it happens with the rotating logo that we've created. It's of course possible to alter this behavior and make something more interesting, such as a "bouncing" effect:

```
late final AnimationController _controller;
late final CurvedAnimation _curved;

@Override
void initState() {
    super.initState();
    _controller = AnimationController(
        duration: const Duration(seconds: 2),
        vsync: this,
    )..repeat();
    _curved = CurvedAnimation(
        parent: _controller,
```

Chapter 14. Animations

```

        curve: Curves.bounceIn,
    );
}

Widget build(BuildContext context) {
    return RotatingLogo(controller: _curved);
}

```

Instead of rotating at a constant speed, the logo will have a bouncing animation executing at different speeds. Instances of `CurvedAnimations` don't need to be disposed; you can choose between a myriad of effects (* is a placeholder):

- `Curves.bounce*`: bouncing effects such as `bounceIn` or `bounceOut`;
- `Curves.easeIn*`: ease in effects such as `easeInCubic` or `easeInSine`;
- `Curves.easeOut*`: ease out effects such as `easeOutQuartic` or `easeOutExpo`;
- `Curves.elastic*`: elastic effects such as `elasticIn` or `elasticOut`;

If you want to have a quick video preview of how an animation type looks like visit the official documentation⁵ about the `Curve` class. Do you want to have a bouncing animation executed in the opposite direction?

```

late final AnimationController _controller;
late final CurvedAnimation _curved;
late final ReverseAnimation _reverse;

@Override
void initState() {
    super.initState();
    _controller = AnimationController(
        // code...
    )..repeat();
    _curved = CurvedAnimation(
        // code...
    );
    _reverse = ReverseAnimation(_curved);
}

```

⁵<https://api.flutter.dev/flutter/animation/Curves-class.html>

RotatingLogo(`controller: _reverse`) will now run the same bouncing animation but in the opposite direction. If you wanted the original spinning logo animation to rotate backwards, you'd simply have to do this:

```
@override  
void initState() {  
    super.initState();  
    _controller = AnimationController(  
        // code...  
    )..repeat();  
    _reverse = ReverseAnimation(_controller);  
}  
}
```

It repeats the animation forever but running backwards. You could have used `reverse()` instead of `repeat()` but then the animation would have ran only once and not forever.

```
_controller = AnimationController(  
    // code...  
)..reverse();
```

It does the same thing as `ReverseAnimation` but you cannot loop forever.

14.2.2.2 Tweens

By default an `AnimationController` produces a series of `double` values ranging from 0 to 1. Thanks to `Tween<T>` you can alter this behavior for the cases where something different from numbers in [0..1] is required.

```
late final AnimationController _controller;  
late final Animation<double> _tween;  
  
@override  
void initState() {  
    super.initState();  
    _controller = AnimationController(  
        duration: const Duration(seconds: 3),  
        vsync: this,  
    )..repeat();  
    _tween = Tween<double>(  
        begin: 0,  
        end: 2 * math.pi,
```

Chapter 14. Animations

```
    ).animate(_controller);
}
```

This tween alters the behavior of the controller which now ranges from 0 to 6.2831... and not anymore from 0 to 1. With this change, going back to the original spinning logo example, we could have changed the code to make it look like this:

```
// This isn't needed anymore because the controller already
// produces values from 0 to 2pi, so no conversions at all!
//
// static const _fullRotation = 2 * math.pi;
@Override
Widget build(BuildContext context) {
    return Transform.rotate(
        // Earlier it was '_controller.value * _fullRotation'
        angle: _controller.value,
        child: const FlutterLogo(
            size: 80,
        ),
    );
}
```

The values produced by the controller are already in the range [0, 2pi] so we can directly pass `_controller.value`. We had to make a conversion in the previous example because values were emitted in the default [0, 1] range. With `Tween<T>` you can create a wide range of animations that aren't strictly tied to numbers. You could for example animate the color transition of a button:

```
late final AnimationController _controller;
late final Animation<Color> _colorTween;

@Override
void initState() {
    _controller = AnimationController(...);
    _colorTween = ColorTween(
        begin: Colors.white,
        end: Colors.green
    ).animate(_controller);

    super.initState();
}
```

```
}

@Override
void dispose() {
    _controller.dispose();
    super.dispose();
}

void _buttonTapped() {
    if (_controller.status == AnimationStatus.completed) {
        _controller.forward();
    } else {
        _controller.reverse();
    }
}
```

Thanks to `ColorTween` the `color` property of any widget is animated from the starting value to the ending value. As you can see it's not difficult at all.

```
AnimatedBuilder(
    animation: _colorTween,
    builder: (context, child) {
        return RaisedButton(
            child: child,
            color: _colorTween.value,
            onPressed: _buttonTapped,
        );
    },
    child: const Text("Animate me!"),
),
```

Notice how we've efficiently cached the `Text` widget which doesn't require rebuilds as it's not part of the animation. When the button is tapped, the animation runs forward or backwards according to the completion status. There are many kinds of tweens:

- `BorderRadiusTween`,
- `IntTween`,
- `DecorationTween`,

- `ShapeBorderTween`,
- `SizeTween`,

And much more! Visit the official documentation ⁶ to see them all.

14.3 Custom animations

If you're looking for specific kinds of animations, such as 3D ones or simply motions that aren't directly supported by Flutter with a particular widget, it's time to do some maths.

- i** While we will try to be as clear as possible, you'll need some skills on matrices and trigonometry to get the most out of this section. Here we're working in the 3D space so math is really required.

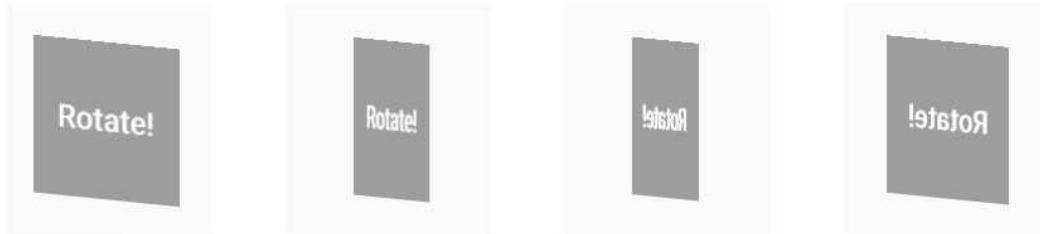
Marcin Szałek made a great talk at Flutter Europe about this topic, which is available on YouTube ⁷. Creating complex animations in Flutter is really a matter of being able to master the combination of three classes only:

- `AnimationController`: emits a certain range of values with a given frequency in order to animate a widget;
- `Transform`: scales, rotates, skews or alters the aspect of a widget using matrices;
- `Stack`: overlaps a series of children and allows custom positioning using the `Positioned` widget.

Let's try to create a container with a little skew that rotates on the Y-axis periodically. It looks like a 3D animation where the container rotates around its left side. Here are some frames of the animation:

⁶<https://api.flutter.dev/flutter/animation/Tween-class.html>

⁷Implementing complex UI with Flutter - Marcin Szałek



While it might seem quite hard to achieve, it's just a matter of calling two `Transform` methods and wrapping them in the usual `AnimatedBuilder` (which does the actual animation by emitting new values).

```
class _DemoAppState extends State<DemoApp>
  with SingleTickerProviderStateMixin {

  late final AnimationController _controller;

  @override
  void initState() {
    super.initState();
    _controller = AnimationController(
      vsync: this,
      duration: const Duration(seconds: 15),
    )..repeat();
  }

  @override
  void dispose() {
    _controller.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {...}
}
```

Nothing new up to here, it's just the "traditional" setup for animations where you declare a

Chapter 14. Animations

controller which is properly disposed inside `void dispose();`. The interesting part is the build method:

```
@override
Widget build(BuildContext context) {
    return AnimatedBuilder(
        animation: _controller,
        builder: (context, child) {
            return Transform(
                transform: Matrix4.skewY(0.1)
                    ..rotateY(_controller.value * 2 * math.pi),
                child: child,
            );
        },
        child: Container(
            decoration: const BoxDecoration(...),
            // 'child' is simply a 'Text' widget with
            // some styling declared via 'TextStyle'
            child: ...
        ),
    );
}
```

The usage of a `AnimatedBuilder` caches the container and the text, which are not animated. Let's analyze the `Transform` class:

- the `skewY(double value)` method keeps the widget sides parallel while inclinating the whole children in a certain direction, determined by `double value`.
- calling `rotateY(double radians)` rotates the children along the given axis, in our case Y, making it look like if it were a 3D motion. There are three kind of rotations:
 1. `rotateX(double radians)`
 2. `rotateY(double radians)`
 3. `rotateZ(double radians)`

Feeding the `rotateY` method with new values at 60 fps, shows a smooth 3D rotation. On a side note, it's possible making the animation going on forever but it'd only move forth and back rather than always in the forward direction.

```
// This is like calling 'forward()' forever
```

```
_controller.repeat();
// This is like calling 'forward()' and then 'reverse()' forever
_controller.repeat(reverse: true); // 2.
```

While in the first case the animation always goes in a single direction (`forward`), in the second one it goes back and forth. We're now showing another simple example in which a container, once tapped, scales and moves to another position of the screen with a sliding animation.



Scaling widgets and moving them is actually very easy because `Transform` exposes both `scale()` and `translate()`. We're able to decide the direction in which the animation has to run thanks to the `AnimationStatus` enum.

```
// 'onPressed' callback for the button
void _moveMe() {
    if (_controller.status == AnimationStatus.completed)
        _controller.reverse();
    else
        _controller.forward();
}
```

On the first tap the animation goes forward while on the second one the widget "restores" to its

Chapter 14. Animations

original position and size. In order to easily move on the screen without interfering with other widgets, the combination of `Stack` and `Positioned` is the perfect choice.

```
AnimatedBuilder(
  animation: _controller,
  builder: (context, child) {
    final scale = 1 - (_controller.value * 0.5);      // scaling logic
    final newPos =  20 + (_controller.value * 400); // position control

    return Stack(
      children: [
        Transform(
          transform: Matrix4.identity()
            ..scale(scale)
            ..translate(newPos, newPos),
          child: child,
        ),
        const Text("Hello"),
      ]
    );
  },
  child: RaisedButton(
    child: Text("Move me"),
    onPressed: _moveMe
  ),
);
```

The identity matrix returned by `identity()` is the "starting point" when scaling or translating widgets. Very easily, `Matrix4` is a 4x4 matrix which is used to represent the coordinates of a 3D space and apply calculations on them. The screen has two dimensions (width and height) but thanks to matrices we're able to work in three dimensions (width, height and depth) even if the available plan is "flat".

1. `final scale` is used to determine how much the child is going to resize (getting smaller in `forward()` or bigger in `reverse()`);
2. `final newPos` moves the child from a position on the screen to another. The `translate()` method can take three parameters indicating the x, y and z coordinates of the widget. By changing them, you're able to move the child;

3. We recommend using a `Stack` so that the moving widget doesn't cause problems to the position of other widgets. Since a `Stack` is a series of overlapped "levels", widgets can move without interfering with others.

You could make countless different animations since the only limit is your creativity. The point is that you really just need a controller, a `Stack` and the `Transform` widget along with its matrices and calculations. Don't be afraid to get lost in the "animation" world; just by using those three classes together you possess a great power!

14.4 Good practices

There's been a lot of information up to here about how you can animate widgets in Flutter and a recap might be very useful. Our suggestions come from both the documentation and the personal experience we've acquired: mastering animations is very doable but a lot of practice is required!

- When it comes to implementing an animation, the first thing we recommend is trying to see if Flutter has an *implicit animation* that might fit for the use case. They are pre-made animations offered by the framework which can be implemented with a minimal effort (sometimes you don't even need to setup any parameter):
 1. `AnimatedAlign`
 2. `AnimatedOpacity`
 3. `AnimatedPadding`
 4. ... and in general, many `AnimatedFoo` widgets.

They basically are the animated counterparts of standard Flutter widgets with a few additions required to setup the animation. They are very easy to use but actually not too much customizable if compared to an `AnimatedBuilder`. Using implicit animations have a some advantages:

- They keep consistency between the animated and non-animated version of the same widget. If you know how the `Padding` widget works `AnimatedPadding` is exactly identical (with the addition of motion).
- They're already in your Flutter installation ready to be used, why should you ignore them and create something from scratch? It would be a waste of time!
- If what you're looking for is not available as an *implicitly animated* widget, start creating

your own using an `AnimatedWidget` or an `AnimatedBuilder`. Both are the same thing but a builder requires a bit less code from your part; since developers are lazy, it might be the default choice.

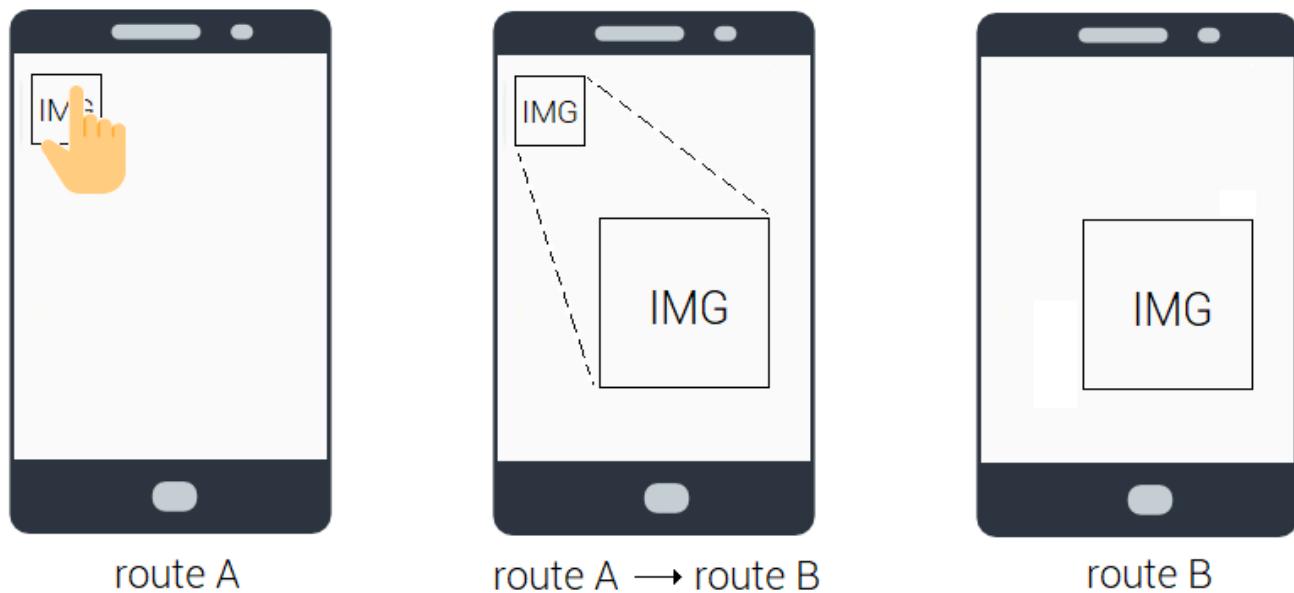
- Flutter tries to run at least at 60 frames per second, or anyway at the highest refresh speed supported by the hardware, so animations can be a problem if you don't optimize your code. For this reason, remember to:
 1. use `const` constructors as much as possible in order to avoid unnecessary rebuilds that might impact the performance of the app;
 2. if a constant constructor cannot be used, it's good practice caching the portion of the sub-tree either by storing it in a `final` variable or by using the `AnimatedBuilder` widget.
- The `AnimationController` class gives "life" to your widgets: it produces, by default, values from 0 to 1 which are used to animate certain properties. Of course you're not tied to numbers alone, in fact controllers can emit different type of values at different speed rates:
 1. with `class Tween<T>` you can change the range of values periodically emitted by a controller;
 2. use widgets like `CurvedAnimation` or `ReverseAnimation` to have even more power on animations.
- Very specific animations need to be created from scratch but there isn't the need to get lost in the vastness of the animation API provided by Flutter. It's just a matter of combining together three classes:
 1. `AnimationController`;
 2. `Stack`
 3. `Transform`

`Transform` is quite "low level" as it works with matrices and trigonometry but at the same time it's very powerful. It relies on values emitted from a controller so that the UI refreshes quickly and the animation runs smoothly.

14.4.1 Hero animations

A hero transition is a kind of animation which keeps the focus on a certain item while navigating from one route to another. If you tapped on an image in a certain route, there would be the

possibility to open a new route and animating the `Image` widget so that it always stays in the foreground of screen.



The image widget will nicely animate, remaining in the foreground, while the routes on the background are changing with another animation. In this way, the user's attention is kept on what he's tapped (the image, in our case). As always, implementing hero transitions in Flutter is quite easy:

```
class RouteA extends StatelessWidget {
    const RouteA();

    @override
    Widget build(BuildContext context) {
        ...
        Hero(
            tag: 'FlutterLogo'
            child: SvgPicture.asset("assets/flutter_logo.svg"),
        ),
        ...
    }
}
```

```
class RouteB extends StatelessWidget {
    const RouteB();

    @override
    Widget build(BuildContext context) {
        ...
        Hero(
            tag: 'FlutterLogo'
            child: SvgPicture.asset("assets/flutter_logo.svg"),
        ),
        ...
    }
}
```

On both routes, there must be a `Hero` widget with the same `tag` property so that Flutter knows which hero has to be animated. Since `Hero` doesn't provide tap callbacks, generally it's wrapped it in a `GestureDetector`:

```
GestureDetector(
    onTap: () => Navigator.of(context)?.pushNamed(
        RouteGenerator.routeB
    ),
    child: Hero(
        tag: 'FlutterLogo'
        child: SvgPicture.asset("assets/flutter_logo.svg"),
    ),
),
```

In this way, when tapping on the image, a hero animation will happen while navigating from a route to another. Be sure to check the official documentation ⁸ for more examples and videos about hero animations.

14.4.2 Custom route transitions

When moving from a page to another using `Navigator.of(context).pushNamed(...)`, Flutter applies a default material transition. Of course, you can easily change the animation type and it's even more straightforward with the `RouteGenerator` technique we've explained in chapter

⁸<https://flutter.dev/docs/development/ui/animations/hero-animations>

12.

```
class RouteGenerator {
  RouteGenerator._();

  static Route<dynamic> generateRoute(RouteSettings settings) {
    switch (settings.name) {
      case homePage:
        return MaterialPageRoute(
          builder: (_) => HomePage(),
        );
      case secondPage:
        return MaterialPageRoute(
          builder: (_) => SecondPage(),
        );
      default:
        throw Exception("Route not found");
    }
  }

  static const String homePage = '/';
  static const String secondPage = '/second';
}
```

As it is, page transitions are handled by `MaterialPageRoute` which uses the default material animation. Let's say we want to customize our app by running a sliding transition when moving to the `SecondPage` route. You just need to create a new type of `PageRouteBuilder`, preferably inside a new file called `routes_transitions.dart`.

```
class SlidingPageRoute extends PageRouteBuilder {
  final Widget navigateTo;
  SlidingPageRoute({
    required this.navigateTo
  }) : super(
    pageBuilder: (context, _, __) => navigateTo,
    transitionsBuilder: (context, animation, _, child) =>
      SlideTransition(
        position: Tween<Offset>()
```

Chapter 14. Animations

```

        begin: const Offset(-1, 0),
        end: Offset.zero,
    ).animate(animation),
    child: child,
),
);
}

```

Subclasses of `PageRouteBuilder` can define the `transitionBuilder` callback which is used to create a custom type of animation to be run while navigating to the route. The callback passed to `pageBuilder` is used to build the page contents but since it is created in the `build` method of the page itself, just return the child itself (for flexibility and reusability purposes, don't implement route-specific logic here).

```

case HomePage:
    // Default material animation transition
    return MaterialPageRoute(
        builder: (_) => const HomePage(),
    );
case SecondPage:
    // Custom sliding transition
    return SlidingPageRoute(
        navigateTo: const SecondPage(),
    );

```

Now you can replace `MaterialPageRoute` with `SlidingPageRoute` to apply the newly created transition when navigating to '`/second`'. However, if you find `SlideTransition` boring, just create another class and use another type of transition:

```

// 1.
ScaleTransition(
    scale: Tween<double>(
        begin: 0,
        end: 1,
    ).animate(animation),
    child: child,
)

// 2.
RotationTransition(

```

```
turns: Tween<double>(
    begin: 0,
    end: 1,
).animate(animation),
child: child,
),

// 3.
FadeTransition(
    opacity: Tween<double>(
        begin: 0,
        end: 1,
    ).animate(animation),
    child: child,
),
```

Place all of these transitions in a dedicated class inside the `routes_transitions.dart` file we created earlier so that any route-related animation is located in a single place.

15 | Working with JSON and other formats

15.1 Parsing JSON

Nowadays JSON is one of the most popular data-interchange formats. For example, very likely you'll have to deal with HTTP requests returning a json-encoded string . Flutter has an incredible support for this format as it allows you to convert objects from and to JSON very quickly.

```
{  
    "id": 1,  
    "name": "Alberto",  
}
```

What's inside curly brackets is called **json object** and it's always made up of a string, which is the key of the field, and a value which can be a string, a number, a boolean, a list, null or another object. This is another example with a JSON array (or list):

```
[  
    {  
        "id": 1,  
        "name": "Alberto",  
    },  
    {  
        "id": 2,  
        "name": "Patrick",  
    }  
]
```

Lists are simply collections of objects. In Flutter you can decide between manually parsing strings,

using facilities from `import "dart:convert"`, or using automatic code generation, which does most of the work automatically. Both ways are fine but you'll see that code generation simplifies maintenance a lot, especially for large JSON objects.

15.1.1 Manual parsing

In all those cases where there's the need to deal with a list or small objects, parsing and maintaining JSON manually is fine. The best approach to handle the received data is the creation of a *model* class from its JSON representation. In other words, you should convert a json-encoded string into a Dart `class`. Let's say we have to parse this string.

```
{ "id": 1, "value": "json test" }
```

First of all there's the need to use the `jsonDecode` method, from `"dart:convert"`, which translates a json string into a Dart object whose type is `Map<String, dynamic>`. As we told you back in chapter 2, this is one of the rare cases in which `dynamic` has to be used.

```
import "dart:convert";

void main () {
    final jsonString = '{ "id": 1, "name": "Alberto" }';

    Map<String, dynamic> data = jsonDecode(jsonString);
}
```

If you think about it, this design choice really makes sense because a json object is nothing more than a map whose keys are strings and values can be integers, strings, objects or lists. There's the possibility to make a 1:1 comparison which is very expressive:

JSON string

```
{
  "id": 1,
  "name": "Alberto"
}
```

Dart object

```
Map<String, dynamic> data = {
  "id": 1,
  "name": "Alberto"
}
```

The values of the map are `dynamic` because we have to take care of converting the content into a proper data type. The next step is the creation of a **model** class which converts a given object from and to JSON.

```
class Person {
    final int id;
    final String value;

    // 1.
    Person._({
        required this.id,
        required this.value
    });

    // 2.
    factory Person.fromJson(Map<String, dynamic> json) =>
        Person._(
            id: json["id"],
            value: json["name"],
        );

    // 3.
    Map<String, dynamic> toJson() =>
    {
        "id": id,
        "name": value
    };

    @override
    String toString() => "$id - $value";
}
```

All the parsing logic lies inside `class Person` and nowhere else; casts only happen internally so from the outside there's no need to deal with `dynamic`. In practice, calling `Person.fromJson()` is safe because the developer is not asked to make any cast since they're hidden internally.

1. The constructor is private because this class can be instantiated only if a well-formed json string is passed via `factory`.
2. The `factory` constructor takes the result of a `jsonDecode` method and constructs a `Person` object. This is the only place in which we should pay attention to types. We could also have written this...

```
Person._(
    id: json["id"] as int,
    value: json["value"] as String,
);
```

... but the compiler is smart enough to infer the types and perform automatic casts. Note that this approach is required if you set strict implicit casts rules in `analysis_options.yaml`.

- Converts the object into a data structure that can be passed to `jsonEncode`, which will convert the `Map` into a JSON string. It's very easy to use:

```
final jsonString = jsonEncode(personObject);
```

The `jsonEncode` method automatically calls `Map<String, dynamic> toJson()` so be sure to have it defined otherwise an error will occur.

As you can see, parsing and converting is all around the creation of a **model class** which should define `factory MyObj.fromJson` and `MyObj.toJson`. Thanks to these conversion methods, handling strings is very easy:

```
void main () {
    final jsonString = '{ "id": 1, "name": "Alberto" }';

    final decodedMap = jsonDecode(jsonString);
    final user = Person.fromJson(decodedMap);

    print("${user.id}"); // prints "1"
    print("${user.value}"); // prints "Alberto"

    final json = jsonEncode(user);
    print(json); // prints '{"id": 1, "name": "Alberto"}'
}
```

Having a private constructor, a `Person` object can only be created from a JSON string and nothing more. Any model class is asked to, at least, define:

- a `factory` constructor which creates new instances using the returned value of `jsonDecode`;
- a method called `toJson()` that will be automatically called by `jsonEncode`

15.1.1.1 Parsing lists

Very often JSON strings are long lists containing a series of objects and of course Dart has an easy way to deal with them. Suppose you had to work with the following string:

```
[  
  { "id": 1, "name": "Alberto" },  
  { "id": 2, "name": "Felix" },  
  { "id": 3, "name": "Rémi" },  
  { "id": 4, "name": "Matt" }  
]
```

In order to parse a list there's the need to intuitively use a `List<T>` object and then treating each item as a JSON object. We're using the same model class we've created in the previous section.

```
void main() {  
  final jsonString =  
    '''[  
      { "id": 1, "name": "Alberto" },  
      { "id": 2, "name": "Felix" },  
      { "id": 3, "name": "Rémi" },  
      { "id": 4, "name": "Matt" }  
    ]''';  
  
  // It's a list of json objects  
  List<dynamic> jsonList = jsonDecode(jsonString);  
  
  // Each object is converted by the model class  
  final List<Person> people = jsonList  
    .map((value) => Person.fromJson(value))  
    .toList();  
}
```

The decoder is allowed to return a `List<dynamic>` when it encounters a list; each json object inside it is nicely converted by the model class. A list is simply a collection of objects which can be handled by a model class.

- ➊ The usage of a model class is very handy in this case; the code is highly readable

and type safe. If we didn't use `Person.fromJson` there would have been the need to make manual casts for `any` list to be parsed.

A model class is always required to keep your code readable and easily testable.

15.1.1.2 Parsing nested objects

With the manual approach, things start to get complicated when the JSON string is quite complex because, for example, it's made up of nested objects or lists. Imagine having to work with this apparently simple string:

```
{  
  "type": "Developers",  
  "data": [  
    { "id": 1, "name": "Alberto" },  
    { "id": 2, "name": "Felix" },  
    { "id": 3, "name": "Rémi" },  
    { "id": 4, "name": "Matt" }  
  ]  
}
```

If you're used to work with REST APIs you might have already encountered something like this millions of times. Jumping into the code, parsing logic starts to get more difficult to write and thus harder for a developer to read. Now `Person` is not enough anymore and we need another model class to handle the outer object:

```
// Same as before  
class Person { ... }  
  
// New class for the main object containing 'type' and 'data'  
class Data {  
  final String type;  
  final List<Person> people;  
  
  Data._({  
    required this.type,  
    required this.people  
  });
```

```
factory Data.fromJson(Map<String, dynamic> json) {  
    List<dynamic> list = json["data"];  
    List<Person> peopleList = list  
        .map((personObj) => Person.fromJson(personObj))  
        .toList();  
  
    return Data._(  
        type: json["type"],  
        people: peopleList,  
    );  
}  
  
}
```

Performance is not an issue; the problem instead is the fact that for a fairly simple JSON string we've had to create two different model classes. You can always use manual parsing but with very complex and nested objects, you might get in trouble.

 As time goes by the maintenance can also get harder. What if the REST API from which you depend changed the structure? Potentially, you'd have to refactor most of your code. The solution is the usage of code generation.

Every json object requires his dedicated model class. There isn't a strict rule to determine when you should go for the manual approach but that's up to you; when you see that parsing starts to get hard and there's a lot of boilerplate code, consider moving to automatic parsing.

15.1.2 Automatic parsing

When JSON strings are very complex, with many nested objects and lists, it would be better using code generation which handles most of the tedious work automatically. First of all, you need to add a few dependencies to the project making sure to use the latest version of any package:

```
dependencies  
    json_annotation: ^3.0.1  
    json_serializable: ^3.4.0
```

The automatic way of parsing JSON is nothing new from what you've just seen. The process

is identical to manual parsing with the only difference that model classes are generated by a command line tool rather than by you, the developer. In order to make an efficient comparison between automatic and manual parsing, we're going to use the same JSON string as example.

```
{ "id": 1, "value": "json test" }
```

As usual, there's the need to create the famous model class which takes care of converting from and to JSON. This time the code looks a bit different from the usual:

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

// 1.
part 'person.g.dart';

// 2.
@JsonSerializable()
class Person {
    final int id;
    final String name;
    Person(this.id, this.name);

    // 3.
    factory Person.fromJson(Map<String, dynamic> json)
        => _$PersonFromJson(json);

    Map<String, dynamic> toJson()
        => _$PersonToJson(this);
}
```

With this setup the code generator tool will be able to automatically create the bodies of both `fromJson` and `toJson`; there's no need for you to make casts or converting data. For now, ignore the errors underlined by the IDE because we're removing them soon.

1. The code generator will automatically create a file called '`person.g.dart`'. Thanks to the `part` directive, the `Person` class is able to access private members in the generated file.
2. The annotation is fundamental as it tells the code generator that `Person` needs the JSON serialization logic to be created. It's a "marker" to recognize the classes that needs code generation.

Chapter 15. Working with JSON and other formats

3. The syntax is a bit weird but it's telling the generator to create for us the code of these methods. You already know what they do from the previous section.

The setup is now ready and it's time to invoke the code generator. Open the terminal from your IDE and run this command, making sure to be at root of the project (which should be already there by default):

```
$ flutter pub run build_runner build
```

Wait until it completes and at the end you will see a new file called `person.g.dart` which has just been created. It contains the same logic you would have implemented by hand as it does casts from `dynamic` to the correct type according with the structure of the model class.

i Any change made to the model class requires the command `build_runner` to be run again in order to update the generated class. Doing this repeatedly might be tedious or you could simply forget about it:

```
$ flutter pub run build_runner watch
```

Using the command with `watch` rather than `build` automatically runs the code generator whenever a change is made to the class.

The IDE doesn't complain anymore about `person.g.dart` not being declared because it's now been generated by the tool. You're going to use `Person` in the **exact same** way we had already seen in the manual parsing section.

```
void main () {
    final jsonString = '{ "id": 1, "name": "Alberto" }';

    final dataMap = jsonDecode(jsonString);
    final user = Person.fromJson(dataMap);

    print("${user.id}"); // prints "1"
    print("${user.name}"); // prints "Alberto"

    final json = jsonEncode(user);
    print(json); // prints '{"id": 1, "name": "Alberto"}'
}
```

In such a simple example it's hard to see why code generation is useful, in fact you could have

done it by hand. The real usefulness of the code generator comes when, for example, complex objects like the following have to be parsed:

```
[  
  {  
    "id": 1,  
    "roles": [  
      {"id": 1, "type": "Author"},  
      {"id": 2, "type": "Reviewer"}  
    ],  
    "data": {  
      "params": [0, 3],  
      "values": {  
        "data": []  
      },  
      "keys": [13, 6, 7]  
    },  
    "extra": "none"  
  }  
]
```

Each object needs its dedicated model class plus a series of handlers for the lists. Doing a manual parsing is going to require a lot of boilerplate code and by consequence it will also be quite hard to maintain as time goes by. Thanks to code generation, an automated tool takes care of the most tedious part.

```
@JsonKey(required: true)  
final int id;  
  
@JsonKey(defaultValue: "no name")  
final String name;
```

There are some extra annotations to instruct the behavior of the generator in certain situations. In the first case, the `required` marker throws an error if the key is not present in the JSON string. In the second case `defaultValue` puts the given value in the variable if the JSON key doesn't exist or if it's `null`.

15.1.2.1 Parsing lists

Parsing lists doesn't involve any extra effort because, again, the generator does all the work for you. Unless you're parsing a primitive type, such as strings or integers, you need a model class for the objects in the array.

```
{
  "total": 4,
  "values": ["A", "B", "C", "D"]
}
```

Declaring a `List<String>` instance variable is enough to tell `build_runner` to generate the needed code to parse a list of strings. In the next section we will show you what to do in case of lists of objects.

```
@JsonSerializable()
class Example {
  final int total;
  final List<String> values;
  Example(this.id, this.name);

  factory Example.fromJson(Map<String, dynamic> json)
    => _$ExampleFromJson(json);

  Map<String, dynamic> toJson()
    => _$ExampleToJson(this);
}
```

You could have put `JsonKey(defaultValue: [])` above `values` if you wanted it to be initialized with an empty array. Even if default values are not required, it's a good practice always defining them just to be sure that a variable is always initialized as expected.

15.1.2.2 Parsing nested objects

As you probably have already guessed, parsing nested object requires no efforts from your side because the implementation will be automatically generated. Let's take as example the same JSON string we have already seen in the previous section:

```
{
  "type": "Developers",
  "data": [
```

```

        { "id": 1, "name": "Alberto" },
        { "id": 2, "name": "Felix" },
        { "id": 3, "name": "Rémi" },
        { "id": 4, "name": "Matt" }
    ]
}

```

We had already created a model class for the items in the array (`Person`, with the manual approach) but you could easily generate one with `build_runner`. What's missing is a model for the "outer" object, the one made up of `"data"` and `"type"`.

```

@jsonSerializable(explicitToJson: true)
class Data {
    final String type;
    final List<Person> data;
    Data(this.type, this.data);

    factory Data.fromJson(Map<String, dynamic> json)
        => _$DataFromJson(json);

    Map<String, dynamic> toJson()
        => _$DataToJson(this);
}

```

Use `explicitToJson: true` when you have a class that contains another class in one of its fields. In this case, in order to properly parse each `Person` object of the JSON array, you have to set `explicitToJson: true` on `Data` so that the inner-objects are included in the generation flow.

```

@JsonSerializable()
class Person { ... }

@JsonSerializable(explicitToJson: true)
class Data { ... }

```

Only the classes containing other classes need to have the `explicitToJson` parameter set. In case of primitive types, the annotation is not required. If your classes were big and deeply nested, the generator might take some more seconds but still it'd be faster than a manual approach.

i The official documentation discourages the usage of the `part` keyword by the

developer¹. The only case in which it's good is when code generation tools are involved, like in this case. For all the other needs, rely on a package-like structure using `library` and `export` statements.

15.2 Parsing XML

Another well-known data exchange format is XML which is still widely used even with the arrival of JSON, so you might be interested in knowing how to deal with it. One of the best packages available is called `xml`², made by Lukas Renggli.

```
<articles>
  <item>
    <name>Book</name>
    <quantity>5</quantity>
  </item>
  <item>
    <name>Tablet</name>
    <quantity>2</quantity>
  </item>
</articles>
```

Reading this string and converting it into a handy `xml` object is very easy as you just need to call `XmlDocument.parse()`. Version 4.1.0 and earlier of the package had a top-level method called `parse()` which is now deprecated.

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

void main() {
  var xmlString = "<articles> ... </articles>";

  try {
    final xmlDoc = XmlDocument.parse(xmlString);
  } on XmlParserException {
    // error while parsing the string
  }
}
```

¹<https://dart.dev/guides/libraries/create-library-packages>

²<https://pub.dev/packages/xml>

```
}
```

If the XML string is malformed, for example due to a syntax error, an exception is thrown. The `xmlDoc` variable is of type `XmlDocument` which provides many functionalities. It overrides the `toString()` so that the object can easily be printed.

```
print("$xmlDoc");
```

The returned string is "pretty" encoded, meaning that it's properly indented with spaces and tabulations. If you want to have more control on the output formatting, use `indent`:

```
final output = xmlDoc.toXmlString(  
    pretty: true,  
    indent: '-'  
)  
  
print(output);
```

Having set `pretty: true` the XML tree is not printed verbatim because useless leading spaces are removed and indentation is fixed. With `indent: '-'` the indentation spaces are replaced with - producing this:

```
<articles>  
-<item>  
--<name>Book</name>  
--<quantity>5</quantity>  
-</item>  
-<item>  
--<name>Tablet</name>  
--<quantity>2</quantity>  
-</item>  
</articles>
```

By default the indentation uses a double white space ' ' but you could also use `indent: '\t'` or anything else.

- When calling `XmlDocument.parse()` you need to pass a string in a proper UTF encoding. In Dart, `String` is a sequence of UTF-16 code units so everything works fine.

```
// All good because strings in Dart are UTF encoded
```

```
final xmlDoc = XmlDocument.parse("<articles></articles>");
```

If you're retrieving the XML from the internet, for example via GET request, the string encoding might be different from UTF (it could be Latin1, for example). Since `parse()` works well only with UTF, you could have the need to use the `Latin1Decoder()` class.

```
// More info on GET requests and networking on chapter 17.  
// This is a GET request which returns a 'Response' object  
final response = await http.get("https://website.com/get/myFile.xml");  
  
// Convert the response to a proper UTF format, which is how Dart  
// strings are internally represented  
final String xmlString = Latin1Decoder().convert(response.bodyBytes);  
  
// You're now sure that parsing will execute successfully  
final xmlDoc = XmlDocument.parse(xmlString);
```

The response body (`response.body`) is a `String` but it's not guaranteed to be in UTF encoding. If the response header specifies `charset=latin1` for example, then the string encoding will be Latin1. Thanks to `Latin1Decoder()` you can convert a latin1-encoded string into a Dart string (UTF-16).

```
AsciiDecoder().convert(response.bodyBytes);  
Utf8Decoder().convert(response.bodyBytes);
```

They all work like `Latin1Decoder()`: use the `convert()` method to return a well-formed Dart string. This conversion should be taken into account not only when working with the `xml` package but in general (the same problem might happen when working with JSON as well).

15.2.1 Parsing strings

Once an XML-encoded string has been converted into a `XmlDocument`, there's the possibility to traverse the elements in many ways. Let's say we wanted to print the name of any article in the list:

```
final xmlString = """  
<articles>  
  <item>  
    <name>Book</name>
```

```
        <quantity>5</quantity>
    </item>
    <item>
        <name>Tablet</name>
        <quantity>2</quantity>
    </item>
</articles>
""";
```

```
final xmlDoc = XmlDocument.parse(xmlString);

xmlDoc.findAllElements("name")
    .map((item) => item.text)
    .forEach(print);

// Prints:
// Book
// Tablet
```

Very intuitively, `findAllElements(String name)` returns an iterable object containing every XML element whose tag matches the given string. We could also have iterated on `item` and extracted the various values of the children by hand:

```
xmlDoc.findAllElements("item")
    .map((item) {
        final name = item.findElements("name").single.text;
        final qty = item.findElements("quantity").single.text;

        return "$name (amount = $qty)";
    })
    .forEach(print);

// Prints:
// Book (amount = 5)
// Tablet (amount = 2)
```

The `findElements()` method looks for any children of the currently selected node, it doesn't look on the entire tree. In order to obtain a reference to the node you can use:

- **single:** Checks if there's only one child node with the given name and returns it. If not,

an exception is thrown.

- **first**: Returns the first child node that matches the given name and returns it. If it's not present, an exception is thrown.
- **last**: Returns the last child node that matches the given name and returns it. If it's not present, an exception is thrown.

While `findAllElements()` finds **every** children in the tree with a given name, `findElements()` instead looks only for children of the **current** node. Look at this simple comparison to better understand the difference:

- **result** is an iterable containing 2 children, more precisely the two `<item>` tags representing the book and the tablet. With `findAllElements()` you're looking for any node across the entire tree.

```
final xmlDoc = XmlDocument.parse(xmlString);
final result = xmlDoc.findAllElements("item");

// <articles> is at the root and it contains 2 <item> nodes;
// findAllElements for any element whose tag is 'item'
```

- **result** is an iterable containing no children because it looks only for direct children of the current node. The current node is the root which has a single child (`<articles>`).

```
final xmlDoc = XmlDocument.parse(xmlString);
final result = xmlDoc.findElements("item");

// The current node is the root, which only contains
// <articles>; the various <item> elements are not direct
// children of the root
```

When calling `parse()` the currently selected node becomes the root of the tree which only has the `<articles>` node. The root has no `<item>` children so the method finds nothing.

- In this case both methods return an iterable of size 1 which contains `<articles>` and all of its children.

```
final xmlDoc = XmlDocument.parse(xmlString);
final deepChildren = xmlDoc.findAllElements("articles");
final directChildren = xmlDoc.findElements("articles");
```

`<articles>` is a direct children of the root, the currently selected node, so `findElements`

is able to get a result. At the same time, `<articles>` is the only tag in the tree so `findAllElements` will return an identical result.

In general you should use `findAllElements` when you want to get an entire list of elements and then inside it you call `findElements` to retrieve the specific values. This is an example of how they can be used together:

```
// Get every <item> node with 'findAllElements' and then
// use 'findElements' to find only direct children of the
// currently selected element.
final tot = xmlDoc.findAllElements("item")
    .map((item) =>
        int.parse(item.findElements("quantity").single.text)
    )
    .reduce((a, b) => a + b);

print("$tot"); // prints '7', the sum of all quantities
```

15.2.2 Building XML strings

Let's say we want to create an XML string representing a series of personal information such as name, surname and age. The final result has to look like this:

```
<people>
  <person>
    <name>Alberto</name>
    <surname>Miola</surname>
    <age>23</age>
  </person>
  <person>
    <name>Another</name>
    <surname>One</surname>
    <age>36</age>
  </person>
</people>
```

Thanks to the `class XmlBuilder` this job is quite easy but the code could get hard to read due to the big amount of nested functions. For this reason, it would be better creating series of methods for each recurring item, in this case the `<person>` child.

```
void addPerson(XmlDocument.XmlBuilder builder, {
    required String name,
    required String surname,
    required int age
}) {
    builder.element('person', nest: () {
        builder.element('name', nest: name);
        builder.element('surname', nest: surname);
        builder.element('age', nest: age);
    });
}
```

Each time `addPerson(...)` is called, a new `<person>` element is added and the code still remains readable. Using `nest` you can pass a primitive value, such a string/number, or a function which generally builds new XML items.

- ❶ Of course there is also the possibility to set attributes to specific elements. For example, if we wanted to add the `age` attribute we'd have to use `attribute()`:

```
builder.element('person', nest: () {
    builder.attribute("young", age <= 18);
    // name, surname and age...
});
```

The result might look like `<person young="false"></person>` where `true` or `false` depends whether the age is lower or higher than 18.

Now you just need to create an instance of `XmlBuilder` and compose the string; once finished, call the `build()` method. Helper functions such as `addPerson` make the code more readable because nesting too many functions might be quite pesky.

```
final builder = XmlBuilder();

builder.processing('xml', 'version="1.0"');
builder.element('people', nest: () {
    addPerson(builder,
        name: "Alberto",
        surname: "Miola",
```

```
    age: 23);
    addPerson(builder,
        name: "Another",
        surname: "One",
        age: 23);
});

final peopleXML = builder.build();
```

Converting the XML object with `peopleXML.toXmlString(pretty: true)`; is probably the best human-readable solution. However, you can also go for a plain `toString()` which uses no spaces nor indentation.

16 | Testing and profiling apps

16.1 Testing Flutter apps

For demo applications consisting of one or two routes and a few lines of code, it's probably fine if you do "*naive*" debugging just by printing values to the console. The more features your program has, the harder it is to manual testing code and execution flows.

i Nowadays multithreading is a fundamental aspect of computer programming and it increases the logical complexity of the code. As you know, Dart doesn't have synchronization problems since Isolates doesn't share memory and thus testing asynchronous code is much easier in comparison with Java or C# for example.

Multithreading aside, in general having a reliable and robust way of testing your code is essential to guarantee the quality of the product. There are three main testing strategies:

- **Unit test.** Kind of tests made on small pieces of Dart code such as classes or functions. They check whether the logic of the code works as intended or not.
- **Widget test.** Kind of tests made on Flutter widgets. They check if a widget is present in the widget tree or how many times does it appear.
- **Integration test.** Kind of tests in which different parts of the code are combined and tested together as a group.

Imagine you're working on a Flutter application with some friends of yours using, for example, Android Studio and GitHub. Everything works fine but one day you have to do a breaking change which requires a quite important refactoring.

- **With tests.** Right click on the `test/` folder and hit *Run*. If you've broken something,

one or more tests will fail indicating exactly where's the issue so that you can immediately investigate and solve it.

- **Without tests.** You have absolutely no idea if fixed bugs have been broken again or other working components of your software are still healthy. Nothing will automatically tell you where known issues are! You have to scan the entire code seeking for bugs and hoping to not miss anything. You can only hope that major changes won't break the code but you understand that this way of working is really a "no-go".

Writing tests (generally using *testing frameworks*) is basically making sure that errors won't be repeated in the future. Once you've found a bug, solve it and write a test to make sure that future changes won't lead again to the same erroneous behaviors.

- i** Of course, passing tests doesn't assure your program is flawless because there might be bugs you haven't spotted yet. However, a *perfect result* (tests passed with no errors) ensures that **known** bugs have been fixed and they're not present anymore.

Having no tests means having no guarantees that known bugs are gone. Any change, be it big or small, is a potential danger. Manually checking the code is tedious, time-consuming and very risky because you might forget to check certain issues (especially if the codebase is large).

16.1.1 Unit Test

Unit tests are used to test small blocks of code (such as classes or methods) by verifying their logical correctness using some conditions defined by the developer. We're going to test this simple class:

```
// created inside 'lib/fraction.dart'
class Fraction {
    int _num;
    int _den;

    Fraction({int numerator = 0, int denominator = 1}) :
        _num = numerator,
        _den = denominator;

    void negate() => _num *= -1;
    double get toDouble => _num / _den;
```

```

    @override
    String toString() => "${_num} / ${_den}";
}

```

The `test` package¹ from the Dart team is a very powerful testing framework which also integrates with `flutter_test`. In our example, `class Fraction` is said to be the *unit* and here's how to test it:

1. We're going to test "*pure*" Dart code, in the sense that no Flutter widgets are involved. Let's start by adding a dependency to the testing framework:

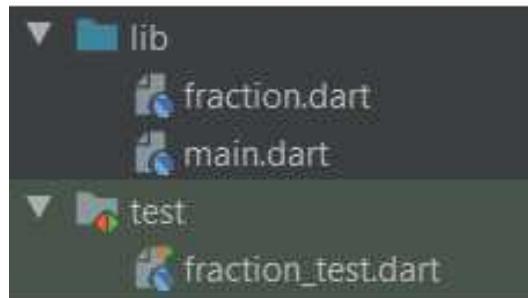
```

dev_dependencies:
  test: ^1.15.3

```

Check <https://pub.dev/packages/test> to get the latest version.

2. Create a file called `fraction_test.dart` inside the `test` library of your project, which is already added by Android Studio or VS Code by default. You should end up with this structure:



Name it whatever you want but by convention test files should always end with `*_test.dart` to be easily recognizable. The must always be a `void main() {}` function.

3. Inside `fraction_test.dart` start implementing the logic of your tests using the API provided by test `test` package. There are mainly two functions to use:

- `void test()`. It accepts a string, which briefly describes what is being checked, and a callback, the actual logic of the test.
- `void expect(a, b)`. The second value is compared to the first one and, if they don't

¹<https://pub.dev/packages/test>

match, the test fails by throwing an exception.

It doesn't matter in which order you pass parameters to `expect` but by convention the value produced by your code goes on the left.

```
void main() {
    test('10 divided by 2 should be 5', () {
        final fraction = Fraction(
            numerator: 10,
            denominator: 2
        );

        expect(fraction.toDouble, 5.0);
    });
}
```

4. Generally, a class has more than a single method to be tested. Using the `group()` function multiple tests can be logically grouped under the same "container".

```
void main() {
    group("Fraction class testing", () {
        test("10 divided by 2 should be 5", () {
            final fraction = Fraction(
                numerator: 10,
                denominator: 2
            );
            expect(fraction.toDouble, 5.0);
        });

        test("'negate' should produce opposed signs", () {
            final fraction = Fraction(
                numerator: 10,
                denominator: 2
            );
            fraction.negate();
            expect(fraction.toDouble, -5.0);
        });
    });
}
```

Chapter 16. Testing and profiling apps

5. Tests are now ready to be run either via command line or using the facilities provided by the IDE. For old school developers, just type this in the terminal:

```
$ flutter test test/fraction_test.dart
```

Otherwise, the official Flutter plugin for both Android Studio and VSCode supports unit testing so that you can run tests in a more friendly way. In **Android Studio** (as of version 4.1.0):

- Open the `fraction__test.dart` file;
- Go on Run > Edit Configurations;
- Click on the + button on the left and select "Dart Test" (or "Flutter Test");
- Give it a meaningful description and be sure that in *Test scope* the *All in file* option is selected;
- Select the test file we've just created. Click OK;

Now in the drop-down next to the **Run** button select the test you've just created and run it; results will appear in the console underneath. In **VSCode**:

- Open the `fraction__test.dart` file;
- Select the *Debug* menu;
- Click on *Start Debugging* and the tests will be run.

If you have set a keyboard shortcut for executing tests immediately, use it.

Creating unit tests is not so hard; once you've your class or function ready to be tested, create a dedicated file in the `test/` folder. Use `test("Description", () {})` to write the logic and possibly `group()` to group more tests together.

- i** To collect code coverage, add the `--coverage <directory>` flag when calling the `test` command line tool. The directory will be automatically created if it doesn't exist and the results will be put there.

So far you've only seen `expect()` being called, which just matches for a single case. There's the possibility to run multiple checks for a single test thanks to `allOf()`. Be sure to check out the documentation of the *matcher*² for a complete list of methods you can use inside `allOf()`.

²<https://pub.dev/documentation/matcher/latest/matcher/matcher-library.html>

```
expect('A short string.', allOf([
  contains('rt'),
  startsWith('A '),
  endsWith('. ')
]));
```

If you want to make sure that your code throws an exception under certain circumstances, you could include it in a `try catch` block and call `expect()` with boolean values. It's doable for sure but there's a better way of dealing with exceptions:

```
expect(() => Fraction(1, 0), throwsA(isA<FractionException>()));
```

The `throwsA()` function catches exceptions and it's generally used together with `isA<T>()` to match the exact type. There's also the possibility to use a "shortcut" in case of common exception types:

```
expect(() { ... }, throwsException);
expect(() { ... }, throwsFormatException);
expect(() { ... }, throwsUnsupportedError);
// and more...
```

Writing `throwsA(isA<Exception>())` is equivalent to `throwsException` but with less code. As always, check out the official documentation ³ to see every kind of common exception type available.

16.1.1.1 Testing asynchronous code and streams

If you use `async` and `await` regularly, tests will work "normally" like if you were working with synchronous calls. The only exception is that they may take a bit longer to execute because the test runner waits until every `Future<T>` completes.

```
void main() {
  test('Testing with async and await', () async {
    final age = await Future<int>.value(25);
    expect(age, equals(25));
  });
}
```

We could have also achieved the same result without using `async` and `await` at all. For consistency and readability, we don't recommend you to go for this kind of approach.

³https://api.flutter.dev/flutter/test_api/test_library.html

```
void main() {
  test('Testing without async and await', () {
    final age = Future<int>.value(25);
    expect(age, completion(equals(25)));
  });
}
```

The `completion()` function is used to test a `Future<T>` object: it ensures that the test doesn't finish until the operation completes. Once available, the value is passed to `equals()`. With streams, nothing new again:

```
final rocketLaunch = Stream<String>.fromIterable([
  "3", "2", "1", "0", "Ignition", "Success!"
]);

test("rocket launch test", () {
  expect(rocketLaunch, emitsInOrder([
    // This is a value and it matches individual events
    "3",
    // Asserts that one of the options in the list is emitted
    emitsAnyOf(["Success!", "Failure"]),
    // By default, more events are allowed at the end. 'emitsDone'
    // makes ensure that nothing else is emitted after this matcher.
    emitsDone
  ]));
});
```

We've created a dummy `rocketLaunch` stream just as test. Details aside, the structure is always the same: create a new test with `test("description", () { ... })` and then use `expect()` to ensure the correctness of the output. There are many matchers available⁴ such as `emitsAnyOf()` or `neverEmits()`.

16.1.1.2 Mocking dependencies

In real-world applications it's very likely you'll have to deal with http connectivity or `Future<T>`s in general. They're very annoying to test using the technique we've just described for a few

⁴<https://pub.dev/packages/test#asynchronous-tests>

reasons:

1. You could successfully run your tests but then, due to a bug in the online API that you don't control, failures start to pop out.
2. Making HTTP requests might slow down the execution time of the tests. They could also fail due to a series of connectivity problems related to the response sent by the server.
3. If the web service goes down for maintenance in the exact moment in which you want to test, you simply can't do it.

Flutter gives you the possibility to **mock** dependencies. The term "*mocking*" indicates the act of emulating a database, a web service or any other external source in local so that it's always available.

i Let's say your app normally sends http requests to an online REST API service. When it's time to test you should create a *mock*, which is a "local fake API" under your control which acts like if it were a real external service.

We're going to create mocks by using the official `mockito`⁵ package, created by the Dart team. We're still doing unit tests but with a different approach. Rather than directly connecting to the server, we emulate a fake one just for testing purposes.

```
import 'package:http/http.dart' as http;

class Todos {
    final _source = "https://myonlineapi/get/todos";

    String getJson(http.Client client) async {
        final response = await client.get(_source);

        if (response.statusCode != 200)
            return response.body;
        else
            return "{}";
    }
}
```

⁵<https://pub.dev/packages/mockito>

Chapter 16. Testing and profiling apps

In chapter 17 you'll see many examples about HTTP requests and data parsing. In this example we're showing how to unit test the class `Todos`, which uses the official `http`⁶ package to retrieve a JSON string.

- Add the `mockito` package in the dependencies list along with the usual `test` package. As always, be sure to use the latest versions.

```
dependencies:
  http: ^0.12.2
dev_dependencies:
  test: ^1.15.3
  mockito: ^4.1.1
```

- Like we've done before, create the file `todos_test.dart` inside the `test` folder which is going to contain our logic. For convenience, mocks can be created in the same file as the `main()`:

```
// The fake API server
class HTTPMock extends Mock implements http.Client {}

void main() {
  // tests go here as usual...
}
```

The `String getJson(...)` method is going to receive an instance of `HTTPMock` rather than `http.Client`. It works because the mock `implements` the `http` client object and thus there's type compatibility.

- Writing tests is nothing new from what you had seen earlier; use `test()` to write the logic and, for a better mental organization, divide the methods in multiple groups with `group()`.

```
class HTTPMock extends Mock implements http.Client {}

void main() {
  final url = "https://myonlineapi/get/todos";

  test("Returns JSON in case of success", () async {
    // 1.
    final todo = Todos();
```

⁶<https://pub.dev/packages/http>

```
// 2.  
final mock = HTTPMock();  
final fakeJson = '[{"id": 1, "desc": "Test!"}]';  
  
// 3.  
when(mock.get(url).whenComplete(() {  
    return http.Response(fakeJson, 200);  
}));  
  
// 4.  
expect(  
    await todo.getJson(mock),  
    // ... or compare the result with 'fakeJson'  
    const TypeMatcher<String>()  
);  
});  
}  
}
```

The `void when(...)` function is provided by the *mockito* package.

1. The instance of the class that is going to be tested.
 2. The instance of the mock, the "fake API" which is going to return a particular json response.
 3. When the test runs on a specific url, the method `get(String url)` is not called on the real `http` client but instead it returns the fake JSON string provided by the mock.
 4. When we call `getJson` we're not passing a real `http` object which connects to the online API. We pass the fake `http` object which emulates a request and returns a JSON string we can control.
- Run the test via command line by typing:

```
$ dart test/todos_test.dart
```

When HTTP and/or database connectivity are involved, use a package like *mockito* to easily run your tests. In every other case where there is just "plain" Dart code to test, with no `Future<T>`s or other time-consuming tasks, use a traditional unit test.

-  Note that even if the web service were down or the connectivity were absent, we

would still be able to run tests thanks to the mock. Everything is local to the current machine so tests don't rely on external sources.

16.1.1.3 Unit testing blocs

State management libraries require to be unit tested like any other logic implemented in the app and thus *flutter_bloc* is no exception. Thanks to the *bloc_test*⁷ package verifying the behavior of your blocs and cubits couldn't be easier.

- ❶ In this example we're unit-testing the simple *CounterBloc* we made in chapter 11. It just expects an event of type `increase` or `decrease` and returns an integer. You could manually test blocs using `test()` but, other than being difficult, there wouldn't be the possibility of verifying certain cases.

Testing blocs is very easy: instead of relying on the classic `test()` we should use `blocTest()` which simplifies the process a lot and makes it very intuitive. Look at this example:

```
void main() {
  blocTest(
    'emits [1] when increment is added',
    build: () => CounterBloc(),
    act: (bloc) => bloc.add(CounterEvent.increment),
    expect: [1],
  );
}
```

With `act` you can send events to the bloc being created in `build` and `expect()` is a list of expected emitted states. Of course there's the possibility to send multiple events and look for multiple results:

```
blocTest(
  "emits [1, 2, 1] when 2 increments and 1 decrement are added",
  build: () => CounterBloc(),
  act: (bloc) => bloc
    ..add(CounterEvent.increment);
    ..add(CounterEvent.increment);
```

⁷https://pub.dev/packages/bloc_test

```
    ..add(CounterEvent.decrement);
},
expect: [1, 2, 1],
);
```

Setting the optional `skip` parameters allows you to ignore a certain number of states. By default `skip: 0` is set and the initial state is excluded from the `expect` list. Does your bloc `await` somewhere and thus you need to wait some time?

```
build: () => MyBloc(),
act: (bloc) => ...,
wait: const Duration(seconds: 2),
expect: [...],
```

In this way, when `act` sends an event to the bloc, the emitted state is returned after the given time span (which is 2 seconds in the example). This is very useful when you need to wait, for example, when debouncing events.

```
build: () => MyBloc(),
act: (bloc) => bloc.add(Something()),
expect: [0],
errors: [
  isA<Exception>(),
]
```

The `errors` parameter is used to catch exceptions, generally together with `isA<T>` to exactly match the type. Of course, bloc tests can be grouped like any other unit test.

```
void main() {
  group('CounterBloc', () {
    blocTest(...);

    blocTest(...);

    blocTest(...);
  });
}
```

16.1.2 Widget Test

Testing Flutter widgets requires the same process you've seen in unit testing with the addition of some new techniques brought by the built-in *flutter_test* package. It's already bundled in the SDK so no need to install it via pub.

```
class Myself extends StatelessWidget {
    final int age;
    final String name;
    const Myself(this.name, this.age);

    @override
    Widget build(BuildContext context) {
        return Row(
            children: [
                Text(name),
                Text("$age"),
            ]
        );
    }
}
```

Before starting the testing journey for this widget, be sure that the `pubspec.yaml` file declares `flutter_test` in the `dev_dependencies` section. Widget tests are meant to check whether one or more widgets have been properly put in the widget tree.

- As usual, create a new file called `myself_test.dart` inside the `test` folder but this time the main player is `testWidgets()` rather than `test()`.

```
void main() {
    testWidgets("Testing if 'Myself' has name and age",
        (WidgetTester tester) async {
            // 1.
            await tester.pumpWidget(MySelf("Alberto", "100"));

            // 2.
            // Basically we're looking for 2 'Text' widgets
            // containing "Alberto" and "100" inside 'MySelf'
            final CommonFinders name = find.text("Alberto");
            final CommonFinders age = find.text("100");
```

```
// 3.
expect(name, findsWidgets);
expect(age, findsOneWidget);
}
);
}
```

1. The `Future<void> pumpWidget()` method is used to build and render a widget in a testing environment. It's like if `MySelf` were inserted in the widget tree and the framework called its `build()` method to render it.
 2. The `CommonFinders find` is part of Flutter's testing utilities so no external packages are required. The `text()` method looks for `Text` widgets in the tree containing the given string.
 3. We're making sure that the two `Text` widgets actually appear in the tree (and thus they're visible on the UI) using a `Matcher`. The method wants to know how many widgets it has to look for:
 - `findsNothing`: asserts that the finder has found no widgets in the tree;
 - `findsOneWidget`: asserts that the finder has found exactly one widget in the tree;
 - `findsWidgets`: asserts that the finder has found one or more widgets in the tree;
 - `findsNWidget`: asserts that the finder has found exactly N widget in the tree, where N is a value defined by you.
- Run the test in the same way you'd do for a classic Dart unit test; the results will appear in your IDE indicating the failures, if any.
- As you've seen, widget testing consists of building a "fake" widget tree with `pumpWidget()` and looking for specific types using a `Finder`. Of course you can look for any kind of widget, not only `Text`:
- `find.text(...)`: you'll often have the need to look for the presence of a `Text` widget and this method does exactly that.
 - `find.byWidget(...)`: it's useful when you want to look for a specific widget in the tree making sure it appears on the screen a certain number of times. The usage is very easy:

```
// You want to look for an error icon
final target = Icon(Icons.error);
```

```

// Build the widget that has to be tested
await tester.pumpWidget(
    Center(
        child: Padding(
            padding: const EdgeInsets.all(10.0),
            child: target,
        )
    )
);

// Search the icon
final finder = find.byWidget(target);

// Ensure the icon actually appears
expect(finder, findsOneWidget);

```

With this code we're ensuring that the widget tree contains exactly one error Icon.

- `find.byType(...)`: looks for widgets of a particular type. It can be used in the following way:

```
expect(find.byType(IconButton), findsWidgets);
```

After having obtained an instance of a Finder, the `expect` method tells us if the finder was able to satisfy the criteria we imposed such as `findsOneWidget` or `findsWidgets`.

- i** Suppose you had to test a `StatefulWidget` which has a series of animations inside. Other than making sure certain widgets are appearing to the UI, you could also want to test its performances:

```

await tester.pumpWidget(MyWidget());
await tester.pump(Duration(milliseconds: 100));

```

Thanks to `Future<void> pump(...)` we can trigger rebuilds of the tree after a given Duration. In our example, after 100 milliseconds `MyWidget()` will be rebuilt.

```

await tester.pumpWidget(MyWidget());
await tester.pump();
await tester.pump();

```

Calls can also be chained for subsequent rebuilds. Passing no parameters to the method causes an immediate rebuild.

Be aware that `Widget build(...)` of the tested widget is called only once, which is when it's being created via `pumpWidget()`. In a testing environment, calling `setState()` has no effect because rebuilds only happen via `tester.pump()`.

16.1.2.1 Testing blocs and providers

It might happen that a particular part of the widget tree you want to test depends on one or more providers of any kind. There's nothing special to do when it comes to testing as just need to normally wrap the widget in the provider you need.

```
// With a provider...
await tester.pumpWidget(
  Provider<Something>(
    child: MyWidget(),
  ),
);

// In case of multiple providers...
await tester.pumpWidget(
  MultiProvider(
    providers: [...]
    child: MyWidget(),
  ),
);
```

Values are still compared using `expect`. The same concept also applies for `flutter_bloc` with the exception that, only for widget testing, the bloc has to be mocked. Taking into account the `CountBloc` example again, you should create a mock being sure to also depend on `mockito`:

```
class MockCounterBloc extends MockBloc<int> implements CounterBloc {}

void main() {
  final counterBloc = MockCounterBloc();

  // Emit a series of states we've decided
  whenListen(counterBloc, Stream.fromIterable([0, 1, 2]))
```

```
// Build a subtree like you'd regularly do in a widget test
await tester.pumpWidget(
    BlocProvider<CounterBloc>.value(
        value: counterBloc,
        child: MyWidget(),
    )
);

// Making sure the state is correct
expect(counterBloc.state, equals(2));
}
```

Basically before using `pumpWidget()` you need to create a "fake bloc" using a mock, similarly to what you'd do with a fake HTTP client. In this way there's the possibility to send a series of states we decide just by using `whenListen()`. It's also possible making sure that states are emitted in a certain order:

```
// Making sure the state is correct
expect(counterBloc.state, equals(2));

// Making sure the stream emits certain values
await expectLater(counterBloc, emitsInOrder(<int>[1, 2]));
```

The first case just makes sure that the latest state equals 2 while the second one ensures that the entire "emission flow" is correct.

16.1.3 Integration testing

This kind of test gathers together both the UI and the business logic so that you can test the app as a whole. With integration tests you're able to verify how the visual components and the code behave together; the app runs on a device (or simulator) and it's told to do certain things automatically, such as pressing on buttons.



+1 0 -1

We're going to test the counter app we've created in chapter 11 with *provider*; the test has to ensure that both buttons properly increment and decrement the counter at the center.

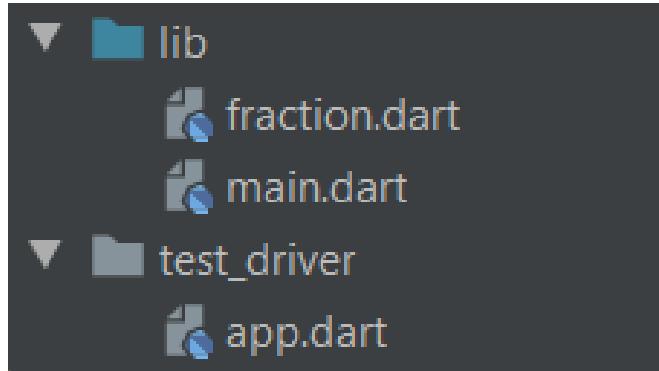
- The full source code can be found in the website at Resources > Chapter 16 > Integration test. The only difference from the original example in chapter 11 is the addition of a `Key` required to reference the specific widgets while testing.

```
FlatButton(  
    key: Key("increment"),  
    child: const Text(...),  
    onPressed: () => counter.increment(),  
) ,  
Text(  
    key: Key("counter"),  
    child: const Text(...),  
) ,  
FlatButton(  
    key: Key("decrement"),  
    child: const Text(...),  
    onPressed: () => counter.decrement(),  
) ,
```

- Integration tests are made with the `flutter_driver` package which has to be added as dependency, as always, in the `pubspec.yaml` file.

```
dev_dependencies:  
  flutter_driver:  
    sdk: flutter  
  test: any
```

- Instead of using the usual `test` folder, you have to create a new one called `test_driver` located at the root of your project (the same level as `lib`).



Integration test tools run don't run in the same process as your app and so, to better represent this separation, Flutter requires you to have a dedicated folder apart.

- Create the file `test_driver/app.dart` but you can name it whatever you want. This file initializes the environment so that it can "automatically" use your app. In practice the code automatically presses on buttons and analyzes the consequences of this action, which is what a human would do in a manual test.

```
void main() {  
  // 1.  
  enableFlutterDriverExtension();  
  
  // 2.  
  runApp(CounterApp())  
}
```

Notice that in widget testing we could only check if a certain widget were present or not in the tree and, at best, count its occurrences. Here instead we're testing the functionalities of the widget combined with the Dart logic behind it.

1. This method prepares the Dart Virtual Machine for executing an "instrumented test" of the app which is, in practical terms, performing actions on various UI widgets (such as tapping buttons, scrolling lists or reading text). Instead of being you tapping buttons, it's the testing driver doing it.
 2. Start the app in the usual way calling `runApp()`. Pass it any widget you want to test.
- The final step is the implementation of the logic inside the `test_driver/app_test.dart` file. In this example we're instructing the driver to press on buttons and read the counter

value from a Text widget.

```
void main() {
    group("Counter App test", () {
        // 1.
        final counterText = find.byValueKey('counter');
        final incrementButton = find.byValueKey('increment');
        final decrementButton = find.byValueKey('decrement');

        late final FlutterDriver driver;

        // 2.
        setUpAll(() async {
            driver = await FlutterDriver.connect();
        });

        // 3.
        tearDownAll(() async {
            driver.close();
        });

        // 4.
        test("Counter increment", () async {
            await driver.tap(incrementButton);

            var readText = await driver.getText(counterText);
            expect(readText, "1");
        });
        test("Counter decrement", () async {
            await driver.tap(decrementButton);

            var readText = await driver.getText(counterText);
            expect(readText, "0");
        });
    });
}
```

1. Here you see why we had set the keys at the beginning; thanks to them, we can locate the widgets in the tree. Each variable holds a "reference" that's going to be used by

the driver to interact with the widget.

2. The `setUpAll()` method is called **before** any test starts and performs initialization. The driver is connected so that it can start interacting with the app.
3. By contrary, `tearDownAll()` is called before the test ends and does the final cleanup of the instances which require a disposing.
4. It's the usual `test()` method you're used to see in this chapter. The `driver` variable is able to interact with our app by doing many things:
 - `Future<void> enterText():` enters the given text in an input field like if the user were tapping on the keyboard;
 - `Future<void> getText():` reads the text from a `Text` widget.
 - `Future<void> scroll():` the driver simulates a finger scrolling on a list. It's possible to also set the direction and the duration.
 - `Future<void> screenshot():` takes a screenshot of the page and stores it in a PNG file.

To see any action that a driver can perform on a widget, visit the official documentation⁸.

- Now you're ready to run the test. First of all, start your app in an Android or iOS emulator pressing the *Run* in your IDE. When the simulator is ready and connected to your IDE, run this command:

```
flutter drive --target=test_driver/app.dart
```

Wait for the initialization to finish and you'll see the driver interacting with the app for you pressing on the buttons, reading text or doing any other action it got told to do.

The driver tool is very powerful: it's a convenient way to systematically and automatically test functionalities of your widgets combined with the business logic. Look what the driver could do for you:

```
test("Counter decrement", () async {
  final textField = find.byValueKey("itsName");

  await driver.setTextEntryEmulation(enabled: true);
```

⁸https://api.flutter.dev/flutter/flutter_driver/FlutterDriver-class.html

```
    await driver.tap(textField);
    await driver.enterText("Flutter is... ");
    await driver.enterText("Awesome!");
});
```

This code writes some strings in a text field (more on this in chapter 19) and also emulates the insertion of the characters like if they were typed by a real person. You should really try the driver at least once to see how useful and user-friendly it is.

16.2 Testing performances

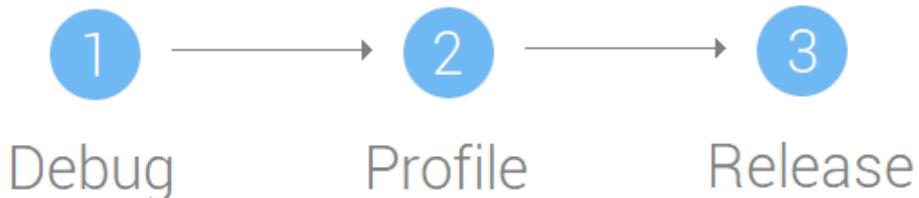
Flutter guarantees to run at 60 fps regularly and at 120 fps in those devices whose display is capable to. If your code wastes resources due to heavy rebuilds, caused by a lack of `const` widgets, or badly-handled animations the frame rate might drop.

 If you follow the numerous good practices we've exposed in this book, there are very high chances that your app won't suffer of performance issues. Caching widgets (entire branches of the tree) and not wasting resources are the most fundamental guidelines to put in practice.

Sometimes you might be able to see with your eyes that the UI doesn't render smoothly but of course it depends on the device. Flutter ships with very useful profilers to help you analyzing the performances of your app while it's running in *profile mode*. Consider that:

- Profile mode works only if your IDE is connected to a physical device, so there's the need to connect an Android or iOS phone to your development machine. Profiling on an emulator might not reflect the actual performances of the app so a real device is required.
- Profile mode is **similar** to release mode but they're absolutely **NOT** interchangeable. In addition, some services and/or features might not be available while profiling.

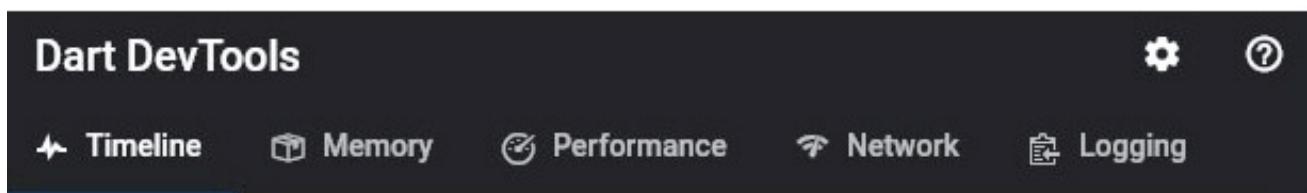
In practical terms, the profile mode exposes a series of graphs and indications about your app's performance. According with the results, the developer should be able to identify the potential bottlenecks and make accurate fixes. It's the last step before proceeding to the release phase.



Debugging happens while testing and developing. Profiling is done before releasing the product to make sure that everything is optimal or at least in an acceptable state.

16.2.1 DevTools

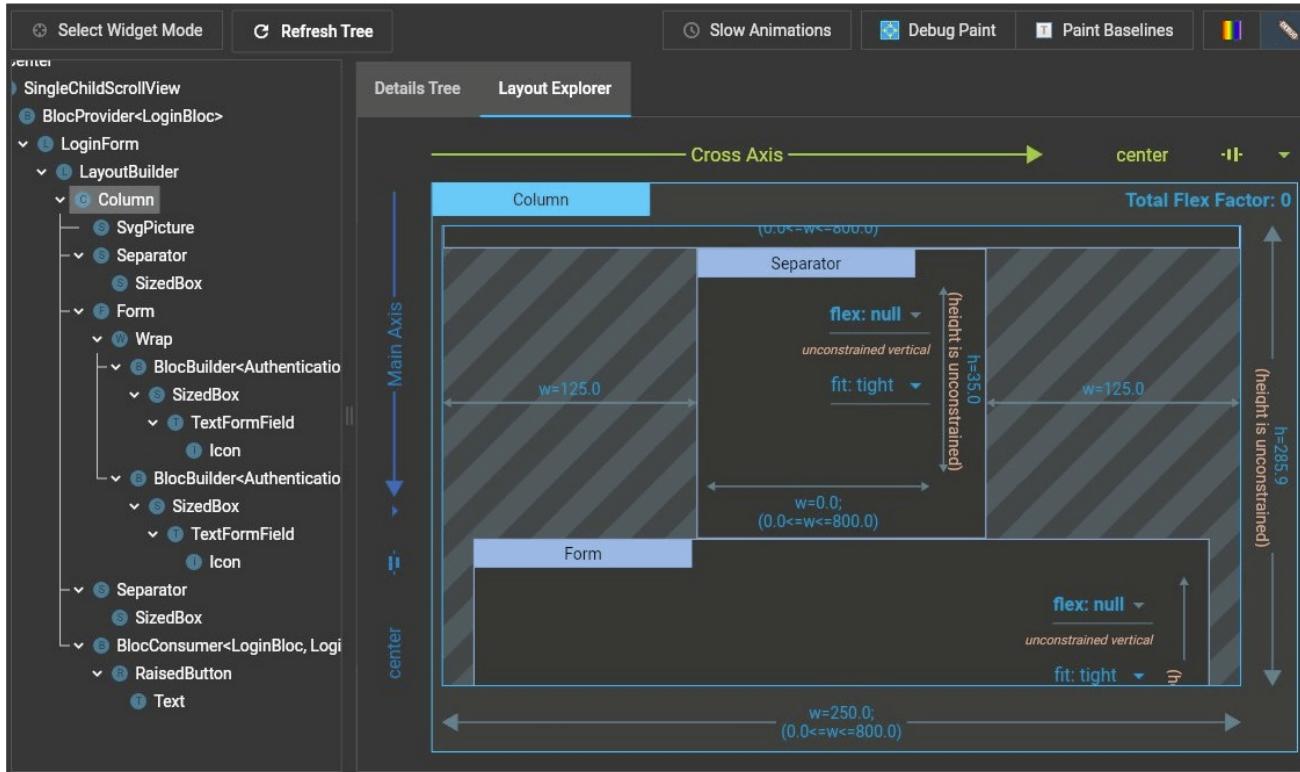
DevTools is a series of debugging and profiling tools for Dart and Flutter. It works in both debug and profile mode but when it's time to test the actual performances of your app, you should use it when profiling. It's like a Flutter-specific debugger with a lot of built-in functionalities. This is how it looks when you're testing your app's performances in **profile** mode:



It's also able to directly interact with the app performing rebuilds, showing grids and slowing down animations to inspect how they affect the performances. The first time you start DevTools, your IDE will automatically download it.

16.2.2 Using the Flutter inspector

The Flutter inspector is a powerful tool for exploring your app's widget tree: it allows you to see literally everything about the widgets you've created. The inspector can also be opened in Android Studio/Visual Studio code but the web version on DevTools is more spacious and user friendly. It can be used in **debug** mode.



In the above image we're analyzing a `Column` with the possibility to also change in real time its properties. For example, you could click on `center`, change it to `end` and see a preview of how the new configuration would look like. On the left, there's a representation of your app's widget tree which can easily be explored. In this page, there are many actions to perform:

- **Slow Animations.** Slows down animations to inspect whether the source of the slowness is a particular animation. It's very useful when you have a jank in the UI and you want to find out if the source of problems is a too expensive animation.
- **Debug Mode Banner.** Enables or disables the red "debug" stripe on the top-right border.
- **Select widget mode.** It allows you to click in a widget on the tree (on the left) to inspect it. You can for example see the associated keys, the sizes, the constraints of the widget itself and so on.
- **Debug Paint.** Adds debugging hints to better see spaces such as alignments, margins and paddings. It's useful when you want to see relative alignments of widgets to make sure they're properly placed on the screen.

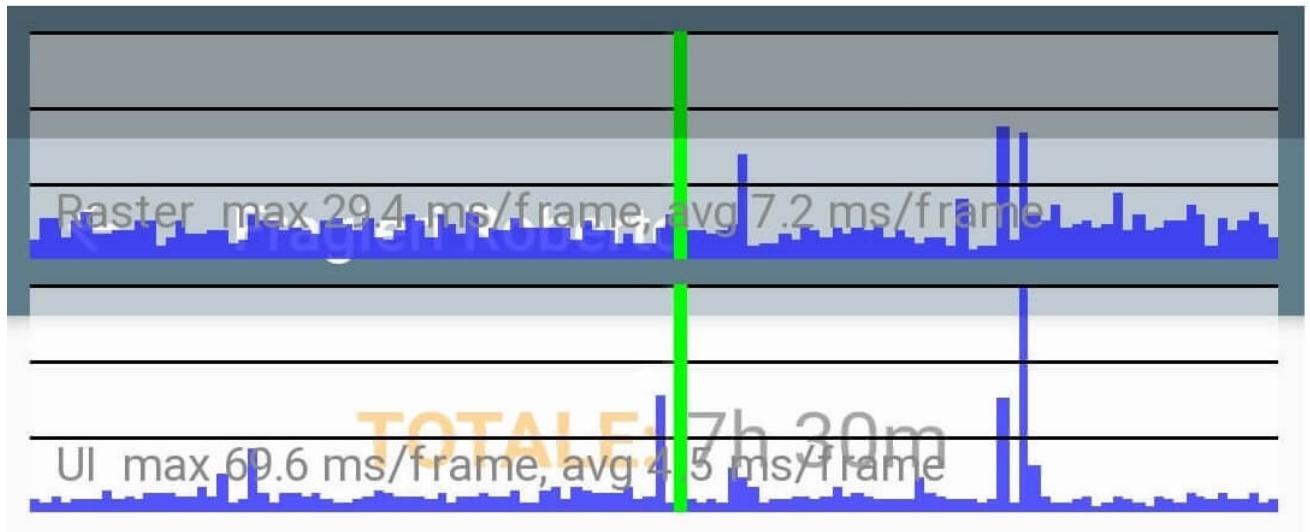
- **Paint Baselines.** Any RenderBox paints a line at each of its text baselines.

16.2.3 Using the Timeline view

This view is made up of multiple bars, each of them representing a single Flutter frame. Clicking on a bar opens a detailed view at the bottom with any single call made by the CPU.



There also is the possibility to show these bars directly in your app clicking on the button next to *Profile granularity*. They appear at the top of your application, when run in profile mode, with two important graphs. You don't have control on them as they aren't regular Flutter widgets; they're painted directly by Skia on top of your app. Both charts are identical: they show the same data but with a different layout.



The charts show the latest 300 frames produced by your app and they're updated whenever Flutter paints something to the UI, which is when `build()` is called. Each bar is a frame. If you want your app to run at 60 frames per second (fps), each frame should be built in about 16 milliseconds.

- ❶ 60fps means that each frame has to be rendered in 1/60th of a second which is about $1/60\text{s} = 0,01667\text{s}$ (16 milliseconds). A frame is said to be **janky** if it's taken more than 16ms to complete (and it's represented by a red bar).

The charts represent data about the raster thread and the UI thread, which are the relevant ones for the developer. Flutter also has an I/O thread, for time-consuming operations, and a platform thread which talks to the underlying OS.

- **UI thread:** here's where your code is executed in the Dart VM. Any action made in the UI thread has consequences on the other threads so it doesn't have to be blocked for a long time. In practical terms, using `async/await` and `const` constructors plays a fundamental role!
- **Raster thread:** Skia runs here. If this thread is slow it's a consequence of the fact that your Dart code (which runs in the UI thread) is also slow. The raster thread executes graphics code from the Flutter engine.

Let's jump to the practical part to see how to actually profile a Flutter application. Once you've connected the IDE to a real device and launched the profile mode, show the performance overlay and start profiling:

- Interact with your app as much as possible: press on buttons, navigate among routes, scroll repeatedly list and so on. Use different devices with different hardware specs if possible. Whenever you see a red bar (jank), start investigating to find the cause.
- The overlay could show red vertical bars indicating that a particular frame took too long to render (more than 16ms). If it happens in the UI thread, then you should review the Dart source code and improve it:
 - try to improve animations (if any) using the caching techniques we've shown in chapter 14 (especially, look at 14.2);
 - have you used `const` constructors? can you refactor to create more immutable widgets (so that they can have a constant constructor)? can you "manually" cache some portions of the tree?

Chapter 16. Testing and profiling apps

- be sure you're lazily-initializing `ListView`s and `GridView`s using their `builder` named constructor. It's very efficient.
- try to see if unnecessary rebuilds are happening. Is `FutureBuilder` using a non cached `Future`?

```
FutureBuider<String>(
  future: _MakeHttpRequest(),
)
```

Doing something like this is absolutely bad and we will explain, in detail, why in chapter 17. Basically, a `Future<T>` object should be cached and initialized inside `initState()` (or with late `final`).

- You can optimize a `ListView` by setting the `itemExtent` property, which is used to determine the height of the children.

```
ListView.builder(
  itemExtent: 75.5,
  itemCount: 350,
  itemBuilder: (context, index) {...}
)
```

In this way Flutter knows in advance the extent and this foreknowledge saves some work making the scrolling more efficient, especially when there are drastic position changes.

- Do you have the possibility to cache data using a `Map<K,V>`? Do you have the possibility to cache data on a local storage like a database? In this way, you could save a lot of computational time.

Red bars on the raster thread are a consequence of your Dart code being too "slow" and thus the app, on average, doesn't run at 60fps.

- It might happen that red bars appear only in the raster chart and not in the UI one. There are a few actions you can take in order to investigate the problem:
 - Some widgets such as `Opacity` are relatively expensive: if you're using them many times (especially inside animations) they could be a problem. If you're using them, try to remove them or reduce their usage and see what happens.
 - Click the Slow animation to run animations slower in the DevTool control panel. See if the problem is caused by the entire animation or only at certain points of the execution.

- Use DevTool to discover, in the TimelineView section, which part of your app is slow (more on this later).

In most of cases, red bars indicate that at certain frames the UI is rebuilt more than required so you should act on the Dart code. Generally, this behavior is caused by animations or `Future<T>`s.

-  Make apps whose frames are ready in less than 16ms (all blue bars) or at least try to stay at 16ms on average. Other than visual benefits you might not notice, low frame rendering leads to better battery life and less device heating.

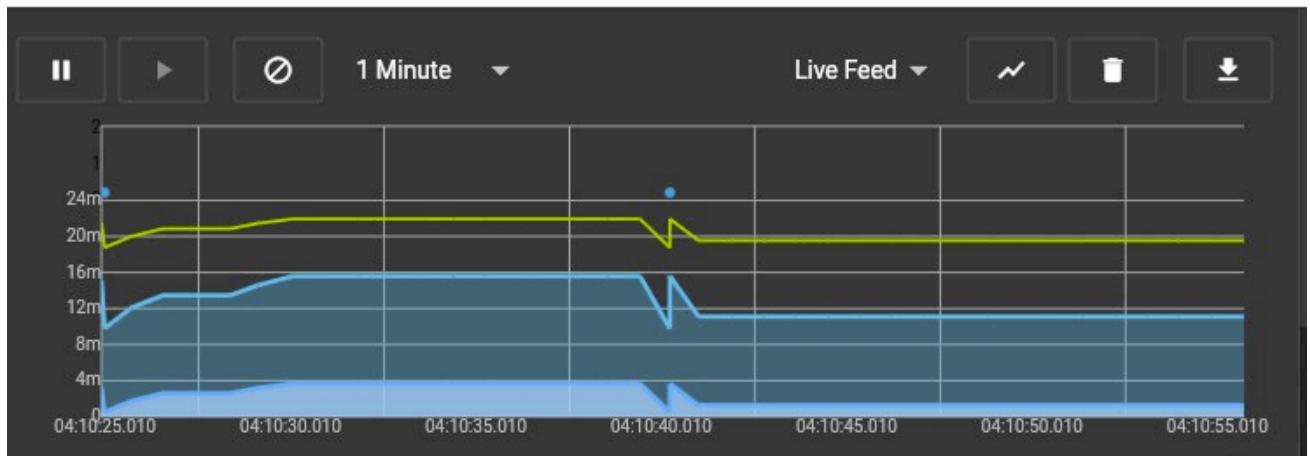
Don't profile your apps exclusively on powerful last generation devices. If your app renders good frames on older devices, whose hardware is not particularly excellent, on better devices results can only improve. You can always display the performance overlay programmatically inside material or cupertino widgets:

```
MaterialApp(  
    showPerformanceOverlay: true,  
    title: 'My App',  
)  
  
CupertinoApp(  
    showPerformanceOverlay: true,  
    title: 'My App',  
)
```

The performance overlay always works, but a meaningful usage only happens in profile mode (which is the closest to the release mode).

16.2.4 Using the Memory view

In Dart there's no need to take care of cleaning the memory from unused objects because there's an efficient garbage collector doing the job for us. Thanks to the *Memory View* tab you can see how an isolate is using the memory at a given moment.



This tab should be used only in **profile mode** as it's the "closest" to release mode in terms of performances. Despite being very accurate, profile mode might show sometimes some slightly higher values because the isolate is running a special service in order to run the profiling. This special service doesn't exist in release mode. There are two main parts on the page:

1. At the top, a chart shows the state of the heap at a certain point of the time. In other words, it's a general overview of the memory while the application is running having some dots and lines updating in real time. Dots represents **GC** events, which is when garbage collection happens.
2. At the bottom, there's the **Snapshot** button which takes a "photo" of the memory at the given instant and reports a detailed analysis of the heap status. Note that *external* refers to data put on the heap by the operating system and thus not managed by Flutter.

Thanks to this chart you could also find potential *memory leaks*, which are unused object lying on the heap wasting memory. They are hard to debug and the garbage collector isn't able to catch those "leaked objects": they waste space, put pressure on the VM and increase memory fragmentation. Finding memory leaks requires experience and skill (as it's not an easy job), but there are two buttons you can use to try spotting them:

- Clicking the **Reset** button in the top-right corner sets to 0 the total instances count of the classes.
- Clicking the **Snapshot** button in the bottom area of the page shows the list of currently active objects on the heap (along with an instance counter).

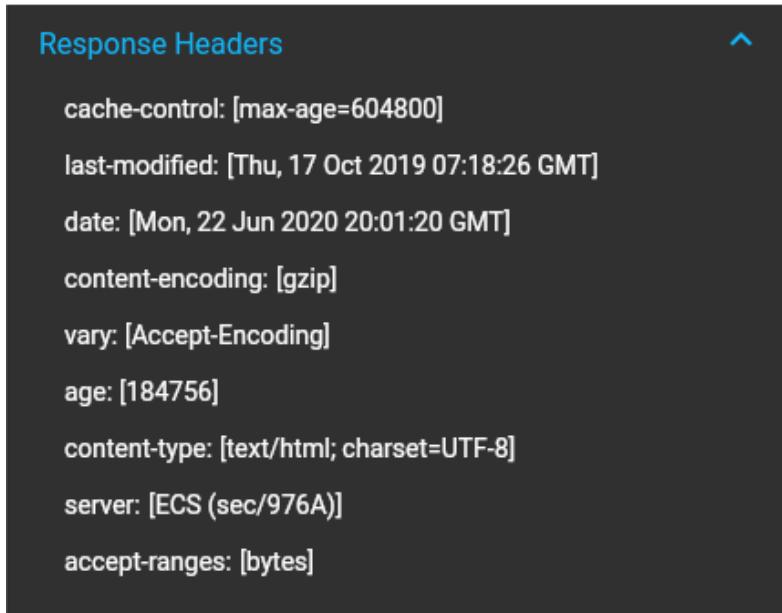
If you click on **Snapshot** after having pressed **Reset**, you'll be able to see how many new instances have been allocated on the heap since the last reset. This can be an useful strategy to find potential leaks, which is the case where there are object that cannot be reached anymore (and cannot be freed by the garbage collector).

-  Analyzing the memory to spot leaks and improve your app's performance is not easy at all. It requires a lot of experience and skills we cannot explain here because they go beyond the scope of this book.

Dart and Flutter themselves don't leak memory. You can leak memory when you forget to call `dispose()` on resources that require to be "cleared" after being used. As you already know from chapter 14, an `AnimationController` has to be disposed in order to avoid leaks.

16.2.5 Using the Network view

Use this tab to inspect HTTP network traffic from your Flutter (or Dart) application. By default the traffic is captured and logged in the big list on the left: clicking on a row, shows all the details about that particular HTTP event.



The screenshot shows the 'Response Headers' section of the Network tab in the Flutter DevTools. The headers listed are:

- cache-control: [max-age=604800]
- last-modified: [Thu, 17 Oct 2019 07:18:26 GMT]
- date: [Mon, 22 Jun 2020 20:01:20 GMT]
- content-encoding: [gzip]
- vary: [Accept-Encoding]
- age: [184756]
- content-type: [text/html; charset=UTF-8]
- server: [ECS (sec/976A)]
- accept-ranges: [bytes]

As you can see, there are a lot of very useful details about requests being sent. We strongly recommend you to use this tab rather than using `debugPrint()` or other logging techniques (see below) to deeply analyze HTTP requests.

16.2.6 Using the Logging view

As the name suggests, this tab simply consists of a logger showing events from Flutter, the Dart runtime and other logging events coming from your application. By default, the page will show you:

- Flutter framework events;
- events from the garbage collector;
- events sent from the application, such as the ones fired by `log()`

When we talk about "*events sent from the application*" we're referring to the fact that you can log messages (or error messages) from your Dart/Flutter apps while debugging. You have two ways to show messages in the logger tab of DevTools:

1. Use the `debugPrint("...")` from Flutter's foundation library. You could also use `print()` but if the string is too long, some lines might be ignored by Android. This doesn't happen if you use `debugPrint`.
2. While `debugPrint` is very simple (it just prints a string), `log` is more powerful as it's able to log more information about something you need to analyze.

```
import 'dart:developer' as developer;
import 'dart:convert';

String getName(BuildContext context) {

    // Getting some data using provider
    final userData = context.read<UserData>();

    // Logging the data
    developer.log(
        'Checking whether the name is correct',
        name: 'event_name',
        error: jsonEncode(userData)
    );
}
```

```
    return userData.name;  
}
```

Using `developer.log()` we can send a log event to the *Logging view* with many mode information. For example, `name` is the name of the source of the log message and `error` generally contains additional data to show along with the message.

```
String getName(BuildContext context) {  
    final userData = context.read<UserData>();  
  
    if (kDebugMode) {  
        developer.log(...);  
    }  
  
    return userData.name;
```

With the following approach, logs are being sent only in debug mode and thanks to tree shaking⁹ `developer.log()` is automatically removed in profile and release mode.

16.2.7 Monitoring widget rebuilds

In Android Studio, when the app is running in debug mode on a real device or emulator, you can easily track the rebuild statistics of any widget. In practice, this table tells you how many times any widget was rebuilt both in the last frame or since when the route has been opened.

⁹See chapter [TODO] about tree shaking

Widget rebuild stats				
Widget	Location	Last F...	Current ...	
_ListIItem	list_operatori.dart:67	1	13	
Consume	list_operatori.dart:117	1	13	
GestureD	list_operatori.dart:119	1	13	
Card	list_operatori.dart:129	1	13	
ListTile	list_operatori.dart:132	1	13	
Icon	list_operatori.dart:135	1	13	
Text	list_operatori.dart:134	1	13	
Icon	list_operatori.dart:133	1	13	
Icon	search_form.dart:46	0	1	
IconButto	search form.dart:45	0	1	

Tapping on the grey circle moves the code editor to the interested widget. With these interesting data you can also see how important `const` constructors are: without them the rebuild count will frequently increase. Instead, constant widgets are built only once and thus you'll see their counter being stuck at 1, which is good.

- When you see red bars in profile mode, run the app in debug mode and use this table to see which widgets rebuild quite often. Try to see if you can reduce calls to `build` by using constant constructors, caching data into fields or using some strategies we've listed above.
- Don't look at the *Frame rendering time* chart above because it shows your app's fps in debug mode, which is not accurate as you know.

PART III

PRACTICAL FLUTTER EXAMPLES

"The most important property of a program is whether it
accomplishes the intention of its user."

CHARLES A. R. HOARE

17 | Networking

17.1 Making HTTP requests

Nowadays almost any app uses internet to fetch data from an API, interact with Firebase or any other action that requires a Wi-Fi or wired connection. This chapter shows how to properly use Flutter to efficiently make HTTP requests to send/receive data from servers.

 Resources > Chapter 17 > HTTP Requests

17.1.1 GET requests

In this example we're connecting to *JSONPlaceholder*¹, an online API used for testing purposes. It's a free service at which you can send GET or POST requests and it returns various types of JSON-encoded strings as response.

```
dependencies:  
  http: ^0.12.2
```

Install the official http² package from the Dart team and follow these step-by-step indications to properly make HTTP requests with Flutter. For brevity, some small classes aren't shown but they're of course available in the *Resources* page of our website.

1. Our code has to be easily maintainable over the time and easy to read. Here's where the single responsibility principle and dependency injection come to the rescue.

```
// To be 'implemented' and not 'extended'
```

¹<https://jsonplaceholder.typicode.com>

²<https://pub.dev/packages/http>

```
abstract class HTTPRequest<T> {
    Future<T> execute();
}
```

This interface will be implemented by all those classes that perform HTTP requests. Other than respecting the SRP, this class will be injected in the UI widgets via constructor.

2. The online API we're connecting to returns a JSON string containing data of a test item. There's the need to implement `HTTPRequest<T>` to perform a GET request and returning a model class (`Item`) representing the received data.

```
import 'package:http/http.dart' as http;

class RequestItem implements HTTPRequest<Item> {
    final String url;
    const RequestItem({ required this.url });

    Future<Item> execute() async {
        // HTTP request
        final response = await http.get(url);

        if (response.statusCode != 200) {
            throw http.ClientException("Oh darn!");
        }

        // Use the model class to make a JSON-to-Item conversion
        return _parseJson(response.body);
    }

    Item _parseJson(String response) =>
        Item.fromJson(jsonDecode(response));
}
```

Thanks to the `http` package we can perform asynchronous GET or POST requests to get an object of type `Response` which exposes many useful properties:

- `body`: the body of the response as a string;
- `statusCode`: the HTTP status code of the request which can be 200, 404 or 500 for example;

- `contentLength`: the size, in bytes, of the response body;
- `headers`: the headers sent from the server in response to our request.

As you can imagine, `class Item` is a simple model that converts a JSON string into an object with the techniques we've covered in chapter 15. It's been put in a separated function just for readability purposes.

- When dealing with futures, like in the case of HTTP requests, there's the need to use stateful widgets in order to cache the `Future<T>` object and use it only once. Let's start with the widget itself:

```
class HTTPWidget extends StatefulWidget {
    final HTTPRequest<Item> _request;
    const HTTPWidget(this._request);

    @override
    _HTTPWidgetState createState() => _HTTPWidgetState();
}
```

The HTTP request is going to return a JSON object represented by the `Item` class so we're requesting for that type in the constructor. Actually, there would be a better way to do this and in 17.3 we will show how to implement a more complete and flexible version of `HTTPRequest<T>`.

```
class _HTTPWidgetState extends State<HTTPWidget> {
    late final Future<Item> futureItem;

    @override
    void initState() {
        super.initState();
        futureItem = widget._request.execute();
    }

    // ...
}
```

Since `initState` executes only once, the API call triggered by `execute()` is also executed only once. What is absolutely **WRONG** is fetching HTTP data, or triggering any other method, inside `Widget build()`:

```
class _HTTPWidgetState extends State<HTTPWidget> {
```

```
  @override
  Widget build(BuildContext context) {
    // This is absolutely BAD
    final futureItem = widget._request.execute();

    // ...
  }
}
```

Flutter calls `build` more than once and you can't predict how many times; with the above code, you're making an HTTP request at every rebuild of the widget.

- ➊ Even if you don't see them, rebuilds happen more often than you think. Putting API calls inside a `build` method is a bad idea as it wastes resources and performs unneeded HTTP request repeatedly.

With a `late final` member variable you can assign a value only once inside `setState` and then, in case of a rebuild, the API call won't be executed again.

4. Now that we have learnt how to "cache" the request, it's time to use the `FutureBuilder<T>` widget. Very simply, you give it a `Future<T>` and it notifies you when it's completed.

```
class _HTTPWidgetState extends State<HTTPWidget> {
  // ... variables and initState

  @override
  Widget build(BuildContext context) {
    return FutureBuilder<Item>(
      future: futureItem,
      builder: (context, snapshot) {
        if (snapshot.hasError) {
          return const ErrorItemWidget();
        }

        if (snapshot.hasData) {
          return SuccessItemWidget(snapshot.data);
        }
      }
    );
}
```

```
        return const Center(
            child: CircularProgressIndicator(),
        );
    }
);
```

Thanks to `FutureBuilder<T>` you can handle the waiting time, the failure and the success of a `Future<T>` very easily. The variable `AsyncSnapshot<T> snapshot` gives you information about the status of the future:

- (a) `hasError`: if an exception occurred while executing the `Future<T>`, this property evaluates to `true`. A proper widget appears to the screen indicating that something has gone wrong while performing the API call.
 - (b) `hasData`: it's `false` by default but when the `Future<T>` completes with no errors it becomes `true`. This is a signal that data have been successfully received and they're ready to appear. Note that `snapshot.data` returns a nullable value.

The final `return` statement shows a waiting spinner at the center of the screen. It appears while the `Future<T>` is still executing; it is very important because it lets the user visually know that some processing is going on in the background.

- To use this widget, you first must create an instance of `HTTPRequest<T>`, outside the `build` method, and then pass the dependency to the widget.

```
static const _url = "https://jsonplaceholder.typicode.com/posts/10";

@Override
Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(
                title: const Text("Request Demo"),
            ),
            body: const Center(
                child: HTTPWidget(
                    RequestItem(
                        url: _url
                    )
                )
            )
        )
    );
}
```

```

        )
),
),
);
}

```

Notice how we've tried to use `const` constructors as much as possible, especially inside the builder of `FutureBuilder<T>` where rebuilds happen very frequently. Caching widgets or using constant constructors is always important.

17.1.2 POST requests and headers

Making a POST request is no different from a GET but in addition you have to provide a `body` of course, which is the payload being sent to the server.

```

final response = await http.post(url,
  body: "send this string via POST",
);

```

You can set `encoding: Encoding.getByName("utf-8")` by passing a lower-case version of the standard charsets names³. The `body` can be of three different types:

- `String`: the content-type of the request is automatically set to `text/plain`;
- `List<T>`: it's used as a list of bytes for the body of the request;
- `Map<K, V>`: it's treated like if it contained form fields and it automatically sets the content-type to `application/x-www-form-urlencoded`

When you do a POST request the returned object is a `Future<Response>`. For both `get()` and `post()` you have the possibility to set headers for the HTTP request; they're simply implemented as a map where both keys and values are strings:

```

// 'http.post(...)' it works in the same way
final response = await http.get(url,
  headers: {
    "Authorization": "your_api_key",
  }
);

```

³<http://www.iana.org/assignments/character-sets/character-sets.xml>

An API might require incoming requests to contain a special header called "**Authorization**" with a password or a long key. For any request header, add a new entry in the map associated to the `headers` parameter.

 Note that "**Authorization**" is the primary header used by clients to authenticate but it's not mandatory. The server might request it under a different name such as "**x-api-key**" or whatever else so be sure to check the documentation before sending the key. The name may vary.

The `Response` object, which is what the request returns, contains the headers returned by the server. It's a `Map<String, String>` so it's easily accessible using the `[]` operator:

```
final response = await http.get(url);

final headers = response.headers; // map of strings
final contentLength = response.contentLength;
final statusCode = response.statusCode;
```

17.1.3 Good practices

The most important thing to keep in mind is the initialization of a `Future<T>` inside the `initState` method so that each rebuild won't trigger an HTTP request. This is also recommended by the Flutter team ⁴ in one of their online examples available in the digital cookbook.

- The best way to wait for a `Future` to complete is the usage of a `FutureBuilder<T>` widget. It allows you to show a loading indicator while waiting and a very easy handling of the request status (running, completed or failed).
- The `builder` function of a `FutureBuilder<T>` class is called many times and thus the usage of `const` constructors is fundamental. Always remember to use them if possible.

When you have to execute a single request, calling `http.get(...)` or `http.post(...)` is enough. However, if you're going to make multiple requests to the same server consider using an instance of an `http.Client`.

```
final client = http.Client();

try {
```

⁴<https://flutter.dev/docs/cookbook/networking/fetch-data#why-is-fetchalbum-called-in-initstate>

```
final one = await client.get(...);
final two = await client.get(...);
final three = await client.post(...);

// do something else...
} finally {
  client.close();
}
```

Keeping an open connection is more efficient than opening and closing single request multiple times. There is less overhead but it's more "*wasteful*" in terms of internet resources so when you're done with all of them, close the client immediately.

```
final response1 = await http.get(url);
final response2 = await http.post(url);
```

Both `get` and `post` methods of the `http` package are global scope functions you can call straight away. If you look at their internals, they instantiate a client which shoots a single request:

```
// This is how 'get(...)' and 'post(...)' internally work. These functions
// are just a shorthand for the creation of a Client object!
var client = Client();
try {
  // 'fn' is declared and used internally
  return await fn(client);
} finally {
  client.close();
}
```

If you wanted to make a POST request, for example, you could either choose to use the convenient method `http.post(...)` or create a `http.Client()` instance. It would be the same. Since there's no need to add boilerplate code to your logic, in case of single requests use the global methods rather than a client.

```
final url = "https://mywebsite.com/posts/profile?id=1&pages=30";

final response = await http.get(url);
```

The url can directly contain query parameters and the request will execute successfully. If you want to have more control over the string, you have to create an `URI` object and pass it to the method.

```
// Use 'var' if you plan to change this Map later
final params = {
    "id": 1,
    "pages": 30,
};

// Domain and path are required in an URI object
final domain = "mywebsite.com";
final path = "/posts/profile";

final uri = Uri.https(domain, path, params);

// Pass headers, if any, using the 'header' named optional param
await http.get(uri);
```

This is more flexible but we've had to write quite a bit more of boilerplate code than what we'd expect. Notice that `Uri` defines `http` and `https` factories and they are of course **not** interchangeable.

17.2 Working with data

The `http` package can also perform I/O operations but dealing with them is actually not so easy. You'd have to work with bytes, encodings, `File` objects and other things that are a bit too "low level".

i It's not a matter of efficiency because `http` does its job very well. The problem is that, for example, there's no `http.downloadFile()` and so you'd have to create it by yourself. It doesn't have many pre-made functions ready to use.

There is a very powerful package called `dio`⁵ which provides an easy way to deal with files download/upload and, of course, many other networking tasks such as HTTP requests. We recommend to follow this guideline:

- If your app is only going to make HTTP requests, choose between the `http` or `dio` packages. They both give you an intuitive and easy way to deal with requests, just pick the one you

⁵<https://pub.dev/packages/dio>

like more. Both are very efficient.

- If your app is going to perform networking operations with data, such as downloads or uploads, we recommend to stick with *dio* as it's simpler to use. It has a lot of pre-made and useful functions that we're going to cover in this section.

As we've already said, with *dio* you can easily make GET and POST requests which return a `Response` object, similarly to what *http* does. There is no need to explain what's going on because the code is self-explanatory (this is great in terms of maintenance):

```
void main() async {
    final dio = Dio();
    final url = "https://website.com";

    try {
        final response1 = await dio.get<String>(url);
        print("${response1.data}");

        final response2 = await dio.post<String>(url, data: {
            "key1": "1",
            "key2": 2,
        });
        print("${response2.data}");
    } on DioError catch (e) {
        print(e);
    }
}
```

Very intuitively, `queryParameters` offers a convenient way to pass query params using a `Map<K, V>`. In *http* this process is a bit more tedious because there's the need for a separated `URI` object which produces a good amount of boilerplate code.

```
final response1 = await dio.get<String>(url, queryParameters: {
    "param1": 1,
    "param2": "2"
});
```

You can also execute multiple requests concurrently and wait for all of them to finish. In this case, instead of repeating the same base URL for every request, just set it up once in a new instance of `BaseOptions` and then make calls with relative paths:

```
// The timeout is expressed in milliseconds
final options = BaseOptions(
    baseUrl: "https://website.com/api/",
    connectTimeout: 3000, // 3 seconds
);

// New instance of dio with the given options
final dio = Dio(options);

// Executes 3 requests concurrently and waits for all of them
// to complete
final response = await Future.wait([
    dio.get<String>("/version"),
    dio.get<String>("/products/list"),
    dio.post<String>("/login",
        options: Options(
            headers: {
                "Authentication": "auth-key",
            }
        )
    ),
]);

```

Thanks to the `BaseOptions` object the `get("/version")` method is actually calling the extended version of the URL (`get("https://website.com/api/version")`) because the value of `baseUrl` is put in front of the URL endpoint.

 Resources > Chapter 17 > Files download

17.2.1 Downloading data

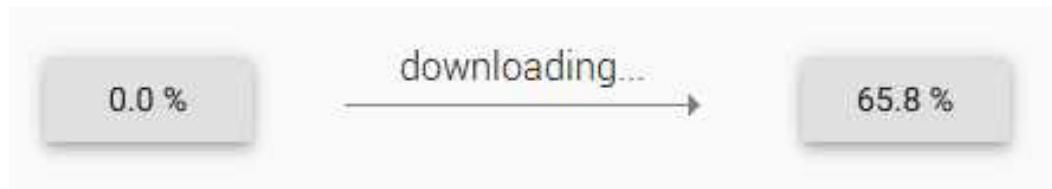
We're now going to see how to create an app that downloads a file from a sever and stores it on the device. It's going to have a button to start the download and a `Text` to show the completion percentage of the operation. As bytes are received, the percentage increases to reflect the progression of the download.

```
dependencies:  
  dio: ^3.0.10  
  provider: ^4.3.2+1  
  path_provider: ^1.6.14
```

The `path_provider` package is maintained by the official Flutter team and it provides a series of utilities to easily get paths to common directories on various operating systems. Just `await` one of its methods, which return a `Directory` object, and use the `path` getter to retrieve the location.

```
final tempDir = await getTemporaryDirectory();  
final extDir = await getExternalStorageDirectory();  
final downloadDir = await getDownloadsDirectory();  
  
// and more...
```

Our app is going to use the temporary directory of the device but of course it's just for the sake of the example, you can use any directory you want. The UI is going to look like this:



In order to change the text of the button there's the need to create a `ChangeNotifierProvider` which will update the percentage while the download is running. `DownloadProgress` is a *model* class encapsulating the network I/O logic so it's a good idea placing it in a dedicated file.

```
class DownloadProgress with ChangeNotifier {  
  // Initial value  
  var _progress = 0.0;  
  double get progress => _progress;  
  
  void start({  
    required String url,  
    required String filename  
}) async {
```

```

    // download logic...
}

void _resetProgress() {
    if (progress != 0) {
        _progress = 0;
        notifyListeners();
    }
}

void _updateProgress(double value) {
    _progress = value;
    notifyListeners();
}

}

```

The class exposes only the `start` method, which starts downloading the given file to a particular location on the device, and the `progress` getter, which indicates the completion percentage of the download.

```

void start(
    required String url,
    required String filename
) async {
    // Reset the percentage in case it isn't at zero
    _resetProgress();

    // Path and name
    final directory = await getTemporaryDirectory();
    final pathName = "${directory.path}/$filename";

    // Download
    await Dio().download(url, pathName,
        options: Options(
            headers: {
                HttpHeaders.acceptEncodingHeader: "*"
            }
        ),
)

```

```
        onReceiveProgress: (received, total) {
            if (total != -1) {
                // The percentage of the received bytes over
                // the total size of the file being downloaded
                var pos = received / total * 100;
                _updateProgress(pos);
            }
        );
    };
}
```

In certain cases it might happen that the total file size is not available due to the *gzip* compression; to avoid this problem set the `accept-encoding` header to `"*"`. If the dimension of the downloaded file is not available, `total` will be set to `-1` so always be sure to make a check.

```
@override
Widget build(BuildContext context) {
    return ChangeNotifierProvider<DownloadProgress>(
        create: (_) => DownloadProgress(),
        child: MaterialApp(
            home: Scaffold(
                appBar: AppBar(
                    title: const Text("Demo Download");
                )
                body: const Center(
                    child: DownloadWidget(),
                )
            )
        ),
    );
}
```

The notifier has to be placed right above the widget listening for progress updates; in this simple application there is a single screen so it just wraps `MaterialApp`. Be sure to use a `Consumer` so that only the button will be rebuilt rather than the whole subtree.

```
class DownloadWidget extends StatelessWidget {
    final String url = "https://website.com/files/test.pdf";
    const DownloadWidget();
```

```
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Consumer<DownloadProgress>(
        builder: (context, status, _) {
          var progress = status.progress.toStringAsFixed(1);

          return RaisedButton(
            child: Text("$progress %"),
            onPressed: () => status.start(
              url: url,
              localPath: "myfile.pdf"
            ),
          );
        },
      ),
    );
  }
}
```

If the download fails for any reason, *dio* automatically stops the download and deletes the contents that have been downloaded up to that point.

17.2.2 Uploading data

In certain cases it might happen you had the need to send form data to the server via POST request to a particular URL. For example, one of the most common scenario for this situation is the case in which you have to fill an HTML form.

```
<form method="post" action="/admin/adduser" enctype="multipart/form-data">
  <input type="text" name="nickname" />
  <input type="file" name="avatar" accept="image/jpeg" />

  <input type="submit" value="Upload" />
</form>
```

When clicking *Upload* on the browser, the client sends a request to the server passing a string and an image. With *dio* it's possible doing the same thing in just a few lines of code; it sends a POST request to an endpoint with the given payload.

```
// Path to the file in the device
final fileDir = await getTemporaryDirectory();
final filePath = "${fileDir.path}/myfile.txt",

// "Fill the form" by passing the data
final payload = FormData.fromMap({
  "nickname": "Roberto",
  "file": await MultipartFile.fromFile(filePath),
});

// Send the request with the payload
await dio.post<String>("/admin/adduser", data: payload);
```

The server expects a POST request with `"nickname"` and `"file"` contents; both the HTML page and the Flutter app are sending the data in the same way. Of course you can also send multiple files at once in the same request:

```
final payload = FormData.fromMap({
  "nickname": "Roberto",
  "moreFiles": [
    await MultipartFile.fromFile(filePath1),
    await MultipartFile.fromFile(filePath2),
    await MultipartFile.fromFile(filePath3),
  ]
});
```

With `FormData` the default encoding is `"multipart/form-data"` so that your request can also send files. If you want another type of encoding, just specify it in the options of the request:

```
await dio.post<String>("/admin/adduser", data: payload,
  options: Options(
    contentType: Headers.formUrlEncodedContentType
  ),
);
```

Like we've done for the download, there's the possibility to show the completion percentage of the upload using a callback that provides the total size of the payload and the current amount of bytes sent. The setup is very similar to what we've done in the previous section:

- Create a `class UploadProgress with ChangeNotifier` which will update our UI with the upload completion percentage. It's basically the same code of the download example

with just the difference that we're using a `post()` rather than a `get()`.

```
class UploadProgress with ChangeNotifier {
    var _progress = 0.0;
    double get progress => _progress;

    void start({
        required String url,
        required String filename
    }) async {
        // upload logic...
    }

    void _resetProgress() {
        if (progress != 0) {
            _progress = 0;
            notifyListeners();
        }
    }

    void _updateProgress(double value) {
        _progress = value;
        notifyListeners();
    }

}
```

- Call the usual `post()` method but in addition listen to completion percentage changes by giving a value to `onSendProgress`.

```
void start({
    required String url,
    required String filename
}) async {
    // See the code of the previous example.
    // Here we reset the progress and get the path to the file

    await Dio().post<String>(url, fileName,
        onSendProgress: (sent, total) {
```

```
        vas pos = sent / total * 100;
        _updateProgress(pos);
    }
);
}
```

- At this point use a `Consumer<UploadProgress>()` in your UI exactly as we did in the previous example to refresh the percentage in the button.

This library has a consistent API which is easy to use and full of useful utilities to conveniently convert data formats or construct objects in a few lines. Another demonstration of this is the fact that you've multiple ways to pass a file to the payload:

- `MultipartFile.fromBytes(...)`: takes as input a `List<int>` which is the series of bytes representing the file;
- `MultipartFile.fromFile(...)`: extracts the file's contents from the given location on the device;
- `MultipartFile.fromString(...)`: converts a string into bytes using the default UTF-8 charset

17.2.3 Good practices

We want to point out again that this library isn't better than `http` for performance reasons or whatever else because both are very good packages. The biggest advantage in favor of `dio` is the vastness of its API and the ease of use. For example, downloading a file is just a matter of calling `download()` while `http` doesn't have this feature, you'd have to implement it by yourself.

- When you want to listen for upload or download percentages, always rely on `provider`, `flutter_bloc` or another state management library. Avoid the usage of `setState` or, if it's really needed (is it?) use it together with `InheritedWidget`.
- For single requests that don't need special configurations, you can create an in-place object and use one of its methods immediately.

```
final result = await Dio().get<T>(...);
```

This is not always convenient; for example, if you're connecting to an API that requires a token you'd better create an instance of `Dio` with `BaseOptions` and store the setting in it. In this way any method will inherit the configurations by default.

```
final options = BaseOptions(
  baseUrl: "https://www.api.website.com/v3/",
  connectTimeout: 4000,
  receiveTimeout: 3000,
  headers: {
    "api-key": "value",
    "something": 5,
  }
);

final dio = Dio(options);

// These request will carry the 'api-key' and 'something'
// headers along with all the other settings
await dio.get<T>(...);
await dio.post<T>(...);
await dio.download(...);
```

If you didn't do this, you'd have to specify the configuration for each single request and of course this is not handy at all.

- When you have to deal with file downloading, advanced POST requests and/or progress completion status, consider using *dio*. You've seen how easy it is to use in such cases; you could do it with *http* as well but you'd have to do most of the work by yourself.
- Look for Reso Coder's online video ⁶ about Dio and *interceptors*, a sort of listeners that run every time an action is performed. It will show you how to easily implement retries in case of connection failures.

 Resources > Chapter 17 > Advanced HTTP

17.3 Advanced REST API calls

Unless you're using Firebase, which will be covered in detail in chapter 22, apps very often need to communicate with a JSON REST service. In this section we're illustrating a possible imple-

⁶Dio Connectivity Retry Interceptor - Flutter Tutorial

mentation of efficient data fetching and parsing from an online API.

- i** In the following example we're working with JSON but of course any other format is fine. Nowadays most of the REST services work with JSON but, for example, you still might encounter a SOAP service which uses XML.

We're going to receive data from <https://jsonplaceholder.typicode.com>, a fake online REST API service for testing purposes. Before starting, we need to create a good folder structure for the HTTP client which might look like this:

```
lib/
  api/
    json_models/
    json_parsers/
    http_client.dart
  main.dart
  routes.dart
```

Let's see how the logic has been splitted into multiple files and folders.

17.3.1 Model classes

Inside `json_models/` we're going to place all those model files required to parse a json-encoded string into a Dart object, using code generation. For example, the endpoint `"/posts/10"` (it could have been any other number) returns this kind of response:

```
{
  "userId": 1,
  "id": 10,
  "title": "abc",
  "body": "abc def"
}
```

As we've seen in chapter 15, thanks to code generation, we can easily create a model class for this JSON string. The code is located in `json_models/post.dart`

```
part 'post.g.dart';

@JsonSerializable()
```

```
class Post {  
    final int userId;  
    final int id;  
    final String title;  
    final String body;  
    const Post(this.userId, this.id, this.title, this.body);  
  
    factory Post.fromJson(Map<String, dynamic> json) =>  
        _$PostFromJson(json);  
  
    Map<String, dynamic> toJson() => _$PostToJson(this);  
}
```

Similarly, inside `json_models/todos.dart`, we're creating another model class to deal with a list of todo items having the following structure:

```
[  
  {  
    "userId": 1,  
    "id": 1,  
    "title": "delectus aut autem",  
    "completed": false  
  },  
  {  
    "userId": 1,  
    "id": 2,  
    "title": "quis ut nam facilis et officia qui",  
    "completed": false  
  },  
]
```

Since the returned JSON is an array, and not an object containing an array, you just need to create a model for the inner object. The array will be manually converted.

```
part 'todo.g.dart';  
  
@JsonSerializable()  
class Todo {  
    ...  
}
```

Note that if the response were a bit different, maybe something like this...

```
{  
  "data": [  
    {  
      "userId": 1,  
      "id": 1,  
      "title": "delectus aut autem",  
      "completed": false  
    },  
  ]  
}
```

... you would have to create two model classes. The first one for the "outer" object, the one with the list, and the second one to handle the contents of the list:

```
part 'todo.g.dart';  
  
@JsonSerializable(explicitToJson: true)  
class TodoObject {  
  final List<Todo> data;  
  const TodoObject(this.data);  
  
  factory TodoObject.fromJson(Map<String, dynamic> json) => ...  
  Map<String, dynamic> toJson() => ...  
}  
  
@JsonSerializable()  
class Todo {  
  ...  
}
```

17.3.2 Parsing JSON

Once the HTTP request has completed with success, as you know, we need to convert the JSON string into a convenient Dart object. We should think about a good way of doing it because there'd be the need to:

- keep the maintenance easy so that the addition or removal of JSON parsers shouldn't affect any other part of the code;

Chapter 17. Networking

- classes should be small, concise and to the point;
- there shouldn't be strong dependencies in the hierarchy.

In other words, our architecture should respect the SOLID principles. Let's start with a good setup for a base class that will be used by parsers:

```
// lib/api/json_parsers/json_parser.dart
library json_parser;

export "post_parser.dart";
export "todo_parser.dart";
export "object_decoder.dart";

abstract class JsonParser<T> {
    const JsonParser();

    Future<T> parseFromJson(String json);
}
```

This kind of structure is the same you'd use to create Dart/Flutter libraries, as we will see in detail in chapter 23. We're creating the "json_parser" library and, at the same time, exporting the various parsers we need to translate JSON strings into Dart objects.

i Thanks to the `export` keyword we're able to import together with the `json_parser.dart` file also other files. This is very useful because you can do the following:

```
import 'package:myapp/api/json_parsers/json_parser.dart';

final todo = TodoParser();
final post = PostParser();
```

There's only one `import` directive to be able to use any kind of parsers. If you didn't use `export`, you had instead to import classes one by one:

```
import 'package:myapp/api/json_parsers/todo_parser.dart';
import 'package:myapp/api/json_parsers/post_parser.dart';

final todo = TodoParser();
final post = PostParser();
```

This `export`-way of writing libraries is exactly what the Flutter team suggests when it comes to writing package. This technique is also well described in 23.1 where we build a Dart/Flutter package from scratch.

We have created some convenient `mixins`. We constrained the mixin (on `JsonParser<T>`) so that only its subtypes will be able to use those methods.

```
mixin ObjectDecoder<T> on JsonParser<T> {
    Map<String, dynamic> decodeJsonObject(String json) =>
        jsonDecode(json) as Map<String, dynamic>;
}

mixin ListDecoder<T> on JsonParser<T> {
    List<dynamic> decodeJsonList(String json) =>
        jsonDecode(json) as List<dynamic>;
}
```

Since our parsers are often going to use `jsonDecode()`, we want to avoid code duplication among classes and so a `mixin` is what we're looking for. Here's the implementation of the various parsers:

```
// lib/api/json_parsers/post_parser.dart
class PostParser extends JsonParser<Post> with ObjectDecoder<Post> {
    const PostParser();

    @override
    Future<Post> parseFromJson(String json) async {
        final decoded = decodeJsonObject(json);
        return Post.fromJson(decoded);
    }
}

// lib/api/json_parsers/todo_parser.dart
class TodoParser extends JsonParser<List<Todo>>
    with ListDecoder<List<Todo>> {
    const TodoParser();

    @override
```

```
Future<List<Todo>> parseFromJson(String json) async {
    return decodeJsonList(json)
        .map((value) => Todo.fromJson(value as Map<String, dynamic>))
        .toList();
}
```

In the future you may have the need to parse data from the `"/info"` endpoint, for example. No problem! Create the `class Info` model and then add a new implementation of `JsonParser<T>`:

```
class InfoParser extends JsonParser<Info> with ObjectDecoder<Info> {
    @override
    Future<List<Album>> parseFromJson(String json) async {}
```

Adding new features doesn't involve touching existing code but just creating new classes (open-closed principle) and each class is specific for a single type of parsing (single responsibility principle).

17.3.3 HTTP Client

Now that we can parse JSON strings and represent them as Dart objects, there's the need to make the actual request. The `http` package is fine but `dio` is more customizable and easier to use on large scale:

```
// lib/api/http_client.dart
class RequestREST {
    final String endpoint;
    final Map<String, String> data;

    const RequestREST({
        required this.endpoint,
        this.data = const {},
    });

    /// HTTP dio client
    static final _client = Dio(
        BaseOptions(
            baseUrl: "https://jsonplaceholder.typicode.com/",
            connectTimeout: 3000, // 3 seconds
```

```
        receiveTimeout: 3000, // 3 seconds
        headers: <String, String>{
            "api-key": "add_one_if_needed",
        },
    ),
);
}

Future<T> executeGet<T>(JsonParser<T> parser) async {...}

Future<T> executePost<T>(JsonParser<T> parser) async {...}
}
```

We've decided to make the client **static** because those settings will always be the same, so no need to create a new Dio instance for each request. In the constructor we're asking for two parameters:

- **String endpoint**: the API endpoint at which the request has to be sent;
- **Map<String, String> data**: it could hold query parameters for a GET request or the payload of a POST. It's up to you and it could also be removed if not needed.

The only public methods are **executeX<T>**: they're called in order to make the actual request. Note how the parser is asked via **method injection** rather than being directly hard-coded inside the class.

```
Future<T> executeGet<T>(JsonParser<T> parser) async {
    final response = await _client.get<String>(endpoint);
    return parser.parseFromJson(response.data);
}

Future<T> executePost<T>(JsonParser<T> parser) async {
    final formData = FormData.fromMap(data);
    final response = await _client.post<String>(
        endpoint,
        data: formData,
    );

    return parser.parseFromJson(response.data);
}
```

Thanks to generics the code is not tied to a specific model class; both methods can be called on

any type of object. The parser is passed from the outside so that the methods know nothing about parsing details, which is not their business.

- ❶ If your app connects to different APIs you shouldn't make the internal client of the request class `static`. Instead, the client should come from the outside via dependency injection:

```
class RequestREST {
    final Dio client;
    final String endpoint;
    final Map<String, String> data;

    const RequestREST({
        required this.client,
        required this.endpoint,
        this.data = {},
    });
}
```

In this way, there's the need to pass a client with the given settings for each request (endpoint, timeouts etc).

You could make requests in a bloc or in a `FutureBuilder<T>`, caching the future as you've learnt, but they're always performed in the same way. After having instantiated a `RequestREST` instance, make a POST or a GET passing the proper parser for the data.

```
// Example of a request in the state of a StatefulWidget
late final Future<List<Todo>> todos;

@Override
void initState() {
    super.initState();

    todos = RequestREST(endpoint: "/todos")
        .executeGet<List<Todo>>(const TodoParser());
}
```

Thanks to this setup you can easily write requests: for the simplest cases, you just need to assign

an endpoint and pass an instance of the JSON parser. The UI is going to use a `FutureBuilder<T>` with the classic setup:

```
late final Future<List<Todo>> todos;

@Override
void initState() {
    super.initState();
    todos = RequestREST(endpoint: "/todos")
        .executeGet<List<Todo>>(const TodoParser());
}

@Override
Widget build(BuildContext context) {
    return FutureBuilder<List<Todo>>(
        future: todos,
        builder: (context, snapshot) {
            if (snapshot.hasData) {
                // Remember that 'snapshot.data' returns a nullable
                final data = snapshot.data ?? [];
                return ListView.builder(
                    itemCount: data.length,
                    itemBuilder: (context, index) { ... },
                );
            }

            if (snapshot.hasError) {
                return const ErrorWidget();
            }

            return const Center(
                child: CircularProgressIndicator(),
            );
        },
    );
}
```

In case `snapshot.data` was `null`, we could also have shown the error widget. We have provided

Chapter 17. Networking

an empty list as default value just for the sake of the example, but of course you have multiple ways to handle the nullability.

```
final data = snapshot.data;

if (data != null) {
    return ListView.builder(
        itemCount: data.length,
        itemBuilder: (context, index) { ... },
    );
} else {
    return const ErrorWidget();
}
```

18 | Assets, images and multimedia

18.1 Assets and images

In Flutter you have the possibility to add a series of resources to your app, technically known as **assets**, like images and videos. Assets are bundled with the binary file in order to be used at runtime; there are no restrictions on the file formats:

- images of various formats such as JPG, PNG or GIF;
- videos;
- textual files such as those with `txt`, `xml` or `json` extension;
- databases such as SQLite files;
- any other kind of static data.

In order to bundle an asset within your app, and thus being able to use it in the code, it has to be declared in the `pubspec.yaml` file. For example if you wrote this...

```
flutter:  
  assets:  
    - myassets/
```

... it would mean that Flutter will look for assets in a folder called `myassets`. Paths are relative to the root of the project so writing `myassets/` implies that a directory called `myassets` is located at the same level as the `pubspec.yaml` file.

```
flutter:  
  assets:  
    - myassets/logo.png  
    - myassets/client_config.json
```

Chapter 18. Assets, images and multimedia

It's also possible selectively including certain files to the app; in this way only those assets you've specified will be bundled while the others will simply be ignored. From your code, you can reference assets using `class AssetBundle` which provides two important methods:

- `loadString(String path)`: loads a text asset;
- `load(String path)`: loads a binary asset which can be an image or any other type of file;

Again, the path is relative to the root of the project and it doesn't have to start with a slash. Let's say your project were made up of the following files and directories:

```
lib/  
  myassets/  
    file.txt  
  pubspec.yaml
```

Flutter would access the asset using `load("myassets/file.txt")` and not `load("file.txt")`. You have to specify the path from the root to the file because directories are not implicitly deducted at build time.

```
import 'package:flutter/services.dart' show rootBundle;  
  
class ConfigLoader {  
  void loadConfig() async {  
    final cfg = await rootBundle.loadString("myassets/some_cfg.json");  
    doSomething(cfg);  
  }  
}
```

Any Flutter app exposes a `rootBundle` object that allows you to easily load assets but it shouldn't be used inside a `StatelessWidget` or a `StatefulWidget`. In the example, `ConfigLoader` is simply a model class made by us.

```
class MyWidget extends StatelessWidget {  
  const MyWidget();  
  
  Future<String> loadConfig(BuildContext context) async =>  
    await DefaultAssetBundle.of(context)  
      .loadString("myassets/some_cfg.json");  
}
```

When you have the need to retrieve an asset from a widget, use `DefaultAssetBundle` instead of

the `rootBundle` object. Both approaches return the same type of object so you'll always end up calling `load()` or `loadString()`.

 We've only shown json files in the examples but `load()` is able to read any kind of binary resource from the bundled assets. It returns a `ByteData` object which is simply a fixed-length sequence of bytes representing the resource.

There is also the possibility to load different variants of an image in different ways according to the context in which they are rendered. While this sentence might sound difficult to understand, this example will clarify what it's saying. The following PNG image is the same but it comes in three different sizes:



If you created three different PNG files, one for each size, and you gave them the **same** name, Flutter would automatically pick the best image according to the pixel ratio of your device. This is quite easy to achieve: just place the smallest "default" image at the root of the asset directory and then make a series of sub-directories with a specific name:

```
myassets/
  logo.png
  2.0x/logo.png
  3.0x/logo.png
```

Here `myassets/logo.png` is the default image with a resolution of 1.0 (32x32); if your app is run on a device with a 2.1 aspect ratio, Flutter will automatically pick the `myassets/2.0x/logo.png` variant. It's important having the names of the folder reflecting the actual sizes, so inside `2.0x` the image should double the size of the default one. In fact:

- `logo.png`: 32 x 32 (default)

- 2.0x/logo.png 64 x 64 (which is $32 * 2$)
- 3.0x/logo.png 96 x 96 (which is $32 * 3$)

Different variants of the image **must** have the same name: notice that we've always used `logo.png`. You also must call directories following the " $Rx/$ " pattern where R is the number indicating the ratio. You can also use floating point numbers such as `1.5x/` which is valid.

 We highly recommend using asset variants for your images because Flutter automatically picks the one that suits better according to the device in which the app is being used.

Keep in mind that the more assets you bundle in your app, the bigger the size of your final executable will be. Try to reduce the sizes of your assets with techniques such as minification. We recommend you to try following these guidelines as much as possible:

- Don't create too much variants for the same image, make your decision according to the devices that your app is going to run on.
- The size of a PNG image can be reduced up to 75% by optimizing it with dedicated software or even online tools. Try to use them as much as possible if you use PNG images so that the final size of your executable won't be bloated too much.
- Instead of resizing and scaling an image at runtime, which might result in a low quality picture, use assets variants and bundle different sizes of the same file.

On a side note, your app's configuration can also be stored using shared preferences (more on them in chapter 20), a secure storage or simply on a web service. Storing data in json assets and loading them is not common.

18.2 Working with images

Once you've declared the images you're going to use as assets via `pubspec.yaml` they're ready to be used in the code. Very intuitively you're going to use the `Image()` widget to deal with all the supported formats and it will automatically pick the best *variant* according to the pixel ratio of the device.

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

```
    return const Center(
        child: Image(
            image: AssetImage("myassets/something.png"),
        )
    );
}
```

Notice the usage of the `const` constructor which is as always very important. Alternatively, you can use a named constructor which does the same thing with the exception it doesn't provide a constant constructor:

```
@override
Widget build(BuildContext context) {
    return Center(
        child: Image.asset("myassets/something.png"),
    );
}
```

If you have to load an image stored as a sequence of bytes, use the `Image.memory()` variant of the constructor which supports **only** compressed formats (such as *png*). Uncompressed formats such as `rawRgba` will lead to undesired runtime exceptions.

 Resources > Chapter 18 > Network images

18.2.1 Loading from the network

In this example we are going to see how to load an image stored in a server rather than loading it from the assets. We're first going to see how to do it with a progress indicator, which is useful to show the fetching progress, and then in another way, that just shows a *Loading...* placeholder.

1. In this first example we're creating a widget showing an image obtained from the network and, while it's being downloaded, a progress indicator appears. All of this is possible thanks to `class Image` and its very convenient `Image.network()` constructor.

```
class ImageFromWeb extends StatelessWidget {
    final String url;
    const ImageFromWeb({ required this.url });
```

```
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Image.network(url,
        loadingBuilder: (context, child, progress) {
          if (progress == null)
            return child;

          return const Center(
            child: CircularProgressIndicator(),
          );
        },
      ),
    );
  }
}
```

Given an url pointing to a valid image, while it's being loaded an animated circular progress indicator appears at the center of the screen. Instead of having a rotating circle spinning around, you might decide to show the actual progression percentage:

```
var percentage = 0.0;
final total = progress.expectedTotalBytes;

if (total != null) {
  final current = progress.cumulativeBytesLoaded;
  percentage = current / total;
}

return Center(
  child: CircularProgressIndicator(
    value: percentage
)
);
```

Alternatively, instead of using a circle, there's the possibility to use a classic linear progress bar using a different class name but with the same setup:

```
return Center(
  child: LinearProgressIndicator()
```

```
        value: percentage
    ),
);
```

2. In this example we're doing the same thing as before but instead of showing a progress indicator, we are just using a "placeholder" widget that doesn't care about the progression percentage. It just tells the user that the image is being fetched but it gives no info about the progress.

```
class ImageFromWeb extends StatelessWidget {
    final String url;
    const ImageFromWeb({ required this.url });

    @override
    Widget build(BuildContext context) {
        return Center(
            child: Image.network(url,
                frameBuilder: (context, child, _, loaded) {
                    if (loaded)
                        return child;

                    // 'Text' or anything else that doesn't
                    // depend on the prgoression value
                    return const Text("Loading...");
                },
            ),
        );
    }
}
```

The text "*>Loading...*" remains visible until the first frame of the image is available and then it's replaced with the picture. Basically this example is the same as before but instead of a progress indicator there's a simple text.

Use the approach that suits better the use-case you're going to implement; for example if you're loading a big image, it might be good using the `frameBuilder` to show progress. If you have something like a custom widget to display while waiting for the download, and it doesn't depend on the progress, use `frameBuilder`.

- i** When using `loadingBuilder` the `Image()` widget is rebuilt very often until the image is ready to be displayed. If you're showing a loading progress indicator it's fine, but in any other case (for instance when you use text such as *Loading ...*) prefer using `frameBuilder`.

The biggest problem is that there isn't the possibility to directly handle any possible error while fetching the image. A solution would be trying to instantiate a `dio` client and make a GET request to check if the resource is available, or directly use it for the entire fetching process. However this would require a lot of boilerplate code.

```
Image.network(url,
    frameBuilder: (...){},
    errorBuilder: (...) {} // it doesn't exist
)
```

These problems are solved by a very popular package called `cached_network_image`¹ which implements error handling, caching and image fetching. It's very straightforward to use.

```
class CachedImgNetwork extends StatelessWidget {
    final String imgUrl;
    const CachedImgNetwork({ required this.imgUrl });

    @override
    Widget build(BuildContext context) {
        return CachedNetworkImage(
            imageUrl: imgUrl,
            placeholder: (context, url)
                => const CircularProgressIndicator(),
            errorWidget: (context, url, error)
                => const Icon(Icons.error),
        );
    }
}
```

When the download completes, the progress indicator goes away and the image appears on the screen with, by default, an animation of type `Curves.easeIn`. Any issue, such as invalid format or connection error, is handled by `errorWidget` which also provides the description of what's gone wrong. You can also set:

¹https://pub.dev/packages/cached_network_image

- the type of animation and the duration;
- HTTP headers if needed;
- the size of the images and the box fit.

By default images are cached and stored in the temporary directory of your device; if you try to load the image multiple times it will be downloaded only once. You can change this behavior by subclassing `BaseCacheManager` and creating your own manager.

```
CachedNetworkImage(  
    imageUrl: imgUrl,  
    progressIndicatorBuilder: (context, url, status) {  
        return CircularProgressIndicator(  
            value: status.progress,  
        );  
    },  
    errorWidget: (context, url, error)  
        => const Icon(Icons.error),  
) ;
```

This is a variant of the preceding example in which we're still using a loading indicator but it's also showing the progression percentage of the download.

i We recommend the usage of `CachedNetworkImage` as it's very simple to use and it offers many functionalities. With `Image.network(...)` you have more work to do to achieve the same results:

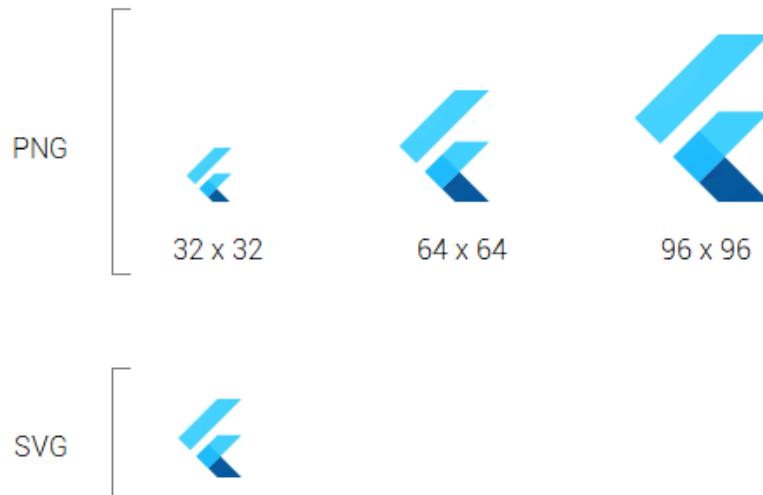
- both `loadingBuilder` and `frameBuilder` can be animated too but you have to do everything by yourself;
- it doesn't have an internal cache so you'd have to create it from scratch and handle it separately;
- it has no error detection so you have to invent something.

18.3 Scalable vector graphics

Images in the `.svg` format are the ones that you should really consider while developing your app. Potentially, they lead to bandwidth savings, simplified workflow and high image quality. The main big advantages of vector images are the following:

1. This file format does **NOT** depend on the resolution of the screen, the size or any other hardware spec. SVG images are based on shapes, paths and fills so they have no resizing or scaling problems; they'll always be painted in high quality.
2. Generally a `.svg` file is smaller in size than a `.png`².

We've seen that PNG images require different variants for the same asset because the file format is dependent on the screen. The pixels of the image are well-defined on a matrix of colors and, when resizing, the result will drop in quality.



If you migrated to SVG images you'd just need to have a single file; it has no resizing or pixel density problems at all and it can infinitely scale. This is also a practical advantage because there's no need to maintain n different variants but only one.

i SVG files can be optimized as well and there are many online tools for this such as *nano*³ that we recommend to use.

Vector images guarantee high quality images and (in most of the cases) a small file size but

²<https://vecta.io/blog/comparing-svg-and-png-file-sizes>

³<https://vecta.io/nano>

they're not always the best choice for your project. While the perfect file format for any use case doesn't exist, we can give you this general guideline:

- Use vector images when you need your images to scale well without losing quality at all. This can be the case of a background image whose size can vary a lot because, for example, you might have the need to show it on a mobile phone (6 inches) and a tablet (12 inches).
- Use PNG images in all those cases in which the image is very complex, full of details and it also has to preserve the transparency. Always remember to optimize the file to reduce its overall size. Also, if the image has a fixed size and it doesn't need scaling, maybe a PNG would work better than a SVG.

When the image is very detailed and complex, the SVG representation is hard to create and also relatively expensive for the graphic engine to decode. A PNG would be better because it's just a map of colors which needs no paths calculations, it still has a high quality but it doesn't resize well. There are a lot of compromises for both formats, so wisely test your apps before deciding which one suits better.

18.3.1 Working with SVG files

If you want to work with SVG images you should use the famous *flutter_svg*⁴ package made by Dan Field. In this simple example we're going to load an vector image from the assets (assuming it contains no errors inside).

- i** Add the package as dependency in the `pubspec.yaml` file following the instructions at <https://pub.dev>. Before putting the `.svg` in your assets, be sure to optimize the file with a proper tool such as nano.

The `SvgPicture` class is able to load vector assets from various sources, similarly to what we've seen earlier with the `Image()` class.

```
class SvgDemo extends StatelessWidget {  
  const SvgDemo();  
  
  @override  
  Widget build(BuildContext context) {  
    return Center(  
      child:
```

⁴https://pub.dev/packages/flutter_svg

```
        child: SvgPicture.asset(
            "myassets/question.svg",
            width: 120,
            placeholderBuilder: (_) =>
                const CircularProgressIndicator(),
        )
    );
}
```

If the vector is particularly complex and it requires some time to load, there's the possibility to show a loading indicator. As you can see, this class is very easy to use and it exposes many useful parameters such as:

- `width` and `height` to specify the dimensions of the image;
- `fit` to determines how to draw the picture in the box that contains it;
- `alignment` that defaults to `Alignment.center`

18.3.2 Loading from the network

Loading an image from the network with `flutter_svg` follows the same pattern you already saw in the previous section. There's the possibility to show a progress indicator while the file is being downloaded and processed, but the class doesn't provide a way to handle connection errors.

```
SvgPicture.asset(
    "https://website.com/images/theimg.svg"
    height: 100,
    placeholderBuilder: (_) =>
        const CircularProgressIndicator(),
)
```

We're now going to show a more complete approach that also takes care of error handling and ensures that, in case of problems, the user is notified. We're doing it with `dio` but you could have used any other HTTP package that easily allows you to handle connection failures.

1. The first step involves the creation of a model class about networking; once the image has been downloaded, it will be passed to the `SvgPicture` widget which will take care of painting it to the UI. We've made the client variables `static` so that they're created only once (the same instance can be reused).

```
class Downloader {  
    final String url;  
    const Downloader({ required this.url });  
  
    static final _opt = BaseOptions(  
        baseUrl: "https://fluttercompletereference.com/",  
        connectTimeout: 3000,  
        responseType: ResponseType.bytes  
    );  
  
    static final _client = Dio(_opt);  
  
    Future<List<int>> start() async {  
        final request = await _client.get<List<int>>(url);  
        return request.data;  
    }  
}
```

We're not downloading the vector as a string but as raw bytes; they will be passed to a `SvgPicture` object which will handle the decoding. For this reason, there's the need to set the response type to `ResponseType.bytes` which returns a list of integers. Note:

```
class Downloader {  
    final String url;  
    const Downloader({ required this.url });  
  
    Future<List<int>> start(Dio client) async {  
        final request = await client.get<List<int>>(url);  
        return request.data;  
    }  
}
```

If each request needs a particular setup, don't make the client and its options `static` but instead ask them as external dependency (method injection).

2. Using a `FutureBuilder<T>` we're able to place a loading indicator while `dio` performs the request but we can also show an error message or whatever else in case of problems. You can find the code of the error widgets in the *Resources* section of our website.

```
class SvgFromWeb extends StatefulWidget {
```

```

const SvgFromWeb();

@Override
_SvgFromWebState createState() => _SvgFromWebState();
}

class _SvgFromWebState extends State<SvgFromWeb> {
    late final Downloader downloader;
    late final Future<List<int>> svgImage;

    @override
    void initState() {
        super.initState();
        downloader = const Downloader(url: "/demoimages/firefox.svg");
        svgImage = downloader.start();
    }

    // build method...
}

```

As always, we need to create a stateful widget because we don't want the `FutureBuilder<T>` to execute multiple unneeded requests. We had already covered this topic in chapter 17 with the same situation; this pattern **always** applies when working with `Future<T>s` inside widgets.

```

FutureBuilder<List<int>>(
    future: svgImage,
    builder: (context, snapshot) {
        if (snapshot.hasError) {
            return const ErrorWidget();
        }

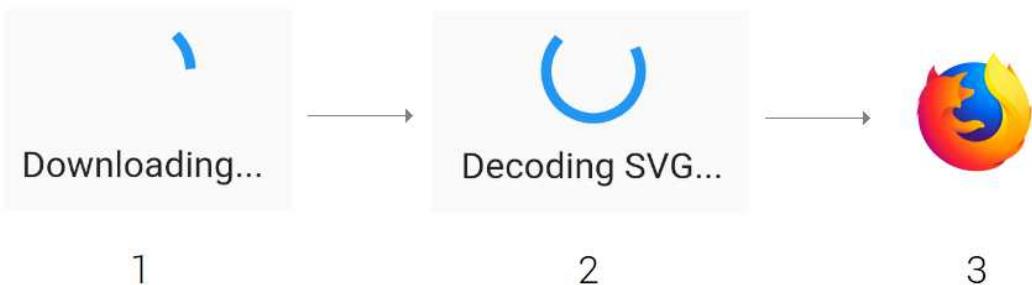
        if (snapshot.hasData) {
            if (snapshot.data != null) {
                return SvgPicture.memory(
                    Uint8List.fromList(snapshot.data!),
                    placeholderBuilder: (_) => const DecoderLoader(),
                );
            } else {

```

```
        return const ErrorWidget();
    }
}

return const NetworkLoader();
),
);
```

By default, a loading indicator appears while the image is being received from the network. If all goes well, we want `SvgPicture` to show another loader in case the image takes some time to be decoded; the user will always be aware that there's some work going on.



The raw bytes downloaded by `dio` are passed to the `SvgPicture.memory()` constructor which decodes an SVG from a series of bytes. `memory()` loads an in-memory representation of a vector stored as a `List<int>` and that's why we set

```
responseType: ResponseType.bytes
```

in our client's options. Those bytes are stored in memory and then the `SvgPicture` decoder will read them and paint the image to the screen.

In this example we've used `dio` to deal with the networking part and all of its details, `SvgPicture` to paint the image to the screen and `FutureBuilder<T>` to take care of loading spinners and error messages, if any. To sum up, this is the execution flow:

1. get the SVG as raw bytes from the network using `dio`;
2. thanks to `FutureBuilder<List<int>>` show a loading indicator;
3. use the `memory()` constructor to decode the in memory representation of the vector and paint it.

18.4 Audio and video with Flutter

 Resources > Chapter 18 > Playing a video

18.4.1 Playing a video

In this example we're going to see how to load a video from the network; we will interact with it changing the volume and using the play/stop button. The Flutter team has created the *video_player*⁵ package which provides an easy way to handle video files from assets, network or file system. The app is going to look like this:



The buttons interact with the video by playing or pausing it and the slider below changes the volume. While the UI seems very poor, there are many aspects to consider when writing the

⁵https://pub.dev/packages/video_player

actual code.

1. First of all there's the need to add *video_player* as external dependency in the pubspec file. Be sure to open *pub.dev* and check which is the latest version.
2. The slider has to change the volume of the video and update its current position so the state of the widget is going to change. We're going to handle this with *provider* using the same approach you've seen many times up to now.

```
class VolumeManager with ChangeNotifier {  
    var _volume = 50.0;  
    double get volume => _volume;  
  
    void setVolume({  
        required double volumeValue,  
        required VideoPlayerController controller  
    }) {  
        _volume = volumeValue;  
        controller.setVolume(_volume);  
  
        notifyListeners();  
    }  
  
}
```

The *VideoPlayerController* class comes from *video_player* package and it's used to change the volume or play/pause/stop the video; it's an external dependency. What we really want to do here is notify the *Slider* that the volume has changed and refresh it.

3. The widget containing the video, the buttons and the slider have to be stateful because there are initialization and finalization tasks to execute. They are needed to setup the video handler and release it when it's not needed anymore in order to not waste resources.

```
class _VideoWidgetState extends State<VideoWidget> {  
    late final VideoPlayerController controller;  
    late final Future<void> initVideo;  
  
    @override  
    void initState() {  
        super.initState();
```

```
// In this example we're loading it from the assets
// but you could also get it from the network!
controller = VideoPlayerController.asset(
    "assets/butterfly.mp4"
);

controller.setLooping(true);
initVideo = controller.initialize();
}

@Override
void dispose() {
    controller.dispose();
    super.dispose();
}

// build...
}
```

We're going to use a `FutureBuilder<void>` to await the loading of the video so in `initState` we need to assign the returned value of `initialize()` to an instance variable. Always remember to dispose the controller inside `dispose()` to avoid potential memory leaks. Note that `class VideoControllerPlayer` has many constructors:

- `VideoControllerPlayer.asset()`: loads a video from the assets;
- `VideoControllerPlayer.network()`: loads a video from the network;
- `VideoControllerPlayer.file()`: loads a video from a file located somewhere in the filesystem of your device.

Their `initialize()` method returns a `Future<void>` so the interface doesn't change; just use the named constructor you need and all the other code will stay the same.

4. Since the controller doesn't return any data, in `FutureBuilder<void>` we don't have to look for `snapshot.hasData` but instead `snapshot.connectionState`, which tells us something about the state of the future.

```
@override
Widget build(BuildContext context) {
    return FutureBuilder<void>(

```

```
        future: initVideo,
        builder: (context, snapshot) {
            if (snapshot.connectionState == ConnectionState.done) {
                return PlayWidget(controller);
            }

            return const Center(
                child: CircularProgressIndicator(),
            );
        },
    );
}
```

When the status is `ConnectionState.done`, the video has been loaded with success and it's ready to be played. The buttons and other UI widgets that interact with the user have been moved to another widget (`PlayWidget`) to separate the concerns and keep the method short.

5. Passing a reference to the controller via constructor injection is very important because we don't want a strong dependency with the object responsible of playing and stopping the video. We've created two private methods that will be called by the buttons to start or pause the video.

```
class PlayWidget extends StatelessWidget {
    final VideoPlayerController controller;
    const PlayWidget(this.controller);

    // build method...

    void _play() {
        if (!controller.value.isPlaying) {
            controller.play();
        }
    }

    void _pause() {
        if (controller.value.isPlaying) {
            controller.pause();
        }
    }
}
```

```
    }
}
```

This is the body of the `build` method with the most important parts. Of course the entire source code is available online in the *Resources* area of our official website.

```
Column(
  mainAxisSizeAlignment: MainAxisSizeAlignment.spaceAround,
  children: <Widget>[
    // The video
    AspectRatio(
      aspectRatio: controller.value.aspectRatio,
      child: VideoPlayer(controller),
    ),
    // "Play!" and "Pause" buttons
    Row(...),
    // The slider
    Consumer<VolumeManager>(
      builder: (context, manager, _) =>
      Slider(
        value: manager.volume,
        onChanged: (value) {
          manager.setVolume(
            volumeValue: value,
            controller: controller,
          );
        },
      ),
    ),
  ],
);
```

The `AspectRatio` widget is very important because it ensures that the size of the widget is consistent with the aspect ratio of the video. If you're playing a video that has a 4:3 width:height aspect ratio, this widget automatically sets the size.

```
AspectRatio(
  aspectRatio: 4/3,
```

```
        child: VideoPlayer(controller),  
    ),
```

You could have set the ratio value by hand, which is just a double, but if you assign it with the `aspectRatio` property of the controller you get the proper value of any video automatically. In general, this widget is very useful when you don't care about the dimensions of the child but you want to preserve its width:height ratio.

```
onChanged: (value) {  
    manager.setVolume(  
        volumeValue: value,  
        controller: controller,  
    );  
},
```

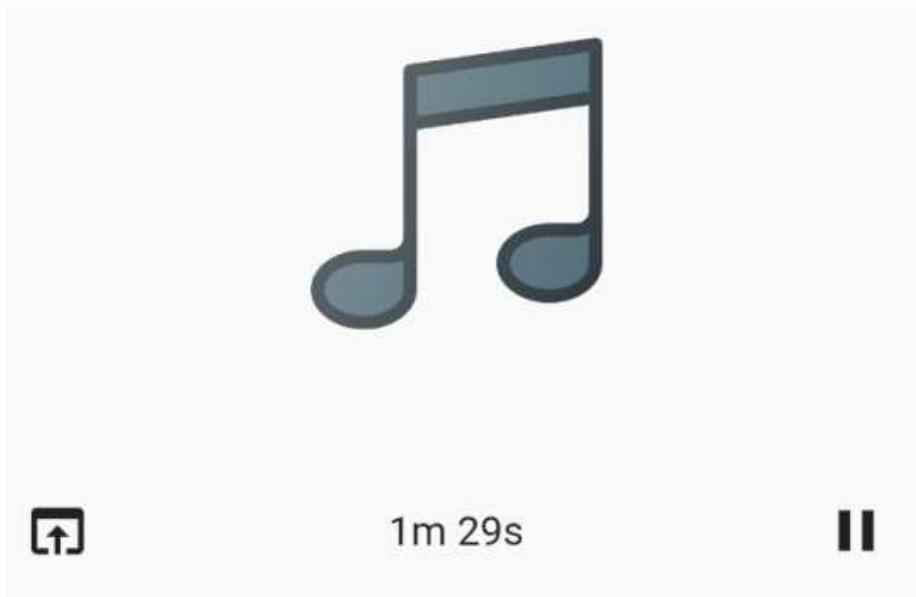
The `Slider` has an `onChanged` listener which gives as parameter the current position of the pointer (in our case, the current value of the volume). We use it to update the volume thanks to the object provided by a `Consumer<VolumeManager>()` which also takes care of rebuilding the slider.

 Resources > Chapter 18 > Listening to music

18.4.2 Listening to music

In this example we're showing how you can listen to music, whether it be a single audio or a playlist of any dimension, using the `assets_audio_player`⁶ plugin. Once the music has been bundled as asset, it will be possible to play it.

⁶https://pub.dev/packages/assets_audio_player



The button on the left opens and starts the audio while the one on the right plays/pauses the track. In the middle a `Text` widget is showing the elapsed time. We're going to work `class AssetsAudioPlayer` which is very easy to use:

```
final player = AssetsAudioPlayer();
player.open(Audio("assets/music/song1.mp3"));

// And then interact with it!
player.pause();
player.playOrPause();
player.stop();
```

You just need to call `open("path/to/file.mp3")` to load the file and then with the `player` object you interact with the audio track. Of course, be sure to have your audio files declared as assets in the pubspec file. You can also load a playlist and the player will play, in sequence, every track of the list:

```
final player = AssetsAudioPlayer();
player.openPlaylist(
    Playlist(
```

```
        assetAudioPaths: [
            "assets/music/track1.mp3",
            "assets/music/track2.mp3",
            "assets/music/track3.mp3"
        ]
    )
);

player.playlistNext();
player.playlistPrev();
player.stop();
```

Skip to a particular track using `player.playlistPlayAtIndex(n)`. In our example we're going for a single audio file but you can easily change it to load a playlist and maybe also load a random image at the top.

1. As usual, start by importing the latest version of the package in the `pubspec.yaml` file. Create a directory called `music/` and add in there the audio tracks you're going to play.

```
dependencies:
  assets_audio_player: ^2.0.9+2

flutter:
  assets:
    - music/
```

2. This package works with streams so we're not going to use the classic `FutureBuilder<T>` setup. This is the initialization of the audio player and the SVG image at the top (see the previous chapter for more details about vector images).

```
class MusicWidget extends StatelessWidget {
    static final _assetsAudioPlayer = AssetsAudioPlayer();
    const MusicWidget();

    @override
    Widget build(BuildContext context) {
        return Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
                // It would have been better if we used 'LayoutBuilder'
```

```
// to make the image responsive, but for the sake of
// the example hard-coding 100 is fine.
SvgPicture.asset(
    "assets/music-note.svg",
    height: 100,
    placeholderBuilder: (_) =>
        const CircularProgressIndicator(),
),
// Buttons and time elapsed...
],
);
}
}
```

It's possible making the audio player `static` when the music, or the playlist, is always going to be the same. There is no need to create the object many times: thanks to `static`, we can define a `const` constructor and "cache" the instance.

3. Now we're going to see how the buttons and the elapsed time indicator have been implemented. To better discuss the pieces of code, we've divided them in three parts.

```
children: <Widget>[
    // Start listening
    IconButton(
        icon: const Icon(Icons.open_in_browser),
        onPressed: () {
            _assetsAudioPlayer.open(Audio("assets/music.mp3"));
        },
    ),
    // Time elapsed
    StreamBuilder<Duration>(...),
    // Play or pause
    StreamBuilder<bool>(...),
]
```

Very simply, this button tells the player to look for "music.mp3" and play it.

4. The `Text` widget is frequently updated by a stream containing information about how much time has passed since the audio has started. Note that `asyncSnapshot.data` is nullable so we need to do a null-check or any other kind of validation.

```
children: <Widget>[
    // Start listening
    IconButton(...),

    // Time elapsed
    StreamBuilder<Duration>(
        stream: _assetsAudioPlayer.currentPosition,
        builder: (context, asyncSnapshot) {
            if (asyncSnapshot.hasData) {
                final time = asyncSnapshot.data;

                if (time != null) {
                    return Text("${time.inMinutes.remainder(60)}m "
                        "${time.inSeconds.remainder(60)}s");
                } else {
                    return const Text("No time available.");
                }
            }

            return const Text("0m 0s");
        }
    ),
    // Play or pause
    StreamBuilder<bool>(...),
]
```

Thanks to `_assetsAudioPlayer.currentPosition` you can subscribe and listen for updates on the elapsed time, since the beginning, with the usual `Duration` object. We have used this condition...

```
if (asyncSnapshot.hasData) { ... }
```

... to know whether there's an audio playing or not. If nothing is going on (no music is playing), we just return the "0m 0s" placeholder.

5. The last part is about listening to a stream which tells us whether there is an audio playing or not. This information can be useful to swap the play/pause button according to the value of the `bool` flag.

```
children: <Widget>[
    // Start listening
    IconButton(...),
    // Time elapsed
    StreamBuilder<Duration>(...),
    // Play or pause
    StreamBuilder<bool>(
        stream: _assetsAudioPlayer.isPlaying,
        builder: (context, AsyncSnapshot<bool> isPlaying) {
            if (isPlaying.data ?? false) {
                return IconButton(
                    icon: const Icon(Icons.pause),
                    onPressed: _assetsAudioPlayer.pause,
                );
            } else {
                return IconButton(
                    icon: const Icon(Icons.play_arrow),
                    onPressed: _assetsAudioPlayer.play,
                );
            }
        },
    ),
]
```

If `isPlaying` is `true` it means that the audio is currently playing so there's the need to show a button that can pause it. On the other hand, if it's `false` then the music is not playing and thus the play button must appear. We have used a tear-off rather than a lambda just to reduce the amount of code:

```
// Tear-off (which is a sort of "method reference")
onPressed: _assetsAudioPlayer.play,
// Lambda
```

```
 onPressed: () => _assetsAudioPlayer.play(),
```

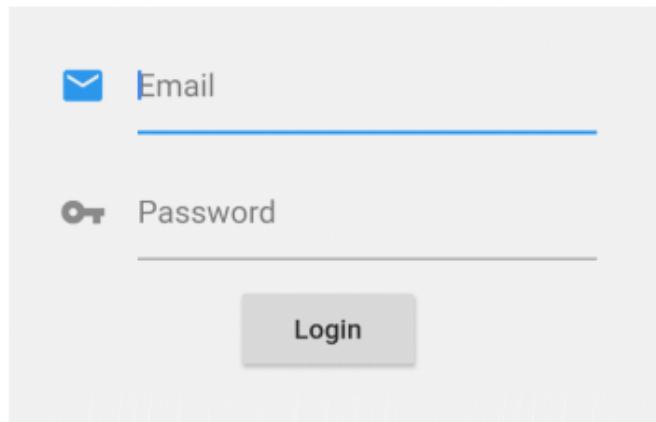
Both versions are equivalent but tear-offs should be preferred.

19 | Forms and gestures

💻 Resources > Chapter 19 > Login form

19.1 Forms and validation

A form consists of a series of fields (to be filled by the user with some information) that can be processed. A typical example is a login form, in which the app generally could ask you for email and password. Once a confirmation button is pressed, an HTTP request or another type of authentication process happens.



In this section we are going to create the login form you see in the above picture and, in the following subsections, we will add more functionalities.

1. The two inputs and the button are positioned at the center of the screen but they take only 2/3 of the width of the containing widget. If the user rotates the screen for example, we still want to have the form at the center and with the same ratio in order to keep consistency.

```
LayoutBuilder(  
    builder: (context, dimensions) {  
        // (a)  
        final width = dimensions.maxWidth / 1.5;  
        final height = dimensions.maxHeight / 3;  
  
        return Center(  
            // (b)  
            child: SizedBox(  
                width: width,  
                height: height,  
                child: LoginForm(),  
            ),  
        );  
    },  
) ,
```

This is the perfect situation in which the `LayoutBuilder` widget has to be used because it gives the possibility to work with the actual available space. In particular:

- writing `maxWidth / 1.5` we can get the value of the width so that it takes 2/3 of the horizontal space. This is very flexible because if you rotated the screen, the widget would automatically rebuild and the ratio would remain identical;
- we could have used a `Container` but since no styling or particular effects are needed, a `class SizedBox\(\)` fits better. Its dimensions are defined by responsive values calculated by `LayoutBuilder`.

This is not only useful when the screen rotates but also if your app runs on a big tablet. You could also decide to implement a more complex logic for the sizes of the box but in this case we recommend you to encapsulate the calculations in a separated function:

```
LayoutBuilder(  
    builder: (context, dimensions) {  
        final width = _getWidth(); // calculated width  
        final height = _getHeight(); // calculated height  
    }  
)
```

-) ,
2. The `Form` widget is a container for multiple form fields such as `TextField()` which takes the user's input. In order to provide fields validation, we need a `GlobalKey<FormState>` object that uniquely identifies the form in the widget tree.

```
class LoginForm extends StatelessWidget {
    final _key = GlobalKey<FormState>();

    @override
    Widget build(BuildContext context) {
        return Form(
            key: _key,
            child: Column(
                mainAxisAlignment: MainAxisAlignment.spaceAround,
                children: <Widget>[...],
            ),
        );
    }
}
```

3. Inside the `Column` we're going to put a series of `TextFields` the user has to fill. Thanks to the `decoration` parameter we can customize the look of the text box adding a leading icon and a hint text. There's the possibility to customize the widget even more; visit the official documentation for `class InputDecoration`¹ to get a better overview of the various styling possibilities.

```
// Email
TextField(
    decoration: const InputDecoration(
        icon: Icon(Icons.mail),
        hintText: "Email"
    ),
    validator: _validateEmail,
),

// Password
```

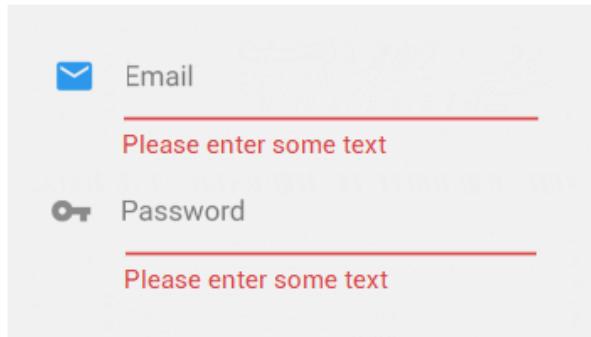
¹<https://api.flutter.dev/flutter/material/InputDecoration-class.html>

```
TextField(  
    decoration: const InputDecoration(  
        icon: Icon(Icons.vpn_key),  
        hintText: "Password"  
    ),  
    obscureText: true,  
    validator: _validatePassword,  
,  
  
// Confirm Button  
RaisedButton(...)
```

To validate the input, we pass a callback to `validator` which returns an error message if the string doesn't satisfy certain conditions. If there are no errors, the function must return `null`.

```
String? _validateEmail(String value) {  
    if (value.isEmpty) {  
        return "Field cannot be empty";  
    } else {  
        return null;  
    }  
}  
  
String? _validatePassword(String value) {  
    if (value.length < 8) {  
        return "At least 8 chars!";  
    } else {  
        return null;  
    }  
}
```

When the "Login" button is tapped but at least one of the two form fields doesn't return `null`, it means that there's a problem and a red error text appears below. If multiple fields have a `validator`, they all must return `null` in order to proceed with the submission of the data otherwise the errors will appear.



4. The last step is the creation of the submission button which is going to check if no validator returns `null` (meaning that all inputs are ok). If this is the case, we can start the login logic which might be, for example, a `dio` HTTP request.

```
children: <Widget>[
    // Email
    TextFormField(...),
    // Password
    TextFormField(...),
    // Login
    RaisedButton(
        child: const Text("Login"),
        onPressed: _login,
    ),
]
```

The login logic should be placed in a dedicated class in order to keep the UI logic separated from the business logic as much as possible.

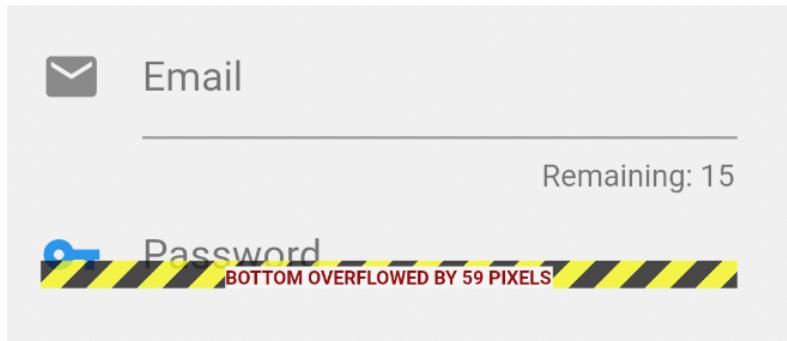
```
void _login() {
    if (_key.currentState?.validate() ?? false) {
        final login = LoginClass();
        // ...
    } else {
        // Show an error or something else to alert
```

```
// that the input fields contain some errors  
}  
}
```

You see now why we had to use a `GlobalKey<FormState>` at the beginning of the example. It's used to identify the form and check if every field inside is validated or not thanks to `bool validate()`.

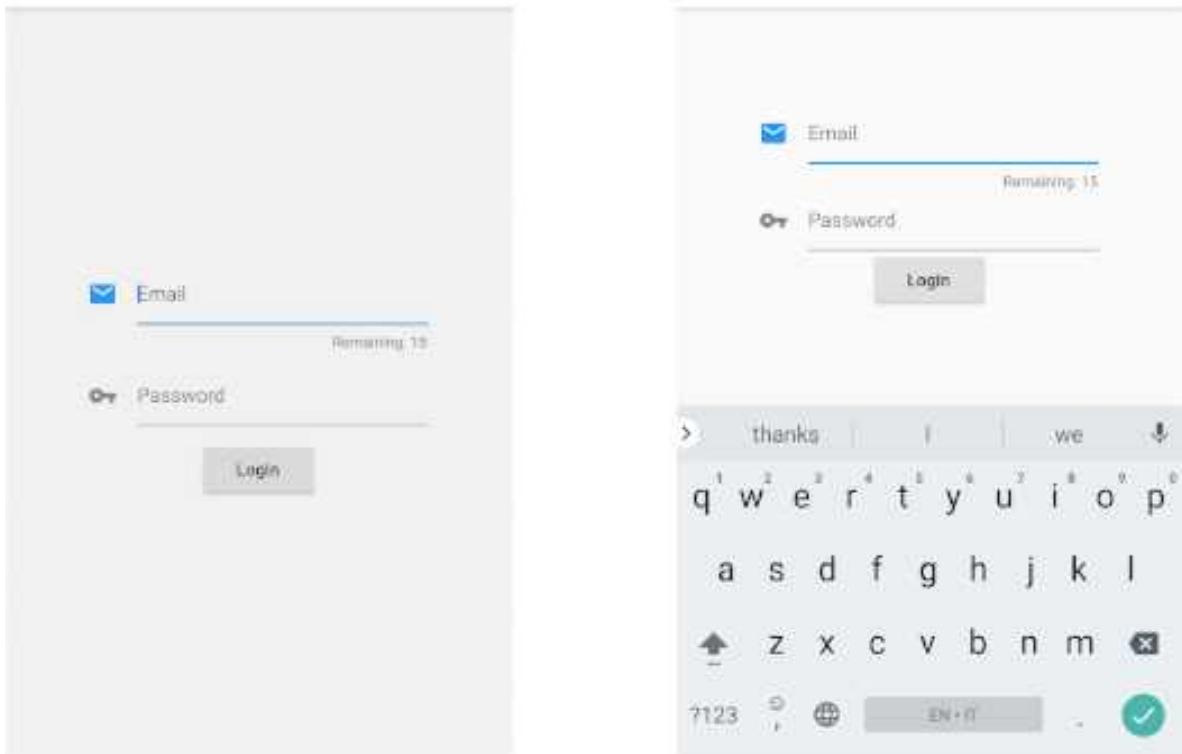
19.1.1 Keyboard and overflows

In our example the `Form` is at the center of the screen and there's enough vertical space so it's all good. However, when you press on a form field, your device's keyboard comes out reducing the available height and an overflow error might occur if there isn't enough space anymore in the vertical axis.



The problem is that the keyboard reduces the available height and so there won't be enough space anymore to entirely show the form. To fix this problem there are two possibilities, depending on how you want the form to behave.

1. A `class SingleChildScrollView` is useful when a widget is normally visible but you want to add scrolling behavior in case there weren't enough space. Basically it ensures that, in case of overflows, scrolling is activated. This can be the case where you have an entirely visible form but when the screen rotates or the keyboard appears, some parts might get "cut off".



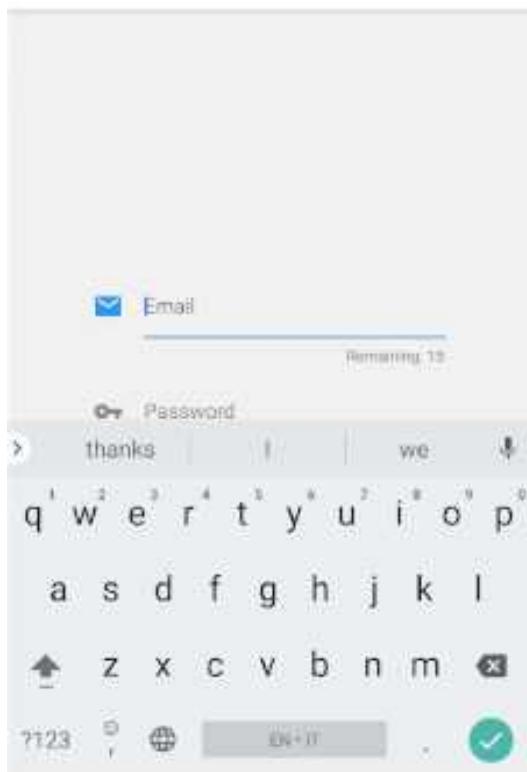
As you can see from the image, the login box moves up and still remains at the center of the remaining space. If the screen were smaller and the box didn't fit the entire height, you would be able to scroll it up and down very conveniently. No overflow errors at all.

```
LayoutBuilder(  
    builder: (context, dimensions) {  
        final width = dimensions.maxWidth / 1.5;  
        final height = dimensions.maxHeight / 3;  
  
        return Center(  
            child: SingleChildScrollView(  
                child: ConstrainedBox(  
                    constraints: BoxConstraints(  
                        minHeight: height,  
                        maxWidth: width,  
                    ),  
            ),  
    ),
```

```
        child: LoginForm(),
    ),
),
);
},
),
),
);
```

The direct child should really be a `ConstrainedBox` as it imposes its children to have the sizes given by the `constraints` parameter. Since they are taken from a `LayoutBuilder`, we're sure about the responsiveness of the layout.

2. The other solution doesn't involve any scrolling; it simply puts the keyboard "in front of your app" but it covers the widgets behind it. This is not optimal in the case of a form because the user cannot see what he's typing and he'd have to close the keyboard.



You cannot scroll the login box up or down because there is no scroll bar. If this is what you

need, go to the Scaffold containing the form and add set `resizeToAvoidBottomInset`:

```
Scaffold(  
    resizeToAvoidBottomInset: false, // add this line  
    appBar: AppBar(...),  
    body: MyBody(...),  
)
```

Forms aside, the solutions we've exposed are valid for any widget with this kind of problem. When you aren't sure that the height will always be enough to show a widget entirely, consider using a `SingleChildScrollView` to avoid unexpected overflow errors.

19.1.2 Getting the text from a text field

So far we've only seen how to make sure the user has filled the inputs in a proper way (**data validation**). The second step is learning how to get the text from the form fields (**data acquisition**).

1. In a `StatefulWidget`, create an instance of a `TextEditingController` and assign it to a `TextField`. We need to convert the previous `LoginForm` widget into a stateful one because the controller has to be disposed.

```
class LoginForm extends StatefulWidget {  
    const LoginForm();  
  
    @override  
    _LoginFormState createState() => _LoginFormState();  
}  
  
class _LoginFormState extends State<LoginForm> {  
    final emailController = TextEditingController();  
    final passwController = TextEditingController();  
  
    @override  
    void dispose() {  
        emailController.dispose();  
        passwController.dispose();  
  
        super.dispose();  
    }  
}
```

```
    @override
    Widget build(BuildContext context) { ... }
}
```

You have to create one controller for each form field you want to get the text.

2. Assign the controller to a form field using the `controller` property.

```
children: <Widget>[
    TextFormField(
        controller: emailController,
        ...
    ),
    TextFormField(
        controller: passwController,
        ...
    ),
    RaisedButton(
        child: const Text("Login"),
        onPressed: _login,
    ),
]
```

3. Get the current text of the field using the `text` property of the controller.

```
void _login() {
    if (_key.currentState?.validate() ?? false) {
        final email = emailController.text;
        final passw = passwController.text;

        final login = LoginClass(
            user: email,
            password: passw,
        );
        // ...
    } else {
        // ...
    }
}
```

```
    }  
}
```

Other than `text`, another very interesting property is `selection`² which lets you handle the text the user has selected. For example, if the user selected **LUTT** in the word **FLUTTER**, this code...

```
debugPrint("Selection: ${myController.selection.start} - "  
        "${myController.selection.end}");
```

... would print `"Selection: 1 - 5"` because the selected chars start from position 1 of the string and end at 5 (not inclusive). If you want to get the selected text, work with the offsets to extract a substring.

```
final start = myController.selection.start;  
final end = myController.selection.end;  
  
final selectedText = myController.text.substring(start, end);  
debugPrint(selectedText); // prints "LUTT"
```

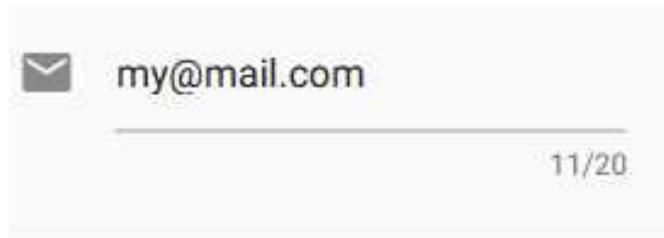
19.1.3 Constraining the input

By default the user is able to type as many characters as he wants because there are no restrictions on the length. If you want to limit the length of the input, just set the `maxLength` property with the max char count.

```
TextField(  
    maxLength: 20,  
    ...  
) ,
```

With this simple addition, the user is able to type up to 20 characters. A counter automatically appears on the bottom-right corner of the field to show how much space left there is in the field.

²<https://api.flutter.dev/flutter/widgets/TextEditingController/selection.html>



The counter can be disabled setting `counterText: ""` in the `InputDecorator` assigned to the `TextField`. A custom text combined with a counter requires a bit more of setup but of course it's still doable:

1. Create a variable to hold the maximum allowed length and preferably make it `static const` if you know that it doesn't have to change.

```
class _LoginFormState extends State<LoginForm> {  
    static const maxLength = 10;  
  
    final emailController = TextEditingController();  
}
```

2. We're going to use the `text` property of the controller because it immediately updates whenever a new character is added or removed in the form field. Thanks to this, we can still show the remaining space but with a different label:

```
TextField(  
    controller: emailController,  
    maxLength: maxLength,  
    decoration: InputDecoration(  
        counterText: "Chars left: "  
            "${maxLength - emailController.text.length}",  
    ),  
,
```

If your input field contains sensible data such as a password, you can constrain it to show dots instead of the actual text with `obscureText: true`. Lastly, we show you a possible way to implement a dropdown menu in which the user doesn't have the freedom to write but he's forced to pick an option from a set of predefined items.

 Resources > Chapter 19 > Dropdown menu

1. The dropdown has to rebuild whenever a new option is picked and we also need to store somewhere the selected item. All of this can easily be achieved with the classic `provider + ChangeNotifier` approach.

```
class DropDownText with ChangeNotifier {
    static final _list = ["Pasta", "Pizza", "Maccheroni"]
        .map<DropdownMenuItem<String>>((item) {
            return DropdownMenuItem<String>(
                value: item,
                child: Text(item),
            );
        })
        .toList();

    final menuItems = UnmodifiableListView(_list);

    var _text = "";
    String get text => _text;

    void setText(String value) {
        _text = value;
        notifyListeners();
    }
}
```

Instead of manually writing each single `DropdownMenuItem<T>` by hand, you can work on a private array (`_list`) which is easier maintain. The `map()` method will automatically build a list of items for the dropdown widget.

2. Create a `ChangeNotifierProvider` right above the widget that is going to have the dropdown picker. Other than rebuilding the dropdown, it will also expose the selected item via `text` property.

```
ChangeNotifierProvider<DropDownText>(
    create: (_) => DropDownText(),
    child: Center(
        child: const SizedBox(
```

```
        width: 200,
        child: DropDown(),
    ),
),
),
),
```

In a production app, instead of hard-coding the width, you should calculate it using a `LayoutBuilder` to get a responsive widget.

3. The `DropdownButtonFormField<T>` widget must contain a list of `DropdownMenuItem<T>` children, where `T` is the data type the widget represents.

```
@override
Widget build(BuildContext context) {
    return Consumer<DropdownText>(
        builder: (context, dropdown, _) {
            return DropdownButtonFormField<String>(
                items: dropdown.menuItems,
                value: dropdown.text,
                onChanged: (value) =>
                    dropdown.setText(value),
            );
        },
    );
}
```

Since we have a list of ingredients, we want them to be strings. Thanks to `provider` we're able to attach the list of items via `items` and also set the currently visible element via `value`. The dropdown gets automatically updated when the selected item is changed thanks to `setText(value)`.

Whenever we want to obtain the currently selected value of the dropdown picker we need to use `provider` and ask for `dropdown.text` which returns a string. Please keep in mind that an exception will be thrown if:

- `items` is null or has no elements;
- there are duplicates in `items`;
- `value` doesn't appear somewhere in `items`;

19.2 Gestures

A **gesture** is a semantic action (such as "tap", "slide" or "drag") recognized by a pointer event on the UI. In practice, with the term *gesture* you indicate all those actions that can be done with one or more fingers such as tapping or sliding in a certain direction. Flutter handles gestures using the `class GestureDetector` which has many listeners³, such as:

- `onTap`,
- `onDoubleTap`,
- `onLongPress`,
- `onVerticalDragStart`,
- `onVerticalDragEnd`
- and much more...

A `GestureDetector` is very useful because it can make any widget "clickable". For example, an image has no `onPressed` event by default (or similar) but you can assign it one or more gestures:

```
GestureDetector(  
    onTap: () => debugPrint("Click!"),  
    onDoubleTap: () => debugPrint("Double click!"),  
    child: Image.asset("..."),  
)
```

In this way a lot of widgets gain the possibility to interact with the user in many different ways. Nevertheless, keep in mind that Flutter also has dedicated widgets for single tap events that should be preferred. For instance:

- **DO** use the `class IconButton` if you want to make an icon clickable because there already is a tap handler (the `onPressed` callback).

```
// OK  
IconButton(  
    icon: const Icon(Icons.add),  
    onPressed: () {}  
)
```

³<https://flutter.dev/docs/development/ui/advanced/gestures#gestures>

- **DO NOT** use a `GestureDetector` if you want to make an icon clickable. There already is a dedicated widget made by the Flutter team.

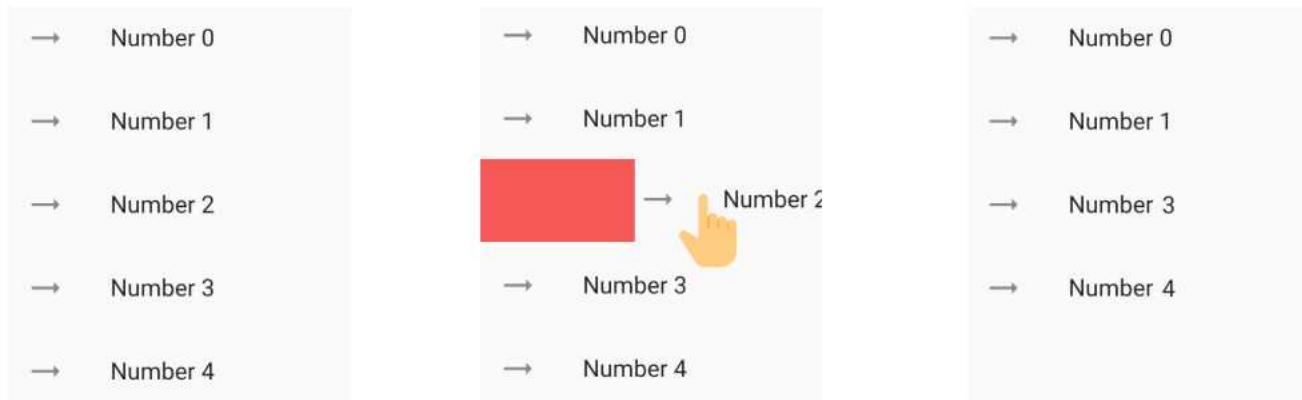
```
// It works but it's not a good idea; prefer using an 'IconButton'  
GestureDetector(  
    onTap: () {},  
    child: const Icon(Icons.add),  
)
```

When you want to implement a gesture for a widget, whether it be a tap, a swipe or anything else, first try to see if there's something already crafted in the material or cupertino library. If it's not the case then you can start working on your own gesture management using a `GestureDetector`.

 Resources > Chapter 19 > Swipe to dismiss

19.2.1 Swipe to dismiss

When there's an item of a list that needs to be deleted, rather than clicking on a button the user can swipe to the left/right to remove it. This is a common UI pattern known as "swipe to dismiss" and we're going to implement it in Flutter:



Given a list of widgets, when you swipe an item to the right it gets deleted. Later on we will also implement a deletion confirmation dialog.

Chapter 19. Forms and gestures

1. Since the list is potentially going to remove elements, we need to manage the state of the widget. We're using the usual model class with `ChangeNotifier` so that a provider can listen for changes.

```
class SourceList with ChangeNotifier {
    final _myList = List<String>.generate(10, (i) {
        return "Number $i";
    });

    List<String> get values => UnmodifiableListView(_myList);

    void removeItem(int index) {
        _myList.removeAt(index);
        notifyListeners();
    }
}
```

Rather than returning the list itself, we return an unmodifiable view of the list to ensure that the only way to delete items is via `removeItem(int)`. This is compulsory because the method contains a call to `notifyListeners()` while `removeAt()` alone doesn't.

2. Now we need to setup the provider making sure it's placed exactly one level above the widget with the list. In this way the scope doesn't get polluted and all those widgets that don't care about the list aren't involved.

```
ChangeNotifierProvider(
    create: (_) => SourceList(),
    child: const SwipeItems(),
),
```

As always, we need a `Consumer<T>` to retrieve the value held by provider which is, in this case, a list of strings.

```
class SwipeItems extends StatelessWidget {
    const SwipeItems();

    @override
    Widget build(BuildContext context) {
        return Consumer<SourceList>(
            builder: (context, list, _) {
                return ListView.builder(...)
```

```
        },
    );
}
}
```

3. For long lists, the `builder()`⁴ constructor of `ListView` is very convenient as it automatically generates the children. The "swipe to dismiss" pattern is implemented with the `Dismissible` widget.

```
return ListView.builder(
    itemCount: list.values.length,
    itemBuilder: (context, index) {
        var item = list.values[index];

        return Dismissible(
            // a.
            key: Key(item),
            // b.
            background: Container(color: Colors.redAccent),
            // c.
            onDismissed: (direction) => list.removeItem(index),
            direction: DismissDirection.startToEnd,
            child: ListTile(
                leading: const Icon(Icons.trending_flat),
                title: Text(item),
            ),
        );
    },
);
```

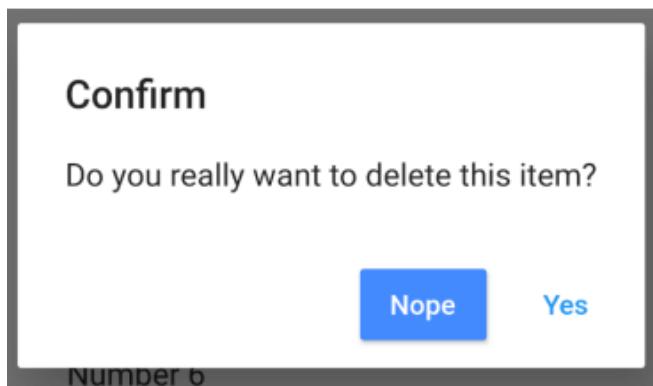
By default, each item of the list can be swiped to the left or to the right; thanks to `direction` we've forces that the only allowed action to delete is "swipe to the right". Of course you can change it to whatever you want because `DismissDirection` provides many directions.

- (a) The `Key` is fundamental because it uniquely identifies an item of the list and it's internally used by `Dismissible`. Be sure to create the key with an unique value for each item, otherwise you'll incur an error.

⁴<https://api.flutter.dev/flutter/widgets/ListView/ListView.builder.html>

- (b) This is the background widget that will appear while the item is being moved to the right. Of course it doesn't always have to be a `Container`: any widget is fine.
- (c) This callback is triggered when the swiping action is finished, which is when the user's finger leaves the screen and the item has completely slid to the right. The element will be removed and the notifier will rebuild the list immediately.

As you can see, the `Dismissible` widget makes the implementation of the "swipe to dismiss" pattern very easy. It also offers the possibility to confirm a pending deletion by setting the `confirmDismiss` callback.



One of the best confirmation widgets is `AlertDialog`, from the material library, and we're going to use it to obtain the above result. Basically it informs the user about situations where an action from his part is needed.

1. Let's start by defining a callback to be attached to the `confirmDismiss` parameter. When set, it waits for a `Future<T>` to return either `true` or `false` to decide if the swiped item has to be deleted or not.

```
Dismissible(  
    confirmDismiss: (direction) => _getConfirm(context, direction),  
    ...  
)
```

2. Create the `Future<bool> _getConfirm(...)` method and use `showDialog()` inside it to actually show a dialog on the UI.

```
Future<bool> _getConfirm(BuildContext context,  
                           DismissDirection direction) {  
    return showDialog<bool>(
```

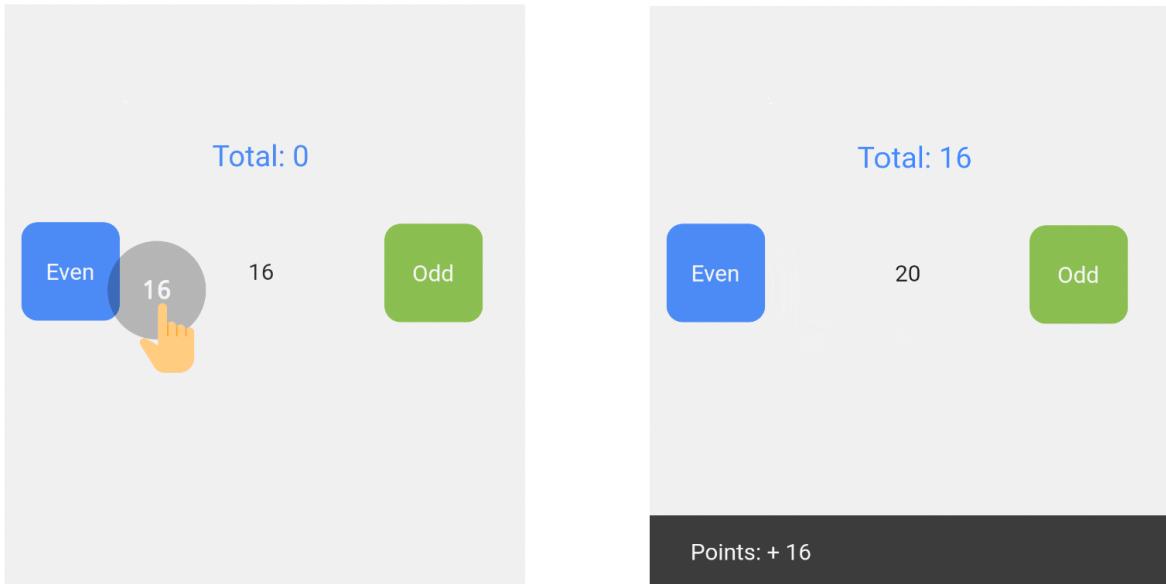
```
context: context,
builder: (BuildContext context) {
    return AlertDialog(
        title: const Text("Confirm"),
        content: const Text("Delete this item?"),
        actions: <Widget>[
            RaisedButton(
                child: const Text("Nope"),
                onPressed: () =>
                    Navigator.of(context)?.pop(false),
            ),
            FlatButton(
                child: const Text("Yes"),
                onPressed: () =>
                    Navigator.of(context)?.pop(true),
            ),
        ],
    );
},
```

The `showDialog<bool>()` method returns a `Future<bool>` whose computed value is the argument of the `pop()` function. In fact, for example when you press on "Yes" the `true` value of the `pop(true)` call is "captured" by the dialog and returned.

 Resources > Chapter 19 > Drag and drop

19.2.2 Dragging items

In this section we're going to see how easy it is to implement the "drag and drop" functionality in Flutter by creating the skeleton of a simple game. Basically a random number appears at the center of the screen and you have to move it to the left or to the right whether it's even or odd.



If you move a number in the correct box, you get points. You could try to use our example as a starting point and then add more functionalities such as a timer, a scoreboard or a more complex point management system.

1. The logic of the game has to be encapsulated in a dedicated class or in a well-designed hierarchy if you plan to make it more complex. In our case, we just need to handle the total amount of gained points and the generation of new values.

```
class GameScore with ChangeNotifier {
  var _score = 0;
  var _currentValue = _random();

  int get score => _score;
  int get currentValue => _currentValue;

  static int _random() => Random().nextInt(100) + 1;

  void addPoints(int pts) {
    _score += pts;
```

```
    _currentValue = _random();  
  
    notifyListeners();  
}  
  
}
```

While `_score` keeps track of the total points earned, `_currentValue` holds the current number that has to be dragged in the "Even" or "Odd" container of the UI.

2. Now we're creating the widget for the game itself. We've removed some styling widgets such as `Padding` and `TextStyle` to focus on the important ones; you'll find the complete code in the online resources.

```
// This is inside a stateless widget called "DragWidget"  
Column(  
  mainAxisAlignment: MainAxisAlignment.center,  
  children: <Widget>[  
    // The blue total score at the top  
    Consumer<GameScore>(  
      builder: (context, game, _) {  
        return Text("Total: ${game.score}");  
      },  
    ),  
  
    // The two boxes and the random number at the center  
    Row(  
      mainAxisAlignment: MainAxisAlignment.spaceAround,  
      children: const <Widget>[  
        EvenContainer(),  
        NumberContainer(),  
        OddContainer(),  
      ],  
    ),  
  ],  
,
```

This is the content of `class DragWidget`, the "container" of the game. At the top there is a `Consumer<GameScore>` that listens to score updates and refreshes the text; below there is the draggable number and the two target boxes.

- With the `Draggable<T>` widget you're able to drag and move around any widget. It has the `data` property which represents the type of value (`T`) it's holding. In our case we're dealing with an `int`.

```
class NumberContainer extends StatelessWidget {
  const NumberContainer();

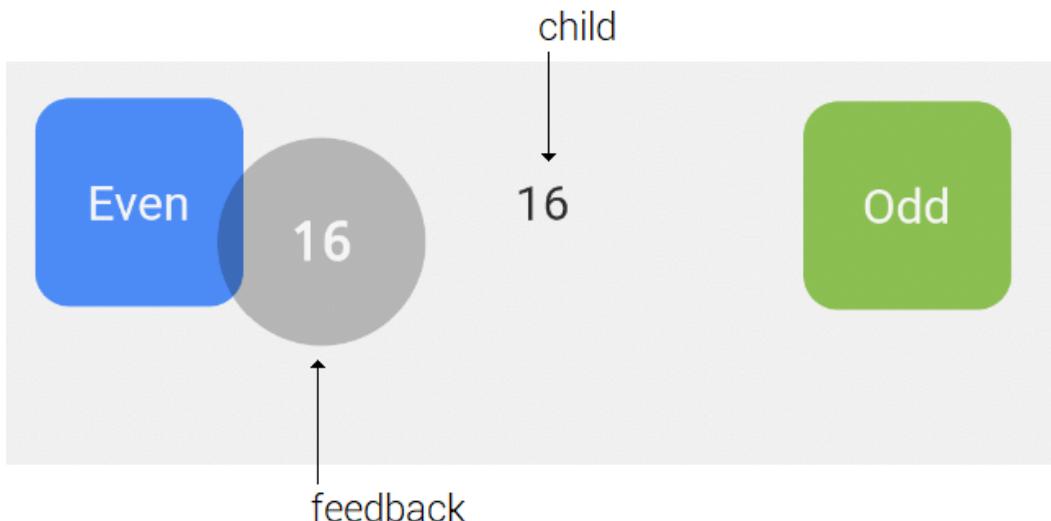
  @override
  Widget build(BuildContext context) {
    return Consumer<GameScore>(
      builder: (context, game, _) {
        return Draggable<int>(
          data: game.currentValue,
          feedback: ...
          child: ...
        );
      },
    );
  }
}
```

The `game.currentValue` variable is the random number appearing at the center of the screen and it has to be linked to the `data` property. In this way, when the widget will be freely dragged around the screen it will also carry the `int` value (the even/odd number).

```
Draggable<int>(
  data: game.currentValue,
  feedback: Container(
    width: 60,
    height: 60,
    decoration: BoxDecoration(
      borderRadius: BorderRadius.circular(60),
      color: Colors.black26,
    ),
    child: Center(
      child: Text("${game.currentValue}"),
    ),
  ),
  child: Container(
```

```
        width: 60,  
        height: 60,  
        child: Center(  
            child: Text("$$game.currentValue"),  
        ),  
    ),  
);
```

The `child` widget usually displays the data that the `Draggable` is holding and `feedback` is how the item should look while it's being dragged.



As you can see, while the number is being dragged there's a circular transparent container around represented by the `feedback` property. It's been styled with a `Container`.

4. Since we want to show a snackbar at the bottom of the screen every time the user scored new points, we're creating a `mixin` to share the implementation across multiple widgets.

```
mixin SnackMessage {  
    void showMessage(BuildContext context, String text) {  
        Scaffold.of(context).showSnackBar(  
            SnackBar(
```

```
        content: Text(text),
        duration: const Duration(milliseconds: 600),
    )
);
}
}
```

The `EvenContainer` and `OddContainer` widgets are going to be the landing zones of the dragged item, depending if it's respectively an even or an odd number. The `_willAccept` method determines whether the received item is valid or not.

```
class EvenContainer extends StatelessWidget with SnackMessage {
    const EvenContainer();

    @override
    Widget build(BuildContext context) {
        return Container(
            width: 60,
            height: 60,
            decoration: BoxDecoration(
                borderRadius: BorderRadius.circular(10),
                color: Colors.blueAccent,
            ),
            child: DragTarget<int>(...),
        );
    }

    void _onAccept(BuildContext context, int data) {
        context.read<GameScore>().addPoints(data);
        showMessage(context, "Points: + $data");
    }

    bool _willAccept(int? data) => data != null && data % 2 == 0;
}
```

A `Draggable<T>` widget can land only in a `DragTarget<T>` widget of the same type. In our specific case we can release a `Draggable<int>` only inside a `DragTarget<int>` widget, and our UI contains two of them.

```
Container(
```

```
...
    child: DragTarget<int>(
        onAccept: (data) => _onAccept(context, data),
        onWillAccept: _willAccept,
        builder: (context, _, __) {
            return const Rectangle(text: "Even");
        },
    ),
),
```

Very intuitively, the `builder` method returns the widget that will be displayed in the UI, which is in our case a blue rectangle with a white text. The `Rectangle` class is a custom widget we've made for reusability.

- `onWillAccept`: the callback is used to determine if this widget is interested in accepting a piece of data being dragged over this target. In practice, it checks if we're trying to put in here a `Draggable` whose value is an even number.
- `onAccept`: if a dropped piece has been accepted because `onWillAccept` returned true, this method is called. In other words, when we're sure that the user dropped an even number, we can give him the points and show the snack bar.

The `OddContainer` widget is very similar; it just changes some styling and the `onWillAccept` callback evaluates to true only if the number is odd.

Note that instead of `context.read<GameScore>()` we could have used the non-extension version, remembering to set `listen: false` because the call is made outside of the widget tree.

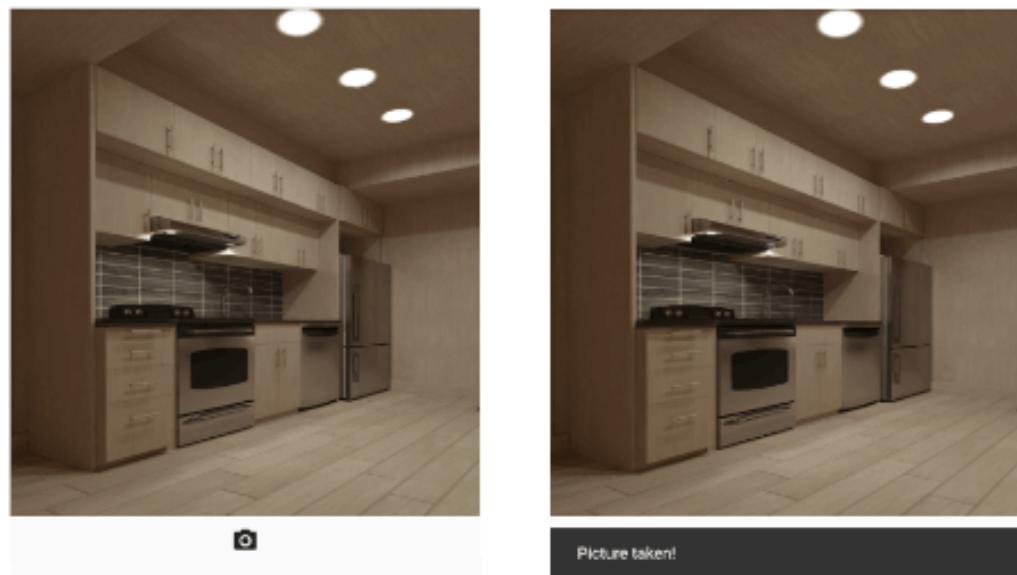
```
// Equivalent to calling 'context.read<GameScore>()'
Provider.of<GameScore>(context, listen: false).addPoints(data);
```

20 | Interacting with the device

(Resources > Chapter 20 > Taking photos)

20.1 Taking a picture

It wouldn't be a surprise if you were asked to take a photo with your app and eventually store the image file you've acquired. The official *camera*¹ package allows you to work with any camera of your device, display previews and acquire photos.



¹<https://pub.dev/packages/camera>

The app we're going to build shows a **live** preview of what's going to be captured and an icon at the bottom which stores the picture in the temporary directory. There are some packages you need to import:

- *camera*, to work with the cameras of the device;
- *path_provider*, to easily find folders' paths;
- *path*, to easily get paths on any platform.

Please note that in order to successfully compile the *camera* plugin you have to change the minimum Android sdk version to 21 at `app/build.gradle`. Look for the `minSdkVersion` label and set it to be 21 (the default value might be lower, such as 16). For iOS users instead add two new entries in `ios/Runner/Info.plist`

- Privacy - Camera Usage Description
- Privacy - Microphone Usage Description

Associate to each key a meaningful description that will be prompted to the user. At this point the setup is ready and we can start creating the app.

1. In the initialization phase, we need to retrieve the list of all available cameras of the device by using `availableCameras()`. Generally there are two, one at the front and one at the back.

```
void main() async {
    // This call makes sure the camera plugin has been
    // properly initialized and it's ready to be used.
    WidgetsFlutterBinding.ensureInitialized();

    final cameras = await availableCameras();
    final firstCamera = cameras.first;

    runApp(...);
}
```

The `cameras` variable is of type `List<CameraDescription>` so calling `cameras.first` is equivalent to writing `cameras[0]`.

2. In order to establish a connection with the device's camera you need a `CameraController` and, like any other controller, it has to be initialized and disposed. For this reason, create a `StatefulWidget` which will take care of the controller's lifecycle.

```
class TakePicture extends StatefulWidget {
    final CameraDescription camera;
    TakePicture({required this.camera});

    @override
    _TakePictureState createState() => _TakePictureState();
}

class _TakePictureState extends State<TakePicture> {
    late final CameraController _controller;
    late final Future<void> _initController;

    @override
    void initState() {
        super.initState();
        _controller = CameraController(
            widget.camera,
            ResolutionPreset.high,
        );
        _initController = _controller.initialize();
    }

    @override
    void dispose() {
        _controller.dispose();
        super.dispose();
    }

    // build...
}
```

You'll see that `_initController` is used in a `FutureBuilder<T>` to wait for the loading of the camera preview screen. You can decide to take high or low quality pictures with one of these controller's constants:

- `ResolutionPreset.low` (240p),
- `ResolutionPreset.medium` (480p),

- ResolutionPreset.high (720p),
- ResolutionPreset.veryHigh (1080p),
- ResolutionPreset.ultraHigh (2160p)

Alternatively, if you don't know which is the maximum resolution supported by the device, you can use `ResolutionPreset.max` which automatically picks the highest preset possible. For the sake of the example we've simply used a `Column` but you could try to make a more complex layout.

```
@override
Widget build(BuildContext context) {
    return Column(
        mainAxisAlignment: MainAxisAlignment.spaceAround,
        children: <Widget>[
            FutureBuilder<void>(...),
            IconButton(...),
        ],
    );
}
```

3. The `FutureBuilder<void>` widget is going to `await` for the initialization of the camera because, as soon as it's ready, we want it to give a live preview of what it's framing.

```
FutureBuilder<void>(
    future: _initController,
    builder: (context, snapshot) {
        if (snapshot.connectionState == ConnectionState.done) {
            return Expanded(
                child: Center(
                    child: AspectRatio(
                        aspectRatio: _controller.value.aspectRatio,
                        child: CameraPreview(_controller),
                    ),
                ),
            );
        }
    },
)
return const Center(
```

```
        child: CircularProgressIndicator(),
    );
},
),
,
```

We recommend the usage of the `AspectRatio` widget as it shows the preview with proper proportions between width and height. It's not compulsory, you could have put the `CameraPreview` widget directly inside `Expanded` and it would have covered the entire available space.

```
return Expanded(
    child: CameraPreview(_controller),
)
```

Since we're inside a `Column` we need somehow to constrain the height and `Expanded` does exactly this. Alternatively you could have used a `Container` or a `SizedBox`.

4. In the `FutureBuilder<void>` we've placed the button to actually take the photo and store the image in the temporary directory with a "random" filename.

```
IconButton(
    icon: const Icon(Icons.photo_camera),
    onPressed: () => _takePhoto(context),
)
```

We underline again the fact that it's a good idea putting the business logic outside of the UI logic and thus the body of the `onPressed` callback has been created in a separated function.

```
void _takePhoto(BuildContext context) async {
    // Ensure the controller is ready
    await _initController;

    // File name and path
    final dir = await getTemporaryDirectory();
    final name = "mypic_${DateTime.now()}.png";

    // Store the picture at the given location
    final fullPath = path.join(dir.path, name);
    await _controller.takePicture(fullPath);

    Scaffold.of(context).showSnackBar(
```

```
        SnackBar(  
            content: const Text("Picture taken!"),  
            duration: const Duration(milliseconds: 600),  
        )  
    );  
}
```

A common strategy to avoid file name conflicts is relying on the current date and time in the name. While this approach is not bullet proof, for simple tasks it's good enough. Note the usage of *join*:

```
final fullPath = path.join(dir.path, name);
```

Combining paths in cross-platform apps is not easy because you have to take into account the filesystem structure of every single supported OS. This is something low-level that you don't want to deal with, so Flutter has a package ready for you and it's called *path*!

 Resources > Chapter 20 > Sensors

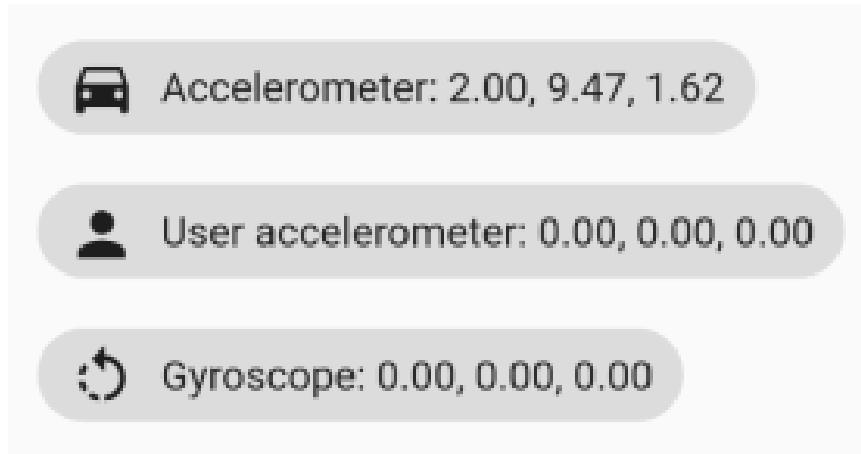
20.2 Working with sensors

Nowadays almost any device has built-in sensors to measure orientation, motion or other environmental conditions with a very high accuracy. To monitor the three-dimensional data the device is able to provide you can use the official *sensor*² package.

- **GyroscopeEvents**: data about the rotation of the device;
- **AccelerometerEvent**: these data let you know in which direction the device is moving, keeping into account the gravity;
- **UserAccelerometerEvent**: similar to **AccelerometerEvent** but it doesn't care about the gravity

We're creating a very basic app that shows data obtained from the sensors at the center of the screen. They are useful when you want to implement motion-based functionalities in your app.

²<https://pub.dev/packages/sensors>



The API of the sensor package is very simple to use and it relies on streams.

1. The `Wrap` widget places the children side by side in the horizontal or vertical direction. It's very similar to `Row` and `Column` but with the difference that its sizes are **constrained**: when there is no more space to lay out the children, it wraps to the next column/row.

```
class SensorWidget extends StatelessWidget {
  const SensorWidget();

  @override
  Widget build(BuildContext context) {
    return Wrap(
      spacing: 0.9,
      direction: Axis.vertical,
      children: const <Widget>[
        AccelerometerData(),
        UserAccelerometerData(),
        GyroscopeData()
      ],
    );
  }
}
```

Here we're placing a series of widgets vertically with a certain space in between. If you try to rotate the screen and set a big gap, such as `spacing: 150`, you'll see that widgets are rearranged differently and no overflow errors appear.

- As you know, the `StreamBuilder<T>` widget deals with streams and we strongly recommend you to **always** supply some initial data to it. This ensures that the first frame will show useful data rather than `null`, which is safe.

```
class AccelerometerData extends StatelessWidget {
    static final initialData = AccelerometerEvent(0, 0, 0);
    const AccelerometerData();

    @override
    Widget build(BuildContext context) {
        return StreamBuilder<AccelerometerEvent>(
            stream: accelerometerEvents,
            initialData: initialData,
            builder: (context, accel) {
                final data = accel.data;

                if (data != null) {
                    final x = data.x.toStringAsFixed(2);
                    final y = data.y.toStringAsFixed(2);
                    final z = data.z.toStringAsFixed(2);

                    return ChipWithIcon(
                        icon: Icon(Icons.drive_eta),
                        text: "Accelerometer: $x, $y, $z"
                    );
                }

                return const Text("No data to display.");
            },
        );
    }
}
```

The `accelerometerEvents` stream is provided by the `sensor` package and it continuously sends a series of three-dimensional data we're showing on the UI. At the center of the screen

there's a material Chip, a rounded box representing short spans of text.

```
class ChipWithIcon extends StatelessWidget {
    final Icon icon;
    final String text;
    const ChipWithIcon({
        required this.icon,
        required this.text
    });

    @override
    Widget build(BuildContext context) {
        return Chip(
            avatar: icon,
            padding: const EdgeInsets.fromLTRB(10, 5, 10, 5),
            labelPadding: const EdgeInsets.only(left: 10),
            label: Text(text),
        );
    }
}
```

We haven't discussed the implementations of the other two listeners because they're basically identical to `AccelerometerData` with the only difference that they subscribe to a different stream.

 Resources > Chapter 20 > Geolocation

20.3 Working with Geolocation

A very popular geolocation package for Flutter is `geolocator`³, well-known for its easy way to access the device's location services. Before starting with the examples, apply the following changes to your project:

- On **Android** you need to add two permissions in the manifest at `android/app/src/main`.

```
<uses-permission
    android:name="android.permission.ACCESS_FINE_LOCATION" />
```

³<https://pub.dev/packages/geolocator>

```
<uses-permission  
    android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

In addition, make sure the `gradle.properties` file has AndroidX enabled by checking if these two lines are present. They are required for version 3.0.0 and above of the plugin to work.

```
android.useAndroidX=true  
android.enableJetifier=true
```

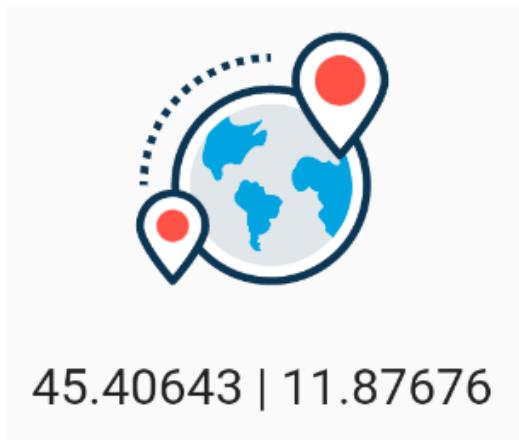
- On **iOS** you have to setup the permission in the `Info.plist` file.

```
<key>NSLocationWhenInUseUsageDescription</key>  
<string>Give me location permissions pls</string>
```

Note that in early versions you had to access members using `class Geolocator` which implemented the *singleton* design pattern via `factory` constructor. Now the class doesn't exist anymore: the package just exposes a series of top-level functions.

```
var position = await getCurrentPosition(  
    desiredAccuracy: LocationAccuracy.high  
)  
  
var lat = position.latitude;  
var lon = position.longitude;  
var alt = position.altitude;
```

You could also use `position.speed` to indicate the speed at which the device is moving but it's not always available. There's also the possibility to listen for position changes thanks to `getPositionStream()` which returns a `Stream<Position>`.



In this very simple example we have an SVG image at the top and below there is a string reporting the current latitude and longitude. If we went out for a walk, we would see the text automatically updating to reflect the new coordinates thanks to the stream.

1. The setup for the `GPSWidget` is very easy to understand. We have decided to make the `_stream` variable `static` to be able to declare a `const` constructor.

```
class GPSWidget extends StatelessWidget {
    static final _stream = getPositionStream();
    const GPSWidget();

    @override
    Widget build(BuildContext context) {
        ...
    }
}
```

2. As usual, we're going to use the `StreamBuilder<T>` widget to subscribe to the stream and constantly update the text below the image.

```
return Wrap(
    spacing: 20,
    direction: Axis.vertical,
    crossAxisAlignment: WrapCrossAlignment.center,
    children: <Widget>[
```

```
SvgPicture.asset(  
    "assets/geolocate.svg",  
    height: 70,  
,  
  
StreamBuilder<Position>(  
    stream: _stream,  
    builder: (context, positionData) {  
        if (positionData.hasData) {  
            final data = positionData.data;  
  
            if (data != null) {  
                final lat = data.latitude.toStringAsFixed(5);  
                final lon = data.longitude.toStringAsFixed(5);  
  
                return Text("$lat | $lon");  
            } else {  
                return const Text("No data available.");  
            }  
        }  
  
        return const CircularProgressIndicator();  
    },  
,  
)  
,  
);
```

Alternatively to `Wrap` you could have achieved the same result with a column but it would have required more boilerplate code. It has no `spacing` property so you'd have to deal with `Padding` or another "spacer" widget.

```
// A 'SizedBox' can be good to add spaces between widgets  
const SizedBox(  
    height: 20  
,
```

If you wish to calculate the distance between two points in the world, use `distanceBetween()` and pass latitude and longitude of the starting and ending point. The distance is expressed in meters.

```
final double distance = await distanceBetween(
  52.2165157, 6.9437819, // starting point lat-long
  52.3546274, 4.8285838 // ending point lat-long
);
```

Another very interesting package is called *geocoding* because it gives you the possibility to translate latitude/longitude coordinates into an address and vice versa. It's very easy to use:

```
final placemarks = await placemarkFromCoordinates(41.909986, 12.3959135);

if (placemarks.isNotEmpty) {
  final result = placemarks[0];
  debugPrint("${result.administrativeArea}"); // Lazio
  debugPrint("${result.locality}"); // Rome
  debugPrint("${result.country}"); // Italy
}
```

Ideally the `placemarks` list will contain only 1 item that exactly matches what you're looking for but in certain cases there might be multiple results. It's of type `List<Placemark>` and it has a ton of properties ⁴ (in addition to the ones we've used above).

```
var locations = await locationFromAddress("Rome");

if (locations.isNotEmpty) {
  var result = locations[0];
  debugPrint("${result.latitude}"); // 41.9...
  debugPrint("${result.longitude}"); // 12.3...
}
```

This is very similar to the previous example but instead of passing the latitude and the longitude, we pass the name of the desired location.

20.4 Platform-specific packages

Flutter's official pub ⁵ page has many packages to interact with some platform-specific APIs. Be sure to also have a look at the GitHub repo ⁶ for a summary table that redirects you to the install page of the packages.

⁴<https://pub.dev/packages/geocoding>

⁵<https://pub.dev/flutter/packages>

⁶<https://github.com/flutter/plugins>

20.4.1 Battery level

The `battery` package lets you extract various information about your device's battery such as the current charge level. Once it's been imported you just need to create an instance of `class Battery` and then use its asynchronous API.

```
Future<int> level() async {
  var b = Battery();
  return await b.batteryLevel;
}
```

If you wanted your device to listen for battery changes, subscribe to the `onBatteryStateChanged` stream which exposes a `BatteryState`. It is an `enum` representing the current state of the battery.

```
enum BatteryState {
  /// The battery is completely full of energy.
  full,

  /// The battery is currently storing energy.
  charging,

  /// The battery is currently losing energy.
  discharging
}
```

20.4.2 Device info

To get a lot of info about the device in which your app is running you can use the `device_info`⁷ package from the official Flutter team. It's just a matter of instantiating `DeviceInfoPlugin()` and then using the getters.

```
final deviceInfo = await DeviceInfoPlugin().androidInfo;
//final deviceInfo = await DeviceInfoPlugin().iosInfo;

debugPrint(deviceInfo.display);
debugPrint(deviceInfo.brand);
debugPrint(deviceInfo.product);
```

⁷https://pub.dev/packages/device_info

```
debugPrint(deviceInfo.manufacturer);  
// and much more...
```

There is a very long series of data this package is able to extract so be sure to check out the documentation.

 Resources > Chapter 20 > Connectivity

20.4.3 Internet connectivity

So far we've used *dio* and *http* to make HTTP requests, download/upload files and handle potential errors. They're not able to detect the connection status before making the actual request but the *connectivity*⁸ package can.

```
var status = await Connectivity().checkConnectivity();  
  
switch (status) {  
  case ConnectivityResult.wifi:  
    debugPrint("Connected via wifi");  
    break;  
  case ConnectivityResult.mobile:  
    debugPrint("Connected via mobile");  
    break;  
  case ConnectivityResult.none:  
    debugPrint("not connected");  
    break;  
}
```

The `checkConnectivity()` method returns `enum ConnectivityResult` which lets us know if we are re connected to a network (via wifi or mobile) or not. In addition, we can also listen to connectivity changes while the app is running thanks to streams.

⁸<https://pub.dev/packages/connectivity>



Thanks to a `StreamBuilder<ConnectivityResult>` we are able to listen to changes and show proper widgets accordingly. You've seen this pattern many times up to now so it should be easy to understand.

1. The `Connectivity` class implements the singleton pattern so it will always return the same instance via `factory` constructor. For this reason, we can "cache" an instance in a `static` variable to define a constant constructor.

```
class ConnectionWidget extends StatelessWidget {  
    static final _conn = Connectivity();  
    const ConnectionWidget();  
  
    @override  
    Widget build(BuildContext context) {  
        ...  
    }  
}
```

2. For each connection type (wifi, mobile or none) we're creating the widget representing the status with an icon and the text.

```
class WifiConnectionWidget extends StatelessWidget {  
    const WifiConnectionWidget();  
  
    @override  
    Widget build(BuildContext context) {  
        return Wrap(  
            spacing: 20,  
            crossAxisAlignment: WrapCrossAlignment.center,  
            direction: Axis.vertical,  
            children: const <Widget>[
```

```
        Icon(Icons.wifi),
        Text("Connected to WiFi")
    ],
);
}
}

class MobileConnectionWidget extends StatelessWidget { ... }

class NoConnectionWidget extends StatelessWidget { ... }
```

3. At this point we can setup the `StreamBuilder<ConnectivityResult>` so that it can listen to connectivity changes and display the proper widget. A result of type `none` indicates that there's no internet connectivity at all.

```
@override
Widget build(BuildContext context) {
    return StreamBuilder<ConnectivityResult>(
        stream: _conn.onConnectivityChanged,
        builder: (context, status) {
            if (status.hasData) {
                final data = status.data;

                if (data != null) {
                    switch (data) {
                        case ConnectivityResult.wifi:
                            return const WifiConnectionWidget();
                        case ConnectivityResult.mobile:
                            return const MobileConnectionWidget();
                        case ConnectivityResult.none:
                            return const NoConnectionWidget();
                    }
                } else {
                    debugPrint("Whoops");
                }
            }
            return const CircularProgressIndicator();
        },
    );
}
```

```
    );  
}
```

20.4.4 Shared preferences

The `shared_preferences`⁹ package is very useful when you have a small collection of key-value data to save. Those data persist on the disk so even if you close the app, they will be available again at the next startup. Some scenarios in which you could use them are:

- storing some flags your app has to read at startup, such as styling configurations set by the user;
- storing some app's preferences such as the currently selected download directory;
- storing all those "small" data like numbers or strings you want to quickly read without having to setup a database.

Use `class SharedPreferences` when you want to work with `int`, `double`, `String` or `bool`. If you have a lot of configurations to store or a complex data structure, consider using a more reliable data structure such as a database¹⁰.

```
final prefs = await SharedPreferences.getInstance();  
  
// Store a value  
await prefs.setInt('age', 23);  
  
// Read a value  
var value = prefs.getInt('age') ?? -1;
```

If `age` wasn't an integer an exception would occur. Do not store critical data using shared preferences nor rely on them to build your app's storage; they are just a simple way to store small data without having the overhead of managing a database.

```
SharedPreferences.setMockInitialValues({  
  "name": "Robert",  
  "age": 28  
});
```

Use `setMockInitialValues()` in your tests to manually populate `SharedPreferences` with some initial values.

⁹https://pub.dev/packages/shared_preferences

¹⁰See appendix B.2

21 | Widgets showcase

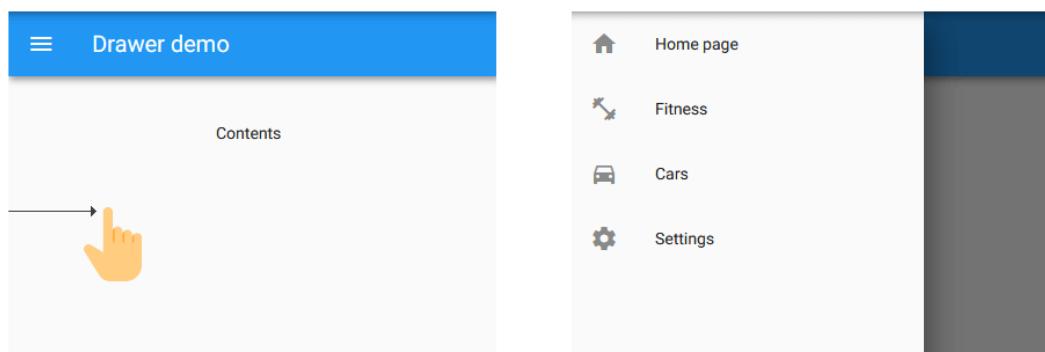
In this section we're going to list a series of well known widgets that follow the "Material" or "Cupertino" design styles. A complete and updated list of any widget made by the Flutter team is available in their official online catalog¹.

21.1 Material

- Reference: <https://material.io>

21.1.1 Drawer

A navigation drawer is a panel that slides from a side to the center of the screen and contains a series of options. The material implementation of a drawer, called `Drawer`, is generally declared inside a `Scaffold` because they nicely integrate together.



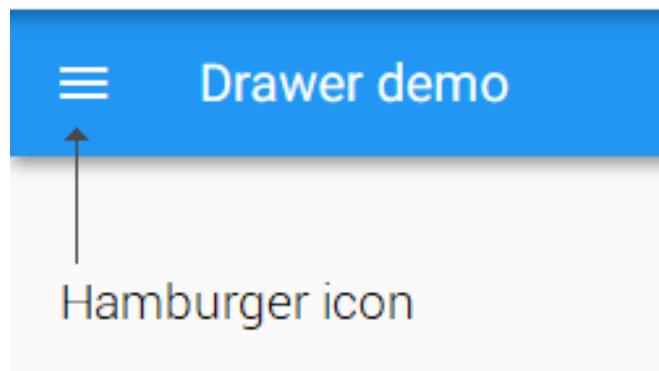
¹<https://flutter.dev/docs/development/ui/widgets>

A drawer generally is made up of a header (with an image or some user data) and then a series of clickable options below. In general, when an item is tapped the user navigates to another route. A drawer is very useful when there are a lot of possible routes but there isn't enough space in the UI to show them all.

1. By default, a drawer can slide from the left side of the screen to the center hovering the current view. Even if it's not required, you should always use a `ListView` as child in order to enable scrolling in case the vertical space weren't enough.

```
Scaffold(  
    appBar: AppBar(),  
    body: const MyPage,  
    drawer: Drawer(  
        child: ListView(  
            children: [...]  
        ),  
    ),  
) ,
```

The `Scaffold` automatically adds an hamburger icon in the top bar so that the drawer can also be opened by tapping on that button.



There's also the possibility to add `endDrawer: Drawer(...)` which is simply a drawer sliding from the opposite side: from the right of the device to the center.

2. In general it's a good idea having a drawer in a dedicated widget, so you should create a stateless widget (called for example `class DrawerMenu`) and put all the logic there. Use a

`ListView` to group the children.

```
class DrawerMenu extends StatelessWidget {
    const DrawerMenu();

    @override
    Widget build(BuildContext context) {
        return ListView(
            children: [...],
        );
    }
}
```

3. If you want to create a header section in your drawer, Flutter has the `DrawerHeader` class. It's very customizable but of course, if it doesn't fit your needs, you can always create a completely new widget from scratch.

```
ListView(
    children: [
        DrawerHeader(
            padding: EdgeInsets.all(15),
            child: Wrap(
                direction: Axis.vertical,
                children: const <Widget>[
                    Icon(Icons.person),
                    Text("myemail@gmail.com")
                ],
            ),
        ),
    ],
)
```

We're going to show how to create items of a navigation drawer with a very Android-like implementation. Tapping on each item opens a new route because generally it's the expected behavior from a drawer item.



By convention, drawer items are represented by `class ListTile` which shows an icon and a short piece of text. You can also visually group tiles using a `Divider` and maybe give a name to the group with a `Text` widget.

```
ListView(  
  children: [  
    ListTile(  
      leading: const Icon(Icons.email),  
      title: const Text("My e-mail"),  
      onTap: () => Navigator.of(context)?pushNamed(...),  
    ),  
  
    const Divider(  
      height: 1,  
      color: Colors.grey,  
    ),  
  
    const DrawerTitle("App management"),  
  
    ListTile(  
      leading: const Icon(Icons.settings),  
      title: const Text("Settings"),  
      onTap: () => Navigator.of(context)?pushNamed(),  
    ),  
  
    ListTile(  
      leading: const Icon(Icons.info),  
      title: const Text("Info"),  
    ),  
  ],
```

```
        onTap: () => Navigator.of(context)?.pushNamed(),
    ),
    const Divider(
        height: 1,
        color: Colors.grey,
    ),
],
);
```

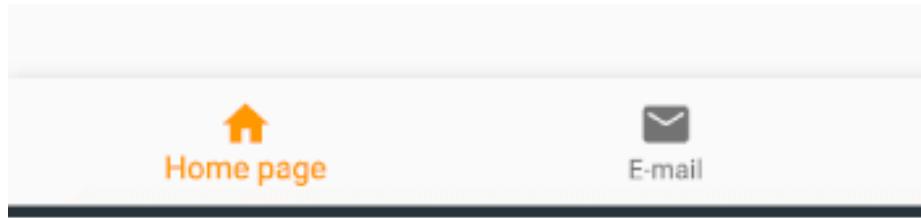
Since there might be more than a single section title in your drawer, we've decided to create a simple reusable `DrawerTitle` widget that styles some text. Note that a `Divider` can also have a custom thickness and color.

```
class DrawerTitle extends StatelessWidget {
    final String text;
    const DrawerTitle(this.text);

    @override
    Widget build(BuildContext context) {
        return Padding(
            padding: const EdgeInsets.fromLTRB(15, 15, 0, 5),
            child: const Text("App management",
                style: TextStyle(
                    fontSize: 15,
                    fontWeight: FontWeight.w500
                ),
            ),
        );
    }
}
```

21.1.2 BottomNavigationBar

This material widget is displayed at the bottom of the screen and it's generally used to navigate among a small amount of pages. There are no restrictions on the maximum icon number but you shouldn't use more than 4 items otherwise the bar becomes too dense.



The selected icon is highlighted with a custom color. You could also decide to only show the text or only the image. To create this kind of navigation bar, you need again a `Scaffold`, the basic building block of any material app, and then a `BottomNavigationBar`.

```
Scaffold(  
    appBar: AppBar(...),  
    body: const MyPage(),  
    bottomNavigationBar: BottomNavigationBar(  
        currentIndex: 1,  
        selectedItemColor: Colors.orange,  
        onTap: (int index) {...},  
        items: const [  
            BottomNavigationBarItem(  
                icon: Icon(Icons.home),  
                label: "Home page"  
            ),  
            BottomNavigationBarItem(  
                icon: Icon(Icons.email),  
                label: "E-mail",  
            ),  
        ],  
    ),  
)
```

The `currentIndex` parameter indicates which icon at the bottom has been tapped and thus selected. `onTap` is a callback for any children in `items` and it should act on the `currentIndex` property. In the above example we have hard coded it but in reality your code would look like this:

```
bottomNavigationBar: BottomNavigationBar(
    currentIndex: selectedIndex,
    onTap: (int index) => _changePage(index),
    items: const [...],
),
```

Using *provider* or *flutter_bloc* you can make it so that `int selectedIndex = 0` is declared somewhere and then, using `void _changePage(int index)`, you change the value to rebuild the `Scaffold`. In other words, changing the value of `currentIndex` also changes the currently visible page.

21.1.3 NavigationRail

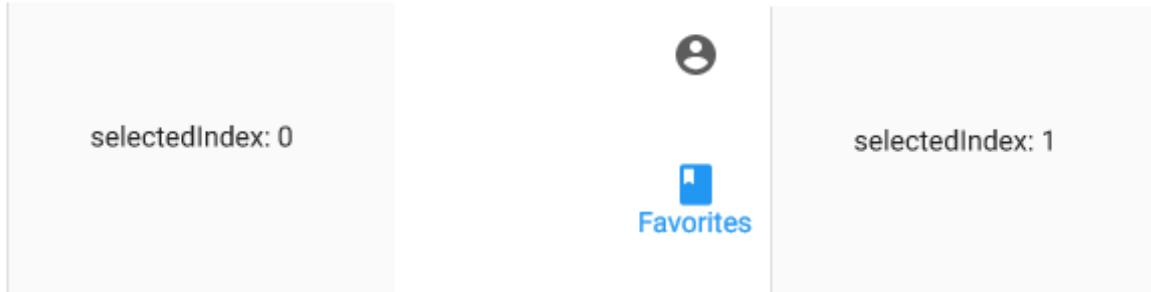
A `class NavigationRail`² is basically a `BottomNavigationBar` that appears on the left or right of the screen (rather than at the bottom). It's generally used to navigate among a small group of routes, typically four or at maximum five.

```
Scaffold(
    body: Row(
        children: <Widget>[
            // Contains the circles
            NavigationRail(
                selectedIndex: currentPage,
                onDestinationSelected: (int index) {...},
                destinations: const [
                    NavigationRailDestination(
                        icon: Icon(Icons.account_circle),
                        label: Text('User'),
                    ),
                    NavigationRailDestination(
                        icon: Icon(Icons.book),
                        label: Text('Favorites'),
                    ),
                ],
            ),
            const VerticalDivider(
                thickness: 1,
```

²<https://api.flutter.dev/flutter/material/NavigationRail-class.html>

```
        width: 1
    ),
    // Actual content of the page
    Expanded(
        child: Center(
            child: Text('selectedIndex: $currentPage'),
        ),
    ),
],
),
);
```

The `int currentPage` variable should be handled by a state management library as it indicates which page is currently visible. Changing the value of `currentPage`, which has to happen inside the `onDestinationSelected` callback, also changes the selected item in `destinations`.



The tappable circles are represented by `class NavigationRailDestination` where the only required parameter is the icon. If you want the navigation rail to be at the right of the screen, just put it as last child of the `Row()`:

```
Row(
    children: <Widget>[
        Expanded(...),
        const VerticalDivider(...),
        NavigationRail(...),
    ],
);
```

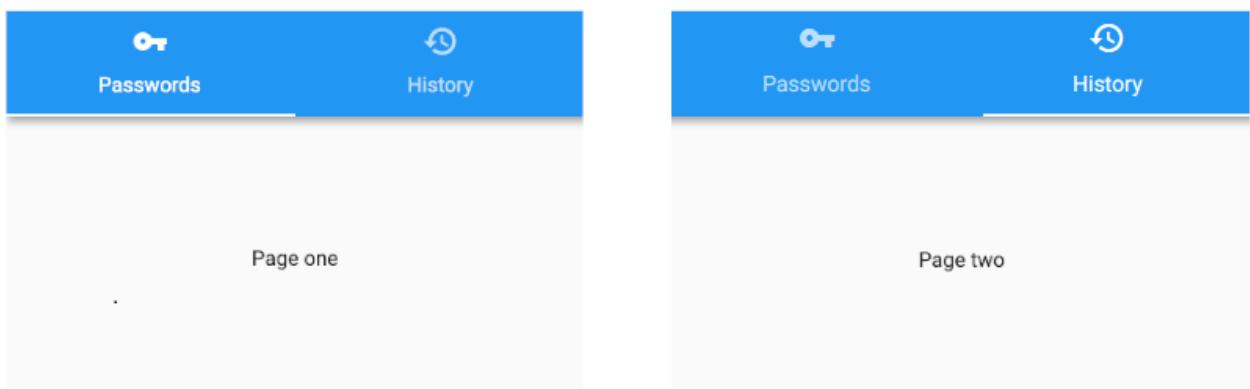
There's also the possibility to change the icon of a selected element by setting the `selectedIcon` property. For example:

```
NavigationRailDestination(  
    icon: Icon(Icons.info_outline),  
    selectedIcon: Icon(Icons.info),  
) ,
```

By default the `Icons.info_outline` icon appears but when the destination is tapped, it's replaced by a `Icons.info`.

21.1.4 TabBar

Working with tabs in Flutter is very easy because it's just a matter of using a controller and a `Scaffold`; on Android instead there would be a lot of styling and coding to do. A tabbed layout is used to group multiple pages in a single view: the user can move among tabs by swiping or by tapping on the top bar.



Similarly from what we've suggested for a `BottomNavigationBar`, you should display three or at maximum four tabs in the page. There aren't any limitations but if you have too many icons the bar becomes too dense and not optimal in terms of user experience (which is essential).

```
DefaultTabController(  
    length: 2,  
    child: Scaffold(  
        appBar: AppBar(
```

```
title: const Text("Using tabs"),
bottom: const TabBar(
    tabs: [
        Tab(
            icon: Icon(Icons.vpn_key),
            text: "Passwords",
        ),
        Tab(
            icon: Icon(Icons.history),
            text: "History",
        ),
    ],
),
body: const TabBarView(
    children: [
        PageOne(),
        PageTwo(),
    ],
),
),
```

There's the need to use a `DefaultTabController` widget to make everything work and the `length` parameter must reflect the actual number of tabs, otherwise an exception is thrown. The `TabBar` widget handles the tabs in the top bar, `TabBarView` instead handles the body of the tabs (the actual pages).

```
class PageOne extends StatelessWidget {
    const PageOne();
    ...
}

class PageTwo extends StatelessWidget {
    const PageTwo();
    ...
}
```

Be sure to create the children of a `TabBarView` into separated widgets so that each page lays in

its dedicated file/class. The `DefaultTabController` widget internally contains a `TabController` which is automatically managed. However, if you want to have more control of your tabs, you should directly work with a controller:

```
class _PageState extends State<Page> with SingleTickerProviderStateMixin {
    late final TabController tabController;

    @override
    void initState() {
        super.initState();
        tabController = TabController(
            vsync: this,
            length: 2,
        );
    }

    @override
    void dispose() {
        tabController.dispose();
        super.dispose();
    }

    void _changePage(int index) {
        if (index >= 0) {
            tabController.animateTo(index);
        }
    }

    @override
    Widget build(BuildContext context) {
        return TabBarView(
            controller: tabController,
            children: const [...],
        );
    }
}
```

There's more code to write because now you have to manually create, initialize and dispose a controller while earlier the `DefaultTabController` did it for you. The advantage is that you

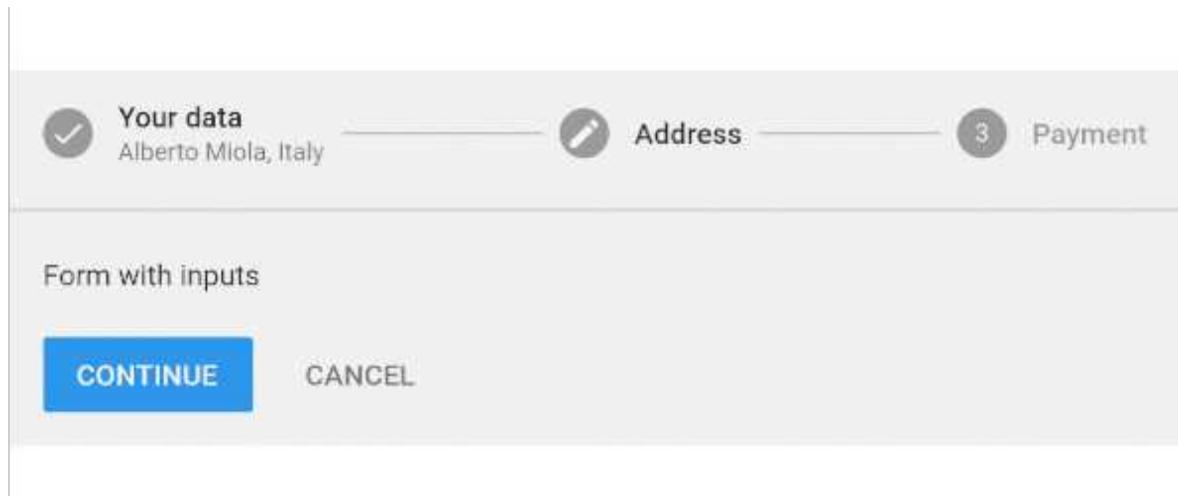
have more control of the tabs because, for example, you can change tabs programmatically:

```
tabController.animateTo(index);
```

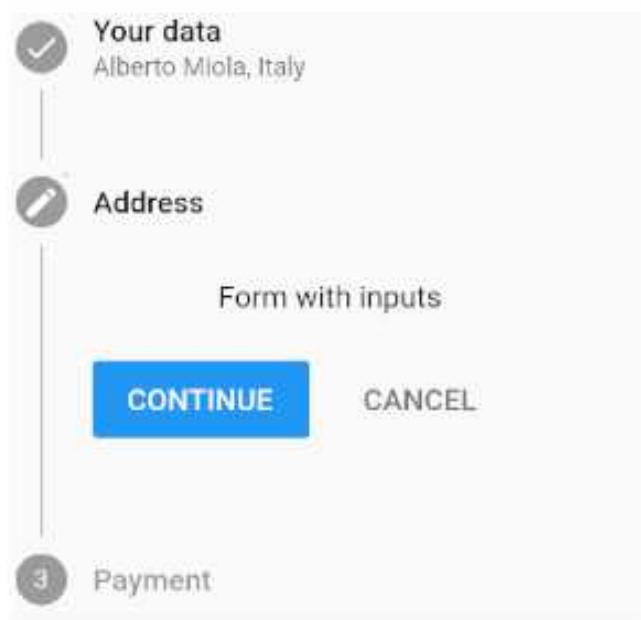
Thanks to `animateTo()` you can move to the desired tab with a swipe animation without needing the user's finger to swipe/tap. Unless you need to do special manual work on tabs, go for a `DefaultTabController` which is much less boilerplate code.

21.1.5 Stepper

If you need a widget that guides the user over a series of steps before getting a result, a `Stepper` might be the right choice. It is very useful when there is a form to fill and certain types of data require the submission of other data in order.



There's the possibility to control the contents of the circles, which can be a tick or a pencil to indicate the current status, and the user can navigate back/forth. A `Stepper` can also be used in vertical mode which is probably better when there is a lot of available height:



You could decide to always use the stepper in a single direction, which is fine as it provides automatic scrolling in case of overflow. However, you should consider the usage of `LayoutBuilder` to make the widget responsive:

```
LayoutBuilder(  
    builder: (context, dimensions) {  
        var stepperType = StepperType.vertical;  
  
        if (dimensions.maxWidth > 550) {  
            stepperType = StepperType.horizontal;  
        }  
  
        return Stepper(  
            type: stepperType,  
            steps: const [...],  
        );  
    },  
,
```

A `Stepper` takes a list of `Step` indicating the single actions the user is asked to accomplish. Note

that `class Step { ... }` is **not** a widget but it still requires some parameters:

- `currentStep`. It's the currently visible step and it should be handled via `provider` or `flutter_bloc`, for example. The `int` associated with this value has to change to reflect the actual step the user has to see.
- `onStepContinue`. Callback triggered when the "Continue" button is pressed; it should be used to increment the value of `currentStep`.
- `onStepCancel`. Callback triggered when the "Cancel" button is pressed; it should be used to decrement the value of `currentStep`.

In this example we've hard coded some values just for the sake of simplicity. Your state management library should also handle the `state` parameter which changes the icon associated with the name of the step in the UI. If you don't set it, numbers are shown by default.

```
Stepper(  
    currentStep: 1,  
    type: stepperType,  
    onStepContinue: () {...},  
    onStepCancel: () {...},  
    steps: const [  
        Step(...),  
  
        Step(  
            title: Text("Address"),  
            content: Text("Form with inputs"),  
            state: StepState.editing,  
        ),  
  
        Step(...),  
    ],  
);
```

Callbacks should be defined into separated functions rather than directly inside the `Stepper`.

21.1.6 DataTable

Sometimes certain type of information can only be represented in a table because other kind of layouts just don't fit well. You could start using the `Table` widget but you'd have to manually

Chapter 21. Widgets showcase

code any extra feature (like sorting) by yourself. A very good alternative is the `DataTable` widget.

Language	Likes	Mascot
Java 	6	Duke
Dart 	8	Dash

Thanks to the `DataTable` widget you're able to easily display tabular data, containing of course any kind of widget. Columns automatically resize to fit the contents. Other than the following, there are many built-in options to enable such as the possibility to sort the entire table just by tapping on the header.

```
SingleChildScrollView(  
    child: DataTable(  
        sortColumnIndex: 1,  
        sortAscending: false,  
        columns: const [...],  
        rows: [...],  
    ),  
,  
,
```

You can decide the sorting "direction" thanks to `sortAscending`, define callbacks for rows, setting margins and much more. We recommend wrapping the table in a `SingleChildScrollView` to ensure that overflows will be automatically handled with scroll bars. Use a `DataColumn` widget to setup a column of a `DataTable`.

```
columns: [  
    DataColumn(  
,
```

```
        label: const Text("Language"),
        onSort: (int colIndex, bool ascending) {...}
),
const DataColumn(
    label: Text("Likes"),
    numeric: true
),
const DataColumn(
    label: Text("Mascot")
),
],
],
```

With the `column` parameter you give the header a name, set whether it's numeric (which is `false` by default) and/or set a callback for sorting purposes. If `onSort` is not defined, the column is considered to be "fixed" and thus it **cannot** be sorted.



Each cell of the table can be populated with `rows`, which is allowed to only contain a list of `DataRow`s. The `DataCell` widget can contain anything such as images, icons, text, SVG and so on. In our example we've populated the cell with a `Wrap` widget.

```
rows: [
  DataRow(cells: [
    DataCell(
      Wrap(
        spacing: 5,
```

```
        children: const [
            Text("Java"),
            Icon(Icons.verified_user,
                color: Colors.green,
                size: 15,
            )
        ],
    ),
),
const DataCell(Text("6")),
const DataCell(Text("Duke")),
]),
DataRow(cells: [
    DataCell(...),
    const DataCell(Text("8")),
    const DataCell(Text("Dash")),
]),
],
],
```

The number of children in a `DataRow` (the table cells) must be equal to the number of columns defined in the table otherwise an exception is thrown. Each `DataCell` also defines an `onTap` callback and the possibility to show an icon to edit the field.

```
DataCell(
    onTap: () {...}
    showEditIcon: true,
),
```

The official `DataTable` documentation says that tables are relatively expensive to be rendered so when you have a lot of data to display you should paginate the results. Indicatively, 10x10 or 12x12 tables are considered "big" but actually it also depends a lot on the physical device. An old and slow mobile phone for example might be in trouble even with a few dozens of data.

Language	Mascot	Language	Mascot
Flutter	Dash	PHP	ElePHPant
Java	Duke	MySQL	Dolphin
Delphi	Helmet		

1-3 of 5 < > 4-6 of 5 < >



The above result is obtained with a `PaginatedDataTable` widget: it limits the number of visible rows by paginating the results. In this way, data can be lazy-loaded avoiding an expensive build of the table and the user is not presented a too long list of data on the screen.

```
PaginatedDataTable(  
    rowsPerPage: 3,  
    header: const Text("Info"),  
    columns: const [  
        DataColumn(  
            label: Text("Language"),  
        ),  
        DataColumn(  
            label: Text("Mascot"),  
        ),  
    ],  
    source: Source(),  
) ;
```

You can decide how many rows have to appear per page with `rowsPerPage`. The header is required and it's just a widget that appears at the top of the table. Rows aren't assigned in the "classic" way using a `rows` parameter but instead there's the need to subclass a `DataTableSource`, which exposes data to the table.

```
class Source extends DataTableSource {
    // The data of the table. They're here as a static list just
    // to keep the example simple.
    static final values = {
        "Flutter": "Dash",
        "Java": "Duke",
        "Delphi": "Helmet",
        "PHP": "ElePHPant",
        "MySQL": "Dolphin",
    }.entries.toList();

    @override
    DataRow? getRow(int index) {
        final data = values[index];

        return DataRow.byIndex(
            index: index,
            cells: [
                DataCell(Text(data.key)),
                DataCell(Text(data.value)),
            ]
        );
    }

    @override
    bool get isRowCountApproximate => false;

    @override
    int get rowCount => values.length;

    @override
    int get selectedRowCount => 0;
}
```

It's declared as `abstract class DataTableSource extends ChangeNotifier` because whenever the data source changes, you need to call `notifyListeners()` to refresh the table. In our example we've simply hard-coded a list with a few values, but in reality you should pass the data via constructor injection.

- `getRow`: returns the actual data represented by a `DataRow` type. You should use the `byIndex` constructor in order to avoid worrying about using keys to uniquely identify rows. Remember that:

- the number of cells must match the column count;
- call `notifyListeners()`; whenever you add/remove a row or any content is changed.

Note that `DataTableSource` `extends ChangeNotifier` so you're able to triggers rebuilds calling `notifyListeners()` inside your model class.

- `isRowCountApproximate`: a value of `false` means that the row count is fixed (the table won't change). When set to `true`, then you're estimating the row count which will be finalized in a second moment (and updated via `notifyListeners()`). This is useful when the table rows aren't immediately ready: some might appear later because of waiting for a `Future<T>` or an external event to finish.
- `selectedRowCount`: number of currently selected rows, which we've set to 0 by default.
- `rowCount`: if `isRowCountApproximate` returns `false`, then this method should just return the exact length of the data source (because the data source won't change). Instead, if it returned `true` you should change the body of `getRow()`:

```
DataRow? getRow(int index) {
    // Trying to access an item not yet in the table. Return
    // 'null' to show an animated circular loading indicator.
    if (index >= values.length)
        return null;

    final data = values[index];
    return DataRow.byIndex(...);
}
```

Basically, if the table tries to access an element not yet in the data source, you need to return `null`. In this way, a loading indicator automatically appears; it will disappear once the element has been added and updated via `notifyListeners()`.

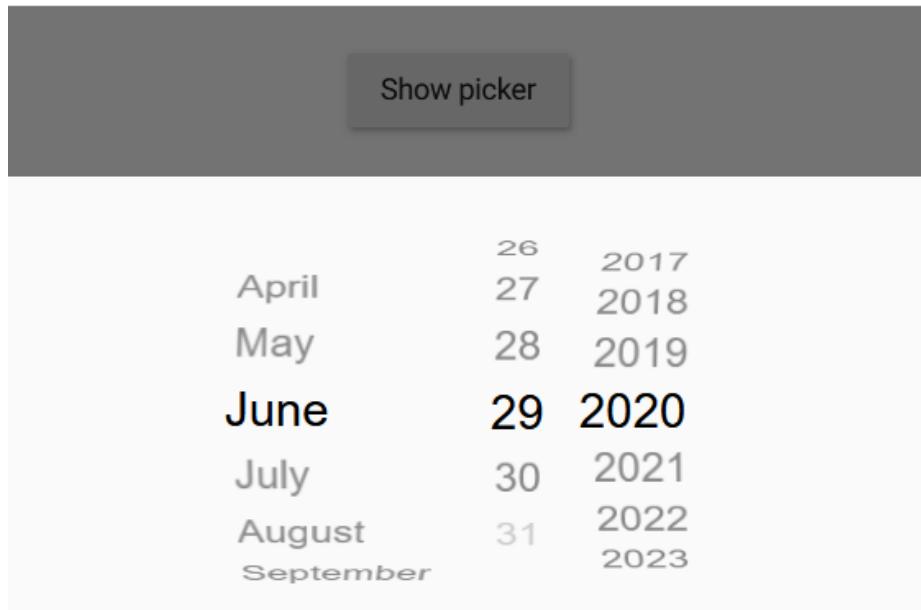
You should set `isRowCountApproximate => true` when you plan to add/remove data on the table at runtime or when a `Future<T>/Stream<T>` completed. The table automatically shows a loading indicator as a placeholder; when data are ready and the table won't be modified anymore, change this to returning `false`.

21.2 Cupertino

- Reference: <https://developer.apple.com/design/human-interface-guidelines/ios/>

21.2.1 CupertinoDatePicker

It's a traditional iOS-styled picker where data are laid out in columns and they're scrollable in the vertical direction. The widget is localized because it automatically translates according to the device's language. The order of the columns may vary according to the current locale.



You can get this result using a `showModalBottomSheet<T>` to wrap the `CupertinoDatePicker` widget. Since the popup overlays a large part of the screen, we need its height to be calculated according to the physical device screen dimensions. For this reason, we're using `MediaQuery` rather than `LayoutBuilder`.

```
void _showPicker(BuildContext context){  
  showModalBottomSheet(  
    context: context,  
    builder: (BuildContext context){
```

```
final size = MediaQuery.of(context)?.size.height;
final panelHeight = size == null ? 150 : size / 2.8;

return SizedBox(
    height: panelHeight,
    child: CupertinoDatePicker(
        initialDateTime: DateTime.now(),
        onDateTimeChanged: (DateTime newDate) {},
        minimumYear: 2010,
        maximumYear: 2050,
        mode: CupertinoDatePickerMode.date,
    ),
);
}
);
```

The widget is quite easy to setup; the `_showPicker` method is invoked on a button tap and a dialog slides up (with an animation) from the bottom. Very easily, the picker can only show the date, the time or both just by changing the mode:

```
mode: CupertinoDatePickerMode.date,
mode: CupertinoDatePickerMode.dateAndTime,
mode: CupertinoDatePickerMode.time,
```

The `onDateTimeChanged` callback can be used to get the currently selected date from the picker. It's very convenient and efficient because there's no need for controllers or keys.

21.2.2 CupertinoActionSheet

If you have the need to show a list of selectable options in an iOS style, the `CupertinoActionSheet` could really be the right choice. It implements the "action sheet" UI design which is basically a list of actions sliding up from the bottom of the screen.

```
CupertinoActionSheet(
    actions: <Widget>[
        CupertinoActionSheetAction(
            child: const Text("Do something"),
            onPressed: () {},
            isDefaultAction: true,
```

```

),
CupertinoActionSheetAction(
    child: const Text("Delete"),
    onPressed: () {},
    isDestructiveAction: true,
),
],
),

```

With `isDefaultAction: true` and `isDestructiveAction: true` you set the text to bold or red respectively; the latter is used when you're implementing a button that deletes something. There's also the possibility to create a separated "Cancel" button at the bottom of the list:

```

CupertinoActionSheet(
    cancelButton: CupertinoActionSheetAction(
        child: const Text("Cancel"),
        onPressed: () {...},
    ),
    actions: <Widget>[...]
),

```

21.2.3 CupertinoSegmentedControl

This widget is typically used when the user is asked to select between a number of mutually exclusive options. The idea is to associate a type to a widget, which is generally a `Text()`, so that when it's pressed, the data it points to are returned.

```

CupertinoSegmentedControl(
    children: const <int, Widget>{
        0: ChildIcon("Option 1"),
        1: ChildIcon("Option 2"),
        2: ChildIcon("Option 3"),
    },
    onValueChanged: (int value) {
        _currentValue = value;
    },
    groupValue: _currentValue,
)

```

The `children` parameter takes a map that associates an `int` to a `ChildIcon` widget. When the

user chooses one of the three options, the `onValuechanged` callback is called with the currently selected value.



For example if you tapped on "Option 3", in `onValuechanged` the `int value` variable would contain 2 because it's the value associated with the selected widget. We have created a reusable widget in order to keep consistency in the control and avoid code duplication, as usual:

```
class ChildIcon extends StatelessWidget {
    final String text;
    const ChildIcon(this.text);

    @override
    Widget build(BuildContext context) {
        return Padding(
            padding: EdgeInsets.fromLTRB(10, 5, 10, 5),
            child: Text(text),
        );
    }
}
```

21.3 Community widgets

At <https://pub.dev> there are lots of packages and the number is at a constant growth thanks to your help. We're listing now some high quality packages you might find useful during your development journey. The descriptions have been taken from their package's home page.

- **sqflite**. Flutter plugin for SQLite, a self-contained, high-reliable, embedded database engine.
- **mobx**. MobX is a library for reactively managing the state of your applications. Use the power of observables, actions, and reactions to supercharge your Dart and Flutter apps.
- **RxDart**. RxDart is an implementation of the popular reactiveX api for asynchronous programming, leveraging the native Dart Streams api.

- **carousel_slider.** A carousel slider widget, support infinite scroll and custom child widget.
- **flutter_slideable.** A Flutter implementation of a slidable list item with directional slide actions that can be dismissed.
- **flutter_staggered_grid_view.** A Flutter staggered grid view which supports multiple columns with rows of varying sizes.
- **flutter_secure_storage.** Flutter Secure Storage provides API to store data in secure storage. Keychain is used in iOS, KeyStore based solution is used in Android.
- **overflow_view.** A widget displaying children in a line with an overflow indicator at the end if there is not enough space.
- **hive.** Lightweight and blazing fast key-value database written in pure Dart. Strongly encrypted using AES-256.

In chapter 23 we will talk in detail about packages, likes and the scoring system.

21.3.1 Flutter Hooks

A `StatefulWidget` is essential in some cases: when using animations for example, its `dispose()` method must be used to perform cleanup operations on the controllers. Following this case, here's the typical skeleton of a `StatefulWidget` with two animations:

```
class Example extends StatefulWidget {
    const Example();

    @override
    _ExampleState createState() => _ExampleState();
}

class _ExampleState extends State<Example>
    with SingleTickerProviderStateMixin{
    late final AnimationController _controller1;
    late final AnimationController _controller2;

    @override
    void initState() {
        super.initState();
        _controller1 = AnimationController(
```

```
        vsync: this,
        duration: const Duration(seconds: 2),
    );
    _controller2 = AnimationController(
        vsync: this,
        duration: const Duration(seconds: 1),
    );
}

@Override
void dispose() {
    _controller1.dispose();
    _controller2.dispose();
    super.dispose();
}

@Override
Widget build(BuildContext context) {
    return MyWidget();
}
}
```

Indeed there's a lot of code to write because controllers generally require to be initialized and disposed. It would have been the same if we had used a `TextEditingController` for example, because it also has to be initialized and disposed. We can notice a few problems with the "controllers pattern":

1. a lot of boilerplate code to write;
2. we'd like to create reusable instances of our controllers but they're tied to `initState()` and `dispose()`;
3. a `mixin` could be a good idea but it can be used only once (if your class required two controllers, you'd have to use the `mixin` twice but that's not possible).

In other words, the code inside a `StatefulWidget` is difficult to reuse. Performance here don't matter: the problem is the potentially big amount of code duplication "caused" by the structure of the class itself. There's no built-in way to solve this maintenance problem but thankfully the Flutter community comes to the rescue!



Thanks to the *flutter_hooks* package (by Rémi Rousselet), you can create "reusable stateful widgets" to get rid of all those code duplication issues we exposed above. Using hooks, the same animation example at the beginning of the section can be rewritten in the following way:

```
// - same thing as before but less boilerplate and more reusability
// - initState/dispose are not needed because 'useAnimationController' does
//   the job for us
class Example extends HookWidget {
    const Example();

    @override
    Widget build(BuildContext context) {
        final controller1 = useAnimationController(
            duration: const Duration(seconds: 2)
        );
        final controller2 = useAnimationController(
            duration: const Duration(seconds: 1)
        );

        return Container();
    }
}
```

This is a huge improvement. Hooks do everything automatically: internally they initialize and dispose the controllers so that you don't have to write too much code. Basically, a `HookWidget` is a "reusable `StatefulWidget`" which automatically handles your widget's resources. Before going on, there's something more to say:

- Even if declared inside `build()`, hooks "survive" to rebuilds and they can be reused an infinite number of times.
- Hooks cannot be used in the `build()` method of `StatefulWidget` or `StatelessWidget`.

- `useAnimationController()` is a "hook", like many others we will see in a few lines.
- There's nothing weird under the hood because internally `HookWidget` inherits from a stateless widget so it's still a "normal" Flutter widget.

Of course, hooks are not required but they are very convenient and you might get used to them very quickly. Here's a side-by-side comparison between the `StatefulWidget` and `HookWidget` implementation of a simple animation example.

Animation with `StatefulWidget`

```
static const R = 2 * 3.1415;

late AnimationController ctl;

@Override
void initState() {
  super.initState();
  ctl = AnimationController(
    vsync: this,
    duration: const Duration(
      seconds: 2
    ),
  )..repeat();
}

@Override
void dispose() {
  ctl.dispose();
  super.dispose();
}

Widget build(BuildContext ctx) {
```

Animation with `HookWidget`

```
static const R = 2 * 3.1415;

@Override
Widget build(BuildContext context) {
  final ctl = useAnimationController(
    duration: const Duration(
      seconds: 2
    )
  )..repeat();

  return AnimatedBuilder(
    animation: ctl,
    builder: (context, child) {
      return Transform.rotate(
        angle: ctl.value * R,
        child: child
      );
    },
    child: const FlutterLogo(
      size: 50,
    ),
  );
}
```

```
        return AnimatedBuilder(
            animation: ctl,
            builder: (context, child) {
                return Transform.rotate(
                    angle: ctl.value * R,
                    child: child
                );
            },
            child: const FlutterLogo(
                size: 50,
            ),
        );
    }
}
```

The type and the usage of the `ctl` variable is **identical** because, in both cases, it's always a `AnimationController`. Initialization and disposal happen in the hook as well but they're "hidden". So far we've only shown a single type hook which works with animations but there are many more you can use, such as:

1. `useTextEditingController()`. Creates a `TextEditingController` and automatically takes care of disposing it when not needed anymore. Optionally, it can be initialized with some text.

```
@override
Widget build(BuildContext context) {
    final emailController = useTextEditingController("default@me.com");

    return TextFormField(
        controller: emailController,
    );
}
```

2. `useTabController()`. Creates a `TabController` and automatically takes care of disposing it when not needed anymore. As you've already seen in 21.1.4, it's used to manually move among tabs on a `TabBarView`.
3. `useStream<T>()`. Subscribes to a `Stream<T>` and uses an `AsyncSnapshot<T>` to return the current state.

4. `useFuture<T>()`. Subscribes to a `Future<T>` and uses an `AsyncSnapshot<T>` to return the current state.
5. `useMemoized<T>()`. Caches an instance of an object for a later use. As you already know, hooks "survive" to rebuilds so this method persists the data.

```
@override
Widget build(BuildContext context) {
    final cachedUser = useMemoized<User>(() => const User(
        name: "Alberto",
        surname: "Miola".
));
}

return UserWidget(
    data: cachedUser
);
}
```

The `User` instance is immediately stored; when the `HookWidget` rebuilds, `useMemoized()` returns the previous instance.

`useContext()`. Returns the `BuildContext` of the current `HookWidget`.

Be sure to visit the official documentation ³ to see all the types of hooks you can use. If you can't find what you're looking for, there's even the possibility to create your own hooks just by overriding `Hook`. Let's see how we can create a hook to generate random numbers:

1. Create a new file called `time_hook.dart`.
2. Create a top-level function to "hide" the actual class implementing the hook and be sure to start with the word `use`. These are just conventions you should follow to keep consistency with the "hooks environment".

```
int useRandomGenerator(int value) {
    return use<int>(_RandomGenerator(maxValue: value));
}
```

3. In the same file, create the actual hook extending `Hook<T>` which has the same structure as a `StatefulWidget`. When inside the state, you can reference members defined in the class using the `hook` property (where in a `StatefulWidget` you'd have used `widget`).

³https://pub.dev/documentation/flutter_hooks/latest/

```
class _RandomGenerator extends Hook<int> {
    final int maxValue;

    const _RandomGenerator({
        required this.maxValue
    });

    @override
    _RandomGeneratorState createState() => _RandomGeneratorState();
}

class _RandomGeneratorState extends HookState<int, _RandomGenerator> {
    late final Random random;

    @override
    void initHook() {
        super.initHook();
        random = Random();
    }

    @override
    int build(BuildContext context) => random.nextInt(hook.maxValue);

    @override
    void dispose() {
        debugPrint("Disposed the 'RandomGenerator' hook");
        super.dispose();
    }
}
```

This is also how `useAnimationController` and others work under the hood. Define your initialization and finalization logic inside `initHook` and `dispose`. The `build()` method returns the data type you need which is an `int` in this case.

21.3.2 State notifier

You've seen many examples where we used `ChangeNotifier` to create "listenable" model classes that can be watched by a `ChangeNotifierProvider<T>` (from the `provider` package). It's the

most common way of working and it's very efficient.

```
class DateCache with ChangeNotifier {  
  DateTime _date = DateTime.now();  
  
  Date get currentDate => _date;  
  
  void refresh() {  
    _date = DateTime.now();  
    notifyListeners();  
  }  
}
```

This small model class simply exposes a date and notifies its listeners when it's updated. All good here but thanks to the *state_notifier* package we can do the same thing with less code (and probably readability would benefit as well).

```
class DateCache extends StateNotifier<DateTime> {  
  DateCache(): super(DateTime.now());  
  
  void refresh() {  
    state = DateTime.now();  
  }  
}
```

`StateNotifierProvider` is the equivalent of `ChangeNotifierProvider` but in the "state notifier" world. They do the same thing but of course the former works with `StateNotifier<T>` and the latter with `ChangeNotifier`.

```
// Requires the 'flutter_state_notifier' package to be imported  
StateNotifierProvider<DateCache, DateTime>(  
  create: (_) => DateCache(),  
  child: const Something(),  
)  
  
// Inside the 'Something' widget  
override  
Widget build(BuildContext context) {  
  return Column(  
    children: [
```

Chapter 21. Widgets showcase

```
        Text("$$\{context.watch<DateTime>()\}"),
        RaisedButton(
            child: const Text("Refresh"),
            onPressed: () => context.read<DateCache>().refresh(),
        )
    ],
);
}
```

Very intuitively, `watch<T>()` is used to rebuild the widget whenever the date changes and `read<T>` just returns a reference without rebuilding anything. Note that `watch<T>()` needs the type of the **state** while `read<T>()` needs the type of the **notifier**.

22 | Using Firebase with Flutter

22.1 Installation

In this chapter we're working with **FlutterFire** plugins, a series of packages that connect your Flutter apps to Firebase. Before any of the Firebase services can be used, you always need to setup your platform and thus, for new projects, the following guide is required. We're only covering the most commonly used package: visit the official documentation ¹ to get a complete overview.



<https://firebase.flutter.dev>

As example, we're going to configure the app we will build in the next section. Regardless, these steps will always be the same for any new project. Go to the Firebase console ² and log in with your Google account. Click on "Add project", give it a name (once decided, it cannot be changed) and leave all the other settings as they are.

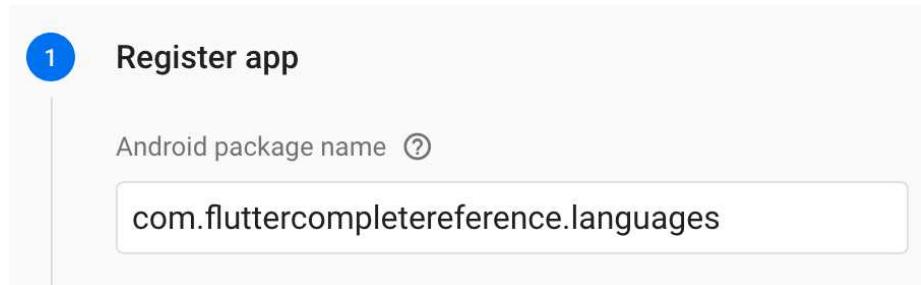
¹<https://firebase.flutter.dev>

²<https://console.firebaseio.google.com/u/0/>



After a few seconds you'll be redirected to the new project page from which you can link Firebase's services (database, storage, analytics...) to your apps. Let's start by clicking on *Add Firebase to your Android app* and follow the steps:

1. Type the package name you want and be sure to also use this same name when the Flutter project will be created. Having an identical name is **fundamental** in order to link an app to your Firebase cloud project.

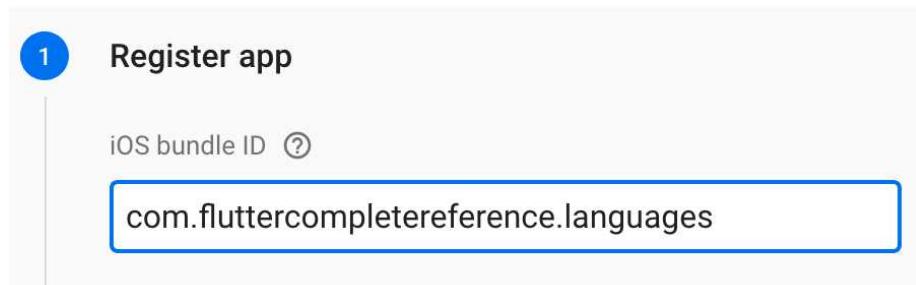


2. Download the `google-services.json` file which we're going to use soon. It's a configuration file needed in our projects in order to recognize which Firebase project your app has to point at.

The Android setup is now ready so we can add more platforms clicking on "Add another app". In case you had the need to use Firebase for iOS, just hit *Add Firebase to your iOS app* and follow the same procedure as before:

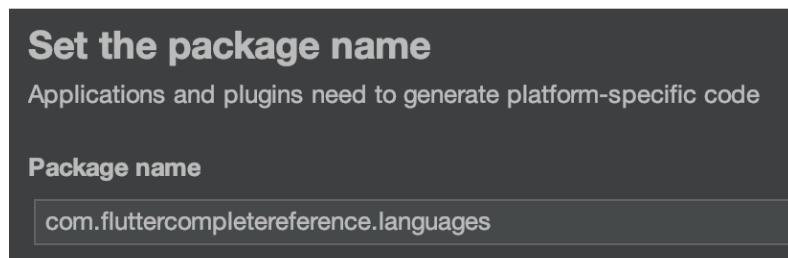
1. Type the bundle ID you want and, again, be sure it matches the bundle ID name of your iOS app. In practice, it must be identical to the string you put earlier for Android since

the project is the same.



2. Download the configuration file.

Since we're creating a cross-platform app with Flutter, both Android and iOS should really have the same value for *package name* and *bundle ID* respectively. Firebase is now completely setup so it's time to create a new Flutter project with the correct package name (screenshot from Android Studio):



It's the same string we've put earlier when registering both the Android and iOS projects. Lastly, there's the need to get the configuration files in place, the ones downloaded at the second step of the platform setup. For Android:

1. Move `google-services.json` in your Flutter project at `android/app`, which is the same directory of the `build.gradle` file
2. Open the `android/build.gradle` file and inside `dependencies` add a reference to google services (be sure to use the latest version):

```
buildscript {
```

```
repositories {  
    google()  
    jcenter()  
}  
  
dependencies {  
    ...  
    classpath 'com.google.gms:google-services:4.3.3'  
}  
}
```

3. Open the `android/app/build.gradle` file and at the bottom, below the `dependencies` group, add this line:

```
apply plugin: 'com.google.gms.google-services'
```

For iOS instead you need to open Xcode, select the `.xcworkspace` file with your project's name, right click it and choose "Add files to 'Runner'". Select the `GoogleServices-Info.plist` file and import it. Don't manually move it via filesystem because it won't work.

```
dependencies:  
    flutter:  
        sdk: flutter  
    firebase_core: ^0.5.0
```

Be sure to add the `firebase_core` plugin in your dependencies list because it's required by any FlutterFire package. It links your Flutter app to Firebase. It has to be initialized before any Firebase interaction happens so ideally `Firebase.initializeApp()` should be called in the home page.

```
class HomePage extends StatefulWidget {  
    const HomePage();  
  
    @override  
    _HomePageState createState() => _HomePageState();  
}  
  
class _HomePageState extends State<HomePage> {  
    late final Future<FirebaseApp> _initialization;  
  
    @override
```

```
void initState() {
    super.initState();
    _initialization = Firebase.initializeApp();
}

@Override
Widget build(BuildContext context) {
    return FutureBuilder<FirebaseApp>(
        future: _initialization,
        builder: (context, snapshot) {
            if (snapshot.hasError) {
                return const ErrorWidget();
            }

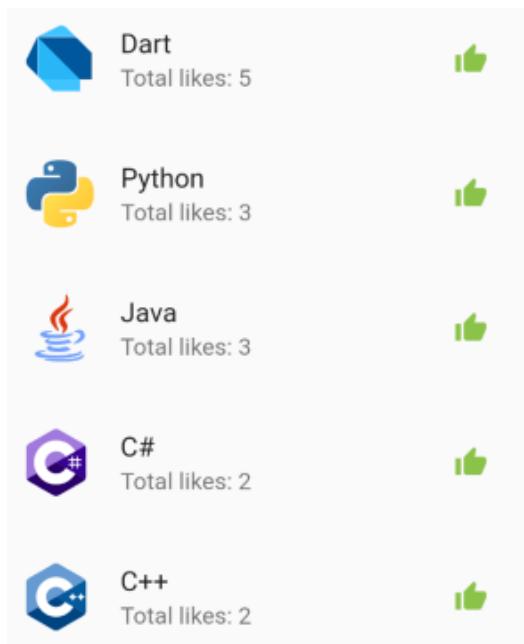
            if (snapshot.connectionState == ConnectionState.done) {
                return const HomePageBody();
            }

            return const Center(
                child: SomeLoadingWidget(),
            );
        },
    );
}
```

For more info on how to gracefully perform startup initialization, see appendix B.3. If you use a FlutterFire plugin without having called `initializeApp()` first, your app won't work.

22.2 Using Firestore as backend

Firestore is Google's backend solution for your desktop, web or mobile apps which offers scalability, reliability and real-time data sync. In this section we're going to create an app to keep track of votes submitted by users about their favorite programming language.



Very simply, pressing the green thumb for a certain language increases the "like" counter by 1. We're going to see the implementation of this app, backed by a Firestore NoSQL database, in the very next section. We now want to focus a bit on the advantages brought by this cloud-based service:

1. When you tap on the "like" button, the database is updated and changes are reflected immediately on any device. Data are updated in real-time so anyone looking at the app can see the counters increasing... in real-time! No need to press on a "refresh" button or perform any action to update the UI.
2. The data, such as "likes" and programming language names, are stored in Firestore (a NoSQL database) and the integration with the app happens thanks to the *cloud_firestore* plugin.
3. Firestore is a very robust, distributed and reliable service maintained by Google, along with its Flutter implementation. You don't have to worry about compatibility or the complexity of the API because there are a series of Dart classes ready for you to use.
4. You don't have to create/test a web service, manage a database, setup periodic backups and all those important backend tasks. Firestore takes care of everything so that you can fully concentrate on app development.

Firebase is very powerful and handy but of course it's not the only way to create a backend for a Flutter application. For example, we could have used a Linux server with a traditional MySQL database and a webservice exposing a REST API. In this case, we'd have to take care of both the backend and the frontend.



Firebase instead exposes a series of services for our apps and thanks to FlutterFire, there's even more integration and ease of use. You can only focus on the frontend. Before adopting it, you should explore its capabilities and doing some demo projects in order to be able to evaluate its pros and cons.

22.2.1 Building the backend

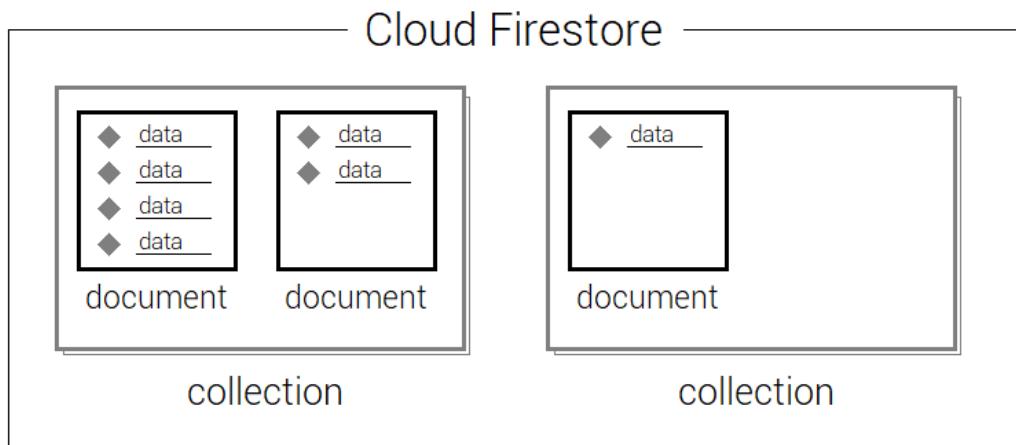
Go to the Firestore console, choose the newly created project, click on "Database" on the left and choose "Cloud Firestore" as project database. For our demo app, going for **test mode** is fine but if you consider keeping your project alive for a long time, you should later switch to **locked mode**. It's more secure.



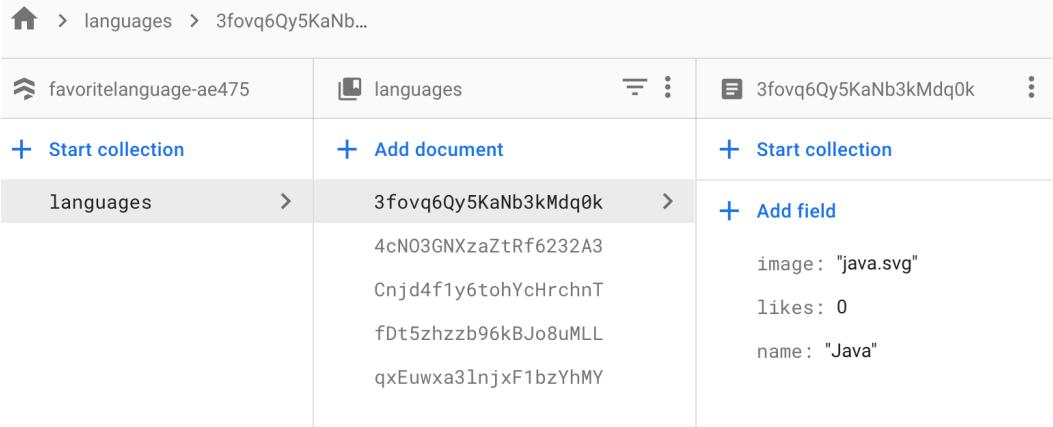
Start in test mode

Get set up quickly by allowing all reads and writes to your database

Wait a minute while the Firestore instance prepares and then you're ready to start. We're going to work with a NoSQL database, which has no relations and foreign keys as it's structured like a JSON file with key-value pairs. It's made up of three important parts:

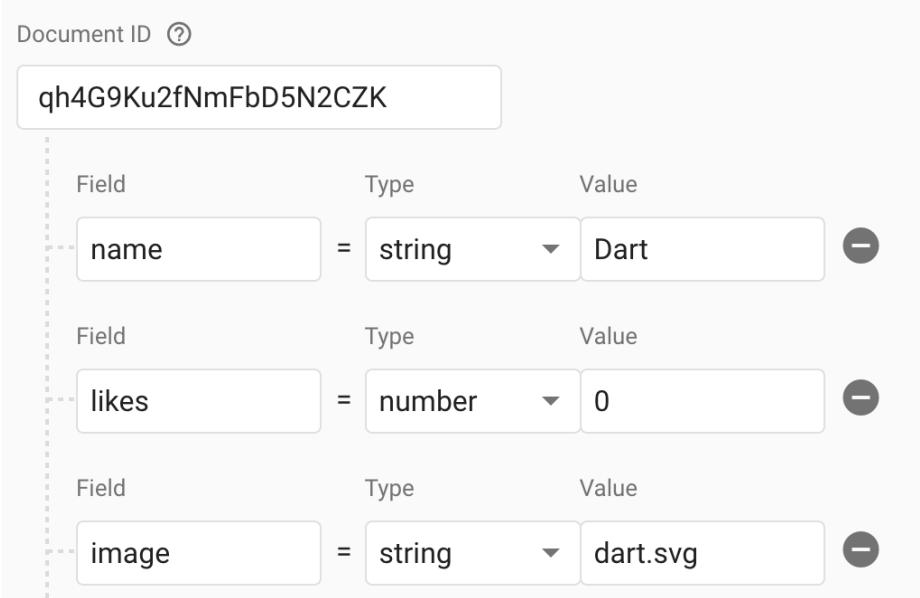


A series of **data** is grouped together in a **document**, which must have an unique id. One or more documents are grouped inside a **collection**. To be more specific, in our example we've decided to use a collection called "languages" which is going to contain a series of documents about programming languages:



The screenshot shows the Firebase Firestore interface. At the top, there's a navigation bar with a home icon, followed by 'languages' and a document ID '3fovq6Qy5KaNb3kMdq0k...'. Below this is a table structure. The first row has three columns: 'favoritelanguage-ae475' (with a 'Start collection' button), 'languages' (with an 'Add document' button), and '3fovq6Qy5KaNb3kMdq0k' (with a 'Start collection' button and a three-dot menu). The second row shows a collection named 'languages' with a single document '3fovq6Qy5KaNb3kMdq0k'. This document has four fields: 'image' with value 'java.svg', 'likes' with value 0, and 'name' with value 'Java'. There's also a placeholder field '4cN03GNXzaZtRf6232A3'.

Data can be of various types such as arrays, integers, strings or time. Each document has an id, represented by a weird long string (it's been automatically generated by Firestore but you could manually assign one). Each document holds data about a certain programming language; you should use the same names across the entire collection in order to make easy queries.



The screenshot shows the Firebase Firestore document editor for the document '3fovq6Qy5KaNb3kMdq0k'. The 'Document ID' is 'qh4G9Ku2fNmFbD5N2CZK'. The table below lists the fields:

Field	Type	Value
name	string	Dart
likes	number	0
image	string	dart.svg

As you can see, "Document ID" is the unique identifier we've decided to automatically assign just to keep the example easy. You should really look at the best practices³ section of the documentation which is very useful, especially for beginners. We're good with these settings so we can move to the frontend section.

 Resources > Chapter 22 > Cloud Firestore

22.2.2 Building the frontend

Add the official `cloud_firestore`⁴ package as dependency in your `pubspec.yaml` file and be sure you've already called `Firebase.initializeApp()`. In order to receive real-time updates from the database, we're going to work with streams.

```
class FavoriteList extends StatelessWidget {
    static Stream<QuerySnapshot> getStream() =>
        FirebaseFirestore
            .instance
            .collection("languages")
            .orderBy("likes", descending: true) // 1
            .snapshots(); // 2

    const FavoriteList();

    // build...
}
```

Thanks to `collection("languages")` we can reference the "languages" collection on Firestore created in the previous step. The `FirebaseFirestore.instance` already knows how to point to the correct database in the cloud because settings are taken from the `json` or `.plist` configuration files.

1. You can build a query and filter the data received from the database. In this example we're ordering rows but you can also use other functions such as `limit()` or `where()`.
2. The `snapshot()` method returns a stream listening to the given collection: new values are emitted whenever something changes. Thanks to this approach, we're able to show

³<https://firebase.google.com/docs/firestore/best-practices>

⁴https://pub.dev/packages/cloud_firestore

live changes in our app without having to manually refresh the page. Data are always automatically kept in sync with the database.

The stream is handled in the usual way: the `hasData` property tells whether data are ready to be displayed or not. As always, remember to use `const` constructors as much as possible because builder functions are called quite often.

```
override
Widget build(BuildContext context) {
    return StreamBuilder<QuerySnapshot>(
        stream: getStream(),
        builder: (context, languages) {
            if (languages.hasError) {
                return const ErrorWidget();
            }

            if (languages.hasData) {
                final data = languages.data;

                if (data != null) {
                    return ListView.builder(
                        itemExtent: 80.0,
                        itemCount: data.docs.length,
                        itemBuilder: (context, index) =>
                            LanguageItem(data.docs[index]),
                    );
                } else {
                    return const ErrorWidget();
                }
            }

            return const Center(
                child: CircularProgressIndicator()
            );
        },
    );
}
```

To favor code readability, making a reusable widget to represent each item of the list is a good

idea so we created `class LanguageItem`. The `docs` list contains the list of documents returned by the query executed on the "languages" collection.

```
class LanguageItem extends StatelessWidget {
    final String name;
    final int likes;
    final String asset;
    final QueryDocumentSnapshot _snapshot;

    LanguageItem(this._snapshot) :
        asset = _snapshot.get("image") as String,
        likes = _snapshot.get("likes") as int,
        name = _snapshot.get("name") as String;

    // build and updateVote...
}
```

The `QueryDocumentSnapshot` object holds data about a document of the collection, which is in our case the data about a programming language. We have to pay attention to write the correct field names otherwise an exception will be thrown. In case you wanted to add another field, go to the online console and click on "Add field" for each object in the collection.

<code>LanguageItem(this._snapshot) :</code> <code> name = _snapshot["name"],</code> <code> likes = _snapshot["likes"],</code> <code> asset = _snapshot["image"];</code>		<code>image: "java.svg"</code> <code>likes: 0</code> <code>name: "Java"</code>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------

Each programming language is represented by a `ListTile` widget which is very handy when it comes to showing data in a `ListView`. The `image` field we set in the Firestore collection indicates the name of the `.svg` asset the app has to load.

```
@override
Widget build(BuildContext context) {
    return ListTile(
        leading: SvgPicture.asset("images/$asset",
            height: 40,
    ),
```

```
        title: Text(name),
        subtitle: Text("Total likes: $likes"),
        trailing: IconButton(
            icon: const Icon(Icons.thumb_up),
            onPressed: _updateVote,
            ...
        ),
    );
}
```

When tapping on the icon we want to update the item in the database so that the "like counter" increases by 1. We need to reference again the collection via `FirebaseFirestore.instance`, increase the value on the correct document and send the update to Firestore. It will then take care of notifying listeners about changes.

```
void _updateVote() {
    FirebaseFirestore.instance.runTransaction((transaction) async {
        // 1.
        final secureSnapshot = await transaction.get(
            _snapshot.reference
        );

        // Getting the current likes count
        final int currentLikes = secureSnapshot.get("likes") as int;

        // 2.
        transaction.update(secureSnapshot.reference, {
            "likes": currentLikes + 1
        });
    });
}
```

This approach is secure and free from data races. We're guaranteed that there won't be concurrency problems thanks to the `runTransaction()` method, which is basically the safe way to update data on Firestore. Inside the scope of a transaction you can update data and be sure that the action will execute "atomically".

1. The `secureSnapshot` object contains a fresh new copy of the language item from which we can safely read and, most importantly, write values.

2. With `update()` you can change the content of the given field, assuming you're passing the correct name. In our case the like counter has to be incremented by 1 so we assign the new value to the field "`likes`".

Firebase has built-in support for offline mode. When reading and writing data, an internal **local** database is kept in sync with the cloud. In case of no internet connection available, Firestore continues to work and when connectivity comes back data are automatically synchronized.

```
void main() {
    FirebaseFirestore.instance.settings = Settings(
        persistenceEnabled: false,
        cacheSizeBytes: Settings.CACHE_SIZE_UNLIMITED,
    );

    runApp(MyApp());
}
```

This is very powerful but you might want to rely only on the internet connection. For this reason, setting `persistenceEnabled: false` disables the "offline mode". The default cache size value is 40 MB and the minimum value you can use is 1 MB. The local database can be removed but the following call has to be made before any Firestore interaction.

```
await FirebaseFirestore.instance.clearPersistence();
```

Calling `clearPersistence()` after even a single Firebase call will clear the local database on the next app startup.

22.2.3 Working with data

In the previous example, we've seen how to read data and receive real-time updates whenever something changes. Other than listening for changes on collections, you can also listen for single documents in a similar way.

```
// Real-time changes on the collection
final Stream<QuerySnapshot> collection = FirebaseFirestore
    .instance.collection('my_collection').snapshots();

// Real-time changes on the document
final Stream<DocumentSnapshot> document = FirebaseFirestore
    .instance.collection('my_collection').doc('doc_id').snapshots();
```

In both cases, you get a `Stream<T>` instance which can be used in a `StreamBuilder<T>` widget. To reference a specific document on the collection, you need to provide its ID. If you aren't interested in real-time updates, you can execute an one-time read:

```
// Get all the documents in a collection
final Future<QuerySnapshot> documents = FirebaseFirestore
    .instance.collection('my_collection').get();

// Get all the data inside a single document
final Future<DocumentSnapshot> documents = FirebaseFirestore
    .instance.collection('my_collection').doc('doc_id').get();
```

The result is now a `Future<T>` because data are one-time received and not listened. In both cases (real-time and one-time) you can query collections but of course there's no possibility to query documents. Here are some examples based on our "languages" collection:

- **Sorting.**

```
FirebaseFirestore.instance
    .collection("languages")
    .orderBy("likes")
    .snapshots();

FirebaseFirestore.instance
    .collection("languages")
    .orderBy("name", descending: true)
    .snapshots();
```

Ascending is the default direction.

- **Limiting.**

```
FirebaseFirestore.instance
    .collection("languages")
    .limit(10)
    .snapshots();
```

Limits the number of documents retrieved by a query. Use `limitToLast()` to limit but from the bottom.

- **Filtering.**

```
FirebaseFirestore.instance
```

```
.collection("languages")
.where("likes", isLessThan: 5)
.snapshots();

FirebaseFirestore.instance
.collection("languages")
.where("likes", isGreaterThanOrEqualTo: 2)
.orderBy("likes", descending: true)
.snapshots();
```

You can also use `arrayContains` for collections.

To add a new document in a collection, simply use `FirebaseFirestore` to get a reference to the collection and then call the `add(Map<String, dynamic> data)` method which returns a `Future<T>` object.

```
// Generates an unique random ID
FirebaseFirestore.instance
.collection("languages")
.add({
  "image": "java.svg",
  "likes": 0,
  "name": "Java"
});
```

With this approach, a new document is added with an auto-generated ID. If you want to be able to manually specify an ID, use `set(Map<String, dynamic> data)` on a document reference. Pay attention because a duplicate ID **replaces** the other document rather than throwing an exception.

```
// Uses the ID you give
FirebaseFirestore.instance
.collection("languages")
.doc("my_id_012")
.set({
  "image": "java.svg",
  "likes": 0,
  "name": "Java"
});
```

Call `delete()` on a `DocumentReference` to delete the document. You can also selectively delete

fields of a document using a combination of `update()` and `FieldValue.delete()`.

```
// Remove the 'image' value from the document
FirebaseFirestore.instance
    .collection("languages")
    .doc("my_id_012")
    .update({
        "image": FieldValue.delete(),
    });
}
```

Using `update()` alone might introduce data race problems because concurrency in Firestore plays a very important role. For this reason, you'd better update your documents' data inside a transaction.

22.2.4 Transactions and batches

You've already seen an example of a transaction in the "languages" example where the like counter is safely incremented by one. Transactions are generally used to ensure a safe read/update of a field based on its current value or the value of other fields. Let's see an example where we try to increase the age of a person:

- Without transactions, Firestore updates a value in two different steps. It first executes `get("age")` to retrieve the value and then, with another separated operation, it executes the `update()` method.

```
final currValue = docSnapshot.get("age") as int;
await docSnapshot.reference.update({
    "age": currValue + 5
});
```

The problem is that between `get()` and `update()`, the field `"age"` might be changed by another call so our `update()` is actually working on "outdated" data. The final result is undefined because it depends on the order in which calls happen. This problem is also known as "data race".

- With transactions, Firestore updates a value in a single step because all the operations are guaranteed to be executed together (like if they were an unique method call).

```
FirebaseFirestore.instance.runTransaction((transaction) async {
    final secureSnapshot = await transaction.get(docSnapshot.reference);
    final currValue = secureSnapshot.get("age") as int;
```

```
        transaction.update(secureSnapshot.reference, {
            "age": currValue + 5
        });
    };
}
```

In this case, `get()` and `update()` are executed together like if they were a single call. There won't be data race problems.

A **transaction** is generally made up of a series of `get()` followed by a series of `delete()`, `set()` or `update()` calls. When it fails, an exception is thrown and no data are written in the database. A **batch** instead is basically a transaction in which you don't need to call `get()` to read data.

```
Future<void> deleteDocuments() {
    final batch = FirebaseFirestore.instance.batch();
    final langs = await FirebaseFirestore.instance
        .collection("languages")
        .get();

    langs.documents.forEach((document) {
        batch.delete(document.reference);
    });

    return batch.commit();
}
```

This batch deletes all the documents inside the "languages" collection in a single operation. The gist is the same as transactions (the `delete()`s are executed all together like if they were a single call), but no read operations happen in between (only `delete()`, `set()` or `update()`).

22.3 Monetizing your apps with AdMob

Google has two main platforms to give you money in charge of showing ad banners: **AdSense**, for the web world, and **AdMob**, for the mobile world. Giving a detailed explanation on how they work and the techniques to maximize the earnings are out of the scope of this book. However, we can give you a general overview of the system from a practical point of view.

1. The main requirement is an active Google account (and thus a Gmail email) which will be linked to Adsense. Before starting monetizing your websites and/or mobile apps, there's the need to submit a website or a YouTube channel to the Adsense team. They need to **approve** it because not every product is eligible for monetization.



When your account will be approved, you will be able to start placing ads in your products and receive money back according to views of banners, clicks and many other factors. Without an approved account, there's absolutely **no** way to show ads. We suggest you to do the following:

- (a) If you have a YouTube channel, you can submit it to the AdSense team. If it meets some specific criteria you can further investigate in the official website ⁵, your account will be approved.
- (b) If you have a website, you can submit the link to the AdSense team and they'll review it. If it meets their specific approval criteria, your website will be accepted and your account will be validated. Check their resources page ⁶ to see how you can improve a website to raise the chances of being approved.

Once you get an approved account, you got most of the work done. Generally the approval phase is the longest one as it requires some technical time and not every website might be accepted at the first try. You might have various improvements to do before having success.

2. When developing mobile apps you have to deal with the **AdMob** platform, which is integrated in the Flutter environment. You need to have a validated AdSense account in order to use AdMob because the two services are linked together. Even if they have two different names, AdSense is still the "main provider".



From now on, we assume that you have a verified AdSense account and thus you're able to properly show ads and receive money. Activating an AdMob account is immediate because it's directly linked to AdSense so both work together.

⁵https://www.youtube.com/account_monetization?nv=1

⁶<https://www.google.com/adsense/start/resources/>

22.3.1 Ad banners

Let's say you have an app published in the official stores for Android and iOS, or at least one of them. To get started, login with your verified account at AdMob and register your product in order to let Google know it has to start distributing ads for your app.

1. On the home page, look at the right and click on "App": a popup menu appears and click on "Add app" at the bottom. If your app has already been published in the App store or in the Play store choose **Yes** and search it.

Have you published your app on Google Play or the App Store?

YES **NO**

If you've just published your app in the stores, you might not be able to see it at first but it's normal; just wait up to 24 hours and then try again. Google needs some time to index the apps in the stores and refresh its databases.

2. Click on "Create ad unit" and select "Banner". The name you're asked to give is only used internally by AdMob to label a banner so that you can easily find it among the others. Once saved, you'll arrive at this point:

Follow these instructions:

1. Complete the instructions in the [Google Mobile Ads SDK guide](#) using this app ID:



- Follow the [banner implementation guide](#) to integrate the SDK. You'll specify ad type, size, and placement when you integrate the code using this ad unit ID:



3. Review the AdMob policies to ensure your implementation complies.

These two codes are very important and they should be kept in a secure place. App ID and unit ID are used in both Android and iOS to identify your account and show the ads;

don't expose them to the public!

Create a new Flutter project making sure it has support for AndroidX (it should be enabled by default, don't untick it). Be sure to also setup a new Firestore project to download the .json configuration files as explained in the installation guide. If your app is running on Android devices, open the `AndroidManifest.xml` file and add this line inside the `<application>` tag:

```
<meta-data
    android:name="com.google.android.gms.ads.APPLICATION_ID"
    android:value="ca-app-pub-#####~#####"/>
```

The long code starting with `ca-app-pub` and containing a tilde is the **app ID** we obtained earlier from the website. Similarly, for iOS open the `Info.plist` file and add the following entry to the list:

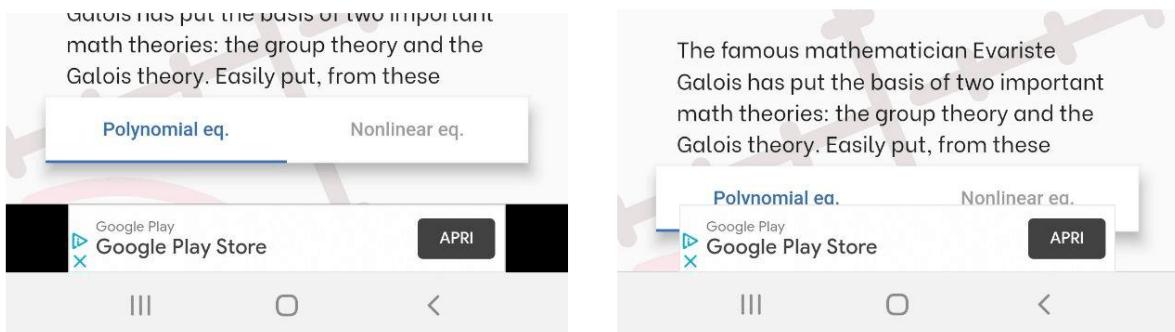
```
<key>GADApplicationIdentifier</key>
<string>ca-app-pub-#####~#####</string>
```

If you forget to add these settings, your app will crash at startup. Be sure to properly place both files, `google-services.json` for Android and `GoogleServices-Info.plist` for iOS, following the installation guide.

 At the time of writing this book, there isn't a Flutter widget to display ads. You can't freely put them where you want, they're just shown "in front" of your app using platform-specific code. The official GitHub ⁷ page of the FlutterFire project states that, in the Future, the `firebase_admob` package will be improved with a dedicated Flutter widget.

At this point the setup is over and it's time to start using `firebase_admob`. An ad banner is **NOT** a widget and thus you can't freely decide to put it wherever you want. It's always placed "in front" of your app: there's no possibility to control this behavior so the best thing to do is placing it at the bottom.

⁷<https://github.com/firebaseextended/flutterfire>



On the right you see how the UI would look if we didn't add any spacing; the banner stays in front of the whole app and it covers some contents. On the left we've solved the problem using the `Padding` widget. Basically it creates a gap from the bottom of the screen equal to the banner's height, which is fixed:

- `AdSize.banner` is 320x50 (where the height is 50);
- `AdSize.largeBanner` is 320x100;
- `AdSize.mediumRectangle` is 300x250;
- `AdSize.fullBanner` is 468x60;
- `AdSize.leaderboard` is 728x90;
- `AdSize.smartBanner` the Mobile Ads SDK adjusts at runtime the width and the height but this is hard to handle.

If you want to be sure that the dimensions of the banners are always under your control, which is probably a good idea, don't use a smart banner. The `AdSize.banner` is the shortest and least intrusive from an user-perspective while `AdSize.large` and `AdSize.full` are intrusive. Here's how you can setup your app:

1. The Firebase AdMob service requires startup initialization in order to show ad banners so we're going to create a `StatefulWidget`. You should make the initialization in the root widget so that it's executed only once (at startup) and not repeatedly. We're going to use a `FutureBuilder<T>` following the classic pattern.

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

```
class DemoApp extends StatelessWidget {
  const DemoApp();

  @override
  _DemoAppState createState() => _DemoAppState();
}

class _DemoAppState extends State<DemoApp> {
  late final Future<bool> initializer;

  Future<bool> loadAds(BannerAd banner) async {
    await FirebaseAdMob.instance.initialize(
      appId: "ca-app-pub-#####/#####~#####"
    );

    await banner.load();
    return banner.show();
  }

  @override
  void initState() {
    super.initState();

    initializer = loadAds(
      BannerAd(
        adUnitId: "ca-app-pub-#####/#####",
        size: AdSize.banner,
      )
    );
  }

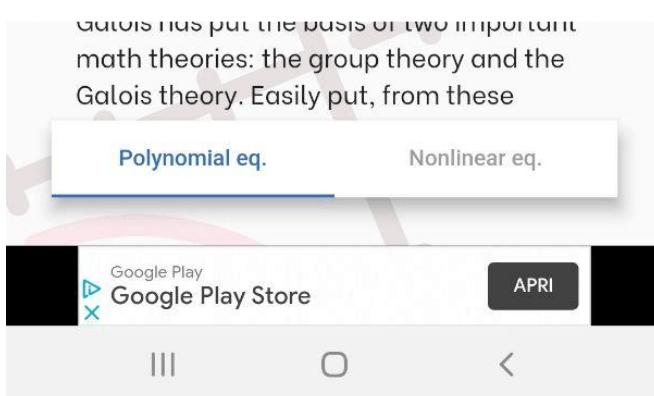
  @override
  Widget build(BuildContext context) => FutureBuilder<bool>(...);
}
```

If you forget to call `initialize()` your app will crash at startup, even if the `.json` configuration file is at the correct place. It's also important calling `load()` before `show()` otherwise the ad might not appear.

2. In the same file, create a package private widget which is going to contain the body of the app with a small gap at the bottom. In this way we can use a constant constructor (`const _AppBody()`) in the `FutureBuilder<void>` rather than working with `Padding()` which cannot be constant.

```
void main() => runApp(DemoApp());  
  
class DemoApp extends StatefulWidget { ... }  
class _DemoAppState extends State<DemoApp> { ... }  
  
class _AppBody extends StatelessWidget {  
    const _AppBody();  
  
    @override  
    Widget build(BuildContext context) {  
        return Padding(  
            padding: EdgeInsets.only(  
                bottom: AdSize.banner.height.toDouble()  
            ),  
            child: MaterialApp(...),  
        );  
    }  
}
```

Thanks to `Padding`, your entire app is visible because it has a bottom padding which gives enough space to the ad to show. The final result will look like this:



3. We can now use a `FutureBuilder<void>` and always return widgets with constant constructors.

```
class _DemoAppState extends State<DemoApp> with AdBanner {
    late final Future<bool> initializer;

    Future<bool> loadAds(BannerAd banner) async {...}

    @override
    void initState() {...}

    @override
    Widget build(BuildContext context) => FutureBuilder<bool>(
        future: initializer,
        builder: (context, snapshot) {
            if (snapshot.hasError) {
                return const ErrorWidget();
            }

            if (snapshot.connectionState == ConnectionState.done) {
                return const _AppBody();
            }

            return const Center(
                child: CircularProgressIndicator(),
            );
        });
}
```

Alternatively, you could remove the progress indicator and immediately show the app. In this case, simply use `const _AppBody()` as default fallback and only handle errors. Ads will still appear at the bottom but later.

```
FutureBuilder<bool>(
    future: initializer,
    builder: (context, snapshot) {
        if (snapshot.hasError) {
            return const ErrorWidget();
```

```
    }

    return const _AppBody();
}

);
```

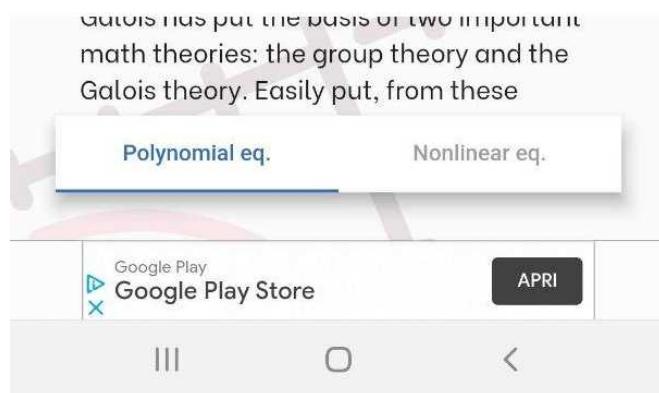
While this solution works, you might not like having a black background behind the banner or so much "empty" space at the bottom. A better solution would involve the usage of a `Column` to contain the app and a `Container` to style the "empty space".

```
class _AppBody extends StatelessWidget {
    const _AppBody();

    @override
    Widget build(BuildContext context) {
        return Column(
            children: <Widget>[
                Expanded(
                    // The app itself
                    child: MaterialApp(...),
                ),

                // What's behind the banner
                LayoutBuilder(
                    builder: (context, constraints) {
                        return Container(
                            width: constraints.maxWidth,
                            height: AdSize.banner.height.toDouble(),
                            decoration: const BoxDecoration(...),
                        );
                    },
                ),
            ],
        );
    }
}
```

Thanks to `Expanded` we're guaranteed that our app will cover as much space as possible. At the bottom there is a `Container` covering the entire width and it has the same height as the banner; it can be styled via `decoration` as you prefer.



Instead of having black spaces behind the banner, now there's a white background which fits better with the style of the UI. It might require some work but an `Image()` or `SvgPicture` could end up with an even better result.

22.3.2 Rewarded video ads

This kind of advertising is different from banners as it's not always visible. Video rewards are used very often in mobile games: the user is asked to watch a video until the end and, as a prize, in-game credits are added. Here's a common execution flow:

- The player taps a button to get coins/gems/credits;
- a full screen ad video appears and it generally lasts a few seconds (it can of course be manually closed);
- if the video has been watched until the end, the player gets a reward.

Video ads don't need to be initialized only once because internally they implement the singleton pattern. Nevertheless, it's still a good practice initializing them only once in the same way as banners, so we're going to use the classic `FutureBuilder<T>` pattern:

```
late final Future<bool> initializer;

Future<bool> loadRewardedVideo() async {
    await RewardedVideoAd.instance.load(
        adUnitId: "ca-app-pub-#####/#####"
    );
}
```

```
    RewardedVideoAd.instance.listener = _listener;  
}  
  
@override  
void initState() {  
    super.initState();  
    initializer = loadRewardedVideo();  
}  
  
void _listener(RewardedVideoAdEvent event, {  
    String? rewardType,  
    required int rewardAmount  
) {  
    if (event == RewardedVideoAdEvent.rewarded) {  
        gamePoints.add(100);  
        gameCredits.increase(rewardAmount);  
    }  
}
```

The `listener` of a `RewardedVideoAd` is called every time the user interacts with a video ad. Using `event` we can check if the video has been watched until the end and decide whether it's the case or not to give a prize.

```
RaisedButton(  
    child: const Text("Show video ad"),  
    onPressed: () =>  
        RewardedVideoAd.instance.show(),  
)
```

To display the video, you can call `show` everywhere in your app and the listener associated to the `RewardedVideoAdEvent` will be triggered. The ad appears in front of your app covering it almost completely but it will always have a close button, generally in the top-right corner.

22.4 Flutter ML Kit

Face detection, image labelling, text parsing and machine learning might sound hard to implement due to the complexity of the topic and the actual skills required. With Flutter's ML Vision plugin realizing all these tasks becomes very easy because almost all of the complex work is pow-

ered by Firebase.

i Firebase's ML Kit is a powerful collection of machine learning modules ready for you to use. They're well-integrated in the Flutter environment thanks to the official plugins made by the Google team so you should definitely give them a try.

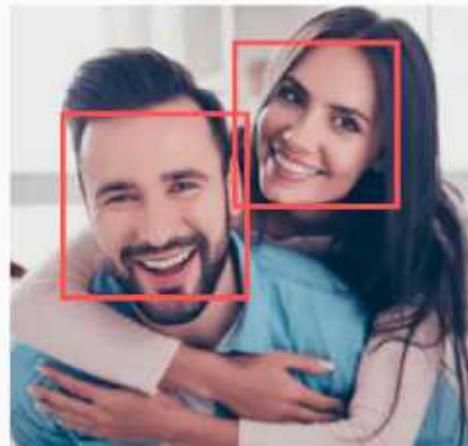
22.4.1 Detecting faces

In this example we're going to create a face detection app that takes an image from our device's gallery and detects if there are one or more humans portrayed on it. It will be also able to detect where faces are with additional information (whether the person is smiling or not, for example). We're going to need two plugins:

- *image_picker*: the official Flutter plugin for picking images from the gallery or any other available location in the device;
- *firebase_ml_vision*: a very powerful plugin that uses the ML (Machine Learning) kit from Firebase⁸

The app is made up of a single page in which the user is asked to select an image from the gallery, but it could have been taken from the camera or anywhere else. The Firebase plugin will detect any human face, head rotation, smiles and some red rectangles will surround the results (if any). The app will look like this:

⁸https://pub.dev/packages/firebase_ml_vision



As always, be sure to install `google-services.json` and/or the `GoogleServices-Info.plist` configuration files as we've explained in the installation guide at the beginning of the section.

1. Let's start with the creation of a class that is just going to contain the results of the face detection process. Since Dart cannot return multiple values from a function, we need to create a "wrapper" to expose more than a single value:

```
import "dart:ui" as ui;

class FaceDetectorData {
    final List<Face> faces;
    final ui.Image image;
    const FaceDetectorData(this.faces, this.image);
}
```

Note that we're using `ui.Image` rather than `Image` because the canvas (more on it later) understands a different kind of image format. The type `Face` is given by the ML Kit package: it contains data about the detected face.

2. Now we need a model class which uses the ML vision plugin to detect faces in a given image. We're hard-coding the size of the image to keep the example simple, but in general you should handle the sizes of the images dynamically according to the screen's dimensions.

```
class FaceDetectorModel {  
  const FaceDetectorModel();  
  
  Future<FaceDetectorData> detectFaces() async {  
    // Choose the image  
    final picker = ImagePicker();  
    final imageFile = await picker.getImage(  
      source: ImageSource.gallery,  
      maxWidth: 260,  
      maxHeight: 250,  
    );  
  
    // Load the image and setup the Vision kit  
    final file = File(imageFile.path);  
    final image = FirebaseVisionImage.fromFile(file);  
    final detector = FirebaseVision.instance.faceDetector();  
  
    // Process data  
    final faces = await detector.processImage(image);  
    final canvasImage = await _decodeImage(file);  
    return FaceDetectorData(faces, canvasImage);  
  }  
  
  Future<ui.Image> _decodeImage(File file) async {...}  
}
```

Calling the `getImage()` method, your device opens the gallery and lets you select one of the available images returning a reference to the chosen file. You could have also taken the image directly from the camera:

```
source: ImageSource.camera,
```

The `processImage()` method looks for faces in the given image and returns a list of `Face` objects. It contains a lot of useful information we will explore later. As you can see, processing images and detecting faces using a machine learning library takes very few lines of code.

```
Future<ui.Image> _decodeImage(File file) async {  
  // 'file' is a reference to the image we picked earlier from the  
  // gallery.
```

```
final rawFile = await file.readAsBytes();

// The 'Codec' class is used by the Flutter engine and it shouldn't
// be directly instantiated. Instead, it can be properly created
// with the 'instantiateImageCodec'
final Codec codec = await instantiateImageCodec(rawFile);

// 'FrameInfo' contains information for a single frame of an
// animation. We need it to extract an 'ui.Image' object which
// is a low level representation of an image
final FrameInfo frameInfo = await codec.getNextFrame();

// Finally... the object we need!
return frameInfo.image;
}
```

This function does some low-level data manipulations. In order to show images on the canvas and draw on it, we need to return an instance of `ui.Image`. This is how you "load" images into a canvas. The code is complicated but in practice you'll never use those methods so don't worry too much about them.

3. Now that Firebase models are ready, we can start working on the UI. We're using provider to expose a boolean variable which will decide whether it's the case to show the image picker or the result widget with the detected faces.

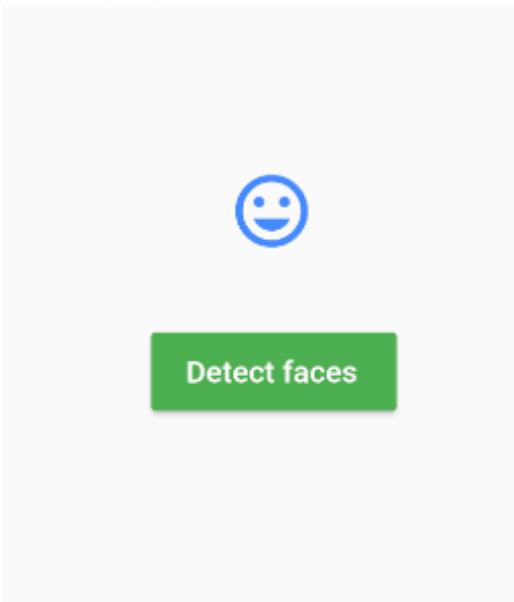
```
class FaceProvider with ChangeNotifier {
  bool _showDetector = false;

  bool get isDetectorVisible => _showDetector;

  void setDetectorVisible(bool isVisible) {
    _showDetector = isVisible;
    notifyListeners();
  }
}
```

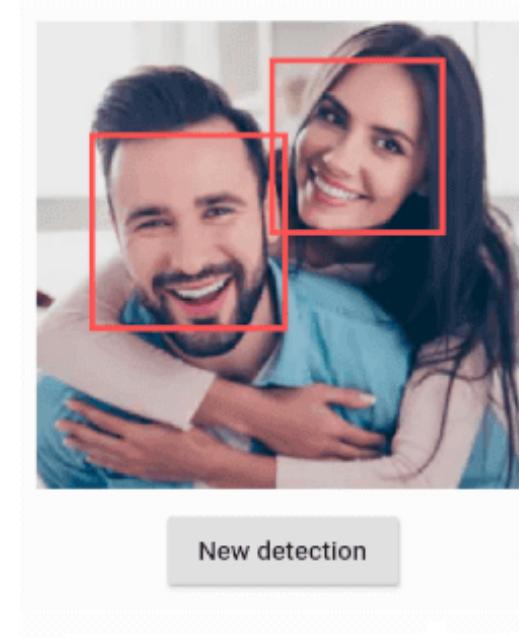
The `isDetectorVisible` getter returns `false` by default because when the app loads, we want the user to click on the "Detect faces" button to select an image from the gallery.

```
class DetectionImgPicker
```



isDetectorVisible: `false`

```
class DetectFacesFromImage
```



isDetectorVisible: `true`

With this technique we're able to easily change the visible widget setting `true` or `false` in the model class, exposed with a provider. In particular, a `Consumer<T>` will read the value and will decide which widget to show.

```
Consumer<FaceProvider>(  
    builder: (context, faceDetector, _) {  
        if (faceDetector.isDetectorVisible) {  
            // Shows the image with the detected faces  
            return const DetectFacesFromImage();  
        }  
  
        // Shows the button 'Detect faces' and opens the gallery  
        return const DetectionImgPicker();  
    },  
)
```

4. The `DetectionImgPicker` widget is nothing special as it's just a `Wrap()` containing a button and an icon. We need a `Consumer<FaceProvider>` to move from the currently visible widget (image picker) to the result page widget (detected faces).

```
Consumer<FaceProvider>(
    builder: (context, faceDetector, _) {
        return RaisedButton(
            child: const Text("Detect faces"),
            onPressed: () => faceDetector.setDetectorVisible(true),
        );
    },
),
```

As we've explained earlier, changing the value to `true`, the currently visible widget becomes `DetectionImgPicker` instead of `DetectFacesFromImage`

5. The `DetectFacesFromImage` widget has to be stateful because there's a `Future<T>` to await and thus we're going to implement the classic `FutureBuilder<T>` pattern.

```
class _DetectFacesFromImageState extends State<DetectFacesFromImage> {
    late final Future<FaceDetectorData> _faces;

    final _model = FaceDetectorModel();

    @override
    void initState() {
        super.initState();

        _faces = _model.detectFaces();
    }

    @override
    Widget build(BuildContext context) {
        return FutureBuilder<FaceDetectorData>(
            future: _faces,
            builder: (context, facesList) {
                if (facesList.hasData) {...}

                return const Center(
                    child: CircularProgressIndicator(),

```

```
        );
    },
);
}
}
```

The `hasData` property becomes `true` when the plugin has successfully decoded the image. We want to show the original image and, in addition, a series of red rectangles surrounding any face recognized; in order to do so, we need a `CustomPaint` widget.

```
Wrap(
  children: <Widget>[
    SizedBox(
      width: 260,
      height: 250,
      child: CustomPaint(
        painter: RectanglePainter(
          facesPositions: data.faces,
          selectedImage: data.images,
        ),
      ),
    ),
  ],
),
Consumer<FaceProvider>(
  builder: (context, detector, _) {
    return RaisedButton(
      child: const Text("New detection"),
      onPressed: () =>
        detector.setDetectorVisible(false),
    );
  },
),
],
),
```

The button at the bottom sets `isDetectorVisible` to `false` to come back to the image selection widget. In the next section we're exploring in detail the `RectanglePainter` class.

22.4.1.1 CustomPainter and the canvas

A `CustomPainter` exposes a canvas in which you can draw lines, shapes or any kind of custom painting. While Flutter is very powerful and customizable, there could be certain cases in which very sophisticated paintings are required for a particular UI so it's the case to use a *painter*.

```
class RectanglePainter extends CustomPainter {
  @override
  void paint(Canvas canvas, Size size) { ... }

  @override
  bool shouldRepaint(RectanglePainter oldDelegate) { ... }
}
```

If you want to implement a painter, which is **NOT** a widget, you have to subclass `CustomPainter` and override its two methods. It can be seen as a low level tool to freely draw on the UI.

- the `paint()` method exposes a canvas in which you can paint anything. All the drawing commands should occur within the bounds given by the `size` object to avoid undesired misalignments of the shapes. Some popular methods are:

```
canvas.drawRect(...); // draws a rectangle
canvas.drawCircle(...); // draws a circle
canvas.drawOval(...); // draws an oval
canvas.drawPath(...); // for Bézier curves
// and much much more...
```

Check out the official documentation of `Canvas`⁹ to see the painting methods you can use.

- the `shouldRepaint()` method controls when the painter should redraw. If your painter has no mutable properties, you can safely return `false` otherwise there will be the need to setup a proper logic.

Given a picture, we want to draw red rectangles around the faces that have been detected. We already have the coordinates because they're inside the `Face` objects so we're ready to paint.

1. Other than drawing the rectangles, we also want the original image to be in the background. A canvas doesn't understand the `Image` type given by Flutter but instead it works with the `Image` class of the '`dart:ui`' package.

```
import 'dart:ui' as ui show Image;
```

⁹<https://api.flutter.dev/flutter/dart-ui/Canvas-class.html>

We need to import "`dart:ui`" and give it an alias just to be sure to not get confused by the names. Inside `class FaceDetectorModel` we declared the `canvasImage` property which exactly returns an instance of `ui.Image`; it's created right after the image is selected from the gallery.

```
class RectanglePainter extends CustomPainter {  
  final List<Face> facesPositions;  
  final ui.Image selectedImage;  
  
  const RectanglePainter({  
    required this.facesPositions,  
    required this.selectedImage  
});  
  
  // paint and shouldRepaint...  
}
```

Here we have `selectedImage`, used as background, and `facesPositions`, which tells us how many rectangles will be painted and their exact positions in the canvas.

2. This class will paint a some rectangles according to the contents of `selectedImage` and the items of the list could potentially change every time the instance is used. For this reason we need to override `shouldRepaint` with a logic:

```
@override  
bool shouldRepaint(RectanglePainter oldDelegate) =>  
  selectedImage != oldDelegate.selectedImage ||  
  facesPositions != oldDelegate.facesPositions;
```

If the class didn't depend on mutable external parameters, we could have overridden this method to simply return `false`. In the example below we have a class with no references to external dependencies and the drawing is always the same, so it's a "static piece" that doesn't need to be repainted.

```
class MyPainter extends CustomPainter {  
  const RectanglePainter();  
  
  @override  
  void paint(Canvas canvas, Size size) {  
    canvas.drawArc(...);  
  }  
}
```

```
    @override
    bool shouldRepaint(MyPainter oldDelegate) => false;
}
```

3. Let's now see how we can actually draw red rectangles around the detected faces, if any. Every time we call a `drawX` function we need to pass a `Paint` object which describes how the figure has to look like.

```
static final Paint _painter = Paint()
  ..style = PaintingStyle.stroke // 1.
  ..strokeWidth = 3.0           // 2.
  ..color = Colors.redAccent;   // 3.
```

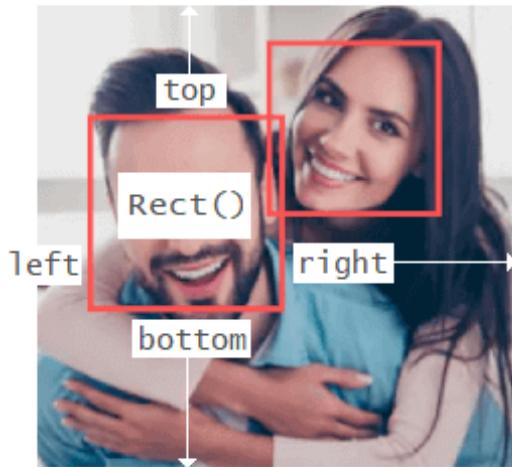
We've decided to create a red (3) shape with no background color, with visible borders (1) and with a certain thickness (2). Instead, if we used `PaintingStyle.fill` the shape would have been entirely filled with the given color.

```
@override
void paint(Canvas canvas, Size size) {
  canvas.drawImage(selectedImage, Offset.zero, Paint());

  for(final face in facesPositions) {
    final coords = face.boundingBox;
    final rect = Rect.fromLTRB(
      coords.left,
      coords.top,
      coords.right,
      coords.bottom
    );

    canvas.drawRect(rect, _painter);
  }
}
```

The `Rect` class represents a 2D rectangle whose coordinates are relative to a certain origin. With `drawRect()` we can exactly tell the engine where the rectangle has to be painted in the canvas. Since each `Face` object contains top, left, right, and bottom offsets we can easily build a `Rect` with no extra effort.



Those red rectangles are drawn by `canvas.drawRect()` and their position/sizes have been calculated by the Firebase plugin earlier.

Since a subclass of `CustomPainter` can't be directly used in the `build()` method, there's the need to wrap it into a widget called `CustomPaint`. The `painter` parameter passes an instance of a `CustomPainter` telling the widget what and how has to be painted.

```
// Inside the 'DetectFacesFromImage' widget
CustomPaint(
  painter: RectanglePainter(
    facesPositions: facesList.data,
    selectedImage: _model.canvasImage
  ),
),
```

22.4.2 Firebase vision kit

If you visit the official pub page of the `firebase_ml_vision` package¹⁰ you'll find many other useful detectors for your apps. They all can be used in the same way: you could organize the code in order to favor reusability and maintainability as much as possible.

- `class BarcodeDetector`. Detector for barcode scanning on an input image.

¹⁰https://pub.dev/packages/firebase_ml_vision

```
final file = FirebaseVisionImage.fromFile(...);
final barCodes = await FirebaseVision.instance
    .barcodeDetector()
    .detectInImage(file);
```

- **class ImageLabeler.** Do you have an image of a family in Paris with the Eiffel tower in the background on a sunny day? This detector will recognize people, things, places and much more with a certain confidence degree.

```
final file = FirebaseVisionImage.fromFile(...);
final labels = await FirebaseVision.instance
    .imageLabeler()
    .processImage(file);
```

Each label has a score indicating the confidence the ML model has in its relevance. In other words, each label has a number indicating how close the algorithm is to the real meaning of the image.

- **class TextRecognizer.** Detector for performing optical character recognition (OCR) on an input image. It can output the whole text on a single string or divide it in multiple blocks.

```
final file = FirebaseVisionImage.fromFile(...);
final textBlocks = await FirebaseVision.instance
    .textRecognizer()
    .processImage(file);
```

If you find these packages useful, you should keep an eye on them because the Flutter team is looking forward into more integration with Firebase ML modules.

22.5 Push notifications with FCM

Push notifications are messages sent outside of the app's UI, even when it's closed, to notify the user about something. They generally appear at the top with a small icon on the left and a short body which describes the purpose of the alert. Firebase is the starring again because it's responsible of dispatching notifications on mobile devices. There's a minimal setup for **Android**:

1. Download, as usual, the `google-services.json` configuration following the installation guide at the beginning of the chapter.
2. In order to be able to react to the user's tap on the notification bar (at the top of your

device) be sure to add the following lines in the manifest file.

```
<intent-filter>
    <action android:name="FLUTTER_NOTIFICATION_CLICK" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

Of course, push notifications for **iOS** are managed by Firebase too and you're asked to do the following setup as well:

1. You need to generate a certificate for your app but it's quite easy to do:
 - (a) Open your developer account page and go to "Certificates, Identifiers and Profiles". Click on **Keys** and add a new one by giving a custom description.
 - (b) Select the checkbox next to "Apple Push Notification service"
 - (c) Confirm your selection and download the file. Be sure to not lose it because it's a one-time download that cannot be retrieved later!
2. Place the `GoogleService-Info.plist` following the installation guide at the beginning of the chapter, unless you haven't already done this step.
3. Still on Xcode, select Runner and inside "Capabilities" turn on push notifications, background modes, background fetch and remote notifications.
4. Check the `pub.dev` package page of `firebase_messaging` to see how to upload the APN certificate and enabling the notification manager via Objective-C/Swift.

After this long preparation, you're finally ready to write some Dart code to make the notification work. Add `firebase_messaging` as dependency and create a dedicated file, which could be called `fcm_setup.dart`, containing the logic to send requests to the FCM cloud service.

```
class FirebasePushManager {
    FirebasePushManager._();

    // A singleton
    static final _instance = FirebasePushManager._();
    factory FirebasePushManager() => _instance;

    bool _initialized = false;

    Future<void> init() async {
```

```
if (!initialized) {
    // Note that 'FirebaseMessaging' is a singleton as well. We
    // create 'firebaseMsg' just for convenience
    final firebaseMsg = FirebaseMessaging();

    // This appears only on iOS but it's really important.
    await firebaseMsg.requestNotificationPermissions();
    firebaseMsg.configure(...);

    _initialized = true;
}
}
```

This class interacts with FCM (Firebase Cloud Messaging) to send and receive push notifications. The `init()` method should be called at startup, maybe in the way we've suggested in appendix B.3. After having called `init()`, your device will be able to successfully receive push notifications. In addition, you probably might find calling another method very useful as well:

```
// still inside the 'init()' method
await firebaseMsg.requestNotificationPermissions();

firebaseMsg.configure(
    onLaunch: (Map<String, dynamic> data) { ... },
    onMessage: (Map<String, dynamic> data) { ... },
    onResume: (Map<String, dynamic> data) { ... },
);
```

When a push notification is received, other than a message and the icon to show in the device's tray icon, it can also carry some data. The `Map<String, dynamic> data` parameter represents the payload coming together with the notification:

- the `onMessage` callback is triggered when the app is in the foreground, which is practically when "the app is opened" (the user can interact with it);
- the `onLaunch` and `onResume` callbacks are called when the app is closed or it's in the background.

When you see a push notification at the top of your device, generally you want to be able to tap on it and open the app. More specifically, it would be even better if you were redirected to a

specific route of your app when tapping on the notification.

22.5.1 Handling push notifications

The `Map<String, dynamic>` `data` parameter contains the additional information received together with the notification. In the next section, we're going to see how to add this kind of data. In the `init()` method, we start by configuring this:

```
firebaseMsg.configure(  
    onMessage: _showAlert,  
    onLaunch: _navigateToRoute,  
    onResume: _navigateToRoute,  
) ;
```

Since `onMessage` is triggered while the user is interacting with the app, moving suddenly to a new page might be confusing and "intrusive". For this reason, an alert dialog (or any other form of message) is better.

```
Future<void> _showAlert(Map<String, dynamic> data) async {  
    final author = data["author"] as String;  
  
    // Actually you should use 'showDialog' or 'showCupertinoDialog'  
    // to nicely display a message to the user  
    debugPrint("Notification sent by $author");  
}
```

The `onLaunch` and `onResume` callbacks are triggered, respectively, when the app is closed or in the background and the notification is tapped. A very common need is the following: the app is closed, a push notification is received, the user taps on it and a specific route has to open.

```
Future<void> _navigateToRoute(Map<String, dynamic> data) async {  
    // When sending a notification, you add as payload the name of  
    // the route to which you want to navigate.  
    final routeName = data["route"] as String;  
  
    // Navigate to the route... but there's no context available!  
    Navigator.of(context)? .pushNamed(routeName);  
}
```

In this way, tapping on the notification opens your app and moves to the given route. However, with our setup there's no `BuildContext` available because the method signature only has a

`Map<K,V>`. You have two solutions:

- change the signature and add a `BuildContext context` parameter but then you might still be in trouble passing a context;
- use a `GlobalKey` and associate to it a `navigatorKey` in your material or cupertino root widget. In order to do this, you could create a new key in `RouteGenerator` as we've seen in 12.3:

```
class RouteGenerator {
    RouteGenerator._();

    // Expose a key to use a navigator without a context
    static final key = GlobalKey<NavigatorState>();

    state Route<dynamic> generateRoute(RouteSettings settings) {...}
}
```

It can now be attached to the navigator in the root widget:

```
MaterialApp(
    onGenerateRoute: RouteGenerator.generateRoute,
    navigatorKey: RouteGenerator.key,
)
```

There's now the possibility to navigate among routes without having a context available. Thanks to the key, we can now use this code so that when the push notification is tapped, the given route is opened.

```
Future<void> _navigateToRoute(Map<String, dynamic> data) async {
    // Name of the route received from the payload
    final routeName = data["route"] as String;

    // Navigate to the route without having a context
    RouteGenerator.key.currentState?.pushNamed(routeName);
}
```

22.5.2 Sending push notifications

Push notifications can be sent very easily in the online Firebase console. Just open it, select your project, click on "Cloud Messaging" and create a new notification. Give it a title, a body and

the URL of an image that will be displayed along with the notification.

Notification title [?](#)

Warning

Notification text

I'm about to send a notification!

Notification image (optional) [?](#)

<https://mywebsite.com/images/icon.png>



You can schedule when the notification should be sent, the audience, give it an ID and much more. In the final step, you have the possibility to insert some optional *custom data* fields: they're the payload of the notification.

Custom data [?](#)

click_action

FLUTTER_NOTIFICATION_CLICK

route

version_page

This is what Flutter will store as `Map<String, dynamic>` in the callbacks you setup in the `configure()` method. You **must** use the `click_action: FLUTTER_NOTIFICATION_CLICK` value in order to be able to correctly receive data and react to tap events.

```
// The "route" param we set earlier as payload is the "route" you see here
```

```
// in the map. It's the landing page to be opened when the notification is
// tapped.
final routeName = data["route"] as String;
```

The console is very convenient but very likely you'll also need to sent push notifications via Flutter. That's not a problem at all because it's just a matter of making a POST request:

```
Future<void> sendPush(String title, String message) async {
    final token = await FirebaseMessaging().getToken();

    final jsonBody = <String, dynamic>{
        "notification": <String, dynamic>{
            "title": title,
            "body": message,
        },
        "data": <String, dynamic>{
            "click_action": "FLUTTER_NOTIFICATION_CLICK",
            "route": "your_route_name_if_needed",
        },
        "to": token,
    };

    await Dio().post("https://fcm.googleapis.com/fcm/send",
        data: jsonBody,
        options: RequestOptions(
            headers: {
                "Authorization": "key=your_fcm_key",
                "Content-Type": "application/json"
            }
        )
    );
}
```

We've kept the code simple for the sake of the example but it'd be better if you moved `Dio()` into a `static` variable to cache the configurations. Note that the `"to"` field has to be assigned with `getToken()` in order to properly send the notification.

```
"Authorization": "key=your_fcm_key",
```

In order to get the key, you need to open the Firebase console and go to the settings. Once there,

choose the "Cloud Messaging" tab and copy/paste the provided key. It's used by Firebase to identify your app so that it knows where the notification has to be sent.

Key	Token
Server key	AAAA-9DpWGI:APA91bHPPMJt_y4l3R3...

Thanks to **topics**, you can send push notifications to specific groups of devices or simply broadcast them to everyone. Let's make a very simple example to get the idea of how topic messaging works on Firebase ¹¹. Pretend you created an app for your friends to invite them at your home parties.

```
Future<void> sendPush(String title, String message) async {
    final token = await FirebaseMessaging().getToken();
    final jsonBody = <String, dynamic>{
        "notification": <String, dynamic>{ ... },
        "data": <String, dynamic>{ ... },
        "to": token,
    };

    await Dio().post(...)
}
```

This is not very useful because we're using "`to`" to send the notification to a specific device. Thanks to topic messaging, you can send a notification to everyone or only to a small group of people. If you wanted to invite your best friends only to the party, the request would be a bit different:

```
Future<void> sendPush(String title, String message) async {
    final jsonBody = <String, dynamic>{
        "notification": <String, dynamic>{ ... },
        "data": <String, dynamic>{ ... },
        "condition": "'best_friends' in topics",
    };
}
```

¹¹<https://firebase.google.com/docs/cloud-messaging/android/topic-messaging>

```
    await Dio().post(...)  
}
```

Using "`condition`" instead of "`to`" you can send a notification **only** to a particular group of people, not to specific devices. Client side, your best friends' app must be registered to the "`best_friends`" FCM topic. You could make the subscription in a settings page or anywhere else: it's just a method to be awaited.

```
await firebaseMsg.subscribeToTopic("best_friends");
```

Calling `subscribeToTopic()` the device is able to receive notifications sent to the given topic. Devices that didn't subscribe, won't see the notification. Actually, a device can subscribe to one or more topics.

```
"condition": "'best_friends' in topics && 'parents' in topics",
```

You can also send a notification to multiple groups combining topics with `&&` (and) or `||` (or). In the above example, you're inviting your best friends and your parents to the party. In summary:

- Using the "`to`" configuration, push notifications are sent to a single device (or group of IDs).
- Using the "`condition`" configuration, push notifications are sent only to devices that subscribed to a certain topic(s). You can create the "`all`" topic to send notifications to everyone.
- A single client can subscribe to multiple topics.

Generally, the "`condition`" approach is the most flexible because it doesn't rely on ids or specific device-related groups.

22.6 Authenticating with Firebase

Nowadays "register" and "login" actions are very common not only in applications, but also in websites and desktop programs. Generally you're asked to register with a valid email address to which a verification code will be sent. Thanks to Firebase, you can easily do the following:

- register an user with email and password;
- send a verification code via email to confirm the address;
- authenticate via Google account, Facebook or other providers;

- send verification codes via SMS, phone call and much more.

We're going to show you how to create a flexible architecture that supports multiple kinds of authentication providers (actually not necessarily tied to Firebase). The example shows how to work with email and password registration/authentication.

1. As always, if you haven't already, follow the installation guide at the beginning of the chapter. Download the `.json` configuration files and place them in the correct folder.
2. Install the official "firebase_auth"¹² plugin and be sure to setup the Android build dependencies. Add the following lines at `/android/build.gradle`:

```
dependencies {  
    classpath 'com.android.tools.build:gradle:3.2.1'  
    classpath 'com.google.gms:google-services:4.3.0'  
}
```

3. In order to enable email/password authentication on Firebase, we need to open the online console and navigate to the "Authentication" tab. From there, there's the need to enable the provider we want to use.

Provider	Status
 Email/Password	Enabled
 Phone	Disabled
 Google	Disabled

Even if they're not shown in the above picture, you can also enable other sign-in providers such as GitHub, Facebook, Twitter, Google, Microsoft and much more. Firebase can even

¹²https://pub.dev/packages/firebase_auth

make a phone call for a direct phone verification.

4. We're now going to use the "*Strategy pattern*" to create a flexible architecture. Our example is going to register users with email and password but in the future you might want to add more authentication strategies. For this reason, we create an abstract class `UserRepository` with basic authentication methods:

```
// Interface
abstract class UserRepository<T> {
  Future<T> register();
  Future<T> signIn();
  Future<void> signOut();
}
```

We're now going to implement this interface in `/lib/models/auth/email.dart` using the Firebase authentication service. Note that `AuthResult` comes from the `firebase_auth` package.

```
class EmailUser implements UserRepository<AuthResult> {
  final String email;
  final String password;
  const EmailUser({
    required this.email,
    required this.password,
  });

  @override
  Future<AuthResult> register() {
    final auth = await FirebaseAuth.instance
      .createUserWithEmailAndPassword(
        email: email,
        password: password
      );

    // We'll deal with this later
    // await auth.user.sendEmailVerification();
    return auth;
  }

  @override
```

```
Future<void> signIn() async =>
    await FirebaseAuth.instance
        .signInWithEmailAndPassword(
            email: email,
            password: password
        );

@Override
Future<void> signOut() async =>
    await FirebaseAuth.instance.signOut();
}
```

Only a few lines of code are required since Firebase will automatically take care of both registration and login of the user. The code is self-explanatory because the class is really small (the library does basically everything). The `AuthResult` class contains data about the logged user which can be accessed and modified in various ways:

- changing the profile picture,
- deleting the account,
- changing credentials...

The registration, if successful, automatically authenticates the user so no need to call `signIn()` later. If the password were too weak or the email were not valid, an exception would be thrown. If you plan to add more authentication providers in the future, just create new concrete implementations of `UserRepository<T>`:

```
// Inside lib/models/auth/email.dart
class EmailUser implements UserRepository<AuthResult> { ... }

// Inside lib/models/auth/facebook.dart
class FacebookUser implements UserRepository<SomeFbObj> { ... }

// Inside lib/models/auth/github.dart
class GithubUser implements UserRepository<SomeGitObj> { ... }
```

With this architecture you're on the good SOLID way!

Firebase takes care of creating a secure authentication process with many important features such as email verification and phone codes. If you had to make all of this by yourself, you'd have a lot of work to do but Firebase is just here, ready to be used and it's very well integrated with

Flutter.

```
// Usage example of the EmailUser class
try {
    final provider = EmailUser(
        email: "hello@email.com",
        password: "_my_c0mpl3x_passw0rd_"
    );

    final user = await provider.register();
} on FirebaseAuthException catch (e) {
    if (e.code == 'weak-password') {
        print('Error: weak password');
    }
    if (e.code == 'email-already-in-use') {
        print('Error: email already in use');
    }
} catch (e) {
    print("Whoops, something's gone wrong :(");
}
```

Authentication errors are exposed via `FirebaseAuthException` which internally has `message`, a textual description of the problem, and `code`, an unique error code. You can use a `Stream<User>` to subscribe to real-time authentication state changes:

```
FirebaseAuth.instance
    .authStateChanges()
    .listen((User user) {
        if (user != null) {
            // signed in
        } else {
            // signed out
        }
});
```

Similarly, there's also the `userChanges()` stream but it's more "general purpose" as it emits events about token refreshes and more. Remember that `FirebaseAuth` automatically persists the user's authentication state so that it will survive page reloads and app restarts.

```
await FirebaseAuth.instance
```

```
.setPersistence(Persistence.NONE);
```

In other words, `FirebaseAuth` automatically stores your authentication state so that it will be available again even if the app is closed and reopened. If you don't like this default behavior, use `setPersistence()` at startup to disable state persistence.

22.6.1 Authentication features

When registering users via email and password, you might want to verify the existence of the given address with a verification code. Firebase can send an email to the given address with a code; the user has to correctly confirm it to validate the account.

```
final auth = await FirebaseAuth.instance
    .createUserWithEmailAndPassword(
        email: email,
        password: password
);

await auth.user.sendEmailVerification();
```

Once the user logged, you can check whether its email has been verified or not and eventually decide to send a validation message. Generally, you should send the email right after the registration but regardless, `sendEmailVerification()` can be called anywhere on a `User` object.

```
final user = FirebaseAuth.instance.currentUser;

if (!user.emailVerified) {
    await user.sendEmailVerification();
}
```

There's no need to sign-out the user and sign-in him again to refresh his email verification status. You can create a form and ask for a code, so that the user can copy/paste it from the email. The verification can happen immediately in the app with this method:

```
Future<void> validateUser(String code) async {
    try {
        await auth.checkActionCode(code);
        await auth.applyActionCode(code);

        auth.currentUser.reload();
    } on FirebaseAuthException catch (e) {
```

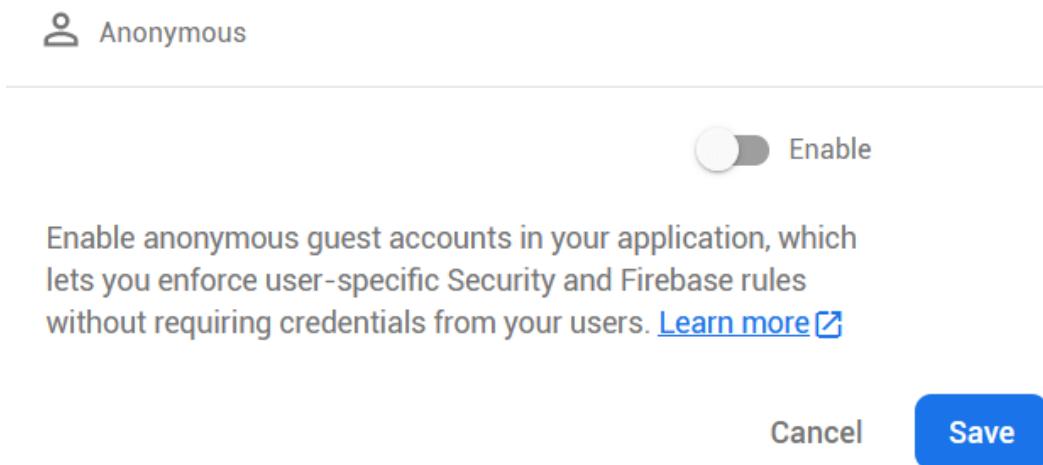
```
    if (e.code == 'invalid-action-code') {
        // wrong code...
    }
}
```

The `reload()` method refreshes the status of the current user, if authenticated, and thus it also updates the value of `emailVerified`. Other than email and password registration, there are also other kinds of authentication strategies:

- Your app might not need authentication but it could still require a way to uniquely identify visitors. For this purpose you can use anonymous sign-in, whose state is persisted on the device (like it happens for "normal" authentication):

```
final credentials = await FirebaseAuth.instance.signInAnonymously();
```

You have to enable this feature in the Firebase console as well otherwise it won't work. Simply go in the "Authorization" page and enable the feature under this "Sign-in method" tab.



- You could authenticate using an external provider, such as Google. In this case, the user will be asked to access with his Google credential rather than going with the classic Firebase authentication.

```
// Authenticate with google and get the data back
```

```
final googleUser = await GoogleSignIn().signIn();
final googleAuth = await googleUser.authentication;

// Creation of a credential object for Firebase
final googleCredential = GoogleAuthProvider.credential(
    accessToken: googleAuth.accessToken,
    idToken: googleAuth.idToken,
);

// Sign in
final credentials = await FirebaseAuth.instance
    .signInWithCredential(googleCredential);
```

In a very similar way, you can also access with Facebook, Twitter and GitHub. The library also exposes classes to work with generic OAuth credentials and Recaptchas.

23 | Publishing packages and apps

23.1 Publishing packages on pub.dev

As you already know, many developers around the world can contribute to Flutter's growth writing packages and publishing them at <https://pub.dev>. It's like a big public repository in which you can find many packages for any platform (web, mobile or desktop). Are you looking for something not present in pub yet? You can be the first to publish it!

💡 We're working with Dart 2.10 and Flutter 1.20.4, which are the latest versions at the time of writing this book. The code we're going to write will be fine for future versions but we can't guarantee compatibility for Dart 2.8 and lower.

In this section we're showing a step-by-step guide on how to publish a package, from its creation up to the release on the official <https://pub.dev> repository. The package we're creating is called *fraction* since it's going to be a facility to work with mathematical fractions. You already know the minimum required building blocks to get started:

1. **pubspec.yaml**: describes the package with many info such as author, version, supported platforms, dependencies and so on.
2. **lib**: contains the Dart source code of your package.

That's really all you need to create a simple package but it's not enough for production because we also need to implement testing, versioning and licensing. Instead of manually creating all the required files from scratch, the `flutter` command line tools can create a template for us.

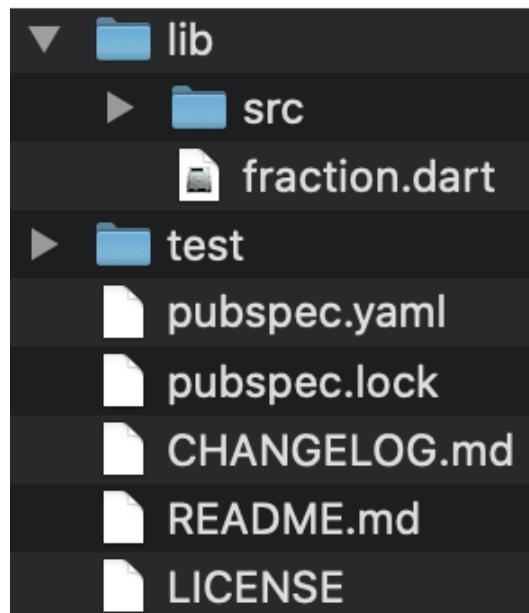
```
$ flutter create --template=package fraction
```

Running this command inside a folder, `flutter` prepares all the files and the folders you need to

create a package. Open `lib/` and start coding.

23.1.1 Creating the package

You could create a single file inside `lib/` and throw all the code in there but it's not a good idea at all. When creating packages, there is a certain structure recommended by the official documentation¹ you should follow (mostly for ease of use and consistency with other people's work).



That's the template the command `flutter create --template` produced in your file system and you shouldn't change it. In addition to the generated contents, we've added something new inside `lib/`:

1. a file with the same name as the package, in this case `fraction.dart`, which exports the public API. This makes sure that using our library will require only a single `import` statement.
2. a `src/` folder containing the actual Dart source code of the package.

The `lib/fraction.dart` is the "heart" of the package because it takes the various classes, located

¹<https://dart.dev/guides/libraries/create-library-packages>

Chapter 23. Publishing packages and apps

in different files, and exposes them to the outside. By convention, any package should contain inside `lib/` a Dart file with this structure.

```
// contents of 'fraction.dart'  
library fraction;  
  
export 'src/core.dart';  
export 'src/extension_num.dart';  
export 'src/extension_string.dart';
```

It is like a "collector": it grabs and exposes your sources all together so that when it's time to use the library, it can simply be referenced using a single `import` statement...

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

... rather than importing each source file one by one

```
import 'package:fraction/core.dart';  
import 'package:fraction/extension_num.dart';  
import 'package:fraction/extension_string.dart';
```

This *fraction* package is available both on GitHub ² and pub ³. We've written a pure Dart package with no Flutter dependencies but we could have easily created a `FractionWidget` simply by importing the material or cupertino packages, for example.

```
example/  
lib/  
test/  
    fraction_test.dart  
    mixed_test.dart  
CHANGELOG.md  
LICENSE  
README.md  
analysis_options.yaml  
pubspec.yaml
```

Of course, we have also written a lot of tests inside `test/` using the `test` package as we've seen back in chapter 16. They've been grouped into logical "categories" according to the type of feature being tested.

²<https://github.com/albertodev01/fraction>

³<https://pub.dev/packages/fraction>

```
void main() {
  group("Constructors", () {
    test("Numerator and denominator", () {
      var f = Fraction(5, 7);
      expect(f.toString(), "5/7");
    });
    // other tests...
  });

  group("Operators", () {
    ...
  });

  group("Methods", () {
    ...
  });
}
```

Packages where performance is critical might also include a `benchmark/` folder with a Dart file executing some tests. Note that this file is **not** meant to test the correctness of the code but its speed, resources consumption and other metrics critical to you.

```
benchmark/
  benchmark_feature1.dart
  benchmark_feature2.dart
lib/
test/
```

23.1.2 Documenting the code

Packages apart, in general Dart code can be documented to explain how methods or classes work and how they can be used. The documentation is fundamental to know what other developers had in mind but it has to be maintained in parallel with the code: they must live together.

- ➊ While writing the code, you know what you're thinking and those thoughts should be properly documented for others (or for "your future self"). Anyone new to your code needs explanations of what and why you wrote the code in that way.

You should always document at least your code's public API but avoid writing long wall-texts; whenever you can, try to be as concise as possible. If it's the case of a complicated method instead, don't be afraid to write a lot. It might happen that a method has more comments than actual code but that's fine.

- i** If you look at Flutter's source files, you'll see that in certain cases there are more comments and examples than actual code. They are useful for you to understand how something works and how it can be used. If Dart and Flutter had no documentation for their classes and methods, it would be a really big problem!

Documenting code is done with triple slashes */// Doc* which is different from usual comments because they use double slashes *// Comment*. You can document classes, methods, getters, setters and so on as long as it's Dart code.

```
/// Use triple slashes to document.  
/// You can also go to a new line  
class HttpRequest {  
    /// Checks whether the API is available or not  
    bool isOnline() {}  
}
```

You could also document the code using only comments but that's not ideal. If you use triple slashes as we've shown above, there's the possibility to use the *dartdoc* utility to generate a very nice HTML page of your documentation. In addition, the IDE can read your docs and show hints when you hover on the text. The process is very easy:

1. document the code with */// Triple slashes docs*;
2. use the *dartdoc* CLI on your Dart file and it will automatically generate a series of HTML pages with the documentation. The documentation of the packages at pub.dev is generated with this tool as well so here's another reason of why you should really use triple slashes.

If you use comments *dartdoc* won't be able to generate the pages for you. The official Dart documentation ⁴ exposes a series of best practices for writing documentation.

- Use triple slashes rather than double slashes to document your code.

```
/// OK - Documentation  
// NO - Comment
```

⁴<https://dart.dev/guides/language/effective-dart/documentation>

- Try to not document every single piece of code you've written; document most of the public API. Of course, private members can be documented too but it's not really needed.
- Don't write too much details that can be omitted; be short and direct to the point.

```
/// Executes an asynchronous GET request with a timeout
/// of 3 seconds
Future<String> fetchData(); // OK

/// This method executes an HTTP GET request to the API
/// configured in the model class (api/model.dart). Each request
/// has a timeout of 3 seconds; in case of success, a string
/// with the body of the response is returned.
Future<String> fetchData(); // NO
```

- If you're documenting a complex method, a lot of text may be needed. Avoid writing wall-texts: consider separating the paragraphs after 4 or 5 lines to make the reading less dense and concentrated.
- You can also write code and *dartdoc* will generate a nice, formatted snippet in the resulting HTML file. Use triple backticks fences to surround the code you want to highlight.

```
/// Returns the longest of two strings
///
/// ````dart
/// var l = longest("abc", "a");
/// ``
String longest(String a, String b) {}
```

Inside backticks you're allowed to use markdown code to format and indent the code; it will be parsed by *dartdoc* and converted into HTML. For example if you want to organize the text in a bullet list, just use a dash.

```
/// Key points of this class:
///
/// - one ...
/// - two ...
```

23.1.3 Reviewing and publishing

Once the code is written, tested and documented it's ready to be published on [pub.dev](#) so that other developers can start using your package. Before submitting it, you should dedicate some time to improve a few files related to the package:

- Check the *LICENSE* file to be sure you are fine with its contents.
- A template for `pubspec.yaml` was automatically generated when you created the package and it already contains the minimal required fields. Be sure to check out the official Dart documentation ⁵ for an overview of any field you can use.
- Write a detailed description of your package in the `README.md` file. It will be placed in your package's home page at [pub.dev](#) so it should look good. Describe the features, use images in case of widgets, write examples and put a link to external references (if any, like your package's website).
- Edit the `CHANGELOG.md` file to reflect the main changes you've applied to the current version that is going to be published.

If you want to compare the setup of the *fraction* package with yours, visit our official GitHub repository. Once all those files have been reviewed, it's finally time to publish the package. Just to be sure that everything is ok, let's run this command in the terminal:

```
flutter pub publish --dry-run
```

It executes a "fake publishing" which doesn't upload your package but it just simulates what would happen if you did so. It might be useful to look for warnings or problems before moving on. If no problems are reported, make the real publishing:

1. Open the console, move to the root of your project and launch these commands:

```
cd lib/  
flutter format .
```

It moves the console to the `lib/` folder and uses `format` to format all of your Dart source code. This is also needed in order to not get penalty points on pub (more on this soon).

2. Publish your package:

```
flutter pub publish
```

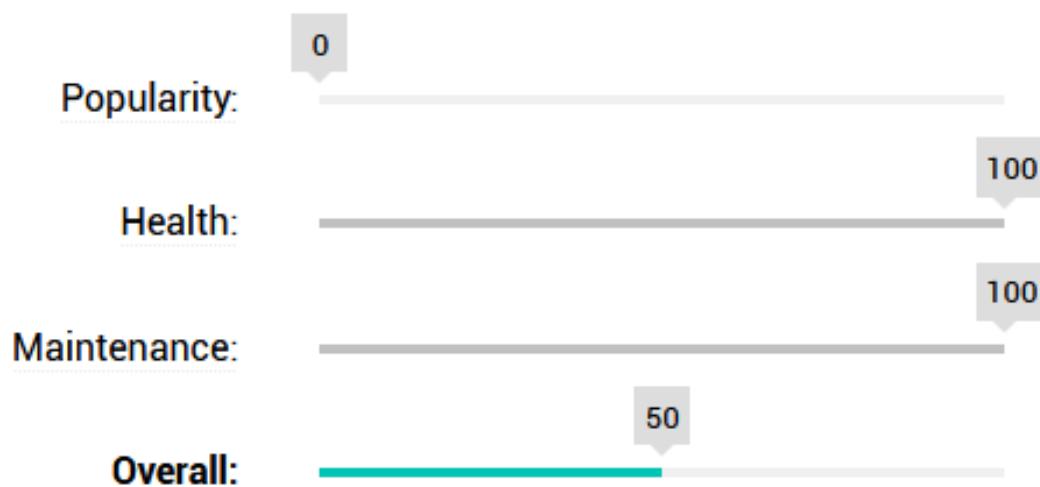
⁵<https://dart.dev/tools/pub/pubspec>

3. The console will tell you to open a link; login with your Google account to confirm the submission.
4. Wait a few minutes for pub to process the files. It will also automatically generate a nice HTML website using the documentation text you've written with triple slashes.

Nothing difficult at all: it's just a matter of running two commands and confirming the submission with a Google login. Keep in mind that once submitted, packages **cannot** be deleted so they're going to stay in the repository forever. The reason is simple: the deletion would break other people's code depending on your package.

23.1.4 Scores and good practices

Any package has a score ranging from 0 to 110 and your goal is of course trying to reach the maximum. While you can maximize **pub points** following some good practices, the other statistics depend on the community. This is the current score of our *fraction* package at the beginning of September 2020.



There's the possibility to calculate pub points before publishing the package using the **pana**⁶ analysis tool (pub uses it as well). Running pana locally will give you a preview of the points and

⁶<https://pub.dev/packages/pana>

the possible suggestions you'd get on the website.

```
# Activate pana
flutter pub global activate pana

# Run it
pub global run pana ~/path/to/package
```

Since pana makes changes to your package, run it on a copy so that the original project won't be affected by modifications. The three main scores of a package are:

- **Likes.** Indicates how many developers liked this package. To like a package, login with your Google account and click on the thumbs up button next to the name. You can see the packages you liked in *My pub.dev > My liked packages*
- **Pub points.** This metric gives insights about the quality of the package. The criteria used to compute the final score are:
 - Follow Dart file conventions. To pass this check, use the `flutter create --template` command we've discussed earlier and carefully review `pubspec.yaml` and the mark-down files.
 - Provide documentation. Document your code using triple slashes, following the good practices we recommended. To get the maximum score, be sure that at least 20% of your public API is documented. Writing a proper, complete documentation goes beyond the success on this metric: do it for quality and maintenance!
 - Support multiple platforms. Currently, your packages should support both native and web platforms but this isn't an issue at all. Whether you're writing a pure Dart package or a Flutter one, in both cases you won't have problems.
 - Pass static analysis. Static analysis determines whether your package contains warnings, errors or styling issues. To maximize the score of this metric, be sure to:
 1. setup the `analysis_options.yaml` file as soon as possible (right after the creation of the package template);
 2. validate a package before publishing it with `flutter analyze`.

For a complete and updated reference, have a look at the official Dart style guide ⁷.

⁷<https://dart.dev/guides/language/effective-dart/style>

- Support up-to-date dependencies. If you have any dependency declared in the pubspec file, be sure to always use the latest version.
- **Popularity.** Indicates how many developers are using your package in their projects. On a scale from 0 to 100, a high percentage indicates that a big number of apps are depending on your package.

You're going to lose points if you **don't** provide an `example/` folder at the root of the project with some runnable demos so you'd better create it. You'll also get penalty points if the `description` in your `pubspec.yaml` file is too short; it should contain 60 - 180 characters.

23.1.5 Verified publishers and Flutter favorite

You may have noticed that certain publishers at [pub.dev](#) have a blue shield next to the name. Thanks to it you know that a package was uploaded by a publisher whose identity has been verified and thus there's more authority to the product. The badge doesn't give info about the quality of the code but it's tied to the *authenticity* of the publisher.



Being a verified published isn't complicated but there are a few steps to follow. You need an active Google account which will be associated with both [pub.dev](#) and the *Google Search Console*.

1. Decide which Google account you want to use and login to the Google Search Console⁸.
2. Any verified publisher must have a domain to associate with his pub account. In our case, the domain is [fluttercompletereference.com](#) which also points to our website. That's not a requirement, you can simply have a valid domain without a website associated.
3. Type your domain in the console and then you'll be asked to verify its identity via DNS. In practice, you have to go to your domain provider settings and add a new DNS text record. Even after the verification, **don't** remove this entry because Google periodically checks your domain.

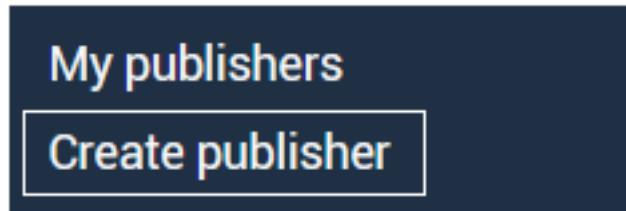
`IN TXT "google-site-verification=XXXXXXXXXXXX"`

The console will give you the exact string to insert; the verification might take some times due to the propagation times of the DNS entries but in 24h you should be able to do it.

⁸<https://search.google.com/search-console/about>

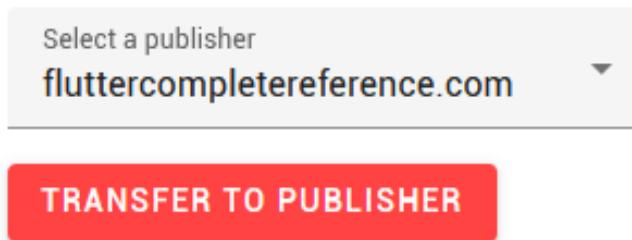
There are also other ways to verify your domain, such as via HTML <meta> tag but it doesn't work with *pub*. You have to do the DNS way.

- Once you've managed to verify your domain with success in the console, open [pub.dev](#) and login. Go to the top-right corner and click *create publisher*.



Type the verified domain name in the box and click "CREATE PUBLISHER". You'll get an error if you try to submit a domain that's not previously been verified via DNS.

At this point you are officially a verified published and the blue badge will always appear next to the name. If you already had published some packages with a non-verified account, just go to the **Admin** area of your package and transfer it to your verified account.



Being a verified publisher is not only a matter of pride but it also gives you a chance to become eligible for the Flutter favorite program. This campaign identifies all those top-quality packages you should first consider when building apps. However, don't blindly prefer them over the other alternatives. They are very good for sure but being a *flutter favorite* doesn't make a package perfect.



It's very unlikely that a newly created package will become a Flutter favorite in short spans of time. There are a lot of criteria to satisfy such as feature completeness, performance, code quality, GitHub integration, being a verified publisher and much more. You can see all the details about this program in the official page ⁹.

23.2 Publishing apps on the stores

At the end of the development cycle, when all the features have been implemented and the testing is done, it's time to put the app in production. When you use the emulator with the hot reload feature, Flutter is working in *debug* ¹⁰ mode which mainly implies a few things:

- the size of the binary file, *.apk/.aab* for Android or *.ipa* for iOS, is very large;
- sometimes the app doesn't run so smoothly or certain transitions might not seem very fluid;
- any `assert()` statement in your code will be evaluated.

The emulator and the hot reload feature work only in debug mode, which is the default one when you execute the `flutter run` command. When it's time to deploy the app for production, switch to release mode to have:

- smaller size of the binary file;

⁹<https://flutter.dev/docs/development/packages-and-plugins/favorites>

¹⁰<https://flutter.dev/docs/testing/build-modes>

- faster execution with fluid transitions, faster startup and general optimizations automatically made by the compiler;
- `assert()` statements are removed.

As you already know from chapter 16, there is also a third mode called profile which enables profiling with "DevTools". In the next sections we will analyze how to prepare an app for the *release* mode and potentially the publication on the official stores.

23.2.1 Releasing Android apps

Before publishing to the Google Play store, there are some final steps to follow to define the details of the app (the icon, the name and the assignments of security certificates). Be sure to check the official documentation ¹¹ for more info about deploying apps on Android devices.

1. You need to have Java installed on your machine because it has the `keytool` command line program. It's used to generate a keystore file to be associated with the binary being built.

```
$ keytool -genkey -v -keystore myAppKey.jks -keyalg RSA  
-validity 30000 -keysize 2048 -alias myKey -storetype JKS
```

Keep this file private and be sure it doesn't get lost otherwise you won't be able to publish updates for this app on the store anymore. It's used to identify **you** as the owner of the app.

2. In the IDE, go to `/android/key.properties` and reference the `.jks` file you've just created. The passwords it asks are the ones you've had to type in the command line program in the previous step

```
storePassword=store_pass  
keyPassword=key_pass  
keyAlias=myKey  
storeFile=location_to_the_file
```

You can place `myAppKey.jks` where you want but be sure to exclude it from version control systems such as GitHub.

3. Now go at `/android/app/build.gradle` and place this code before the `android {}` block:

```
def keystoreProperties = new Properties()  
def keystorePropertiesFile = rootProject.file('key.properties')  
if (keystorePropertiesFile.exists()) {
```

¹¹<https://flutter.dev/docs/deployment/android>

```
        keystoreProperties.load(new FileInputStream(keystorePropertiesFile))
    }
```

In this same file, remove the `buildTypes` block and add all these lines. Basically they instruct gradle to sign your app with the given `keystore` file any time a build in release mode happens.

```
signingConfigs {
    release {
        keyAlias keystoreProperties['keyAlias']
        keyPassword keystoreProperties['keyPassword']
        storeFile keystoreProperties['storeFile'] ?
            file(keystoreProperties['storeFile']) : null
        storePassword keystoreProperties['storePassword']
    }
}
buildTypes {
    release {
        signingConfig signingConfigs.release
    }
}
```

Just to avoid possible caching problems, run `flutter clean`.

- Now move to the manifest file located at `/android/app/src/main`. In the `<application>` tag you can change the `android:label` to give your app the name (the one appearing below the icon on the home screen).

```
<application
    <!-- other settings here ... -->
    android:label="App name" />
```

That way, the name is hard-coded but that's not always what you're looking for. If you want to show different names according to the device's language (so you want to **localize** the title) there's an extra step to do.

```
<application
    <!-- other settings here ... -->
    android:label="@string/appname" />
```

Change the value of `android:label` and then locate the `app/src/main/values` folder; create inside it a file with this specific name: `strings.xml`. Write your app's name in the

native language (in our case, english) following this scheme:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="appname">My app</string>
</resources>
```

Now for any new language you want to support, create a `values-XX` folder containing the localized string for the given locale. For instance, if we wanted to add an Italian value, we should create the `/android/app/src/main/values-it` folder with another `strings.xml` file inside it.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="appname">La mia app</string>
</resources>
```

The name attribute is always the same (`appname`). Our app will be called *My app* in any language, because it's the default value, but when `italian` is set the name will be *La mia app*. You can add as many languages you want.

5. If needed, add the internet permission in the manifest file to allow the connectivity.
6. Your app's icons are located inside the `res/drawable-XXX` folders. It's always the same image but with different sizes because Android will automatically pick the best one according to the device's pixel density. You have to:
 - (a) create one logo for each folder keeping the same sizes of the placeholder image that you find;
 - (b) if you're using Android Studio, you can use the very convenient "*Image Asset Studio*" tool which automatically generates the icons.

With the asset studio tool you can even load SVG images and it will take care of generating all the icons for each folder.

It's been a long way but you've finally made it to the end; you can now build the binary file in release mode. There are two possible formats to choose for the Play Store:

- **.apk**: it is the traditional file extension you're used to see for Android apps. Go to the console and run the following command to generate the binaries:

```
flutter build apk --split-per-abi
```

You'll find three apk files at `/build/app/outputs/apk/release`, one for each ABI¹². This is the best thing to do; you could also generate a single `.apk` with this command...

```
flutter build apk
```

... but you'll end up with a very big file including the code compiled for **each** ABIs. There's no need for this because it would carry useless data for devices that don't support a specific architecture. You'd better split the files and pick only the binary compatible with your target device.

- **.aab**: it's the new file format recommended by the Android team¹³ in place of `apks`. It should be your primary choice when it comes to production apps to be published to the store.

```
flutter build appbundle
```

This command generates at `/build/app/outputs/bundle/release/` a single file with `.aab` extension. It contains the compiled code for the supported ABIs.

Check out appendix A.3 to learn about code obfuscation and a better way of using the `build` command. Both file types contain the compiled code for three ABIs but they're built in different ways. An app bundle is an optimized version that will be further processed by the Play Store while an `apk` is like a "zip" containing all the precompiled binaries.

- i** If you had to share your app, for example, to an internal member of your team to show how it looks, you could create an `apk`. It can be transferred to a device via file manager and then the Android package installer will install it.

You can drag and drop an `apk` in the Android emulator to install it. This is not possible with `aab` files because they're made to be uploaded in the Google Play store; Android doesn't directly "understand" app bundles.

- i** If you want to extract an `apk` from an `aab` file you have to use the `bundletool`¹⁴ tool, which is what the Google Play store uses. An `aab` file is just a better way of handling different ABIs, rather than having multiple `apks`. It's more convenient.

¹²Application Binary Interfaces

¹³<https://developer.android.com/guide/app-bundle>

¹⁴<https://developer.android.com/studio/command-line/bundletool>

23.2.2 Releasing iOS apps

Before publishing to the Apple App store, there are some final steps to follow to define the details of the app (the icon, the name and the assignments of security certificates). Be sure to check the official documentation ¹⁵ for more info about deploying apps on iOS devices.

1. First of all, you need to join the "Apple Developer Program" in order to be able to access the "App Store Connect" (referenced as ASC from now on). This is important because iOS builds happen via Xcode tools which require certificates obtained from the ASC. Open the Apple Developer portal, click on **Identifiers** and add a new "App ID".

Register a New Identifier

● App IDs

Register an App ID to enable your app to access available services and identify your app in a provisioning profile. You can enable app services when you create an App ID or modify these settings later.

Go for **App IDs**. On the next page, type you app's name and be sure to check the **Explicit App ID** option. Enter an ID and pick from the list below the services you app is using, if any. Click on "Register" to confirm.

2. Open the ASC, click "My Apps" and add a new one pressing on + in the top-left corner. Fill the forms with all the required information and then your application tab will appear on the screen. At the end of the process, you'll get something like this on the main page:



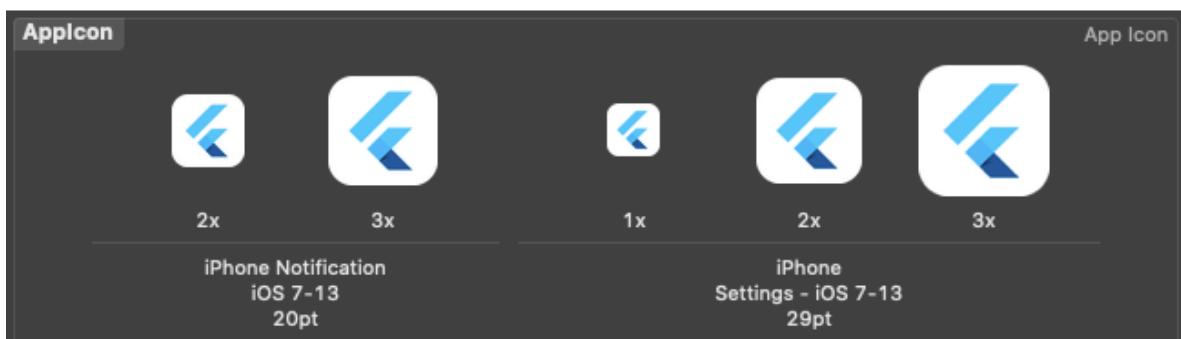
The *My Apps* section of the ASC shows any application you've uploaded to the store with a brief summary of the status. The big image is the icon you're required to upload (more on this later) while at the bottom there's a quick summary of the status:

- (a) Prepare for submission
- (b) Waiting for review
- (c) In review
- (d) Ready for sale

¹⁵<https://developer.apple.com/app-store/submitting/>

Click on the app's logo and provide the various information they need such as screenshots and descriptions. Let's now open Xcode to finalize some other details.

3. Open the iOS folder of your Flutter project. Open `Runner.xcworkspace` and then select "Runner" in the Xcode project navigator. Choose "Runner" as target.
 - In the **Identity** section, change you app's name.
 - In the **Signing** section, make sure that automatic signing is ticked. Choose your team (the one associated to the Apple Developer account) from the dropdown menu. This is quite useful because Xcode will automatically handle profiles, app IDs and certificates.
 - In the **Deployment Info** section be sure that 8.0 or higher is selected.
4. In case you're doing the first release, make sure that `pubspec.yaml` has a version code of 1.0.0 otherwise change it accordingly. Don't manually handle versioning on XCode: for an easier maintenance across multiple platform, rely on the `version:` attribute of `pubspec.yaml`.
5. Add the app icon going to the `Assets.xcassets` folder, located inside `Runner` in the project navigator, and edit the assets. There's a default Flutter placeholder image. Replace it and to be sure it looks as you'd expect, run the app on a Simulator (or a real device).



For a quick prototyping, you can use an online asset generator to build a temporary logo for your app. It will create all the required assets so that you'll just have to download and move the images.

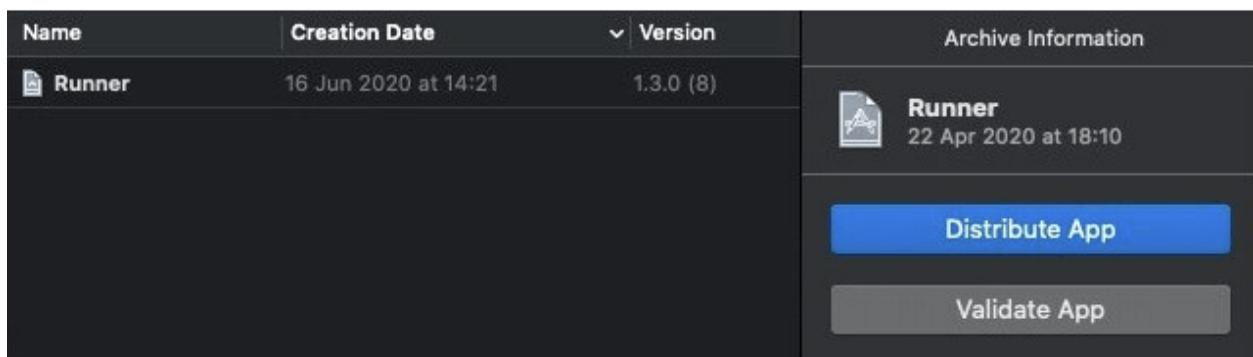
It's finally time to generate the native iOS binary. Close Xcode to avoid refreshing problems with the Flutter build tool and launch the following command:

```
flutter build ios
```

Once the process has completed, return to Xcode again and open the `runner.xcworkspace` file. In sequence, perform the following actions on the menu bar of the IDE:

- Product > Scheme > Runner;
- Product > Destination > Generic iOS Device;
- Project Navigator > Runner > Runner
- Product > Archive

Wait for the completion of the build and then a dialog similar to the one below will appear. It shows all the builds you've done up to now (we've removed some from the image for convenience). You should first click **Validate App** to ensure that everything is ok and then click **Distribute App** to upload the app on the store. This process may take several minutes.



Open the App Store Connect, click on your app, select the build you've uploaded via Xcode and wait for the review of the Apple Store team. Consider reading our appendix A.3 about Dart code obfuscation in release mode.

23.2.3 Splash screens

Some apps might not be ready as soon as they're opened due to time-consuming operations required at startup. For example, they could have to check for updates, load a lot of data from the cache or exchange some data with a server. The problem could also be an old, slow device.

i Think about games for mobile phones. In general, as soon as they're opened, a splash screen appears telling you that the app will start very soon. It gives the feeling of fastness and responsiveness since the very beginning, which is a fundamental UX

aspect. Animations and loading indicators make the waiting time more pleasant.

Flutter may take more than a few milliseconds to start, especially if you're doing a lot of initialization work before calling `runApp`. If it's the case, you'll see a completely white screen appearing (the default splash screen) but you can of course customize it:

1. **iOS.** Very simply, open the `Runner.xcworkspace` file from Xcode, select the assets folder and double click the **LaunchImage** image set. It's simply a PNG file you can customize as much as you want to create a nice and user-friendly splash image.
2. **Android.** Similarly to what we've seen for iOS, all you need to do is editing a certain pre-made template. Opening `src/main/res/values/styles.xml` you'll find the default definition of the splash screen, which is just a `<style>` element pointing to a drawable.

```
<style name="LaunchTheme" parent="@android:style/Theme.Black.NoTitleBar">
    <item name="android:windowBackground">@drawable/launch_background</item>
</style>
```

Go to `src/main/res/drawable/launch_background`, which is the actual splash screen, and style it as you prefer. For example, you could create a PNG image with the logo and a small text below telling the user "Loading..." or something similar.

```
<!-- The background color -->
<item android:drawable="@android:color/white" />

<!-- The PNG at the center of the screen -->
<item>
    <bitmap
        android:gravity="center"
        android:src="@drawable/my_splash_image" />
</item>
```

The file `my_splash_image.png` has been created in the `drawable/` folder.

Keep in mind that even if your app has a splash screen, it shouldn't appear for more than a few seconds. In addition, try to keep it quite simple and be sure to mark the fact that the app is loading and it will be ready very soon. Have a look at the appendix B.3 to see how to gracefully execute startup initialization in Flutter.

23.2.4 Doing CI/CD for Flutter

In the previous sections we've shown you how to prepare applications for production giving a proper name, setting icons and much more. Once it's done, you need to run two different build commands and Flutter will take care of creating the native binaries. To sum it up, the deployment process is the following:

1. run tests to ensure that everything is ok;
2. prepare the app for the Android release (generate keys, configure gradle, fix the manifest and so on);
3. prepare the app for iOS release (setup the certificates, use Xcode to work on the Runner, load assets and so on);
4. publish the native binaries to the stores.

Indeed there's a lot of manual work to do when it comes to publish a new update for the app but you can't do much about it. Icons, names and other details might not change in each new release but still there's the need to go through those four steps. Thanks to certain online services, such as Codemagic, you can automate the testing/deployment phase for your Flutter apps.



Codemagic is a cloud-based CI/CD service that automatically builds, tests and deploys your Flutter native apps to the official stores. When using these kind of services you just need to write the app and the tests; all the other steps (from running tests up to the publication in the official store) are automatic. In particular:

- for each commit to a git repository, there's the possibility to automatically run tests and make builds. For example, if you don't have a macOS and/or an iPhone, you can still successfully test, build and publish an iOS app because the CI/CD service has the hardware;
- any tool is always updated to the latest version. You don't need to manually check for Flutter updates or installing heavy Xcode releases because everything is on the cloud;
- you can run apps on Android emulators, iOS simulators or even real physical devices.

All of this goodness is part of a series of processes called **Continuous Integration** and **Continuous Delivery**. We aren't going to cover CI/CD in detail because it goes beyond the scope of this book but in case you didn't know what they are, here's a description from a purely **practical** point of view.

- **CI.** Developers push a new commit to a git repository (generally on GitHub or GitLab) and a "listener" is triggered: the newly pushed code is built and then tested. With this approach, building and testing **automatically** happen as soon as the new code is committed to the repository so that the team can quickly find bugs.
- **CD.** It starts where CI ends. If tests ran with success and the team is happy with the current status of the product, the CD process starts. The application is packaged, signed (required for iOS and Android) and then deployed.

Codemagic offers 500 free build minutes and two team seats for free every month. For professional developers, they offer a post-paid billing model, which means you will get billed based on usage. Your Flutter and native mobile projects will be up and running with just a few clicks. Alternatively, you could use *fastlane* which is local to your machine and the official Flutter documentation¹⁶ also has a guide on how to use it.

23.2.4.1 GitHub actions

The Dart and Flutter team use GitHub to version their code. If you decide to do the same, which is what we recommend, you can easily setup CI and CD in your repository using *Actions*. For example, our *dart_equations*¹⁷ package has a GitHub action that performs the following tasks for each push (or pull request):

1. install dependencies using pub `get`;
2. run `dartfmt` to make sure the code is well formatted;
3. run tests.

For example, every time you make a `git push` the action installs the dependencies, checks the formatting and runs tests. Once completed, you can see a green tick or a red cross next to the commit telling you respectively whether the action completed with success or failed.

¹⁶<https://flutter.dev/docs/deployment/cd>

¹⁷<https://pub.dev/packages/equations>

Minor documentation fixes



albertodev01 committed 2 days ago ✓

To create a new action, just go on your GitHub repository, click Actions and then choose "set up a workflow yourself". Alternatively, create the `.github/workflows/` folders and create a new YAML file (which is what the website would do for you). In practice, an action is nothing more than a `.yaml` file telling the backend which steps have to be done.

```
name: dart_equations_ci

on:
  push:
    branches:
      - master
  pull_request:
    branches:
      - master

jobs:
  build:
    runs-on: ubuntu-latest
    container:
      image: google/dart:latest
    steps:
      - uses: actions/checkout@v2
      - name: Install dependencies
        run: pub get
      - name: Format
        run: dartfmt --dry-run --set-exit-if-changed .
      - name: Run tests
        run: pub run test
```

This is the action we have implemented for our `dart_equations` and `fraction` packages. This setup

is good for a "pure" Dart application but for Flutter, you need different commands:

```
name: master_ci

on:
  push:
    branches:
      - master

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: subosito/flutter-action@v1.3.0
      - name: Install Dependencies
        run: flutter packages get
      - name: Format
        run: flutter format --set-exit-if-changed lib test
      - name: Run tests
        run: flutter test --no-pub
```

In summary, if you want to quickly setup a GitHub action for your Dart or Flutter projects, just create the folders `.github/workflows` and place a YAML file containing the steps to do. The action itself is a file telling the backend what to do. A repository can run one or more actions.

5 results	Event ▾	Status ▾	Branch ▾	Actor ▾
✓ Fixed some analysis info; removed 'flutter ... master_ci #5: Commit 0eac0fd pushed by albertodev01	master	 5 days ago  1m 57s		...
✗ Updated tests and CI master_ci #4: Commit a963861 pushed by albertodev01	master	 5 days ago  1m 53s		...

For each action, you get the history of the results of runs. GitHub Actions usage is free for public

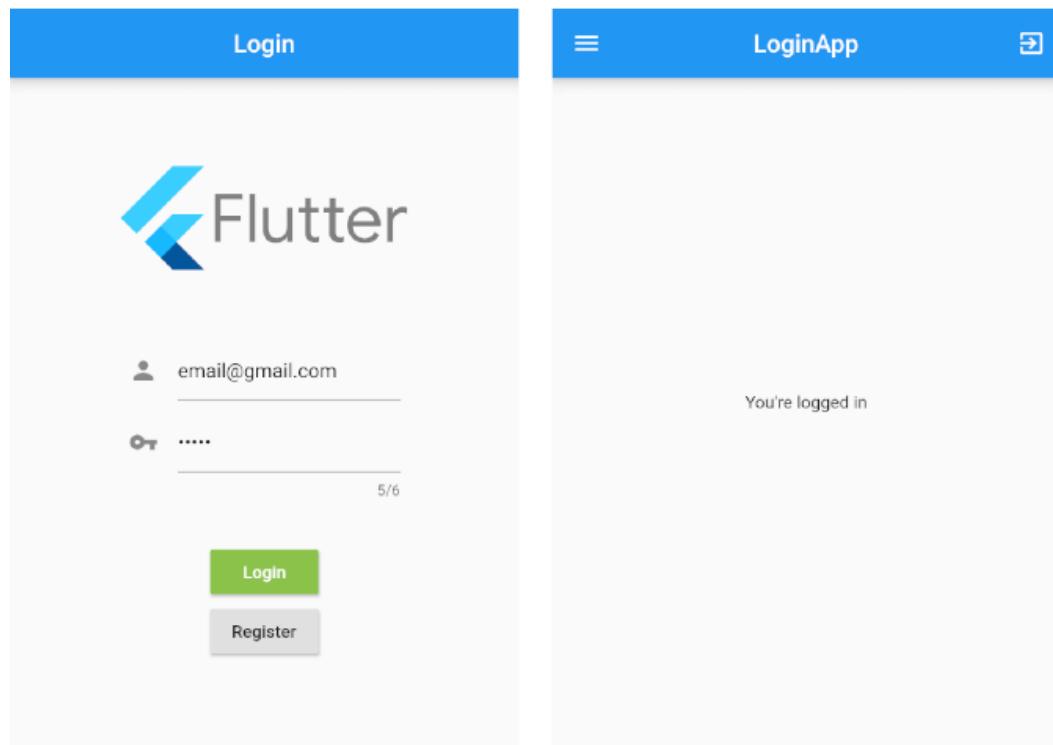
repositories and self-hosted runners. For private repositories, each GitHub account receives a certain amount of free minutes and storage, depending on the product used with the account. For more info on actions, visit the official documentation ¹⁸.

¹⁸<https://docs.github.com/en/free-pro-team@latest/actions>

24 | Complete Flutter project example

24.1 Preparing the project

In this section we're going to show in detail how to create a Flutter application in which an user can register and login using the Firebase authentication system. The project uses email address and password as authentication credentials but any other way would be fine.



Both buttons can be pressed; "Register" adds a new user to the system while "Login" is just authentication for already signed up people. Let's start by opening Firebase and creating a new project called **LoginApp**. As always, follow the installation guide in chapter 22 to correctly place the configuration files. Here's a summary:

- Create a new Android app on Firebase with the correct package name (the same you've used for the Flutter project) and download the `google-services.json` file.
- Move `google-services.json` in the `android/app` folder and setup the Gradle build file.
- Create a new iOS app on Firebase with the correct package name (the same you've used for the Flutter project) and download the `GoogleServices-Info.json` file.
- Move `GoogleServices-Info.json` in the `ios/Runner` folder.

On the home page of your Firebase project, click on "Authentication" on the left sidebar, then choose "Sign-in methods" and enable the email/password login provider. Even before opening the IDE, do a `git init` and version your code!

24.1.1 Folder structures and basic setup

As you already know, managing the state of an app with `setState()` and `InheritedWidget` is difficult and requires too much boilerplate code. For this reason, we're working with the `flutter_bloc` library. We also want to localize the contents and split the UI in multiple reusable widgets so a good folder structure is essential.

```
lib/
  blocs/
  localization/
  repository/
  routes/
  widgets/
  main.dart
  routes.dart
test/
analysis_options.yaml
pubspec.yaml
```

Following good coding practices is very important for our code's health and thus we immediately create the `analysis_options.yaml` file. You can find its definition in the *Resources* section of our website. Inside `lib/localization` create the localization delegate and [class AppLocalization](#)

(see chapter 13), choosing between "manual" or `intl` approach.

```
// main.dart
void main() => runApp(const LoginApp());

class LoginApp extends StatelessWidget {
  const LoginApp();

  @override
  Widget build(BuildContext context) {
    return BlocProvider<AuthenticationBloc>(
      create: (context) {
        // More on blocs in the next section...
        return AuthenticationBloc(repository);
      },
      child: MaterialApp(
        initialRoute: RouteGenerator.homePage,
        onGenerateRoute: RouteGenerator.generateRoute,

        localizationsDelegates: [
          const AppLocalizationDelegate(),
          GlobalMaterialLocalizations.delegate,
          GlobalCupertinoLocalizations.delegate,
          GlobalWidgetsLocalizations.delegate,
        ],
        supportedLocales: [
          Locale.fromSubtags(languageCode: "en"),
          Locale.fromSubtags(languageCode: "it"),
        ],
        onGenerateTitle: (context) => context.localize("title"),
        debugShowCheckedModeBanner: false,
      ),
    );
  }
}
```

Routes are going to be managed inside the `routes.dart` file to keep the navigation logic in a separated file, as we've covered in chapter 12. For safety, we've decided to create a custom

exception type in case the route name were invalid but it won't happen. Since we rely on hard-coded `static` constants, there cannot be typos in the name of the route.

```
/// Routing handler for the app
class RouteGenerator {
    const RouteGenerator._();

    static Route<dynamic> generateRoute(RouteSettings settings) {
        switch (settings.name) {
            case HomePage:
                // We will create 'HomePage' later
                return MaterialPageRoute<HomePage>(
                    builder: (_) => const HomePage(),
                );

            default:
                throw RouteException("Route not found");
        }
    }

    static const HomePage = '/';
}

/// Exception thrown when a given route doesn't exist
class RouteException implements Exception {
    final String message;
    const RouteException( this.message);
}
```

A `switch` statement with a single `case` doesn't make much sense but we're assuming that the app will have many routes. Our example only has the home page but in reality you'll have more than a single route.

24.2 State management and model classes

In order to easily manage user's data, we're going to create an architecture that takes care of authentication tasks. In a more complex scenario, it might also handle authentication tokens or the retrieval from a secured storage of cached username and password.

```
// Inside 'repository/user_repository.dart'  
abstract class UserRepository {  
    /// Creates the repository for authenticating an user  
    const UserRepository();  
  
    /// Email of the signed user  
    String get signedEmail;  
  
    /// Login with username and password  
    Future<bool> authenticate(String username, String password);  
  
    /// Registration with username and password  
    Future<bool> register(String username, String password);  
  
    /// Logout  
    Future<void> logOut();  
}
```

This class has to be implemented in order to create a login provider, such as the Firebase one we're going to show in a moment. This way of modelling classes is known as *Strategy pattern*, a famous design pattern from the Gang of Four (GOF). Note that we're always documenting public methods:

```
// Inside 'repository/user_repository/firebase_repository.dart'  
class FirebaseUserRepository extends UserRepository {  
    /// Firebase authentication repository  
    const FirebaseUserRepository();  
  
    @override  
    String get signedEmail =>  
        FirebaseAuth.instance  
            .currentUser  
            .email ?? "-";  
  
    @override  
    Future<bool> authenticate(String username, String password) async {  
        try {  
            await FirebaseAuth.instance  
                .signInWithEmailAndPassword(
```

```
        email: username,
        password: password
    );

    return true;
} on FirebaseAuthException catch (e) {
    debugPrint(e.message);
    return false;
}
}

@Override
Future<bool> register(String username, String password) async {
    try {
        await FirebaseAuth.instance
            .createUserWithEmailAndPassword(
                email: username,
                password: password,
            );

        return true;
    } on FirebaseAuthException catch (e) {
        debugPrint(e.message);
        return false;
    }
}

@Override
Future<void> logOut() => FirebaseAuth.instance.signOut();
}
```

Note that in case of a failed authentication or registration a `FirebaseAuthException` is thrown but it will be handled later in the bloc. In both cases, the `result` variable is of type `AuthResult` which contains many data about the user such as his email, the avatar image, whether the email has been verified or not and much more.

```
// Inside 'repository/user_repository/test_repository.dart'
class TestUserRepository extends UserRepository {
    /// The email of the user
```

```
final String fakeEmail;

/// Determines whether the methods will fail or not
final bool succ;

/// Mock authentication repository (for testing)
const TestUserRepository({
    required this.fakeEmail,
    required this.succ,
});

@Override
Future<bool> authenticate(String username, String password) {
    return Future<bool>.delayed(const Duration(seconds: 1), () => succ);
}

@Override
Future<bool> register(String username, String password) {
    return Future<bool>.delayed(const Duration(seconds: 1), () => succ);
}

@Override
Future<void> logOut() => Future.delayed(const Duration(seconds: 2));

@Override
String get signedEmail => fakeEmail;
}
```

We've also created the `TestUserRepository` which will be used in unit tests and widget tests. It's a convenient mock for the Firebase authentication process which requires no internet connection or platform setup. Thanks to the usage of the strategy pattern, you can easily add more authentication providers to your application. In the future, if you wanted to also login with Facebook Twitter or Google, you'll just have to implement the interface:

```
// Inside 'repository/user_repository/twitter_repository.dart'
class TwitterUserRepository extends UserRepository {}

// Inside 'repository/user_repository/google_repository.dart'
class GoogleUserRepository extends UserRepository {}
```

```
// Inside 'repository/user_repository/facebook_repository.dart'  
class FacebookUserRepository extends UserRepository {}
```

It's also a very good way of respecting the SOLID principles. You should end up with the following contents inside the `repository/` folder:

```
repository/  
    user_repository.dart  
    user_repository/  
        firebase_repository.dart  
        test_repository.dart
```

24.2.1 Authentication bloc

The `AuthenticationBloc` bloc determines whether the user is authenticated or not. In other words, it's used to decide if the login form has to be displayed or not, according to the current authentication status. Inside `blocs/authentication_bloc/events.dart` we're creating the events:

```
/// Events for the [AuthenticationBloc] bloc  
abstract class AuthenticationEvent extends Equatable {  
    /// Base class for events fired by [AuthenticationBloc]  
    const AuthenticationEvent();  
  
    @override  
    List<Object> get props => [];  
}  
  
/// User has logged with success  
class LoggedIn extends AuthenticationEvent {  
    const LoggedIn();  
}  
  
/// User requested to logout  
class LoggedOut extends AuthenticationEvent {  
    const LoggedOut();  
}
```

When the `LoggedIn` event is fired, the login form disappears and the actual home page is shown.

When `LoggedOut` is fired, the currently visible route disappears and the login form is shown because the user logged out. Notice how we try to use `const` constructors as much as possible. Here's the states:

```
/// States emitted by [AuthenticationBloc]
abstract class AuthenticationState extends Equatable {
    /// Base class for states emitted by [AuthenticationBloc]
    const AuthenticationState();

    @override
    List<Object> get props => [];
}

/// App just opened, login or register actions required
class AuthenticationInit extends AuthenticationState {
    const AuthenticationInit();
}

/// Login made with success
class AuthenticationSuccess extends AuthenticationState {
    const AuthenticationSuccess();
}

/// Logout
class AuthenticationRevoked extends AuthenticationState {
    const AuthenticationRevoked();
}

/// Loading (awaiting for registration or authentication)
class AuthenticationLoading extends AuthenticationState {
    const AuthenticationLoading();
}
```

In case of `AuthenticationLoading()` a loading indicator appears on the UI indicating that the app is awaiting for Firebase to return a response. It's now time to create the bloc itself to actually manage our app's authentication state:

```
/// Manages the authentication state of the app
class AuthenticationBloc
```

Chapter 24. Complete Flutter project example

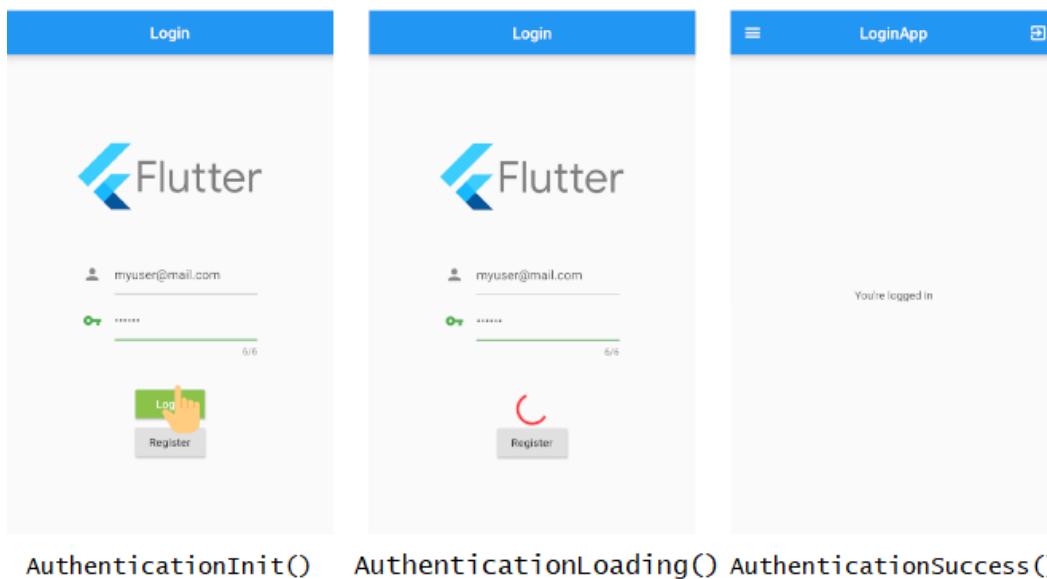
```

    extends Bloc<AuthenticationEvent, AuthenticationState> {
    final UserRepository userRepository;
    AuthenticationBloc(this.userRepository) :
        super(const AuthenticationInit());

    @override
    Stream<AuthenticationState> mapEventToState(AuthenticationEvent e) async*{
        if (e is LoggedIn) {
            yield const AuthenticationSuccess();
        }
        if (e is LoggedOut) {
            yield const AuthenticationLoading();
            await userRepository.logOut();
            yield const AuthenticationRevoked();
        }
    }
}

```

As you can see, when the `LoggedIn` event is fired `AuthenticationSuccess` is emitted by the bloc so that the app knows to change page (move from the login page to the actual home page). Tapping on the login button triggers the login bloc (see next sections) which, in case of success, will fire a `LoggedIn` event causing the actual home page to appear.



`AuthenticationInit()` `AuthenticationLoading()` `AuthenticationSuccess()`

Your code is surely going to reference the bloc file in an `import` statement but state and event files are also required. In some cases you might end up having to import a lot of libraries, especially if multiple blocs are needed by a single class.

```
import 'package:flutter_app/blocs/authentication_bloc/bloc.dart';
import 'package:flutter_app/blocs/authentication_bloc/events.dart';
import 'package:flutter_app/blocs/authentication_bloc/states.dart';
import 'package:flutter_app/blocs/credentials_bloc/bloc.dart';
import 'package:flutter_app/blocs/credentials_bloc/events.dart';
import 'package:flutter_app/blocs/credentials_bloc/states.dart';
```

As you already know from chapter 4, we can use the `library` keyword to group multiple `import` statements in a single one. In other words, each bloc is a "mini library" which can easily be referenced by a single statement.

```
// contents of '/lib/blocs/authentication_bloc.dart'
library authentication_bloc;

export 'authentication_bloc/bloc.dart';
export 'authentication_bloc/events.dart';
export 'authentication_bloc/states.dart';
```

This is very useful because with a single `import` statement we can reference the bloc, its events and its states all together. Maintenance will also benefit from this so you should really use this approach when creating blocs.

```
import 'package:flutter_app/blocs/authentication_bloc.dart.dart';
import 'package:flutter_app/blocs/credentials_bloc.dart';
```

In the next section we're creating the `CredentialsBloc` class, which handles the login and fires the `LoggedIn` event in case of successful authentication or authorization.

24.2.2 Credentials bloc

While `AuthenticationBloc` is used to move between the login and home page, `CredentialsBloc` is taking care of communicating with Firebase to authenticate and/or register an user. Two events are required, one for the *Login* button and the other for the *Register* button:

```
/// Events for the [CredentialsBloc] bloc
abstract class CredentialsEvent extends Equatable {
    /// The username
```

```
final String username;

/// The password
final String password;

/// Events fired by [CredentialsBloc] when a button is pressed. It
/// provides information taken from the form.
const CredentialsEvent(this.username, this.password);

@Override
List<Object> get props => [username, password];
}

/// Event fired when the login button is tapped
class LoginButtonPressed extends CredentialsEvent {
    const LoginButtonPressed({
        required String username,
        required String password
    }) : super(username, password);
}

/// Event fired when the register button is tapped
class RegisterButtonPressed extends CredentialsEvent {
    const RegisterButtonPressed({
        required String username,
        required String paxssword
    }) : super(username, password);
}
```

The bloc is going to use UserRepository to make the actual registration or authentication but it also has to handle the waiting time and the errors. The following states are required:

```
/// States emitted by [CredentialsBloc]
abstract class CredentialsState extends Equatable {
    /// State emitted by [CredentialsBloc] when the form is created
    const CredentialsState();

    @override
    List<Object> get props => [];
}
```

```
}

/// Action required (authentication or registration)
class CredentialsInitial extends CredentialsState {
    const CredentialsInitial();
}

/// Login request awaiting for response
class CredentialsLoginLoading extends CredentialsState {
    const CredentialsLoginLoading();
}

/// Registration request awaiting for response
class CredentialsRegisterLoading extends CredentialsState {
    const CredentialsRegisterLoading();
}

/// Invalid authentication credentials
class CredentialsLoginFailure extends CredentialsState {
    const CredentialsLoginFailure();
}

/// Weak password or invalid email
class CredentialsRegisterFailure extends CredentialsState {
    const CredentialsRegisterFailure();
}
```

Failure states are used to represent errors while trying to authenticate, due to wrong credentials, or register, due to problems with the email. In case of errors, a generic `PlatformException` with many different codes you can find in the source code ¹. In order to not make the example too complicated, we've decided to simply handle exceptions with a single error message indicating to try a different email.

```
/// Manages the login state of the app
class CredentialsBloc extends Bloc<CredentialsEvent, CredentialsState> {
    /// Data about the user
    final UserRepository userRepository;
```

¹https://github.com/FirebaseExtended/flutterfire/tree/master/packages/firebase_auth

```
/// The [AuthenticationBloc] taking care of changing pages
final AuthenticationBloc authenticationBloc;

/// Creates a Bloc taking care of managing the login state of the app.
CredentialsBloc({
    required this.authenticationBloc,
    required this.userRepository,
}) : super(const CredentialsInitial());

@Override
Stream<CredentialsState> mapEventToState(CredentialsEvent e) async* {
    if (event is LoginButtonPressed) {
        yield* _loginPressed(event);
    }

    if (event is RegisterButtonPressed) {
        yield* _registerPressed(event);
    }
}

Stream<CredentialsState> _loginPressed(CredentialsEvent e) async* {}
Stream<CredentialsState> _registerPressed(CredentialsEvent e) async* {}

}
```

We've decided to split the body of `mapEventToState()` into multiple functions in order to keep the code readable. If the authentication fails because of a wrong combination of username and password, an exception of type `PlatformException` is thrown.

```
Stream<CredentialsState> _loginPressed(CredentialsEvent event) async* {
    yield const CredentialsLoginLoading();

    try {
        await userRepository.authenticate(
            event.username,
            event.password,
        );
    }

    authenticationBloc.add(LoggedIn());
```

```
        yield const CredentialsInitial();
    } on PlatformException {
        yield const CredentialsLoginFailure();
    }
}

Stream<CredentialsState> _registerPressed(CredentialsEvent event) async* {
    yield CredentialsRegisterLoading();

    try {
        await userRepository.register(
            event.username,
            event.password,
        );

        authenticationBloc.add(LoggedIn());
        yield const CredentialsInitial();
    } on PlatformException {
        yield const CredentialsRegisterFailure();
    }
}
```

Note that `authenticationBloc.add(LoggedIn());` tells the authentication bloc to move from the login page to the home page because the login happened with success. We've done the same thing in the registration method too, but that's not always desired. If you need to confirm the email address before being able to authenticate for example, remove the line so that a successful registration will **not** move to the home page.

```
// contents of '/lib/blocs/credentials_bloc.dart'
library credentials_bloc;

export 'credentials_bloc/bloc.dart';
export 'credentials_bloc/events.dart';
export 'credentials_bloc/states.dart';
```

Like we did before with `AuthenticationBloc`, we're creating a library here as well in order to minimize the number of `import` in our files.

24.2.3 Localization files

We're going to localize our application using the "manual" approach described in chapter 13.2 because we don't have much content to internationalize. Since you're probably going to use this authentication example in a larger application, you could stick with `intl` instead because there might be a lot of content to localize.

```
/// Localization delegate that builds an [AppLocalization] instance
class AppLocalizationDelegate
    extends LocalizationsDelegate<AppLocalization> {
    /// Localization delegte of the app
    const AppLocalizationDelegate();

    @override
    bool isSupported(Locale locale) =>
        ['en', 'it'].contains(locale.languageCode);

    @override
    Future<AppLocalization> load(Locale locale) =>
        SynchronousFuture<AppLocalization>(AppLocalization(locale));

    @override
    bool shouldReload(LocalizationsDelegate<AppLocalization> old) => false;
}
```

This is the classic setup for a localization delegate. `AppLocalization` is created following the same structure we described in 13.2.1, along with the extension method.

```
/// Adds an useful localization method on a [BuildContext]
extension LocalizationExt on BuildContext {
    String localize(String value) {
        final code = AppLocalization.of(this)?.locale.languageCode ?? "en";
        final database = AppLocalization._localizedValues;

        if (database.containsKey(code)) {
            return database[code]?[value] ?? "-";
        } else {
            return database["en"]?[value] ?? "-";
        }
    }
}
```

```
        }

/// This class is responsible of translating strings into a certain
/// languages defined by the [Locale] passed in the constructor.
class AppLocalization {
    final Locale locale;
    const AppLocalization(this.locale);

    static AppLocalization? of(BuildContext context) =>
        Localizations.of<AppLocalization>(context, AppLocalization);

    static final Map<String, Map<String, String>> _localizedValues = {
        "en": {
            "title": "Login app",
            "login": "Login",
            ...
        },
        "it": {
            "title": "Login app",
            "login": "Entra",
            ...
        },
    };
}
```

Have a look at our GitHub repository to see the complete code.

24.3 Building the UI

The `UserRepository` model is required by two blocs to authenticate the user and only a single instance of it is required. We could cache the repository using `Provider` so that, in the future, if we wanted to change the authentication provider, we'd just need to make a small change in the `void main()` method.

```
void main() => runApp(
    Provider<UserRepository>(
        create: (_) => FirebaseUserRepository(),
```

Chapter 24. Complete Flutter project example

```

        child: const LoginApp(),
),
);
}

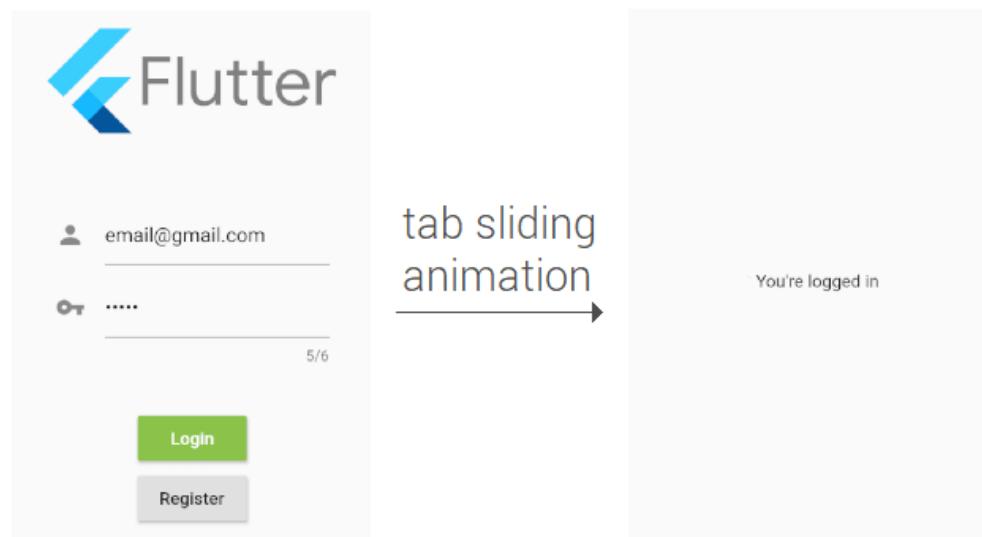
class LoginApp extends StatelessWidget {
const LoginApp();

@Override
Widget build(BuildContext context) {
final repository = context.select((FirebaseUserRepository r) => r);

return BlocProvider<AuthenticationBloc>(
create: (_) => AuthenticationBloc(repository),
child: MaterialApp(...),
);
}
}

```

Create a new file called `routes/home_page.dart` which is going to be the first route to appear in the UI when the app loads. It's a single widget with two children: the login form widget and the welcome page widget. When the authentication successfully happens, there is a sliding transition to the left which brings the user to the welcome page:



This kind of animation is easy to obtain with a simple *trick*: we just need to create a `TabBarView` with no tabs at the top and the removal of the swiping gesture. In this way, we can use a controller to programmatically swipe the pages to the left/right on successful login/logout.

```
return TabBarView(  
    physics: const NeverScrollableScrollPhysics(),  
    controller: tabController,  
    children: const [  
        _LoginPage(),  
        _WelcomePage(),  
    ],  
) ;
```

In order to move tabs programmatically (moving tabs using the code rather than the finger) we need to use a `TabController`. Thanks to the `animateTo` method we can obtain the traditional sliding transition for tabs using code rather than a swiping gesture.

```
class HomePage extends StatefulWidget{  
    const HomePage();  
  
    @override  
    _HomePageState createState() => _HomePageState();  
}  
  
class _HomePageState extends State<HomePage>  
with SingleTickerProviderStateMixin {  
late final TabController tabController;  
  
    @override  
    void initState() {  
        super.initState();  
        tabController = TabController(  
            vsync: this,  
            length: 2,  
        );  
    }  
  
    @override  
    void dispose() {
```

```
        tabController.dispose();
        super.dispose();
    }

    /// Sliding animation to show the welcome page
    void loginTransition() {
        if (tabController.index != 1)
            tabController.animateTo(1);
    }

    /// Sliding animation to show the login page
    void logoutTransition() {
        if (tabController.index != 0)
            tabController.animateTo(0);
    }

    @override
    Widget build(BuildContext context) {...}
}
```

When calling `loginTransition()` the welcome page appears because the `TabBarView` widget is showing the second page, since authentication successfully executed. With `logoutTransition()` instead, the login form is shown because the user requested to logout. Those two methods are called by the bloc according to the emitted state:

```
@override
Widget build(BuildContext context) {
    return BlocBuilder<AuthenticationBloc, AuthenticationState>(
        builder: (context, state) {
            // This state is emitted on successful authentication
            if (state is AuthenticationSuccess) {
                loginTransition();
            }

            // This state is emitted on logout
            if (state is AuthenticationRevoked) {
                logoutTransition();
            }
    }
}
```

```

        return TabBarView(
            physics: const NeverScrollableScrollPhysics(),
            controller: tabController,
            children: const [
                _LoginPage(),
                _WelcomePage(),
            ],
        );
    );
}

```

Thanks to `BlocBuilder` we can "catch" the incoming states and login/logout the user in both the backend (Firebase) and the frontend (moving from welcome page to the login page and vice versa). Since a `const` constructor cannot be directly applied to a `Scaffold` because not all of its contents are constant, we have created (in the same file) two private classes. We're going to analyze them in the next section.

- ❶ This is a quite common strategy which can be used in all those cases where a "root" widget cannot be constant. For example, in this case there's no way to use a constant constructor because certain classes doesn't have one:

```

children: const [ // NOT possible.
Scaffold(
    appBar: AppBar(...), // AppBar doesn't have a const constructor
    body: BlocProvider(...),
),
]

```

If we created a stateless widget dedicated to only contain the `Scaffold` with its content, we could then define a constant constructor. You could place it in a separated file but since the code is quite short and the `Scaffold` is logically a "core" part of the containing widget, we can make a class in the same file. It should be private though, because it's not a public reusable widget: it's an internal part of a widget that's been moved outside for optimization purposes.

```

// create this at the bottom....
class _LoginPage extends StatelessWidget {
    const _LoginPage();

```

```
    @override
    Widget build(BuildContext context) => Scaffold(...);
}

// and then in the tab we can use...
children: const [ //OK now it's possible
    _LoginPage(),
]
```

If the widget can be reused by other widgets, put it in a separated file and use a `const` constructor. If the widget is used only by a single widget because it's part of its "internals", put it in the same file as a private class and use a `const` constructor. However, consider splitting classes into multiple files if the code gets too long and logically you prefer keeping them separated.

Actually it's not so important placing multiple widgets in the same place (with private classes) or splitted in various files. It's up to you deciding what's better, no golden rules to apply: what's fundamental instead is that you try to use `const` as much as possible.

24.3.1 Creating the login form

It's our app's first widget appearing since authentication is required. As you've just seen, we've moved it into a separated class in order to have a useful constant constructor.

```
class _LoginPage extends StatelessWidget {
    const _LoginPage();

    @override
    Widget build(BuildContext context) {
        final repository = context.select((FirebaseUserRepository r) => r);
        final authBloc = context.bloc<AuthenticationBloc>();

        return Scaffold(
            appBar: AppBar(...),
            body: BlocProvider(
                child: const LoginForm(),
                create: (context) => CredentialsBloc(
                    userRepository: repository,
```

```
        authenticationBloc: authBloc,  
        ),  
        ),  
    );  
}  
}
```

We've placed the actual `Form` in a separated widget in order to have the possibility to declare a `const` constructor in the provider of the bloc. It could have been put in the same file as a private widget but the code is quite long and thus we preferred moving it inside `widgets/home_page/login_form.dart` for a better logical order. Text controllers, disposals and form keys are not shown for brevity.

```
class LoginForm extends StatefulWidget {  
    const LoginForm();  
  
    @override  
    _LoginFormState createState() => _LoginFormState();  
}  
  
class _LoginFormState extends State<LoginForm> {  
    // controllers, form key, call to 'dispose' here...  
  
    @override  
    Widget build(BuildContext context) {  
        return LayoutBuilder(  
            builder: (context, data) {  
                var baseWidth = 250.0;  
  
                // For wider screen, such as tablets  
                if (data.maxWidth >= baseWidth) {  
                    baseWidth = data.maxWidth / 1.4;  
                }  
  
                return Column(  
                    mainAxisAlignment: MainAxisAlignment.center,  
                    children: [  
                        SvgPicture.asset("assets/flutter_logo.svg",  
                            width: baseWidth,
```

```
        ),  
  
        ...  
    ]  
);  
}  
);  
}  
}
```

In order to make the app responsive, we've used a `LayoutBuilder` to define the width of the logo. It's fixed by default but if the screen gets bigger, the image changes dimensions according to the available space. In the same way, the `baseWidth` variable is used to determine the width of the `TextEditingController`s of the form.

```
Form(  
    key: _formKey,  
    child: Wrap(  
        children: <Widget>[  
            SizedBox(  
                width: baseWidth - 30,  
                child: TextFormField(...),  
            ),  
  
            SizedBox(  
                width: baseWidth - 30,  
                child: TextFormField(...),  
            ),  
        ],  
    ),  
,
```

Now that the logo and the form are setup, we need to take care of the *Login* and *Register* buttons. They're visible by default but, when tapped, a circular indicator must appear to indicate that a communication with the server is happening. This is quite easy to do with a `BlocConsumer` widget:

```
BlocConsumer<CredentialsBloc, CredentialsState>(  
    listener: (context, state) {  
        if (state is CredentialsLoginFailure) {
```

```

        // Show a snackbar or a dialog to notify the failure
    }
},
builder: (context, state) {
    if (state is CredentialsLoginLoading) {
        return const CircularProgressIndicator();
    }

    return RaisedButton(
        child: Text(context.localize("login")),
        onPressed: () {
            final state = formKey.currentState;

            if (state?.validate() ?? false) {
                _loginButtonPressed(context);
            }
        },
    );
},
),

```

The listener is used to show a message dialog in response of an error state, which is very handy. In the builder instead we swap between a circular indicator and the login button according to the state emitted by the bloc. Note how we've localized the text of the button and used a constant constructor. Of course the action should be performed only if the form has been properly filled:

```

void _loginButtonPressed(BuildContext context) {
    BlocProvider.of<CredentialsBloc>(context).add(
        LoginButtonPressed(
            username: _emailController.text,
            password: _passwordController.text
        )
    );
}

```

There's another `BlocConsumer` for the registration button which is basically identical with exception for the name displayed in `Text` and the different callback which fires an event of type `RegisterButtonPressed`.

24.3.2 Creating the welcome page

When the authentication successfully executes, the user can see the welcome page and thus start actually using our application. This widget is the starting point from which you can open new routes using a drawer, using action buttons and much more. We've provided a logout button at the top-right corner of the app, in the `Scaffold`.

```
class _WelcomePage extends StatelessWidget {
    const _WelcomePage();

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                centerTitle: true,
                title: const Text("LoginApp"),
                actions: <Widget>[
                    IconButton(
                        icon: const Icon(Icons.exit_to_app),
                        onPressed: () => BlocProvider
                            .of<AuthenticationBloc>(context)
                            .add(LoggedOut()),
                ],
            ),
            drawer: const Drawer(),
            body: const Center(
                child: Text("You're logged in"),
            ),
        );
    }
}
```

Sending a `LoggedOut` event to the authentication bloc causes the `TabBarView` to swipe to the right and show the login form (in other words, we've made a logout from the app). The bloc emits a state of type `AuthenticationRevoked` which is the actual logout from Firebase **and** a call to the tab controller in the UI:

```
// Body of the 'HomePage' widget, inside the 'BlocBuilder'
if (state is AuthenticationRevoked) {
```

```
    logoutTransition();
}
```

24.4 Testing the code

Even if the testing section is at the bottom of the chapter, tests shouldn't be written and executed at the end of the development cycle. For example, as soon as the `AuthenticationBloc` is ready to be used in the UI you should also write tests for it. We recommend the usage of this kind of folder structure:

```
test/
  unit/
    authentication_bloc_test.dart
    credentials_bloc_test.dart
  widget/
    login_form_test.dart
  integration/
```

Create one folder for each testing strategy and then organize the files inside as you prefer. It's an intuitive way to logically group entities together for a quicker search and a better mental order.

24.4.1 Unit tests

Create `test/unit/authentication_bloc_test.dart` which will test `AuthenticationBloc`, using the `bloc_test` package. Here's where the fake user repository we created is very useful because it can be easily used as a mock for an authentication provider:

```
void main() {
  final authenticationRepository = const TestUserRepository(
    fakeEmail: "alberto@miola.it",
    success: true,
  );

  blocTest<AuthenticationBloc, AuthenticationEvent, AuthenticationState>(
    'Authentication successful',
    build: () async => AuthenticationBloc(authenticationRepository),
    act: (bloc) async => bloc.add(LoggedIn()),
    expect: <AuthenticationState>[
```

Chapter 24. Complete Flutter project example

```

        AuthenticationSuccess(),
    ],
);

blocTest<AuthenticationBloc, AuthenticationEvent, AuthenticationState>(
    'Authentication failed',
    build: () async => AuthenticationBloc(authenticationRepository),
    act: (bloc) async => bloc.add(LoggedOut()),
    expect: <AuthenticationState>[
        AuthenticationLoading(),
        AuthenticationRevoked(),
    ],
);
}

```

In the same way, inside `test/unit/login_bloc_test.dart` we're writing the code to test the login bloc. Again, thanks to the fake repository we're able to easily emulate the success or failure of the authentication just by changing the value of `loginResult`.

```

void main() {
    final successRepository = const TestUserRepository(
        fakeEmail: "alberto@goodtest.it",
        success: true,
    );

    final failedRepository = const TestUserRepository(
        fakeEmail: "alberto@failtest.it",
        success: false,
    );

    blocTest<CredentialsBloc, CredentialsEvent, CredentialsState>(
        'Successful login',
        build: () async => CredentialsBloc(
            authenticationBloc: AuthenticationBloc(successRepository),
            userRepository: successRepository
        ),
        act: (bloc) async => bloc.add(LoginButtonPressed()),
        expect: <CredentialsState>[
            CredentialsLoginLoading(),

```

```
        CredentialsInitial(),
    ],
);

blocTest<CredentialsBloc, CredentialsEvent, CredentialsState>(
    'Registration failed',
    build: () async => CredentialsBloc(
        authenticationBloc: AuthenticationBloc(failedRepository),
        userRepository: failedRepository
    ),
    act: (bloc) async => bloc.add(RegisterButtonPressed()),
    expect: <CredentialsState>[
        CredentialsRegisterLoading(),
        CredentialsRegisterFailure(),
    ]
);

// more tests...
}
```

Any other model class you will create should be properly tested in a dedicated file. You should write tests for a class as soon as it's ready to be used in the code so that you maintain the actual implementation and the testing in parallel.

24.4.2 Widget tests

Widget testing is also very important and the `bloc_test` package is essential when it comes to testing blocs. We're first going to create inside `test/widget/mock_material.dart` a reusable mock for a `MaterialApp` which will be very handy.

```
class MockMaterialApp extends StatelessWidget {
    final Widget child;
    const MockMaterialApp({
        required this.child
    });

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

```
        localizationsDelegates: [
            const AppLocalizationDelegate(),
            GlobalMaterialLocalizations.delegate,
            GlobalCupertinoLocalizations.delegate,
            GlobalWidgetsLocalizations.delegate,
        ],
        supportedLocales: [
            Locale.fromSubtags(languageCode: "en"),
            Locale.fromSubtags(languageCode: "it"),
        ],
        home: child,
    );
}
}
```

You'd have done the same if the project worked with a CupertinoApp. Having a MaterialApp widget is important because, for example, localization delegates are required in order to test certain widgets relying on context.localize("my_value"). Mocking blocs is no different from what we've seen in 16.1:

```
class MockAuthenticationBloc extends MockBloc<AuthenticationState>
    implements AuthenticationBloc {}

void main() {
    final authBloc = MockAuthenticationBloc();

    whenListen<AuthenticationState>(
        authBloc,
        Stream.fromIterable(<AuthenticationState>[
            AuthenticationLoading(),
            AuthenticationSuccess(),
        ])
    );

    testWidgets("Testing how the authentication behaves", (tester) async {
        await tester.pumpWidget(
            BlocProvider<AuthenticationBloc>.value(
                value: authBloc,
                child: MockMaterialApp(

```

```
        child: const WelcomePage(),
    ),
)
);

final key = Key("HomeMessage");
expect(find.byKey(key), findsOneWidget);
});
```

Again, mind a good folder structure to not mix different types of tests in the same file. For example, like it happens with unit test, each bloc should be tested in a separated file for a better organization of the code.

A | Dart Appendix

A.1 The I/O library

The `"dart:io"` library is used to work with files, directories, WebSockets, HTTP clients and processes. Almost any I/O operation in Dart is executed asynchronously to avoid blocking the application so results are returned either via `Future<T>` or `Stream<T>`.

A.1.1 Files

In Dart you can read a file in two ways: all at once or lazily. Reading a file entirely requires having enough memory. Note that `Uint8List` is basically a `List<int>` so its contents can be traversed regularly like if they were items of a list.

```
final myFile = File('data.json');

// The entire content of 'data.txt' is read
final String content = await myFile.readAsString();

// Each line of the files is represented by a String
final List<String> contents = await myFile.readAsLines();

// Reads the file as a sequence of bytes (raw bytes)
final Uint8List data = await myFile.readAsBytes();
```

Consider using the above approaches when the file is not too big because it's entirely loaded in memory. In mobile apps you generally don't store big text files so calling `File.readAsString()` in Flutter is not a problem. With larger files, you probably should lazily read the contents using a stream.

```
final myFile = File('data.json');
```

```

final Stream<List<int>> stream = myFile.openRead();

final fileLines = utf8.decoder
    .bind(stream)
    .transform(LineSplitter());

try {
    await for (var line in fileLines) {...}
} catch (e) {
    print(e);
}

```

The file is closed as soon as the `await for` terminates. The `utf8` variable is a constant part of the `dart:convert` library. Writing to a file can easily be done using the `File` class:

```

File('something.txt').writeAsString("hello");
File('another.abc').writeAsBytes([0x3, 0xA2]);

```

When creating a `File` object you can also decide the mode, which can be one of the following:

- `FileMode.append`: Read and write at the end of the file (which is created if it doesn't exist)
- `FileMode.read`: File can only be read
- `FileMode.write` (default): Used to read and write the file (which is overwritten if it already exists or created if it doesn't exist).
- `FileMode.writeOnly`: the same as `write` but there's no possibility to read the contents.

By default `File` is in `write` mode but if you don't want, for example, overriding the file every time, just create it with the `append` mode.

A.1.2 Directories

Working with directories is quite easy thanks to `class Directory` which exposes many useful methods. An object of type `Directory` must contain a path on which operations are executed and it can be absolute or relative.

 In Flutter you should avoid using this class directly. Instead, prefer using the `path_provider` package which can be seen as the cross-platform version of `Directory`.

Appendix A. Dart Appendix

It automatically resolves the correct paths to the various system folders.

Like it happened with files, most of the operations are asynchronous (or streams) so that the main application is not blocked by I/O operations.

```
final dir = await Directory('folder1/folder2')
    .create(recursive: true);
```

Thanks to `recursive: true` you're sure that the target directory (*folder2*) is created **together** with its parents (*folder1*) if they don't exist. There are a series of "classic" methods you'd expect to be able to use on this kind of object:

- `createTemp()`: creates a temporary directory in the given directory;
- `rename()`: renames a directory;
- `exists()`: used to determine whether a directory exists or not;
- `delete()`: deletes a directory;
- `list()`: lists all the sub-directories and files

There's also the possibility to use a `Stream<FileSystemEvent> watch()` to listen for actions happening in the directory such as insertions or deletions.

A.1.3 Server side Dart

With Dart you can easily setup an HTTP server using `HttpServer`, which heavily relies on `Future<T>` and `Stream<T>`, just by providing an IP address and a specific port. Thanks to its asynchronous nature, it can listen for requests and handle more than one at the same time:

```
import 'dart:io';

Future<void> main() async {
    // Start server on localhost:8075
    var server = await HttpServer.bind(
        InternetAddress.loopbackIPv4, 8075
    );

    // Asynchronously handle incoming requests
    await for (HttpRequest request in server) {
        request.response.write("""<html><body>
```

Appendix A. Dart Appendix

```
<p>Hello client!</p>
</body></html>""");
}

}

}

await request.response.close();
}
```

The class `HttpRequest` is a server-side object containing information about the received HTTP request. Of course you can have more control on the incoming requests but the approach is quite "low level" as you manually have to handle the entire response lifecycle. This is a more elaborate example:

```
void main() {
  var server = await HttpServer.bind(
    InternetAddress.loopbackIPv4, 8075
  );

  await for (HttpRequest request in server) {
    handle(request);
  }
}

void handle(HttpRequest request) {
  if (request.method == 'GET') {
    handleGet(request);
  } else {
    if (request.method == 'POST') {
      handlePost(request);
    } else {
      request.response
        ..statusCode = HttpStatus.methodNotAllowed
        ..write('${request.method} not handled')
        ..close();
    }
  }
}
```

You can also run an HTTPS server using `bindSecure()` rather than `bind()`. For a complete

Appendix A. Dart Appendix

tutorial about writing HTTP servers with Dart, refer to the official documentation ¹. You can find on <https://pub.dev> a very convenient package called *http_server*, a higher level way of building HTTP servers with Dart. It's easier to use:

```
Future<void> main() async {
  var htmlPages = VirtualDirectory('www_root');

  htmlPages.directoryHandler = (directory, request) {
    final homePath = Uri.file(directory.path).resolve('index.html');
    final homePage = File(homePath.toFilePath());

    htmlPages.serveFile(homePage, request);
  };

  final server = await HttpServer.bind(
    InternetAddress.loopbackIPv4, 8075
  );

  await server.forEach(htmlPages.serveRequest);
}
```

The *VirtualDirectory* class is a secure way to serve files and directories in response to HTTP requests. It can be used to return HTML pages together with mime-types, potential error codes and so on. Be sure to check out the official package documentation ² to get more info.

A.2 Date and time

In Dart the *Duration* class is used to represent time spans and making conversions from one unit to another. It's used to represent the difference between two moments, which can be negative in case the difference were from a later time to an earlier one.

```
// It also has 'days', 'milliseconds' and 'microseconds'
final myWorkToday = Duration(
  hours: 8,
  minutes: 25,
  seconds: 38,
);
```

¹<https://dart.dev/tutorials/server/httpserver>

²https://pub.dev/documentation/http_server/latest/http_server/http_server-library.html

The above time span can be converted into seconds, for example, using the `inSeconds` getter which will return the **total** converted value of the interval. Look at this example to better understand how it works:

```
final total = Duration(
  hours: 9,
  minutes: 11,
  seconds: 37
);

// 9 hours, 11 minutes and 37 seconds are equivalent to 33097 TOTAL seconds:
// 9 * 3600 + 11 * 60 + 37 = 33097
print("${total.inSeconds}"); // prints 33097

// Returns the remainder of the division of the duration by 60. In fact
// 33097 % 60 = 37
print("${total.inSeconds.remainder(60)}"); // prints 37
```

In practice, `inSeconds` returns the representation of the **entire** duration in seconds. If you want to only extract the `seconds` parameter of `Duration`, you need to make a division. The same gist also applies to other getters such as `inMinutes`.

```
final d1 = const Duration(minutes: 15);
final d2 = const Duration(seconds: 548);

final d3 = (d1 < d2) ? d1 + d2 : d1 - d2;
```

Thanks to operator overloads you can easily compare objects and add/subtract values. If you want to represent a point in the time, use the `DateTime` class instead, which is also part of `dart:core`. Instances are generally created with one of the following constructors:

```
final d1 = DateTime.now();
final d2 = DateTime(1997, 5, 20); // May 20, 1997
final d3 = DateTime.utc(1997, DateTime.may, 20); // uses the UTC timezone
final d4 = DateTime.parse("1997-05-20 15:29:10Z");
```

Unless the `utc()` named constructor is used, the object is tied to the current device's date and time. Use a `DateTime` in conjunction with the "intl" package³ to localize date and times. As always, be sure to check out the official documentation for a complete overview of the numerous

³See chapter 13

Appendix A. Dart Appendix

available getters ⁴.

```
final today = new DateTime.now();
final tomorrow = today.add(const Duration(day: 1));
```

A `DateTime` object doesn't change once created and has no operator overloads. Use the `Stopwatch` class if you're looking for a precise tool to measure the elapsed time between one or more calls. The two main methods you'll almost always use are `start()` and `stop()`:

```
final sw = Stopwatch();
print("Startng...");

sw.start();
await executeSomething();
sw.stop();

print("Elapsed = ${sw.elapsedMilliseconds}");
print("Elapsed = ${sw.isRunning}"); // false
```

Enclose the parts of code you want to "benchmark" between `start()` and `stop()` to retrieve the elapsed milliseconds or microseconds. Note that `reset()` doesn't start or stop the timer: it just sets the counter back to 0.

```
sw.start();
await Future.delayed(const Duration(seconds: 2));
sw.reset();
await Future.delayed(const Duration(seconds: 1));
sw.stop();
print("Elapsed = ${sw.elapsedMilliseconds}"); // Elapsed = 1000

sw.start();
await Future.delayed(const Duration(seconds: 2));
await Future.delayed(const Duration(seconds: 1));
sw.stop();
print("Elapsed = ${sw.elapsedMilliseconds}"); // Elapsed = 3000
```

Be sure to call `reset()` whenever you want to reset the timer. If you wish to convert the results in other time units, just wrap the total microseconds (or milliseconds) in a `Duration` object.

```
// Measure
```

⁴<https://api.dart.dev/stable/2.8.3/dart-core/DateTime-class.html>

```
sw.start();
await Future.delayed(const Duration(
  seconds: 2,
  milliseconds: 75
));
sw.stop();

// Wrap the time in a convenient object
final duration = Duration(
  microseconds: sw.elapsedMicroseconds
);
print(duration);
```

A.3 Obfuscating Dart code

After you've published a new project to the official stores, someone could download your Flutter app's binary and use some tools to see the internals (including assets and source code). This process is always doable and it's called **reverse engineering**. You can use code obfuscation to make the binary harder for humans to understand.

- ❶ Note that code obfuscation does **not** encrypt your Dart files: it just makes them harder for a human to read. In this way, the attacker will have harder times trying to figure out the actual logic but data is still visible.

In order to obfuscate a Flutter app, you need to make a build in **release** mode and append the `--obfuscate` flag at the end of the command. In addition, there should be `--split-debug-info` to tell Flutter where output files has to be written.

```
flutter build appbundle --obfuscate --split-debug-info=/path/to/a/folder/
flutter build apk --obfuscate --split-debug-info=/path/to/a/folder/
flutter build ios --obfuscate --split-debug-info=/path/to/a/folder/
```

Files generated at the path specified by `--split-debug-info` are needed if you want to de-obfuscate the binary in a second moment. Keep them in a secure place. We strongly recommend you to obfuscate your binary before publishing it to the official stores.

Appendix A. Dart Appendix

- ℹ The usage of `--split-debug-info` could drastically **reduce** the code size so you should really use it. It also works without `--obfuscate` but you'd better go for obfuscation + splitting.

Code obfuscation doesn't work for web apps simply because there's no need for it (web apps are **minified**, not obfuscated).

B | Flutter Appendix

B.1 Riverpod

Riverpod is a new state management library created by Rémi Rousselet, the technical reviewer of this book, which tries to fix some common problems affecting his *provider* package. It's basically "provider on steroids" that doesn't depend on Flutter and has many interesting features. Before describing it, there are a few points to clarify:

1. At the time of writing this book, Riverpod is a **beta** rewrite of Provider. As such, you can explore and test the library making demo projects but be aware that it's not ready yet to be used for production. Currently, the latest Riverpod version is *0.8.0*.
2. Provider is built on top of `InheritedWidget` to make it easier to use. Riverpod instead is a complete rewrite of `InheritedWidget` from scratch.
3. Riverpod won't be merged with Provider because they have core differences (see the above point).
4. Riverpod **could** replace Provider in the future but it's not sure.
5. If Riverpod will be proven to be a better alternative to Provider, then it might be a replacement. Until that time, if it will ever come, continue using Provider which is stable and highly popular.

As you may have guessed, at the moment Provider should still be your primary choice: Riverpod is currently a beta project that requires more testing and validations by the community. It's born to inherit all the benefits of Provider plus more:

1. Riverpod doesn't depend on `InheritedWidget` so Flutter is not a requirement anymore: this library can also be used with Dart!
2. In general, you can just use a `ProviderScope` as root widget (see the example below)

and you won't have to worry about runtime exceptions. No more surprises caused by a `Provider<T>` not being located at a certain level of the widget tree. It's a **compile-safe** solution.

3. With Riverpod you can have multiple providers of the same type. States can be disposed when not needed anymore.
4. Last but not least... "riverpod" is the anagram of "provider"!

Riverpod comes in three flavors: `riverpod` (Dart), `flutter_riverpod` (Flutter) or `hooks_riverpod` (Flutter + the `hooks` package). For Flutter applications, if you don't use the `hooks`¹ package, just go for `flutter_riverpod`.

B.1.1 Usage

In Riverpod, a "**provider**" is the same concept you've been used to see up to now: it exposes a value that can be shared by one or more widgets in the subtree. We're now showing how to create the simple famous "Flutter counter app", which just does a `+1` each time a button is pressed. Let's start with the basics:

```
void main() {
  runApp(
    const ProviderScope(
      child: CounterApp(),
    ),
  );
}
```

You can decide to not place `ProviderScope` right after `runApp()` but then, be aware it has to be put one level above the values you're trying to access. The `ProviderScope` class enables Riverpod for the entire project and it should really be placed at the root of the tree. Providers can be declared as global variables but since we don't really like them, an equivalent but better looking solution is the following:

```
// Create this in a file called lib/providers/counter.dart
abstract class CounterProvider {
  static final provider = StateProvider<int>((ref) => 0);
}
```

¹See 21.3.1 to read about "Flutter Hooks"

Appendix B. Flutter Appendix

We prefer creating a dedicated class called `CounterProvider` which encloses a `static` reference of the provider our app is going to use. We also recommend creating a `providers/` folder with all the providers you use (one per file). A `StateProvider<T>` is a provider able to react to state changes, similarly to what a `ChangeNotifierProvider` does. We now have two ways to read the value:

1. A `ConsumerWidget` is basically a `StatelessWidget` with the ability to listen to changes on providers. Under the hood, it extends `StatefulWidget` so it can be put in the widget tree as usual.

```
class CounterApp extends ConsumerWidget {
  const CounterApp();

  void _buttonPressed(BuildContext context) { ... }

  @override
  Widget build(BuildContext context, ScopedReader watch) {
    final value = watch(CounterProvider.provider).state;

    return Scaffold(
      body: Center(
        child: Text("$value"),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () => _buttonPressed(context),
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

Rather than extending `StatelessWidget`, we use `ConsumerWidget` which exposes a very useful `watch` parameter. It's used to trigger a rebuild whenever the state of the listened provider changes (so the `Scaffold` will be rebuilt along with its children).

2. Alternatively, you can use the classic `Consumer` widget inside a `StatelessWidget` to obtain the same result. However, in this case you can optimize performance by rebuilding only widgets that actually depend on a provider. It's a better approach.

```
class CounterApp extends StatelessWidget {
```

```
const CounterApp();

void _buttonPressed(BuildContext context) { ... }

@Override
Widget build(BuildContext context) {
    return Scaffold(
        body: Center(
            child: Consumer(
                builder: (context, watch, child) {
                    final count = watch(CounterProvider.provider);
                    return Text('${count.state}');
                }
            ),
            floatingActionButton: FloatingActionButton(
                onPressed: () => _buttonPressed(context),
                child: const Icon(Icons.add),
            ),
        );
    }
}
```

In this case, **only** the `Text` widget is rebuilt and not only the entire subtree. In case you had a nested widget, use the `child` parameter to "cache" what doesn't depend on a provider.

```
Center(
    child: Consumer(
        builder: (context, watch, child) {
            final count = watch(CounterProvider.provider).state;
            return Column(
                children: [
                    child,
                    Text('$count')
                ]
            );
        },
        child: Container(...),
    ),
)
```

Appendix B. Flutter Appendix

) ,

In this way, the `Container` (along with its children) is cached. Changes on the state will only affect `Column` and `Text`.

Both ways are fine: you have more control on rebuilds (and thus performance) with `Consumer` so it's generally a better choice but it reduces the readability of the code. We still need to implement the actual increasing of the counter, which happens using `read()`:

```
void _buttonPressed(BuildContext context) =>
    context.read(CounterProvider.provider).state++;
```

Riverpod's `read()` is the equivalent of Provider's `read<T>()`: it simply reads the value of a provider without listening to it. When outside of the widget tree, like in this case, it's the only way you have to access the state of a provider. The library has then two ways to access the state:

- use `watch()` inside a `ConsumerWidget` or `Consumer` to listen to changes. Rebuilds will happen whenever the state changes. It can be used only inside the widget tree.
- use `read()` when you need to simply read the state of a provider without listening to it. It has to be used when accessing the state of a provider outside of the widget tree.

Note that `context.read()` can be used both inside and outside of the widget tree. When you need to listen to changes, consider using `Consumer` to avoid unnecessary rebuilds. Here's the kind of providers you can use:

- `Provider<T>`. It exposes a read-only value. It's the most simple kind of provider, commonly used as a "cache" to share data (model classes) among widgets. You could use it in the following way:

```
abstract class UserDataProvider {
    static final provider = Provider<UserData>((ref) => UserData());
}

class UserData {
    var name = '';
    var surname = '';
    var age = 0;
}
```

Any UI widget can get access to data about the user regardless their position in the tree. We will see later how the `ref` parameter can be useful.

- **StateProvider<T>**. It exposes a value to the outside so that it can be modified and listened/read from. You've already seen an example in the above "counter" app example where the exposed value can be read or listened: it is something similar to the `ChangeNotifierProvider` of the provider package.
- **FutureProvider<T>**. Kind of provider that asynchronously creates a value: it can be seen as a combination of `Provider<T>` and `FutureBuilder<T>`. For example, let's see how you can easily make a GET request and **synchronously** return data to the UI.

```
abstract class RequestProvider {  
    static final provider = FutureProvider<UserInfo>((ref) async {  
        final url = "https://website.com/api/json/something";  
        final response = await dio.get<String>(url);  
  
        return UserInfo.fromJson(response.data);  
    });  
}  
  
// Exposes some getters such as nickname, age and birthday  
class UserInfo { ... }
```

There's no need for the classic `FutureBuilder<T>` setup because Riverpod will take care of everything for you. The syntax couldn't be more expressive than this:

```
@override  
Widget build(BuildContext context, ScopedReader watch) {  
    final AsyncValue<UserInfo> jsonString =  
        watch(RequestProvider.provider).state;  
  
    return jsonString.when(  
        loading: () => const CircularProgressIndicator(),  
        error: (err, stackTrace) => const SomeErrorWidget(),  
        data: (userInfo) => SomeInfoWidget(  
            nickname: userInfo.nickname,  
            age: userInfo.age,  
            birthday: userInfo.birthday,  
        ),  
    );  
}
```

Appendix B. Flutter Appendix

A `FutureProvider<T>` returns a very useful object called `AsyncValue<T>` which is used to safely handle asynchronous data. With this setup, the UI is automatically rebuild when the data is ready.

- `StreamProvider<T>`. It works exactly as a `FutureProvider<T>` with the only difference that the created value is a `Stream<T>` rather than a `Future<T>`. In particular, when you have one or more resources to be disposed, be sure to use the `autoDispose()` named constructor:

```
abstract class ExampleProvider {
    static final provider =
        StreamProvider.autoDispose<String>((ref) async* {
            final source = SomeStreamSource();

            // You can also perform cleanup operations
            ref.onDispose(() => source.close());

            await for (final event in source.stream) {
                yield "$event";
            }
        });
}
```

The `ref` parameter is a reference to the current `StreamProvider<T>` object and it can be used to dispose resources via `onDispose()`. The provider returns a `AsyncValue<T>` so the syntax is the same as before:

```
Consumer(
    builder: (context, watch, _) {
        final msgStream = watch(ExampleProvider.provider).state;

        return msgStream.when(
            loading: () { ... },
            error: (err, stackTrace) { ... },
            data: (eventData) { ... },
        );
    }
);
```

You can also use `FutureProvider.autoDispose()` to have more control on asynchronous re-

quests. In the official documentation ² there's an article showing how easy it is to cancel HTTP requests when no-longer needed. Even if *Riverpod* is a complete rewrite of the *Provider* package, there are still many similarities in the names and usages.

B.1.2 Combining providers

In Riverpod, the constructor of any provider always exposes a `ref` parameter. Very simply, it's just a reference to the "current" provider object on which you're working. You've already seen it in action:

```
StreamProvider.autoDispose<String>((ref) async* {
  ref.onDispose(() { ... });
});
```

In this case, `ref` is a reference to the current `StreamProvider<T>` instance in which you're working. Other than being useful to call `onDispose()`, it's also great when it comes to combining two or more providers (potentially, infinite providers!):

```
abstract class YearProvider {
  static final provider = StateProvider<int>(_ => 2020);
}

abstract class EventProvider {
  static final provider = FutureProvider<List<Event>>((ref) async {
    final year = ref.watch(YearProvider.provider);
    return downloadEvents(year);
});
```

In this example, `EventProvider` depends on another provider because it's listening to changes. In particular, `ref` is a reference to the current `EventProvider` object which makes possible watching (or also reading) another source.

```
// Don't listen, just read
final year = ref.read(YearProvider.provider);
```

This would have worked anyway. In the *Provider* package you cannot declare two providers of the same type but in Riverpod there isn't this restriction. There can be two providers exposing a variable of the same type (even the same object) without any problem:

²https://riverpod.dev/docs/concepts/modifiers/auto_dispose#example-cancelling-http-requests-when-no-longer-used

Appendix B. Flutter Appendix

- **Provider.** The following code compiles but you'll get a runtime error because there are two providers holding the same type. The `Consumer<T>` doesn't know which provider has to be taken into account.

```
MultiProvider(
  providers: [
    Provider<MyModel>(create: (_) => const MyModel()),
    Provider<MyModel>(create: (_) => const MyModel()),
  ],
  child: Consumer<MyModel>(
    builder: (context, model, _) => Text("$_model"),
  ),
);
)
```

- **Riverpod.** No runtime errors because there's no dependency on the widget tree. We're using abstract classes for providers so Flutter doesn't care about the types of the models.

```
abstract class AlbertoProvider {
  static final provider = StateProvider<String>(_ => "Alberto");
}

abstract class RemiProvider {
  static final provider = StateProvider<String>(_ => "Rémi");
}

// somewhere in the widget tree
Consumer(
  builder: (context, watch, _) {
    final alberto = watch(AlbertoProvider.provider).state;
    final remi = watch(RemiProvider.provider).state;

    return Text("$_alberto $_remi");
  }
)
```

Riverpod doesn't rely on the widget tree so errors like "*Could not find the correct provider above...*" can never happen.

B.1.3 Testing

Testing an application is very straightforward and you don't even need to use particular libraries. Riverpod can be tested in pure Dart applications (Flutter isn't a dependency) along with the usual *test* and *mockito* packages. Here's how you'd setup tests for the counter app.

```
// counter_provider_test.dart
abstract class CounterProvider {
    static final provider = StateProvider<int>((_) => 0);
}

class Listener extends Mock {
    void call(int value);
}
```

For simplicity, we directly create the `CounterProvider` in the same file but actually it should be placed somewhere else, inside a good folders structure. `Listener` is a convenient callable class³ we've created to keep tracks of signals sent by the provider to listeners.

```
// still inside counter_provider_test.dart
void main() {
    test('Notifies listeners when the state changes', () {
        final container = ProviderContainer();
        final listener = Listener();

        // "Catch" changes emitted from the provider and forward
        // them to our listener
        CounterProvider.provider.watchOwner(container,
            (value) => listener(value.state)
        );

        // Make sure the initial state is 0
        verify(listener(0)).called(1);
        verifyNoMoreInteractions(listener);

        // Increment by 1
        container.read(counterProvider).state++;
    });
}
```

³See: 4.5.1 callable classes

Appendix B. Flutter Appendix

```
// Make sure the state is 1
verify(listener(1)).called(1);
verifyNoMoreInteractions(listener);
});
}
```

In Flutter, `ProviderContainer` is internally used by Riverpod so you don't need to care about it. It's a very important class used to store the `state` of a provider. The actual testing is done with `verify()` from the `mockito` package.

1. We make sure the initial state is really 0.

```
verify(listener(0)).called(1);
```

The `called(1)` call makes sure that the method has been called exactly 1 time

2. We make sure that nothing more has happened with `verifyNoMoreInteractions()`.

When it comes to widget testing, you just need to wrap the subtree in a `ProviderScope` and you're already good to go. There's nothing more to setup because it's really a "normal" widget test process.

```
void main() {
    testWidgets('Counter increment', (tester) async {
        await tester.pumpWidget(
            const ProviderScope(
                child: CounterApp()
            )
        );
        // Counter must start at 0
        expect(find.text('0'), findsOneWidget);

        // Tap the FAB and trigger a rebuild
        await tester.tap(find.byIcon(Icons.add));
        await tester.pump();

        // Counter now has to be 1
        expect(find.text('1'), findsOneWidget);
    });
}
```

Be sure to have a look at the official documentation ⁴. It's rich of examples and it is guaranteed to be a growing source of information about this new state management library. Be sure to keep an eye on their official website to stay updated on the latest changes.

B.2 Local databases

It's very common nowadays deferring data storage and processing to the server so that the mobile device doesn't have any heavy computing to execute. In particular, databases are hosted on a server which exposes the contents to the outside with a REST API or an SDK (like it happens with Firebase). This is quite useful:

1. the user can access data from different devices (so no hardware dependencies);
2. no internal memory occupied by databases or other kind of files;
3. no I/O disk operations since everything happens via internet.

You might have guessed that relying on a local data storage is not a good idea. What is absolutely fine instead (and it's also very common) is caching some data on the device using a database. It may be useful for:

- storing user's preferences (settings, values, flags...);
- caching data so that certain features of your app can be used even when there's no internet connection.

We've already seen in chapter 20 the *shared_preferences* package but it's only good for simple key-value data. If you're looking for a better way to persist information on the device, we recommend the usage of one of the following packages.

B.2.1 Hive (NoSQL)

Hive ⁵ is a fast, lightweight and secure NoSQL database for both Dart and Flutter with built-in encryption. It can be initialized whenever you want but you should really do it at startup, at the very first line of the `main()` method:

```
// For Dart apps
void main() async {
    Hive.init("/valid/path/on/filesystem/");
```

⁴<https://riverpod.dev>

⁵<https://pub.dev/packages/hive>

Appendix B. Flutter Appendix

```

    await Hive.openBox("my_data");
}

// For Flutter apps
void main() async {
    await Hive.initFlutter();
    await Hive.openBox("my_data");

    runApp(const MyFlutterApp());
}

```

Data are stored inside "boxes" which are automatically encrypted. You could, for example, create two different boxes called `"app_settings"` and `"cached_contents"` to store different values of your app. Once the box has been opened, reading and writing data is straightforward:

```

// Get a reference to the box containing the data
final data = Hive.box("my_data");

// Store a value
data.put("name", "Alberto");

// Retrieve a value
final name = data.get<String>("name", defaultValue: "-");

```

Don't try to open a box more than once otherwise an exception will be thrown. This might be another reason to open boxes as soon as possible when the app launches (you won't have to call `bool isBoxOpen(String name)` repeatedly). It would be better if you created a separated file only containing the database's keys, such as:

```

// 'hive_keys.dart'
class HiveKeys {
    const HiveKeys._();

    static const name = "name";
    static const password = "passwd";
}

```

In this way you can later use `HiveKeys.name` rather than a plain `"name"` string, which is less error-prone and better in terms of maintenance. Of course you can also store complex objects, not only primitive types such as `double`, `DateTime`, lists and maps. This file represents data

about a particular device configuration:

```
@HiveType()  
class Settings extends HiveObject {  
    // Every field has to be annotated with 'HiveField' and an integer  
    // ascending value for subsequent members (0, 1, 2...)  
    @HiveField(0)  
    String deviceName;  
}  
  
class SettingsAdapter extends TypeAdapter<Settings> {  
    @override  
    final typeId = 0;  
  
    @override  
    Settings read(BinaryReader reader) {  
        return Settings()  
            ..deviceName = reader.read();  
    }  
  
    @override  
    void write(BinaryWriter writer, Settings settings) {  
        writer.write(settings.deviceName);  
    }  
}
```

In this example, `Settings` is the model class we're going to store and it has to be marked with the `HiveType` annotation. The adapter can be automatically generated using `hive_generator`, which is recommended, but for this simple example we've written it by hand.

- ❶ The documentation is very detailed and rich of interactive examples, so you should definitely have a look at it for more info: <https://docs.hivedb.dev..>

Appendix B. Flutter Appendix

B.2.2 SQLite (SQL)

There's a very popular SQLite package called "sqflite"⁶, which is also covered in the official Flutter documentation. SQL databases cannot run on mobile devices because there is no server installed; for this reason, `.sqlite` files are an alternative way to go. The `sqflite` package can easily be initialized in this way:

```
import 'package:path/path.dart' as path;
import 'package:sqflite/sqflite.dart';

void main() async {
    WidgetsFlutterBinding.ensureInitialized();

    // Automatically get the database folder according to the current OS
    final dbPath = await getDatabasesPath();

    // Use 'join' to correctly build the path to the database file
    final dbName = path.join(dbPath, 'settings.db');
    final database = openDatabase(dbName);

    // Maybe expose the reference to the database using Provider
    runApp(const FlutterApp());
}
```

`path` is the cross-platform way to work with filesystem's paths. The above setup is only required if the database is going to be used across the entire app. If you plan to use it only in particular areas of your app (like storing settings), avoid working in `main()` and prefer creating a `Future<T>` which can be simply awaited by a `FutureBuilder<T>`:

```
Future<Database> initialize() async {
    final dbPath = await getDatabasesPath();
    final dbName = path.join(dbPath, 'settings.db');
    return openDatabase(dbName);
}
```

In the `openDatabase()` method there's also the possibility to specify the `version` using an integer value. It's generally used together with `onCreate`, `onUpgrade` or `onDowngrade` callbacks to specify actions to perform when a migration of the database is required.

⁶<https://pub.dev/packages/sqflite>

```

openDatabase(dbName,
    // Called if the database doesn't exist prior to calling 'openDatabase'
    onCreate: (db, version) {...},
    // Called if there is no 'onCreate' callback OR 'version' has changed
    // and it's higher than the previous value
    onUpgrade: (db, version) {...},
    // Called when 'version' is lower than the previous value
    onDowngrade: (db, version) {...},
    version: 1
);

```

In practice, you should always start with `version: 1` and define (if needed) an `onCreate` callback to execute initialization queries on the database for the first time it's created. Use `onUpgrade` when a new version is available to make a migration to the new configuration. Remember that changing the version number triggers the upgrade or downgrade callbacks.

```

openDatabase(dbName,
    onCreate: (db, version) {
        return db.execute(
            "CREATE TABLE something (id INTEGER PRIMARY KEY, descr TEXT)",
        );
    }
);

```

Queries should be placed in a dedicated file: we put a `CREATE` statement directly in the method just to keep the example easy. Rather than manually writing queries, the library exposes a series of useful methods:

- `INSERT`: returns the internal id of the record.

```

final id = await db.insert('table_name', {
    'id': '1',
    'descr': 'custom text',
});

```

- `UPDATE`: the jolly (?) placeholder can be used but it requires a non-empty arguments list. It returns the number of updated rows.

```

final count = await db.update('table_name',
    {'descr': 'some text here'},
    where: 'id = ?'

```

Appendix B. Flutter Appendix

```
    whereArgs: ['1'],
);
```

- DELETE: returns the number of deleted records.

```
final count = await db.delete('table_name',
    where: 'id = ?',
    whereArgs: ['1']
);
```

- SELECT: returns a list of maps.

```
final list = db.query('table_name', columns: ['name', 'type']);
```

If you used one or more placeholders in a query, the arguments are required otherwise the engine wouldn't know how to handle the symbol. Visit the official documentation to learn more about this package, which also supports transactions and many other common SQL commands (such as `LIKE` in combination with wildcards).

B.3 Initializing data at startup

We have seen multiple times that certain packages, such as `hive` or `hydrated_bloc`, require an immediate initialization (even before calling `runApp()`). The `main()` is a good place to perform this kind of setup but it might become a problem:

```
void main() async {
    // Hydrated bloc
    HydratedBloc.storage = await HydratedStorage.build();

    // Firebase
    await Firebase.initializeApp();
    await FirebaseFirestore.instance.clearPersistence();
    await FirebaseAdMob.instance.initialize(appId: "...");
    await FirebaseMessaging().requestNotificationPermissions();

    // Something else
    await LibraryA.initialize();
    await LibraryB.initialize();

    // Finally, start the app
}
```

```
        runApp(const MyFlutterApp());  
    }  
}
```

As you can see, in some cases there might be too much initialization to do before calling `runApp()`. Waiting for the setup of a single component could be fine but when there is a series of `await` (like above) the situation is different. The approach must change: start the app as soon as possible and wait for initialization directly in the UI.

1. Create a model class used to gather all the startup methods required by your app. It should really return a `Future<T>` so that it can be awaited in the home page as we're going to show in a moment:

```
// app_startup.dart  
abstract class AppStartup {  
    static Future<void> setup() async {  
        // Hydrated bloc  
        BlocSupervisor.delegate = await HydratedBlocDelegate.build();  
  
        // Firebase  
        await Firebase.initializeApp();  
        await FirebaseFirestore.instance.clearPersistence();  
        await FirebaseAdMob.instance.initialize(appId: "...");  
        await FirebaseMessaging().requestNotificationPermissions();  
  
        // Something else  
        await LibraryA.initialize();  
        await LibraryB.initialize();  
    }  
}
```

Since we don't like global functions (and global variables) at all, we've created `AppStartup` which just exposes the `setup()` method. The class is `abstract` because it doesn't need to be instantiated (you could have also created a regular class with a private constructor).

2. In your app's home widget, use the `FutureBuilder<T>` pattern to `await` for the completion of the initialization. As usual, `startupFuture` is a variable initialized inside the `initState` method as we've explained in detail in chapter 17.

```
FutureBuilder<void>(  
    future: startupFuture,  
    builder: (context, snapshot) {
```

Appendix B. Flutter Appendix

```
        if (snapshot.connectionState == ConnectionState.done) {
            return const HomePage();
        }

        return const SplashScreen();
    }
)
```

In this way, a splash screen appears (or anything else) while packages and plugins are executing their initialization phases. Once ready, the actual contents of the home page appear.

This is a nice way to gracefully initialize data in your app rather than using a bunch of asynchronous calls directly inside the `main()` method. We recommend you to follow this guideline:

- If you have only one or two initialization methods to call (and they execute quickly), there's no need for a splash screen. Just put them before `runApp()` and you'll be fine.
- If you have a lot of initialization methods to call (and they require time), there's the need for a splash screen. It lets the user know that something is loading but the app is still responsive! Use this approach if you also need to use an internet connection on startup.

In order to decide if initialization methods are "slow" or "quick" run many tests and measure, on average, how much time they take. It's up to you deciding if they are fast or if there's the need for a splash. However, if you want to stay safe, always go for a graceful initialization with a `FutureBuilder<T>`.

B.4 Accessibility

An application is **accessible** when it can be used by a broad range of people without creating barriers for users of any age. For example, someone might need the help of a screen reader to interact with the device or simply a stronger color contrast to be more comfortable. Flutter has some built-in facilities to increase your app's accessibility level:

- Screen readers for Android and iOS are very good at understanding what's being displayed in the UI but they aren't always accurate. You can help them making the user experience even more pleasant with the `Semantic`⁷ widget:

⁷<https://api.flutter.dev/flutter/widgets/Semantics-class.html>

```
const Semantics.fromProperties(  
    child: SomeButtonWidget(  
        child: Text("Play"))  
,  
    properties: SemanticsProperties(  
        label: "Play music button",  
        hint: "Tap to play music",  
        button: true,  
        enabled: true,  
        value: "Play"  
)  
)
```

The `Semantics` widget is used to describe what a piece of UI means. It's very helpful for screen readers because they get a more accurate idea of what's being displayed. In the above example, you can see how we've used `Semantics` to describe a "Play" button widget.

```
// No constant constructor for the default 'Semantics()' '  
Semantics(  
    child: SomeButtonWidget(  
        child: Text("Play"))  
,  
    label: "Play music button",  
)
```

You should create new instances using the `fromProperties()` named constructor since it can be constant. Regardless, in both cases there are more than 50 parameters you can set so be sure to visit the documentation for a detailed overview.

- A natural physical factor is the loss of focus by the eyes when a human gets older so smaller fonts become harder to read. Flutter automatically determines the size of the text according to the hardware of the device but you can manually change the setting:

```
// By default this is 1.0  
Text("Some text",  
    textScaleFactor: 1.2,  
)
```

In this way, the text is 120% of the normal size. It might be a good idea, for example, using `hydrated_bloc` to store user's settings and among them defining a value for the `textScaleFactor`.

Appendix B. Flutter Appendix

- Don't make "tappable" items too small as they might become a difficult target. The minimum recommended size is 48x48⁸ but you could raise this lower limit to 56x56 to be even more safe.
- Make sure that buttons always "do something" and in general, any widget that requires interaction should trigger a visual response. For example, if a button is disabled it shouldn't simply ignore the finger tap. Instead, open a dialog or notify the user about why the button is disabled.

```
// Don't do this
IgnorePointer(
    ignoring: true,
    child: RaisedButton(...))
)
```

Whereas `GestureDetector` is used to add tap callbacks to widgets, `IgnorePointer` does the opposite: it disables any kind of interaction on the children. In the above example, the button cannot be pressed and it doesn't provide animations or visual feedbacks (looks like it's "freezed"). Avoid this pattern and prefer something else, like this:

```
RaisedButton(
    child: const Text("Go!"),
    onPressed: () => _showSnackBar("Cannot press this because..."),
)
```

The button can still be pressed, which is what the user expects, but you give a reason of why it doesn't perform a certain action.

- Background and foreground should be easily distinguishable thanks to efficient color contrasts. For example, you shouldn't use white text on a light grey background since the contrast is not strong enough to clearly recognize the text. Eyes shouldn't be stressed to much.

Other than the usual three trees maintained by Flutter, the usage of `Semantics` creates a fourth tree called "semantic tree" and this is used by screen readers. These tools traverse it to get information about widgets but the accuracy depends on you: the more parameters you define in `Semantics` the better the help to the users will be.

⁸<https://flutter.dev/docs/development/accessibility-and-localization/accessibility#accessibility-release-checklist>

B.5 The Flutter community

If you want to get involved in the community, along with millions of developers around the world sharing your passion for Flutter, we've good news for you. You can attend conferences, watch video tutorials, follow online courses and stay in touch with many developers via Discord, Twitter or Reddit.

- **Flutter Europe** - <https://fluttereurope.dev>

It's the biggest Flutter conference in Europe which took place for the first time in Poland, at the end of January 2020. People from all over the world gather together to attend the talks made by people from the Google team and open source maintainers.



In case you missed the event, check out their official YouTube channel to see the replays. The various talks are held during the day and in the evenings there usually is a party.

- **Flutter Community** - <https://medium.com/flutter-community/>

The "Flutter Community" is an organization aimed at providing a central place for community made Flutter packages and content. They write a lot of articles on Medium and develop popular Flutter packages for <https://pub.dev>⁹, such as:

1. RxDart, which makes Dart's streams and controllers even better;
2. get_it, a simple service locator for Dart and Flutter;
3. flutter_webview_plugin, a native way to communicate with WebView;
4. sticky_headers, headers for scrollable contents;
5. workmanager, to schedule background tasks on Android and iOS;
6. and much more...

⁹<https://pub.dev/publishers/fluttercommunity.dev/packages>

Appendix B. Flutter Appendix

You can get in touch with many passionate developers also on **Reddit**, at r/FlutterDev, and via **Discord**, which is the official channel of the Reddit page.



- **Code with Andrea** - <https://codewithandrea.com>

Code with Andrea is a website about learning Flutter by Andrea Bizzotto. This includes a growing collection of over 50 weekly video tutorials and articles; courses are also on offer for students looking for a more structured approach.



The first part of our book, about Dart, is used by courses at codewithandrea.com to introduce the language to students! There's also a YouTube channel with some free courses, such as the ones about REST APIs, state management and Flutter widgets.

- **Reso Coder** - <https://resocoder.com>

Learn Flutter app development with tutorials built to make the new concepts stick thanks to the Reso Coder YouTube channel. It's a very popular source of informative contents from the Dart / Flutter world which provides both video tutorials and written articles.



The channel is very popular for its TDD and DDD complete courses: a series of 20+ videos about Flutter best practices following specific development designs.

- **Fireship** - <https://fireship.io>

Fireship is a project created by Jeff Delaney which offers a series of videos and courses about Flutter, ranging from state management to best practices and UI tutorials. The main strength of Fireship are courses and tutorials about Firebase, with a notable series called *Flutter Firebase - The Full Course*.



If you're looking for a more direct and "daily" approach to the community, you can join the Discord server or reach the official Reddit channel. You can find every link in the "Community" section of the official Flutter website¹⁰. In particular, we recommend you to keep an eye on:

- the official Flutter YouTube Channel. It's full of interesting contents such as the "Boring show" and the "Widget of the week" series;
- <https://twitter.com/FlutterDev> and https://twitter.com/dart_lang to stay updated on the

¹⁰<https://flutter.dev/community>

Appendix B. Flutter Appendix

latest changes and news on the Flutter and Dart world;

- the official GitHub repositories of both Dart and Flutter in which you can find spoilers about the upcoming new features, join the conversations and contributing by opening new issues.

B.6 Flutter SDK management

As always, the official documentation is the best place to get an updated step-by-step guide ¹¹ to install Flutter in your operating system (Windows, macOS, Linux and Chrome OS). We strongly recommend you the usage of Android Studio or VS Code since they have an official Flutter plugin.

- ⓘ** As of Flutter 1.21, the Flutter SDK includes the full Dart SDK so you don't need to install them separately. This is very convenient because everything is centralized and managed by a single command line tool (called `flutter`).

A fresh new Flutter install downloads files from the **stable** source channel, which is the safest one in terms of stability. Your production software should rely on the stable version of the Flutter and Dart SDK. However, you can also download the SDKs from other channels (sorted from the less to the most stable):

1. **master**. The absolutely latest cutting edge build you can get. It generally works but sometimes it might break. Use it if you want to play with the latest features and fixes but absolutely don't rely on this channel for stability.
2. **dev**. The latest build with a complete test coverage. It's a bit more stable than "master" because everything is completely tested so there's high chance that everything will work well. You shouldn't rely your production code on this channel unless you heavily test your build (but consider if it's worth the effort).
3. **beta**. Once the build has arrived at this stage, it will land to "stable" in a couple of weeks. Even if not completely reliable, contents on this channel are quite stable. If your production app cannot wait for a feature on the "stable" channel, you can switch to "beta" and use it.
4. **stable**. This is the safest build of the engine as well as the recommended channel for production app releases. Very rarely there might be issues on this channel but the team comes to the rescue with hotfixes.

¹¹<https://flutter.dev/docs/get-started/install>

Even if the default Flutter installation is in sync with `stable`, you can easily change the channel to get another build. Run the following commands in sequence to ensure being on the latest build of the selected channel:

```
# replace <name> with 'master', 'dev', 'beta' or 'stable'  
$ flutter channel <name>  
$ flutter upgrade
```

Using the `flutter channel` command you can see in which channel you're currently on. If you want to upgrade your current Flutter installation with the latest version available, just use the `upgrade` command.

```
$ flutter upgrade
```

You should also make sure that all of your dependencies listed in the `pubspec.yaml` are always up to date. Rather than manually checking the versions at pub.dev, which can be very tedious if you have a lot of dependecies, just run this command:

```
$ flutter pub upgrade
```

It automatically updates all of your dependencies to the latest version. However, before running `pub upgrade` you should use the `pub outdated` command which tells you which packages can safely be upgraded without having versioning conflicts.

B.6.1 Web and Desktop

At the time of publishing this book (September 2020), Flutter for web is in the beta channel but the desktop support (Windows, macOS and Linux) is still in early alpha on the dev channel. Once they will be released in the stable channel, we will cover these new platforms in a future version of the book. However, you can still move to the beta channel and unlock the Flutter web preview:

```
$ flutter channel beta  
$ flutter upgrade  
$ flutter config --enable-web
```

Your Android and iOS apps will look exactly the **same** also on web and desktop, with very high performance. The developer really needs to write the code only once because the compiler will take care of porting Flutter natively to any platform. Here's a general overview of how the web support works:

1. There's no need for a browser plugin because Flutter generates web contents by simple using

Appendix B. Flutter Appendix

HTML, CSS and JavaScript. Rather than compiling to native ARM code, as it happens for mobile devices, Dart is compiled to JavaScript so that web browsers can understand it.

2. Thanks to Dart's optimized JavaScript compiler, the entire Flutter framework (including the core) along with your Dart source code are compiled into a single, minified source file that can be deployed to any web server.
3. Any project made for a mobile app can be ported to web. If you have worked with layout responsiveness in mind, there will be no extra work to do. A desktop is like a big horizontal tablet so your layout should adapt to various sizes.

If you want to try building your Flutter app on desktop as well, there's currently a bit more of setup to do. Being desktop support in early alpha, complex apps might not work as expected. Follow the latest update guide on the official Flutter website and then run the following commands before creating a new project:

```
$ flutter channel dev  
$ flutter upgrade  
$ flutter config --enable-macos-desktop  
$ flutter config --enable-linux-desktop  
$ flutter config --enable-windows-desktop
```

You can now create a new project with desktop support. At the time of publishing this book, macOS is the best supported OS since it has a lot of similarities with iOS. Linux is at a good stage as well but Windows still has a long way to go. Creating builds for different OSes, as always, can't be easier than this:

```
$ flutter build macos  
$ flutter build linux  
$ flutter build windows
```

The official documentation ¹² recommends to not release a desktop app until support for this mode lands to the stable channel. You can add desktop support to an existing Flutter project running this command from the root project directory:

```
$ flutter create .
```

Most of the pub packages are available for web but only a few support desktop (you can really count them by hand). Visit <https://flutter.dev/web> and <https://flutter.dev/desktop> to get the latest news about the progress the team is making on those platforms to make them land on the stable channel.

¹²<https://flutter.dev/desktop>

Index

Symbols

-obfuscate	724
-split-debug-info	724, 725
?	142, 146
.()	90
<T>	137
==	51, 125
=>	96
??	55, 102
?[]	48
[]	41, 45, 48
.....	366
.....	447

A

abstract	109
AccelerometerEvent	560
accelerometerEvents	562
add()	621
addPerson	435
AdMob	623
AdSense	623
AdSize	627, 631
AlertDialog	262, 547
allOf()	441
analysis_options.yaml	211
AngularDart	20
AnimatedAlign	388
AnimatedBuilder	399
AnimatedContainer	386

AnimatedCrossFade	391
AnimatedDefaultTextStyle	389
AnimatedIcon	392
AnimatedOpacity	388
AnimatedPadding	389
AnimatedPhysicalModel	390
AnimatedPositioned	390
AnimatedSize	391
AnimatedWidget	392, 393, 397
Animation library	385
AnimationBuilder	397
AnimationController	396, 402, 405
AnimationStatus	408
annotation	93
AOT	19, 20, 29
AppBar	257
AppLocalization	368, 370, 374
ARB file	377, 378
ARM	16, 25, 27
arrayContains	621
arrow syntax	62
ART	26
as	51
AsciiDecoder()	431
AspectRatio	521
assert	60
AssetBundle	503
AssetsAudioPlayer	523
async	160, 163, 165, 166, 172

async* 168
AsyncSnapshot<T> 477, 601, 602
AsyncValue<T> 733
attribute() 435
AuthResult 655
authStateChanges() 657
availableCameras() 556
await 160, 163–167, 172, 182, 183
await for 172

B

bang operator 48
BarcodeDetector 644
BaseOptions 482
Battery 568
BatteryState 568
benchmark/ 664
bind() 720
bindSecure() 720
BLoC 312, 313
BlocBuilder 291
BlocBuilder<B,S> 333
BlocBuilder<T, K> 318
BlocConsumer<B,S> 334
BlocListener<B,S> 322, 333
BlocObserver 323
BlocProvider<T> 316
blocTest() 447
BottomNavigationBar 578, 581
BoxConstraints 278
BoxDecoration 224, 225
BoxShadow 224
break 59
bridge 15, 16, 26–28
build_runner 425
BuildContext 217
ButtonBar 261
ByteData 504

C

CachedNetworkImage 509, 510

call() 64, 99, 100
called() 737
CameraController 556
CameraDescription 556
CameraPreview 559
cancel() 177
CanvasKit 243
cascade notation 75
case 57
cast<T>() 150
catch 130–132
catchError() 159, 160
ChangeNotifier 299
ChangeNotifierProvider 300, 301
checkConnectivity() 569
ChildIcon 595
Chip 563
clearPersistence() 619
code obfuscation 724
collection() 615
ColorTween 404
Column 220, 221
CommonFinders 450
Comparable<T> 127
compareTo() 127
completion() 443
ComponentElement 245
compute() 185
configure() 648
Connectivity 569, 570
ConnectivityResult 569
const 38, 90–94
const constructor 237
constant constructor 91
ConstrainedBox 536
constructor assignment 121
Consumer 729
Consumer<T> 306
ConsumerWidget 729
Container 223, 225

Index

- contains() 154
context.select() 312
context.watch<T>() 311
continue 59
copyWith() 102–104
counterText 540
covariant 107
createElement() 244
createRenderObject() 244
CrossFadeState 391
cubit 330
Cubit<T> 331
CupertinoActionSheet 594
CupertinoActionSheetAction 594
CupertinoAlertDialog 270
CupertinoApp 266, 341
CupertinoButton 271
CupertinoDatePicker 593
CupertinoDatePickerMode 594
CupertinoDialogAction 271
CupertinoLocalizations 366
CupertinoPageRoute 343
CupertinoPageRoute<T> 344
CupertinoPageScaffold 268
CupertinoSegmentedControl 595
CupertinoTabBar 268
CupertinoTabScaffold 268
CupertinoThemeData 287
currentStep 586
Curve 401
CurvedAnimation 402
CurvedAnimations 401
Curves 400
Curves.linear 387
Custom animation 385
CustomPaint 640, 641, 644

D
dart2js 20, 243
dart:convert 418
dart:ffi 249
DartPad 23, 29
DataCell 588
DataColumn 587
DataRow 588
DataTable 587
DataTableSource 590
DateFormat 365
DateTime 365, 722, 723
DDD 186
default 56
DefaultAssetBundle 503
DefaultTabController 582
defaultValue 426
delete() 621, 743
Dependency Injection 193
DeviceInfoPlugin 568
DevTools 29, 459
DI 193
diamond 139
diamonds 139
dio 481, 482
DIP 193
Directory 718
DismissDirection 546
Dismissible 546, 547
dispose 297
distanceBetween() 566
Divider 576
do while 59
DocumentSnapshot 619
download() 485, 490
Draggable<T> 551, 553
DragTarget<T> 553
drain() 171
Drawer 258, 573
DrawerHeader 575
DropdownButtonFormField<T> 542
DropdownMenuItem 542
DropdownMenuItem<T> 541

Duration 721, 722
DVM 19, 21
dynamic 35, 36, 123, 137–139, 146

E

Element 245
else 54, 55, 61
embedder 241
enableFlutterDriverExtension() 455
engine 242
ensureInitialized() 556
enterText() 457
enum 43, 44
equals() 51
Equatable 151, 319
Error 133, 135
event loop 179–183
every() 154
Exception 128, 132, 133, 135
exception 128
Expanded 284, 286
expect() 439, 447
explicitToJson 428
export 495, 663
extend 133
extends 106, 112–114, 139
extension 122
extension methods 121
external 144, 147, 149

F

FAB 256, 259
Face 636
factory 85, 120
FFI 249
FFI.DynamicLibrary 250
FFI.NativeFunction() 250
FieldValue.delete() 622
File 717, 718
File.readAsString() 717
 FileMode.append 718

FileMode.read 718
 FileMode.write 718
 FileMode.writeOnly 718
 final 37, 38, 61, 63, 64, 81, 82, 88, 90–93, 97
 finally 132
 find 675
 findAllElements() 432, 433
 findElements() 432, 433
 Finder 450
 findsNothing 450
 findsNWidget 450
 findsOneWidget 450
 findsWidgets 450
 Firebase.initializeApp(); 609
 FirebaseAdMob 627
 FirebaseAuthException 657
 FirebaseFirestore.instance 615
 FirebaseMessaging 646
 Firestore 611, 612
 FlatButton 261, 290
 Flutter Hooks 597
 flutter_driver 454
 flutter_test 449
 FlutterFire 606
 FlutterLogo 393
 fold() 155
 followedBy() 154
 fontFamily 287
 for 58, 143, 146
 for-in 61, 66
 forEach 150
 Form 531
 FormData 488
 forward() 396
 frameBuilder 508
 Function 63, 64
 Future.delayed() 168
 Future.wait<T>() 160
 Future<T> 158, 159, 162, 163, 166

Index

Future<T>.delayed() 160
 Future<T>.error() 160
 Future<T>.sync() 161
 Future<T>.value() 159, 161, 166
 FutureBuilder<T> 476–479
 FutureProvider<T> 309, 732

G

generator 167, 171
 GestureDetector 413, 543, 544
 get 97, 106
 getPositionStream() 564
 getRow() 592
 getter 95, 151
 getText() 457
 GlobalKey 234
 GlobalKey<FormState> 531
 GlobalKey<NavigatorState>() 359
 google-services.json 608
 google_fonts 209, 210, 218
 GoogleServices-Info.plist 609
 group 440
 group() 441
 GyroscopeEvents 560

H

hasData 477
 hasError 477
 hashCode 150–152
 HashMap<K,V> 149
 Hero 411, 413
 hide 78
 Hive 738
 home 347
 Hook<T> 602
 hot reload 210, 211
 http.Client 479
 HTTPMock 445
 HttpRequest 720
 HttpRequest<T> 474
 HttpServer 719

HydratedBloc 324, 325
 HydratedCubit<T> 333

I

IconButton 229, 262, 543
 identical() 126, 152
 identity() 409
 if 55, 61, 137, 142, 143, 146
 IgnorePointer 747
 Image 505
 Image.memory() 506
 Image.network() 506
 ImageLabeler 645
 immutable class 92
 implement 133
 implements 111–113, 115, 139
 Implicit animation 385
 import 76, 77
 index 43
 InheritedWidget 295, 296
 initializer list 83, 108, 109
 initializing formal 82, 86
 initialRoute 345, 347
 initialState 315
 initState 296, 297
 inMinutes 722
 InputDecoration 531
 inSeconds 722
 insert() 742
 Integration test 437
 interface 111
 Interface Segregation Principle 191
 internationalization 366
 intl 364, 365, 375, 380
 Intl.message 376
 Intl.plural() 381
 invokeMethod<T>() 252
 is 52
 isA<T>() 442
 isBoxOpen 739

isDefaultAction 595
isDestructiveAction 595
isolate 179, 180
Isolate.spawn() 184
Iterable<T> 167
Iterator<T> 169

J

JIT 29
JSON 417
jsonDecode 418–420
jsonEncode 420
JsonKey() 426
JSONPlaceholder 473
JsonSerializable() 424

K

Key 233, 235

L

late 36, 81, 87, 89
late final 81, 88
Latin1Decoder() 431
LayoutBuilder 278, 280, 530
library (keyword) 77
library aliases 77
limit() 620
limitToLast() 620
LinearGradient 224
LinkedHashMap<K,V> 149
linter 211
Liskov Substitution Principle 190
List.filled() 145
List.generate() 145
List.unmodifiable() 145
List<T> 141
listen() 178
ListTile 576, 617
ListView 222, 233
ListView.builder 222
load() 503

loadingBuilder 506
loadString() 503
Locale 367, 369
localization 366
Localization delegate 372
LSP 191

M

machine learning 633
mainAxisAlignment 219
map() 154, 171
Map<K,V> 147
Map<String, dynamic> toJson() 420
Matcher 450
MaterialApp 256, 257, 341
MaterialPageRoute 343
MaterialPageRoute<T> 344
Matrix4 409
maxLength 539
MediaQuery 280
method channels 251
MethodChannel 251
mixin 116–120, 152
mock 444
mockito 444, 445
model class 418
MultiBlocProvider 335
MultiProvider 309

N

named route 344
NavigationRail 579
NavigationRailDestination 580
Navigator 345, 351
Navigator.of() 344
new 80, 89
non-nullable 46–48
notifyListeners() 300
null 37, 46–48, 52, 81, 83, 86
nullable 47, 48
num 39, 140

O

Object 36, 74, 124
 ObjectKey 234
 obscureText 540
 OEM 25
 on 130–132, 134
 onAccept 554
 onBatteryStateChanged 568
 onDateTimeChanged 594
 onDestinationSelected 580
 onDispose() 733
 onGenerateRoute 345
 onSort 588
 onStepCancel 586
 onStepContinue 586
 onWillAccept 554
 OOP 74, 105
 open closed principle 188
 openDatabase() 741
 operator== 151, 152
 orderBy() 620
 OrientationBuilder 282
 override 56

P

package (keyword) 77
 PageRouteBuilder 414, 415
 PageStorageKey<T> 236
 PaginatedDataTable 590
 Paint 643
 paint() 641
 parse() 40
 part 424
 path_provider 484, 718
 Placemark 567
 placemarkFromCoordinates() 567
 playlistPlayAtIndex() 524
 pop() 345, 351
 Positioned 225
 pretty 430

primaryAccent 288
 primaryColor 288
 processImage() 636
 Provider 291
 Provider<T> 731
 ProviderScope 728
 pubspec 211
 pubspec.yaml 204, 206
 pump 451
 pump() 451
 pumpWidget 450
 pumpWidget() 450, 452
 pushNamed() 344, 345, 351
 putIfAbsent() 147

Q

QueryDocumentSnapshot 617
 queryParameters 482
 QuerySnapshot 619
 quiz 18

R

RadialGradient 224
 RaisedButton 261
 Raster thread 462
 ReceivePort 184
 recursive 719
 reduce() 154
 reload() 659
 remainder() 722
 RenderObject 245
 repeat() 396, 402
 ReplayBloc 328
 ReplayBlocMixin 330
 ReplayCubit<T> 333
 requestNotificationPermissions() 646
 required 68, 228
 reset() 723
 resizeToAvoidBottomInset 537
 ResolutionPreset 557
 Response 474, 479, 482

ResponseType.bytes 514
rethrow 135
reverse engineering 724
reverse() 396, 402
ReverseAnimation 402
RewardedVideoAd 632, 633
Riverpod 727
rootBundle 503
rotateX() 407
rotateY() 407
rotateZ() 407
RoundedRectangleBorder 263
route 339, 341
RouteGenerator 342
Router 348
Row 219–221
rowsPerPage 590
runApp() 215
runTransaction() 618

S

Scaffold 257
scale() 408
screenshot() 457
scroll() 457
scrollDirection 222
secondChild 391
selection 539
Selector<T> 310
Semantic 745
semantic tree 747
sendEmailVerification() 658
SendPort 184
set 97, 106
Set<T> 145, 146
setMockInitialValues() 572
setPersistence() 658
setState 229–231
setState() 291
setter 96

setUpAll() 457
SharedPreferences 572
short if 56
shouldRepaint() 641
show 78
showBottomSheet 265
showDialog() 262
showDialog<T>() 548
showFirst 391
showModalBottomSheet<T> 593
shrinkWrap 285
SimpleDialog 264
SingleChildScrollView 534
skewY() 407
Skia 27
skip 448
skip() 154, 171
sleep() 169
Slider 518, 522
SlideTransition 391, 415
SlidingPageRoute 415
smart cast 52
SOLID 187
sortAscending 587
SplayTreeMap<K,V> 149
spread operator 142
SRP 187, 189
Stack 225, 405
start() 723
State<T> 229, 230
Stateful widget 228
StatefulWidget 215, 226, 230, 233
StatelessWidget 215, 226, 227, 233
StateNotifier<T> 604
StateNotifierProvider 604
StateProvider<T> 729, 732
static 86, 91
Step 585
Stepper 584
stop() 723

Index

Stopwatch 723
 stream 166, 167
 Stream<T> 167, 169
 Stream<T>.empty() 171
 Stream<T>.error() 170
 Stream<T>.fromFuture() 170
 Stream<T>.fromIterable() 170
 Stream<T>.periodic() 170
 Stream<T>.value() 170
 StreamBuilder<T> 173, 562, 565
 StreamProvider<T> 733
 StringBuffer 42, 43
 subscribeToTopic() 653
 super 106, 108, 109
 SvgPicture 512
 SvgPicture.memory() 516
 SweepGradient 225
 swipe to dismiss 544
 switch 56
 sync* 169
 SynchronousFuture<T> 372

T

TabBar 581, 582
 TabBarView 582
 TabController 583
 TargetPlatform 274
 TDD 186
 tearDownAll() 457
 test 439
 test() 439
 testWidgets() 449
 Text 218
 TextEditingController 537
 TextFormField 531
 TextRecognizer 645
 TextStyle 218
 Theme 288
 Theme.of() 288
 ThemeData 287, 288, 325

then 159, 163
 then() 159, 160, 162, 163
 this 74, 85, 92, 122
 throw 133
 throwsA() 442
 TickerProviderStateMixin 396
 Timer 177
 toJson() 420
 toString() 41, 43, 124
 toXmlString 436
 Transform 399, 405
 Transform.rotate() 394
 translate() 408, 409
 tree shaking 213
 try 130, 131, 134
 ttf 209
 Tween<T> 402
 typedef 72, 73

U

UI thread 462
 Uint8List 717
 UniqueKey 234
 Unit test 437
 unmodifiable() 150
 UnmodifiableListView<T> 145
 update() 742
 useAnimationController() 599
 useFuture<T>() 602
 useMemoized<T> 602
 UserAccelerometerEvent 560
 userChanges() 657
 useStream<T>() 601
 useTabController() 601
 useTextEditingController() 601
 utc() 722
 Utf8Decoder() 431

V

ValueKey 233
 ValueKey<T> 233

var 35–38, 63, 64
verify() 737
verifyNoMoreInteractions() 737
video_player 518
VideoControllerPlayer 519
VideoControllerPlayer.asset() 519
VideoControllerPlayer.file() 519
VideoControllerPlayer.network() 519
VideoPlayerController 518
virtual 105
VirtualDirectory 721
vsync 396

W

wait() 160
watch 425
WebAssembly 243
when() 446

whenListen() 453
where 153
where() 153, 154, 621
while 58, 59
Widget 215
Widget test 437
widget tree 215
with 116
Wrap 561

X

xml 429
XmlBuilder 434, 435
 XmlDocument 430, 431

Y

YAML 206
yield 168, 169, 172, 174, 175
yield* 175

Special thanks to Felix Angelov, Matej Rešetár, Rémi Rousselet, Matthew Palomba and Alfred Schilken