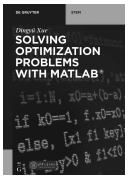
Dieter Meiller

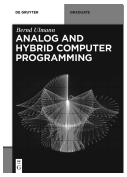
Modern App Development with Dart and Flutter 2

Also of interest

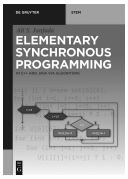


Solving Optimization Problems with MATLAB®
Dingyü Xue, 2020
ISBN 978-3-11-066364-8, e-ISBN (PDF) 978-3-11-066701-1,
e-ISBN (EPUB) 978-3-11-066369-3

Mobile Applications Development
With Python in Kivy Framework
Tarkeshwar Barua , Ruchi Doshi, Kamal Kant Hiran, 2021
ISBN 978-3-11-068938-9, e-ISBN (PDF) 978-3-11-068948-8,
e-ISBN (EPUB) 978-3-11-068952-5



Analog and Hybrid Computer Programming
Bernd Ulmann, 2020
ISBN 978-3-11-066207-8, e-ISBN (PDF) 978-3-11-066220-7,
e-ISBN (EPUB) 978-3-11-066224-5



Elementary Synchronous Programming in C++ and Java via algorithms
Ali S. Janfada, 2019
ISBN 978-3-11-061549-4, e-ISBN (PDF) 978-3-11-061648-4,
e-ISBN (EPUB) 978-3-11-061673-6

Programming in C
Xingni Zhou, Qiguang Miao, Lei Feng, 2020
Volume 1 Basic Data Structures and Program Statements
ISBN 978-3-11-069117-7, e-ISBN (PDF) 978-3-11-069232-7,
e-ISBN (EPUB) 978-3-11-069249-5

Volume 2 Composite Data Structures and Modularization
ISBN 978-3-11-069229-7, e-ISBN (PDF) 978-3-11-069230-3,
e-ISBN (EPUB) 978-3-11-069250-1

Dieter Meiller

Modern App Development with Dart and Flutter 2

A Comprehensive Introduction to Flutter

DE GRUYTER
OLDENBOURG

Author

Prof. Dr. Dieter Meiller
Ostbayerische Technische Hochschule (OTH)
Kaiser-Wilhelm-Ring 23
92224 Amberg
Germany

Dieter Meiller was born in 1970 in Amberg, Germany. He holds a doctorate in natural sciences from the FernUniversität in Hagen, as well as a master's degree in computer science and a diploma in communication design from the Technical University of Nuremberg Georg Simon Ohm. For several years, he worked as a media designer and software developer for various companies, partially on a freelance basis. Since 2008 he is professor for media informatics at the East Bavarian Technical University Amberg-Weiden.
His interests in research and teaching are information visualization, data science, machine learning, human-machine interaction, usability and accessibility, interaction design, web technologies, programming languages for media and computer art.

ISBN 978-3-11-072127-0
e-ISBN (PDF) 978-3-11-072133-1
e-ISBN (EPUB) 978-3-11-072160-7

Library of Congress Control Number: 2021937995

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet at http://dnb.dnb.de.

Cover image: by Dieter Meiller
Printing and binding: CPI books GmbH, Leck

www.degruyter.com

Thanks

My special thanks go to my family, my dear wife Magdalena for her active support and to my children Pauline and Leo. I also thank my colleague Dominik Gruntz from the University of Applied Sciences and Arts Northwestern Switzerland for his expertise and corrections.

https://doi.org/10.1515/9783110721331-202

Preface

At the end of 2018 a new technology was born: Google released Flutter in version 1.0, a new framework to program cross-platform apps for Android and iOS. It seems that it could have a similar impact on the digital world as Java technology at the time.

The development should be easier and faster than before. In addition, the apps should have better performance than previous solutions thanks to the integrated render engine Skia. Google also announced that it is developing a new operating system in addition to Android that will support new types of devices. The application development for this is expected be done with Flutter.

Although there are many tutorials and code examples on the web, a more structured explanation of the topic should not be missed. It is time to introduce the app programming with Flutter and the programming language Dart to interested developers in book form. I would be pleased to receive feedback and suggestions and wish all readers many interesting insights while reading this book and much fun and success while developing!

Kastl, May, 2021

Dieter Meiller

https://doi.org/10.1515/9783110721331-203

Contents

1 Introduction

This book deals with a specific innovative technology for the efficient development of apps for mobile devices. Consequently, it is possible to realize native apps cross-platform in only one programming language. Various practical examples introduce a new technology called "Flutter." The main part of this book is the description of a more complex app, a 2D game called "Chicken Maze." The goal of the game is to navigate a chicken through several levels, where it earns points and fights enemies.

When Steve Jobs presented the iPhone at the beginning of 2007, it was a turning point in the digital world. He also demonstrated a new type of software that can be run on the new device, the so-called apps, which is an abbreviation for "Applications." The abbreviation was probably intended to suggest that these are smaller programs. The first apps were smaller gadgets, for example, the beer-drinker app: You could put the phone to your mouth and tilt it, a beer level that could be seen on the display slowly decreased, and in the end, you heard a burping sound. The amazing thing was that the beer level always remained horizontal, as the program could determine the angle of the device using a built-in sensor. The app is available from the App Store, Apple's cloud-based software sales platform, revolutionary for its time and exclusive to Apple's mobile devices.

People stood in line at Apple stores to buy an iPhone and expected new apps. Developers also started rushing for the new device; it was a kind of new gold rush. In 2008 the first competing mobile phone with an Android operating system was launched. The operating system was mainly developed by Google. Google gave the manufacturers of mobile devices a ready-made system software. A store, the Google Play-Store, was also integrated there. For a short time, there were other competing systems such as Windows Mobile or webOS, but these were unable to establish themselves and eventually disappeared. What remained are iOS and Android, so today's development tools concentrate exclusively on these two. The gold rush is long gone, and meanwhile, you have to do a lot of work as a developer to have commercial success by developing apps.

1.1 Cross-Platform Development

Apps are often developed twice and individually ported to the respective platform. Native Apps[1] for the iOS environment, i.e., for Apple mobile devices, are programmed using the programming languages Objective-C or Swift and Apple's frameworks. Android apps, on the other hand, are usually based on the Java platform with the programming languages Java or Kotlin. If you want to work with the in-house technology of compet-

1 These are apps programmed and optimized specifically for a platform

https://doi.org/10.1515/9783110721331-001

ing companies, you have to laboriously translate the code into the other languages by hand. The need for a cross-platform solution, therefore soon came up and various technologies were developed. Basically, we can distinguish between two approaches: On the one hand, there are frameworks that work with web technologies. However, this should not be confused with web apps. These are merely web pages that adapt as well as possible to the mobile device. For app development, there is the procedure of embedding web pages into the app and presenting them via a browser component that fills the entire screen. Your own code is programmed in JavaScript. Via an API, you can talk to native components, which can then address the specific hardware of the device. This approach has two disadvantages: You can only use features that are available on both platforms. Furthermore, the performance is limited because, in the end, you are dealing with a browser application. The advantage is that web developers can also attempt app programming. Well-known frameworks of this kind are React Native from Facebook and Apache Cordova (formerly PhoneGap from Adobe).

The other approach is the following: You program with platform-unspecific languages and frameworks, which are then cross-compiled into the desired environment and therefore have a better performance. A well-known representative of this technology is Xamarin from Microsoft. There, programming is done with C#. The Flutter framework with the programming language Dart also belongs to this category.

1.2 Motivation

The hype about Flutter started in 2018 when it became known that Google was working on a new operating system called Fuchsia, and there was the rumor that this new operating system should replace Android. Meanwhile, Google denied this and announced that the new operating system would be used on new types of devices in the future. Fuchsia is open source and freely available, it can be tested and installed [35]. Flutter is used for the user interface programming. The book does not discuss Fuchsia itself, but you can use the knowledge taught in the book to develop applications for Fuchsia. The graphical user interface is displayed with a render engine called Skia, which is used in Flutter. The framework was in the alpha and beta phase after the first release and allowed the programming of native apps for iOS and Android systems. In addition, Google recommended that you should better realize your apps with Flutter, as it realizes the Google material design. The interest in Flutter was then awakened in the author, who was planning a course for media computer scientists on app programming at that time.

1.3 Target Group

This book targets app developers who are interested in developing new cross-platform apps. These can be professional programmers or students of courses in computer science or media-related studies.

All such persons should have some basic knowledge of programming. The book does not provide a basic introduction to programming. It does not discuss the meaning of variables, loops and control structures. Although the syntax of these structures in the Dart language is also explained, the meaning of the constructs should be clear. The reader should also have some knowledge of object-oriented programming. Ideally, readers should also have already dealt with program libraries and the command line. Knowledge of Java, C#, JavaScript or C++ as first the programming language would be ideal.

1.4 Structure and Aim of the Book

The book points out the pitfalls that you inevitably run into when you take on the adventure of learning a new programming language and a new framework. The book is divided into two parts: a fundamental section and a practical section. The first part explains the most important constructs of the Dart language. This is followed by an introduction to app programming with Flutter. Simple examples are shown there. In the second section, which is the practical part, more sophisticated examples follow, starting with a messenger app that demonstrates how the app connects to the cloud. A more complex application follows, a 2D game called “Chicken Maze,” where a chicken is the game character. The game is not too complex, but is more than a demo app. It contains features that professional applications need. The code of the examples is explained in detail and specific programming techniques of Flutter and Dart are described.

Part I: **Foundations**

In the first part, the syntax and the special features of the programming language Dart is discussed. Then the Flutter framework is introduced with its philosophy, conventions and structure.

2 Foundations of Dart

In the following, the language Dart is briefly introduced and a complete introduction to the programming language given. However, the basics of programming are not taught here. Basic skills in imperative and object-oriented programming are expected. The chapter tries to teach special features of the language in comparison to other languages and basics, which are important to understand how to use the Flutter-Framework. It deals with Dart in version 2.12

The extended Backus Naur Form (BNF), a meta syntax, is used to explain parts of the syntax. However, this does not provide a completely correct description of the Dart language, as it would be necessary for compilers or standards. Some non-terminals are also not further resolved. It should be clear what is meant by `<condition>` (a boolean expression), for example.

2.1 The Dart Language

Dart is a programming language developed by Google. In addition to Dart, Google developed many other programming languages, including Groovy. This language is compatible with Java and generates code for the Java Virtual Machine. The language Go was released at the same time as Dart. Go is a hardware-oriented language that can be used for system development, similar to C and C++. Go is supposed to be a modern version of these languages and compiles faster.

Dart, in contrast, was designed as a language for application development. Originally, Dart was developed as a language for the web browser Chrome. The reputation of Dart was controversial in the developer community due to its direct competition with JavaScript and ran contrary to efforts to standardize and interoperate web browsers. It was feared that there would be a new browser war like the one in the 2000s when Microsoft introduced Visual Basic as the browser language in competition with JavaScript, the Netscape language of the time. Therefore, the acceptance of the Dart language was rather low, and it became quiet around Dart.

In early 2018 Google announced a new framework for the development of apps, namely Flutter (see chapter 4). The development in Flutter should be done exclusively in Dart. At the start of Flutter, Dart was also modernized and is now available in version 2. In the process of this development, Dart changed from an extension of JavaScript to a language, which has strong similarities to C# and Java. Subsequently, Dart offers dynamic typing as well as static typing.

https://doi.org/10.1515/9783110721331-002

Dart is being developed by Google since 2013. It is available in version 2.12 (stable) at the time of writing this book. There is the ECMA standard: ECMA-408.

2.1.1 Mobile and Web

Dart is mainly available for three types of platforms: Mobile devices, the console and the web [14]. The way it is executed on the three platforms is different. For mobile devices, you can run Dart in a virtual machine (VM) with a just-in-time compiler, which is used in development with Dart and Flutter. Flutter has the "Hot-Reload" feature. You can make changes to the code while the application is running. This is done by running it from the VM. When the app is released, the code is compiled to the native ARM code, which is the second form of execution, i.e., compiling with an Ahead-of-Time compiler and executing the native code.

Additionally, there is the web environment, where the dart code is compiled to JavaScript. An interesting framework for the web environment is AngularDart [4], a dart version of Angular. Although the web version is not part of the book, it is certainly good to know that you are learning a language that has other target platforms. Furthermore, you can run Dart as interpreted or compiled language on your PC or server.

2.1.2 Installation

If you want to program mobile apps with Flutter, it is not necessary to install the Dart SDK and the Runtime contained in it separately, you rather need to install Flutter, which contains Dart.

2.1.2.1 Dart-SDK Installation

However, you can also install Dart and the SDK if you want to use it without Flutter, either for programming experiments or to test the examples in the book. To do this, use the package manager appropriate for your operating system (for detailed instructions, see [15]). On Windows, you can use the package manager Chocolatey in the console as in Listing 2.2 line 1. On Linux (lines 2 and 3), you can use the usual built-in package manager apt. Similarly, for Mac computers, you can use the homebrew package manager (lines 4 and 5).

Listing 2.1: Install Dart

```
C:\> choco install dart-sdk
$ sudo apt-get update
$ sudo apt-get install dart
$ brew tap dart-lang/dart
$ brew install dart
```

Dart also runs on ARM systems like the Raspberry PI. You can get the SDK by using these commands:

```
wget https://storage.googleapis.com/dart-archive/channels/stable/release/2.12.0/
    sdk/dartsdk-linux-arm-release.zip
unzip dartsdk-linux-arm-release.zip
```

2.2 Dart Syntax

Listing 2.2 shows the usual hello "world program" in Dart: The entry function is like in C: *main()*. As line 2 shows the similarity to other typed languages of the C family, the *for* loop is exactly as you would expect. The *print* in line 3 outputs a string.

2.2.1 Start Program

You can start the program from the console with `dart helloWorld.dart`. Also, you can compile the source code first with `dart2aot helloWorld.dart helloWorld.aot` into native code and then start it with `dartaotruntime helloWorld.aot`. Source code files must have the extension ".dart."

Listing 2.2: helloWorld.dart

```
void main() {
  for (int i = 0; i < 5; i++) {
    print('hello ${i + 1}');
  }
}
```

2.2.2 Data Types

Dart allows the use of static and dynamic typing. There is also type inference. Types are permanently assigned to constants.

2.2.2.1 Assignment and Initialization

Before using variables and functions, their types must be specified. However, you can put *var* or *dynamic* in front of variables or functions. There is one important difference: In the line `dynamic x = 42;` the variable *x* is assigned the integer value 42. Afterwards, you can also assign values with other types, for example, `x = "Hello World";`. In contrast, you cannot do this with `var y = 23;`. Here, an integer value is assigned to the variable *y*. However, the type for the variable *x* is fixed now, and this may seem unusual for developers experienced in JavaScript, but C# developers should be able to get along with it. This is called implicit typing, type inference or type derivation.

The same is true when using the term *final*: The assignment `final z = 21;` defines an int-variable *z* with the value 21, but this cannot be changed anymore. There is, however, a subtle but important difference to the assignment `const u = 3;`. Here again, the value 3 is assigned once and the type from *u* is changed to *int*. With *const*, however, the assigned value must already be determined before compilation, i.e. it must be written directly into the code. Values at the final assignment can be calculated once, but only at runtime. So `final x = DateTime.now();` does work, but `const x = DateTime.now();` does not.

2.2.2.2 Number Types in Dart

In Listing 2.3, there are some examples of the use of data types in Dart. Line 2 and 3 define variables of type *int* and double, the two number types of Dart. Variables of type *int* can store integers. In Dart, however, unlike other languages such as Java, there are no other integer types such as *long* or *short*. The type *int* represents all these types. Optimized memory management ensures that only as much memory as necessary is reserved, depending on the size of the number. The same applies to the type *double*. This type stands for real floating-point numbers. The name *double* indicates that it is comparable in accuracy to the type double in Java or other languages. However, if a smaller number is stored, it is not necessary to specify a type such as *float*, as this is regulated by the enhanced memory management so that no superfluous memory space is reserved. Both number types, *int* and *double*, inherit from type *num*. A variable of type *num* can store values of both number types. These are then converted to *int* or *num*, as required. The code

```
1  num x = 42;
2  print(x.runtimeType);
```

shows *int* as output. In my opinion, the type *double* should have been called *float*, since the integer type was called *int* and not *long*. Line 4 shows the use of the truth type *bool*: it can take the values *true* and *false*.

---

i In Dart, there are three types for numbers: *int* for integers and *double* for real numbers. Both are subtypes of *num*.

Attention: All types get the default value zero if the variables have only been declared and have not yet been assigned a value. Thus, a declared bool variable initially has the value *null*, and *null* is not equal to *false*, which can be a problem in comparisons.[2]

Therefore, there is a dart-specific conditional assignment operator ??= for initialization: This says that if the variable has the value zero, the expression on the right is assigned.

All declared variables get the default value *null*. Each variable can also be given the value null. From version 2.12 Dart becomes null-safe, which means that all variables must be initialized and cannot be assigned with the value *null*. Non-null-safe code as used in this book is incompatible with null-safe code and must be rewritten (more about null-safety see 2.2.16).

2.2.2.3 Strings in Dart

As usual in Java, there is a type for strings (*String*), see line 5, capitalized according to the conventions in Java, which is a bit confusing since there is no convention that object types are capitalized, and primitive types are lowercase. In Dart, all types are objects, including the number types just mentioned. You can mark multi-line string values with three quotation marks at the beginning and end. Besides, Dart uses Unicode (UTF-8). So, you can use arbitrary characters in the source code and strings, also characters of other languages, such as the Japanese Kanji, can be used. Strings can be in single and double quote pairs. There is no difference here.

Variables can be evaluated within strings by placing a dollar sign ($) in front of the variables. You can also evaluate whole expressions in strings by placing the expression in curly brackets (`${...}`)– see line 8.

In Dart, strings can be written with single and double quotes. Variables with $ characters will be evaluated in the string, regardless if it was enclosed with single or double quotes.

There is also a type of symbols that stand for themselves. Symbols are marked with a hash (#) (see line 10). However, there are also enums for similar purposes (line 38): A variable of type *Direction* can only take the four listed values here.

As usual in most modern programming languages, functions have a special status and can be defined and processed in different ways. There is the classic definition, with the return value, function name, parameter list, and function body with the return value (lines 35-37).

There is also the lambda notation, which can be used to assign an unnamed function object to a variable, as in lines 11 and 12. In line 11 there is a one-line short notation similar to JavaScript, the fat arrow [3], which you can use if the function value

---

2 There is even the type *Null*, but a variable of type *Null* can only contain the value *null*.

3 FART: Fat-Arrow-(Term?)

can be expressed in a term to the right of the arrow. If you need more than one row, you have to create a body of curly brackets instead of the arrow and return the value with the command *return* (lines 12-16).

---

i Lambda expressions have the form `(<Parameter>) => <Expression>` in Dart, if the function can be written in a single line. With the form `(<Parameter>) {<Code lines> return <Expression>;}`, you can include several lines of code in the function.

In addition, you can specify the type of a function object more precisely than just the general type `Function`. To do this, you must define a new type using the keyword `typedef` by specifying the signature, which is the return value, and the parameter types of the function. You can then use this new type to typify variables that store corresponding function objects (lines 30-34). Moreover, named functions, unlike in many other languages, can contain further function definitions within their body, since they are also function objects.

There are two important types of collections in Dart: maps and lists with several elements. Line 17 shows the definition of an untyped list containing numbers and a string. The values are given in square brackets, separated by commas. The type that is displayed here in line 19 is, therefore, `List<Object>`. You can also create lists constantly (line 18). Here, all values of the list are constant. If, on the contrary, only the variable `list2` [4] had been specified as const, the list would be constant, but not its contents.

Here you can also see that type inference is at work, and the keyword `var` would actually suggest that no type is specified. However, the type was determined and set for the variable `list1`. Thus, you cannot assign values other than lists to the variable, `list = 23;` would cause an error in the static type checking. On the other hand, if you only declare a variable and do not assign an initial value to it, as with `var a;`, then you can store any value in the variable. Its type is then `dynamic`. You can also specify this type explicitly: `dynamic x = 42;`. This means that Dart has static and dynamic typing.

You can also specify the type of the list in pointed brackets: in line 20, only `double` values are allowed in the list. The comma at the end of the list is allowed and common in Dart: this notation has been established especially for Flutter.

A dictionary type with key-value pairs is also available: `Map`, starting at line 21. You can specify key : value pairs here, in curly brackets. Of course, you can mix maps and lists to create a structure reminiscent of JSON objects [12].

---

4 except that this would not be a variable but a constant

Listing 2.3: dartTypes.dart

```
void main() {
  int aNumber = 4;
  double otherNumber = 4.0;
  bool isTrue = true;
  String hallo = '''
Multiline
String
Nummer: ${aNumber + 1}
''';
  Symbol sym = #metoo;
  Function foo = (double x) => x*x;
  Function bar = (x) {
    assert(x is double);
    double val = x*x;
    return val;
  };
  var list1 = [1,2,3, 'Foo'];
  var list2 = const [1, 2, 'Hello'];
  print(list1.runtimeType);
  List<double> list3 = [1.0, 2.0, 3.0,];
  Map map1 = {
    'name': 'Hans',
    'alter': 33,
    'square': (x) => x*x,
  };
  print(map1["square"](3.0));
  print(map1.runtimeType);
}

typedef MyFunction = int Function(int);

MyFunction foo2 = (int x) {
  return x*x;
};
int sq(int v) {
  return v*v;
}
enum Direction {left, right, up, down}
Direction d = Direction.left;
```

As collection types, there are in Dart Lists `[...]` where you separate the values with commas, and there are Maps `{K:V}` with keys and values.

A special feature of handling the initialization of lists and other collections is that you can write *for* loops and *if* conditions directly in the parenthesis term of the list. Listing 2.4 produces the output `[0, 1, 2, 3, 4, 6, 7, 8, 9]`, i.e., a list of the numbers 0-9, without the number 5. In the language Python, this is known as "List-Comprehensions." Note that the loop and the if must not have a body in curly brackets: otherwise, it would be interpreted as a map.

Listing 2.4: collections.dart

```
1 void main() {
2   var numbers = [
3     for (int i=0; i<10; i++)
4       if (i != 5) i
5   ];
6   print (numbers);
7 }
```

2.2.3 Control Flow

The basic options for controlling the program flow are listed here. These are based on the usual C syntax.

2.2.3.1 Loops

The following example loops (listing 2.5) output the numbers from 1-10. Line 4 shows the classic *for*-count loop. In the round brackets after the *for* comes first the initialization part, where variables can be initialized. After the semicolon comes the condition part, where a boolean expression should be placed. As long as this expression is true, another loop run is executed. After the second semicolon comes the update area, all statements there are executed after the loop pass. With Dart, as in other C-like languages, you can write several expressions in the areas in the round brackets, separated by a comma. However, it is recommended to avoid such expressions for readability reasons. After the round brackets, the body follows, i.e., a loop block in curly brackets or a single command.

In line 8, you can see the above-explained list initialization using a *for* loop. The list then contains the numbers 1-10.

In line 10, you see a for-in loop. The variable *i* takes every value in the list `lst`. An alternative is the `.forEach(...)` method. All lists have this method. You use a lambda

function as parameter, this is called for every value in the list and gets the current value as parameter.

In lines 18 and 19, you can see another possibility for iteration of lists. All lists have an iterator. Its method `.moveNext()` lets a pointer move to the first or next entry. If there is no more iteration, the method returns false, otherwise true. The `.current` attribute then returns the entry at the current pointer. The *while* loop runs as long as the condition in the round brackets is true.

Lines 23-27 then show the *do-while* variant for the sake of completeness. Here the block of the loop is executed at least once and repeated until the condition is false.

Dart loops:

Counting loop: `for (<Init>;<Condition>;<Update>) <Body>`,
Do-While loop: `do <Body> while (<Condition>)`,
While loop: `while (<Condition>) <Body>`,
For-In loop: `for (<Variable> in <list>) <Body>`,
Foreach method: `<List>.forEach(<Lambda function>)`.

2.2.3.2 Branches

Now we come to conditional execution of program code: Here we have the classic *if-else* statement, as usual. The *if* block is only executed if the condition in the brackets is true, otherwise, the *else*-block is executed.

From line 37 on, you see a *switch-case* statement. There are a few small special features of Dart to notice. *switch-case* can check several cases of a variable (here *rand*) and then jump to the respective *case* branch. A *break* statement ends the *switch* block. The *default:*-label is jumped to if the parameter variable does not match any of the cases. So far, everything is similar to other C-like languages. In Dart, however, you cannot omit the *break* to fall through to the next case block, as would be possible in C or Java. However, unlike other languages, you can define your own labels (line 47: *onefour*) and jump to them with a *continue* command. You can also end the statement with an exception handling, i.e. with *throw* or *rethrow*. You can call the statement with all kinds of variables with discrete types, i.e., *int*, enums or symbols. It is not possible to call the statement with variables of type *bool* and *double*.

Dart branches:

If-Else: `if (<condition>) <Body> [ else <Body> ]`
Switch case: `switch (<Condition>) { <Case-List> [ [<Label>:] default: <Statements> ]}`
with `<Case-List> ::= <Case-Line> | <Case-Line> <Case-List>`
and `<Case-Line> ::= [<Label>:] case <Value>: <Statements> break;`

Listing 2.5: control.dart

```
import 'dart:math';

void main() {
  for (int i = 1; i <= 10; i++) {
    print(i);
  }

  List<int> lst = [for (int i = 1; i <= 10; i++) i];

  for (var i in lst) {
    print(i);
  }

  lst.forEach((i) {
    print(i);
  });

  var it = lst.iterator;
  while (it.moveNext()) {
    print(it.current);
  }

  it = lst.iterator;
  it.moveNext();
  do {
    print(it.current);
  } while (it.moveNext());

  int rand = Random().nextInt(10);

  if (rand < 5) {
    print("smaller 5");
  } else {
    print("greater 5");
  }

  switch (rand) {
    case 0:
      print("null");
      break;
```

```
41      case 1:
42        print("one");
43        continue onefour;
44      case 2:
45        print("two");
46        break;
47      onefour:
48      case 4:
49        print("1 or 4");
50        break;
51      default:
52        print("none");
53    }
54  }
```

2.2.4 Exception Handling

In Dart, there are several ways to react to errors that may occur during the runtime of the program.

2.2.4.1 Try-Catch

In the code example 2.6 you can see how the exception handling is done in Dart with *try-catch*. A variable *i* is initialized with 0, a string *s* with a character "2," another *int*-variable is also initialized with 0. In the *try* block, an attempt is made to convert the string into a number and then divide it by the contents of the variable *d*[5]. Depending on the variable assignments, two problems can occur: The string s contains characters that cannot be interpreted as numbers. Or the divisor d contains a 0, by which it is not allowed to divide.

After the *try* block in line 10, the first exception is handled by a *catch* block when a format exception occurs, to it is then jumped to. The output of the parameter *e* provides further information about the exception. If the second problem occurs, the system jumps to the second *catch* block. The optional *finally* block is always executed, even if no error occurs. The difference to exception handling in other languages is small: The type of error is specified after the `on` keyword. The sense of this modification is as follows: You do not necessarily have to receive the parameter with `catch(e)`. So, it would be possible to catch the parameter with `on SomeException { ... }` to catch an exception *SomeException*. In addition, you can also omit the `on ...` clause and catch

5 "~/" is the division operator for integer numbers without remainder. In contrast to "%," this is the modulo operator, i.e., the remainder of the division.

unspecified errors with a pure `catch(e)`. Such statement would be useful if all errors are to be caught. It should be placed before the *finally* block and after all named error types have been handled.

2.2.4.2 Error handling with methods

For so-called futures (section 2.2.14), there is also a method `.catchError(<Lambda-Function>)` with which you can catch errors. Additionally, there is a functional counterpart to the *finally* statement: `.whenComplete(<Lambda-Function>)`.

2.2.4.3 Own Exceptions

You can also create your own exception classes (line 31) by implementing the interface *Exception* (see section 2.2.11). This is a pure marker interface where you do not have to implement any methods, but you are allowed to. These exceptions can then be thrown with `throw`. Besides the keyword `throw`, Dart also knows `rethrow`, which can be used to throw the already caught exception. This keyword may only occur in catch and on blocks.

---

i Dart exception handling:

```
try <Body> <Catch statements> [finally <Body>]
with <Catch statements> ::= <Catch statement> | <Catch statement> <Catch statements>
and <Catch statement> ::= <how to catch?> <Body>
and <how to catch?> ::= on <Exception> | [ on <Exception> ] catch(<Variable>)
```

Listing 2.6: control.dart

```
1  void main() {
2    int i = 0;
3    String s = "1";
4    int d = 3;
5    try {
6      i = int.parse(s) ~/ d;
7      if (i < 0) {
8        throw new TooSmallException();
9      }
10   } on FormatException catch (e) {
11     print(e.runtimeType);
12   } on IntegerDivisionByZeroException {
13     print("Division by Zero!");
14     rethrow;
15   } catch(e) {
```

```
16      rethrow;
17    } finally {
18      print("Ready.");
19    }
20  }
21
22  class TooSmallException implements Exception { }
```

2.2.5 Object-oriented Programming

The entry into the programming language Dart is easy for Java or C# experienced programmers. Google intends to keep the hurdles for the change as low as possible. In Listing 2.7, you can see the definition of a class `Person`. Listing 2.8 imports the class file, creates an object and calls its method `info()`. The code is directly readable for the mentioned programmers; with minor deviations, the code would also be immediately executable in the mentioned languages.

The class definition (Listing 2.7) is identical to a definition in Java. Lines 2-4 declare the typed object variables, line 6 is the signature of the constructor, lines 7-9 fill the previously declared variables with the parameter values. The method `void info()` outputs a string with the *print* command, the string concatenation works analogously to a definition in Java. Only the globally available *print* function offers other possibilities due to the variable evaluation in the strings. Slightly different is also the import of code in Listing 2.8, line 1, where you have to specify files and not classes or packages. All definitions of the imported files are then directly visible, not only classes but also other definitions like variables or functions.

Class definition: `class <Class name> { <Object variables> <Constructor> <Methods> }`

It is not required to create one file per class like in Java, you can write arbitrary code into a file. Also, the naming of the file is not bound to certain naming conventions. You can mix classes, functions and other definitions in one file. Furthermore, packages do not exist in the form as in Java: you can store code files in folders in a structured way.

Listing 2.7: Person-1.dart

```
1  class Person {
2    String fname = "";
3    String lname = "";
4    int age = 0;
5
6    Person(String fname, String lname, int age) {
7      this.fname = fname;
8      this.lname = lname;
9      this.age = age;
10   }
11   void info() {
12     print("Person " + this.fname + " " + this.lname +
13     " is " + (this.age.toString()) + " years old.");
14   }
15 }
```

Listing 2.8: main-1.dart

```
1  import 'Person-1.dart';
2  void main() {
3    Person p = new Person("John", "Smith", 33);
4    p.info();
5  }
```

2.2.6 Shorter Spellings

Here are a few examples that build upon the previous one. In 2.9, the following is noticeable: The object variables (lines 2-4) start with an underscore. All variables whose names start with an underscore are protected, which means they are not visible outside their own package and are, therefore, protected from outside access. In Java, this is the same as the default visibility (package scoped), without modifier `protected` in front of the variable. Further, finer control of access protection does not exist in Dart (yet). So also, no `private` or `public`. But this makes the program easier to understand. For example, in Java, the difference between `protected` and Package-Local (no modifier) is not always intuitively clear. In other programming languages such as Python, the underscore often has only a purely semantic meaning, where only variables are marked by convention as private object variables. The compiler for Python makes no difference between variables with and without underscore. In Dart, however, the underscore is part of the syntax.

In line 6, it is noticeable that the constructor does not have a body of curly brackets at all. The variables are initialized there in the round parameter brackets directly (with `this.<Variable>`). This makes the code shorter and more readable. Furthermore, you do not have to specify the types there. As a convention, you should specify the variable type only once (when declaring the object variables): this is suggested in the coding style of Dart. However, this can be seen critically, because then the auto-completion in the editors or development environments might not be able to make type suggestions anymore. There is another small syntax deviation: As you can see in Listing 2.10, line 4, you can simply omit the `new` keyword when creating an object. This has no further effect on the behavior of the program. However, in many source codes you can see, e.g. on GitHub, that `new` is often used there. Later, in flutter layouts, omitting `new` is useful as it allows you to construct layout structures from objects in a declarative style.

Constructor: `<Class name>(<One or more constructor parameter>) <Body> | ;`

Listing 2.9: Person-2.dart

```
1  class Person {
2    String _fname;
3    String _lname;
4    int _age;
5
6    Person(this._fname, this._lname, this._age);
7
8    void info() {
9      print("Person $_fname $_lname is $_age years old.");
10   }
11 }
```

Listing 2.10: main-2.dart

```
1  import 'Person-2.dart';
2
3  void main() {
4    var p = Person("John", "Smith", 33);
5    p.info();
6  }
```

2.2.7 Named Constructors, Optional Parameters, Fat-Arrow

If you want to have several possibilities to call a constructor, i.e., with a varying number of parameters, you have to write several constructors in many languages. In Java, this would be constructors with a varying number of parameters, which may then call each other (Constructor Chaining).

In Dart, there is a more elegant way: You can make parameter[6] optional by putting them in square brackets (see Listing 2.11, lines 6 and 7, Invocation Listing 2.12).

Now it gets a bit complex: In addition to the usual positional parameters, i.e., parameters where the values are assigned in order, Dart offers the possibility to use keyword parameters. In Listing 2.13 you can see the different possibilities of declaration and call. Note that optional positional parameters cannot be mixed with keyword parameters. The function declaration in line 21 does not work this way.

---

i In parameter brackets optional parameters `[...]` or named parameters `{k:v}` can be used in addition to the position parameters.

By the way: The double question mark in line 6 is a special form of conditional expression. In most languages, there is the well-known `x ? a : b` expression. As a reminder, this means that if *x* is *true*, the whole expression takes the value *a*, otherwise *b*. In Dart, there is also the `a ?? b` expression. It means: If *a* is *null*, the expression becomes *b*, otherwise *a*. This syntax is useful because variables can be initialized with null by default, if you explicitly allow this by adding a question mark after the type (`int? y`).

There is a conditional assignment operator (ternary operator) that makes the assignment only if the variable is not null `<Variable> ?? <Expression>`.

In lines 14 to 19 of Listing 2.11, you can see further special features: Dart offers Getter and Setter with the keywords `get` and `set`. If you put them in front of method names, they can be addressed like object variables: `get name {return "Fred";}` can be read using `object.name`. `set name(s) {this.name = n;}` can be assigned using `object.name = "Fritz"`. As mentioned in 2.2.2, Dart offers the possibility to create function objects and to keep the notation short with the Fat-Arrow. This syntax can then be used with the getters and setters. The above examples becomes: `get name => "Fred";` and `set name(s) => this.name = n;`.

From line 9 on to line 12, a so-called named constructor can be found. You can offer several constructors this way, but they have to be addressed with their own name (see Listing 2.12, line 6).

---

6 This applies to parameters of constructors as well as parameters of methods or functions.

Listing 2.11: Person-3.dart

```
class Person {
  String _fname = "";
  String _lname = "";
  int _age = 0;

  Person(this._lname, [this._fname = 'Unknown',
         this._age = 18]);

  Person.fromOther(Person p) {
    this._fname = p.firstName;
    this._lname = p.lastName;
    this._age = p.age;
  }
  get info => "Person $_fname $_lname is $_age years old.";
  String get lastName => this._lname;
  get firstName => this._fname;
  get age => this._age;
  set lastName(String nam) => this._lname = nam;
  printInfo() => print(info);
}
```

Listing 2.12: main-3.dart

```
import 'Person-3.dart';

void main() {
  Person p = Person("Smith");
  print(p.info);
  Person q = Person.fromOther(p);
  q.printInfo(); // same result
}
```

Listing 2.13: parameter.dart

```
int addXYZ1(int x, int y, int z) {
  return x + y + z;
}

int addXYZ2(int x, [int? y, int? z]) {
  return x + (y ?? 0) + (z ?? 0);
}
int addXYZ3(int x, [int y = 0, int z = 0]) {
```

```
9   return x + y + z;
10 }
11
12 int addXYZ4({x: 0, y: 0, z: 0}) {
13   return x + y + z;
14 }
15
16 int addXYZ5(int x, int y, {int z: 0}) {
17   return x + y + z;
18 }
19
20 // wrong
21 int addXYZ6(int x, [int y], {int z: 0}) {
22   return x + y + z;
23 }
24
25 void main() {
26   print( addXYZ1(1, 2, 3) );
27   print( addXYZ2(1, 2) );
28   print( addXYZ3(1, 2) );
29   print( addXYZ4(x: 1, y: 2) );
30   print( addXYZ5(1, 2, z: 2) );
31   // wrong
32   print( addXYZ6(1, 2, z: 2) );
33 }
```

2.2.8 Immutable, Annotations

Dart offers the possibility to make objects "immutable," which means unchangeable. This indicates that these objects cannot change their state, unlike the usual ones in object orientation. The concept is then applied in the Flutter framework, where there are stateless widgets, i.e., elements of the user interface that have no state such as text that does not change. The advantage is an easier, faster memory management.

There are two ways to make a class immutable: The first one is in 2.14. All fields must be final (lines 4-6), additionally, the constructor must be preceded by the `const` keyword. Then, as in Listing 2.15, line 4, you can call the constructor with const instead of using the (otherwise optional) keyword `new`. With this method of making the object immutable, the object is already generated at compile time instead of loading it into memory at runtime. The disadvantage of this method is that all information about the

state of the object must be available. So you cannot load information or generate it by user input and then generate the object unchangeably. But there is a second option:

In Dart you can use annotations, similar to Java. So you can mark parts of code with labels, which can be evaluated by the system. There is the package "meta" in the system library of Dart.

This can be included via `import 'package:meta/meta.dart';`. Then you only have to write the annotation `@immutable` in front of the class definition. With this, the keywords `final` and `const` need not be used. If you then create an object, it is frozen in its state and cannot be changed. The advantage is that you can generate information dynamically before you create objects.

Immutable classes: The use of immutable classes as well as *final* and *const* variables can improve performance. **i**

In line 8 in Listing 2.14, you see an annotation `@required`, which is part of Flutter. Here in Listing 2.16, you can find a definition of the annotation. In Dart, you can define them with the standard programming syntax. This is in contrast to the definition of annotations in Java, where the syntax does not fit the syntax of the rest of the programming language. In Dart, annotations are either constant variables or constant constructors. If you create a constant object from a class, you can use it to decorate the code with a preceding @ symbol and evaluate it if necessary. The evaluation of metadata about the code is done with the `dart:mirrors` library, analogous to Reflections in Java. Besides, there is a modifier keyword `required` which should be used instead of the annotation `@required` in Dart 2.12 and later.

2.2.9 Packages and Dependencies

To resolve the dependencies of your own code, like to the use of the meta package, you first have to create your own package: By convention, the working directory must contain a so-called "pubspec.yaml" file that lists the dependencies. Listing 2.17 contains the meta-package in line 4. The Yaml files have their own syntax. Errors are often caused by the indentation of the lines. Here, `meta: ^1.1.7` is indented, which means that it is hierarchically under `dependencies:`. By the way, the dart source code belonging to the package should be located in a `lib/` folder in the working directory. If you then execute the command `pub get` in the working folder, the package is automatically downloaded and installed from a cloud server, assuming you have installed Dart itself before, of course.

Pub is the Dart's own package management system. Such systems now exist for several programming languages or programming frameworks like Ruby, Python or Node. This simplifies the handling of external program libraries, since they do not have to be collected from the Internet in a time-consuming way. One problem, however, is the

version dependency of the libraries, for example, if package A is updated and a package B depending on it and no longer works, because it builds on the old functionality of package A. For larger projects, this can also cause significant problems with Flutter and Dart and leads to extensive editing of the `pubspec.yaml` file. This is because, to the downside, different packages may depend on the same packages, but on different versions. If the question arises what the caret symbol in line 4 Listing 2.17 means: It says that all main versions ≥ 1.3.0 should be used until the next main release < 2.0.0, and that the largest possible ones should be used.

The `pubspec.yaml` file lists the external libraries used. With the console command `pub get`, these are installed from the cloud to the local computer. An overview of all packages is available under `https://pub.dev/flutter/packages`.

Listing 2.14: Person-4.dart

```
import 'package:meta/meta.dart';

class Person {
  final String fname;
  final String lname;
  final int? age;
  const Person(this.lname, {this.fname: 'Unknown',
       @required this.age});
  get info => "Person $fname $lname is $age years old.";
  printInfo() => print(info);
}
```

Listing 2.15: main-4.dart

```
import 'Person-4.dart';

void main() {
  Person p = const Person("Meier", vname: 'Hans', alter: 23);
  p.printInfo();
}
```

Listing 2.16: meta-4.dart

```
const required = Required("No Arg");
class Required {
  final String _msg;
  const Required([this._msg = '']);
}
```

Listing 2.17: pubspec.yaml

```
name: Persons
description: A Test
dependencies:
    meta: ^1.3.0
```

2.2.10 Factory Constructor and Initialization List

There are other ways to write constructors. In Listing 2.18, you see in line 6 a factory constructor: It must have a return command which returns an object of the type of the class. Here another named constructor is called. Note that it is protected by the underscore in front of it. In line 10, you can see this constructor. There the field "radius" in the parameter brackets is initialized. Since the diameter and the area of a circle depend on the radius, it should be initialized as well. In the example, you see a possibility to do this without writing a body in which you do the calculation. After the colon behind the parameter brackets, you can write assignments. Several assignments can be listed separated by a comma.

Listing 2.18: Circle.dart

```
1  class Circle {
2    double radius;
3    double diameter;
4    double area;
5
6    factory Circle(r) {
7      return Circle._fromRadius(r);
8    }
9
10   Circle._fromRadius(this.radius)
11       : diameter = radius * 2,
12         area = radius * radius * 3.14;
13 }
```

Listing 2.19: main-6.dart

```
1  import 'Circle.dart';
2  void main(){
3    var c = Circle(10.0);
4    print(c.diameter);
5  }
```

2.2.11 Inheritance, Mixins and Interfaces

As there is inheritance in Dart, in Listing 2.20, line 7 a new class `Student` is declared, which derives Listing 2.14 from Person. Right here, you can read that another class is declared after `with`, namely `Learner`. The concept behind this is called "mixin." Mixins allow the use of certain functionalities next to the inherited functionality. In practice, this means that the included class `Learner` must not have a constructor. The class can be `abstract` (line 3), but this is not mandatory. Thus, you could omit `abstract` and also create objects from it. However, it is a good style not to treat the partial functionalities as independent objects. Methods of mixins are transferred to the class. In the example, objects of the class `Student` can call the method `learn()` from `Learner`, see 2.21, line 6.

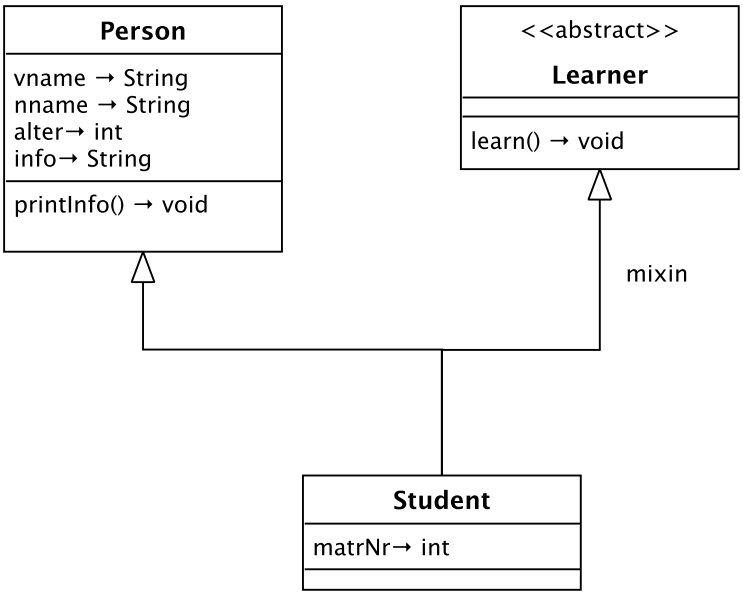
In line 10 of listing 2.20, there is a colon after the constructor parameter bracket, where the super constructor is usually called similar to C# or C++, with `super(...)` (line 12). You can also specify so-called "Assertions" here (and elsewhere in the code). The example checks whether the last name specified in the constructor exists.

---

i Mixins: There is the possibility of a kind of multiple inheritance. With the keyword `with`, you can include abilities and properties of further classes into the existing class. Objects of this class are then also of the type of these classes. However, they must not have a constructor, since the super-constructor of the mixins cannot be called from the subclass.

Dart also knows the keyword `implements`, like you may know it from other languages. All methods declared in the interfaces specified must be implemented in the class. In contrast to other languages, Dart does not know an own keyword `interface` with which you can specify these interfaces. In Dart, you can specify all classes as interfaces. In the sub-class all methods of the interface classes must be implemented. The implementations in the superclasses are no longer executed. So if you want to declare a pure interface like in Java or C#, you should use an abstract class instead.

There is the keyword `implements`, but no keyword `interface`. Interfaces are abstract classes in Dart.



Listing 2.20: Person-5.dart

```
import 'Person-4.dart';

abstract class Learner {
  void learn() => print("thinking...");
}

class Student extends Person with Learner {
  int regNr;
  Student(lname, fname, age, this.regNr)
      : assert(lname != null),
        assert(lname != ""),
        super(lname, fname: fname, age: age);
}
```

Listing 2.21: main-5.dart

```
1  import 'Person-5.dart';
2
3  void main() {
4    Student s = Student("Smith", 'John', 23, 123);
5    s.printInfo();
6    s.learn();
7    print(s is Learner);
8  }
```

2.2.12 Dynamic Extension of Types

In Dart from version 2.7 there are so-called extension methods. With the help of these extension methods, classes and all "primitive" types[7] like int and bool can be extended by methods and also getters and setters. It means that you can easily add features, without having to create a new subclass. In Listing 2.22, there are three methods that allow to convert int-variables into strings of different number systems. In line 1, you simply specify the method to extend int: `.toBinaryString()`, which should return a binary number. The expression "this" is always the object itself, in this case, the numerical value. The int class already provides a method `.toRadixString()` that returns a string in number system with the specified base. For dual numbers, a string with the own value in number system with base 2 is returned.

In line 7, a hexadecimal representation is implemented, in line 13 an Octal number representation. Here the methods are all named `.str()`. The extensions have their own names: When calling the extension methods in lines 21 and 22, the number is passed as a parameter of the corresponding names.

---

i Extension methods:

Dart offers the possibility to extend existing classes with methods, getters and setters: `extension [ <Name> ] on <Type> { <Methods> }`

---

7 But there are no primitive types in Dart like in other languages, because everything is an object.

Listing 2.22: extension.dart

```
extension on int {
  String toBinaryString() {
    return this.toRadixString(2);
  }
}

extension Hex on int {
  String str() {
    return this.toRadixString(16);
  }
}

extension Oct on int {
  String str() {
    return this.toRadixString(8);
  }
}

void main() {
   print (42.toBinaryString() );
   print ( Hex(42).str() );
   print ( Oct(42).str() );
}
```

2.2.13 Generic Classes

2.2.13.1 Type Parameter

Dart offers similar to Java also "Generics," thus type parameter. Since the feature is not used in the same way in all languages and the concepts may not be known to all readers, here a short introduction.

Suppose you want to represent pairs of similar things in a class, numbers (`double`) and strings (`String`). You can create a class for this purpose, as in the listings 2.23 and 2.24. But you can keep the definition more general in Dart, thus "generic," and use a type parameter variable, here called `T`, like in Listing 2.25. You can then use specific types in Listing 2.26, such as `double` and `String`. Then the specific type in the class definition is used instead of T.

Listing 2.23: PairDouble.dart

```
1 class PairDouble {
2   double a;
3   double b;
4   PairDouble(this.a, this.b);
5 }
```

Listing 2.24: PairString.dart

```
1 class PairString {
2   String a;
3   String b;
4   PairString(this.a, this.b);
5 }
```

Listing 2.25: Pair.dart

```
1 class Pair<T> {
2   T _a;
3   T _b;
4   Pair(this._a, this._b);
5   set a(T a) => this._a = a;
6   set b(T b) => this._b = b;
7   T get a => _a;
8   T get b => _b;
9 }
```

Listing 2.26: pairs.dart

```
1 Pair<String> st = Pair<String>("Dieter","Meiller");
2 Pair<double> d = Pair<double>(7.0,2009.0);
3 print(st.a);
```

2.2.13.2 Constrained Parametric Polymorphism

But now the type parameter concept is too open, so Dart offers the possibility to restrict the types with `<Variable> extends <Type>` as in Listing 2.28, line 1. In the particular example, it gets clear why this is needed: You would like to have a definition of the concept of a bottle box (inspired by [57, p. 32]), which can only be filled with bottles and not with any other things (objects of other types in programming). Here it is specified that the box can only ever contain bottles, whereby the type "Bottle" is abstract (Listing 2.27, line 1). Then a `Bottle` of the type `BeerBootle` can only be filled with specific bottles (see 2.29). In Dart, unlike in Java, you cannot restrict the types

with `<variable> super <type>` in the other direction of inheritance and specify that type T should be a super type of another type.

Generics in Dart are similar to those in Java. `<type variable> [ extends <supertype> ]` However, you cannot make a restriction down the inheritance hierarchy like in Java with `super`.

Listing 2.27: Bottle.dart

```
abstract class Bottle {
  int get capacity;
}

class CokeBottle implements Bottle {
  int get capacity => 75;
}

class BeerBottle implements Bottle {
  int get capacity => 100;
}
```

Listing 2.28: Box.dart

```
class Box<T extends Bottle> {
  List<T> bottles;
  Box(this.bottles);
}
```

Listing 2.29: Boxes.dart

```
var bottles = <Bottle>[
  BeerBottle(), BeerBottle(), CokeBottle()
];
var box = Box<Bottle>(bottles);
var obj= Box<Object>([]); //Not working
print(box.bottles);
```

2.2.14 Asynchronous Operations

An important feature in Dart is the asynchronous execution of functions. This makes it possible to wait in a thread, thus an execution thread, for the occurrence of an event without blocking the application. In Listing 2.30, a function is implemented which

creates a new file in line 5 and then waits in line 6 until the file has actually been created. The time needed for this operation is not constant, because it is not possible to tell where the medium is located as it could be on a remote drive in a cloud. Also the writing speed of different drives varies. This operation is representative for many operations in this example. Note that this function does not return a String object, as line 7 suggests, but a future object of the type String (line 4).

Future objects refer to practically a future that can or will occur. All future objects have a method `then(...)`. In normal language, you could express it like this: After the occurrence of ... do this or that. In the calling code in Listing 2.31, the method `.then(..)` of the future object is called, which has an anonymous function. This is called only after the calling function (`getFilePath(...)`) has been completed. As argument, it is then passed a parameter of the type of the future object (here `String`) and can process it in the passed function.

It is interesting that the call in line 2 does not wait (block) until the operation as a whole has ended. Line 5 is called immediately, only then line 3 is executed, unless the operation for creating the file (in the example) is faster than the program execution and the transition to line 5.

As mentioned, errors can also be handled. If something did not work, you can handle it with the method `.catchError(...)`. So this method is executed instead of the `.then(...)` method. Another optional `.whenComplete(...)` method is executed in any case, if available.

---

i Asynchronous methods are marked with `async`. The value of the type `T`, which you want to return with `return x`, is then returned as `Future<T>`. It has a method `.then( (T x) {...}` , where you specify a method to be called when the asynchronous operation, that can be waited on with `await`, has been performed. A method call waited on with `await` must be inside an asynchronous method. The called method must also be an asynchronous operation.

Listing 2.30: getFilePath.dart

```
1 import 'dart:async';
2 import 'dart:io';
3
4 Future<String> getFilePath(String name) async {
5   final file = new File("$name");
6   await file.exists();
7   return file.path;
8 }
```

Listing 2.31: callGetFilePath.dart

```
1 void foo() {
2   getFilePath('Foo.txt').then((String path) {
3     print(path);
4   });
5   print("Before or after");
6 }
```

2.2.15 Generators: Iterators and Streams

The use of Co-Routines is an old concept, which was present in the object-oriented programming language Simula, but those are now being used again in other programming languages. These allow to return from a function, but to continue the execution at the same point at the next call. In Dart, there are also generator functions, which are an extension of this concept: These generate a sequence of results, but not all at once, but "lazy", thus on demand.

In the function `numbersTo(n)` in Listing 2.32, a *while* loop counts up a counting variable `k` to `n`. Note the keyword `yield`, that, like `return`, returns the following expression. In contrast to a normal function with `return`, however, the execution is continued at this point until the condition `k < n` is no longer fulfilled. After the return, the system always waits one second (line 7). The function is marked as iterator with `sync*` (the asterisk means generator function). Just like `async` and `await` belong together, `sync*` or `async*` and `yield` also belong together. The return value is a `Iterable<int>`. In contrast to a list, which is first filled using a loop and then passed through, iterators can in principle have an infinite number of values, because the following value is always formed at the next access.

In Listing 2.33, the output 2.34 is generated. First, "Start" and "0" are output immediately. Then after one second, each "1, 2, 3". After two more seconds then "2." In line 7, `numbersTo(50000000)` describes a high number of data, but only the third element is requested.

There are normal "synchronous" functions, asynchronous functions with `async`, and generator functions with `sync*` and `async*`.

Listing 2.32: numbersToSync.dart

```
import 'dart:io';

Iterable<int> numbersTo(n) sync* {
  int k = 0;
  while (k < n) {
    yield k++;
    sleep(Duration(seconds: 1));
  }
}
```

Listing 2.33: callNumbersToSync.dart

```
import 'numbersToSync.dart';
void main() {
  print("Start");
  for (var i in numbersTo(4)) {
    print(i);
  }
  print (numbersTo(50000000).elementAt(2));
  print (numbersTo(2).runtimeType);
  print("Ready");
}
```

Listing 2.34: Output in terminal

```
Start
0
1
2
3
2
_SyncIterable<int>
Ready
```

Listing 2.35 shows the asynchronous version. The return value here is `Stream<int>` and the function is marked as `async*`. In Listing 2.36, line 3, the function waits for every single element of the stream. A stream consists of single future elements in this case, of future elements of the type `int`. There are only two things to do with future elements: either you wait for them with `await`, or you tell them what to do when they are there with `then(...)`, where you have to pass a function as parameter, which is then called, as already described.

The program now has the following sequence: In the main function, first the numbers 1-3 are output immediately, as shown in the output 2.37, "Start," the type, "Done." and "0," then the numbers 1-3 after one second. The call in line 10 now causes "2" to be output with a delay of one second, then "4."

Listing 2.35: numbersToAsync.dart

```
1  import 'dart:async';
2  import 'dart:io';
3  Stream<int> numbersTo(n) async* {
4    int k = 0;
5    while (k < n) {
6      yield k++;
7      sleep(Duration(seconds: 1));
8    }
9  }
```

Listing 2.36: callNumbersToAsync.dart

```
1  import 'numbersToAsync.dart';
2  void callNumbersToAsync() async {
3    await for (int i in numbersTo(5)) {
4      print("Number $i");
5    }
6  }
7  void main() {
8    print("Start");
9    callNumbersToAsync();
10   numbersTo(5).elementAt(2).then((n) => print(n));
11   print(numbersTo(2).runtimeType);
12   print("Ready.");
13 }
```

Listing 2.37: Output in terminal

```
Start
_ControllerStream<int>
Ready.
Number 0
Number 1
Number 2
2
Number 3
Number 4
```

2.2.16 Null-Safety

Dart is under intense development. In contrast to Java, where backward compatibility is always ensured, the designers of Dart have the courage to introduce innovations that result, that existing code is no longer executable. In this way, the Dart language can be adapted to the latest trends in programming language development, whereas innovations in Java are only introduced with great caution, so that the language lags behind other languages. The disadvantage of this approach, however, is, that existing code no longer works, and you have to rewrite it. It must be made sure, that libraries are used, that correspond to the new standard. These, in turn, must be made consistent with the libraries used there. A chain of dependencies results, so that the new standard cannot be used immediately, developers must wait until all dependencies correspond to the new standard. This is also the case with null-safety, which was introduced in version 2.12 of Dart. There is a recommendation on the Dart homepage on how to proceed when migrating to new versions, see [16]. Here, a short introduction to null-safety is given (see [17]), which also illustrates, what it means when language features change fundamentally. It is expected, that further future improvements of Dart will have similar effects on programming practice.

A variable has the value *null*, if it has been declared, but not yet assigned a value. This may simply have been forgotten, but in practice this gets problematic, if you retrieve the value from a data source, that is not deterministic. For example, if you read a file, this may fail, and the result is not a file, but *null*. At runtime, an exception handling must be provided here. With null-safety, this problem can be handled with static type checking. In interactive programs, like apps, there is no way to ensure, that variables never become *null*, but it is desired to avoid this. The basic idea with null-safety is, that variables must be declared explicitly, if they are allowed to take the value *null*. Syntactically, this is expressed in Dart by adding a question mark (?) to the type. In line 8, in listing 2.38 there is a declaration and definition of the integer variable `x1`. Since null-safety is enabled, it is necessary to ensure, that there is always

a value assigned to the variable. In line 10, the declaration of the variable `x2` can be seen, a value is not assigned to it. Because of the question mark after the type, this variable may explicitly have the value *null*. In line 13, you can see a list type `List<int>`. A list must always be assigned to the variable `x3`, also no element of the list must be *null*. In line 14, the variable `x4` does not have to be initialized. But if it contains a list, no element of the list must be *null*. In line 15, the variable `x5` must be initialized with a list, but its elements may be *null*. Null-safe variables must contain a value, before they can be used for the first time. If you omit line 17 and do not assign a list to the variable `x6`, you will get this message during the static check:
`Error: Non-nullable variable 'x6' must be assigned before it can be used.`
In line 20, an object of class *Foo* is stored in a nullable variable. Since the *bar* property of the class is not initialized after creating the object, it must be explicitly marked with `late` (line 2). Reading and writing of the field must be done with an extension of the point operator `?.`, since the variable *f* itself is nullable. Properties of the object can only be accessed if the object itself is not *null*. Thus, the expression `f?.bar` returns the value of *bar*, if *f* is not *null*, otherwise *null*. However, the static type check recognizes here that the variable has been initialized. Thus, the specification of the question mark is superfluous here (line 22). In classes, you can declare null-safe variables as usual if you give them default values in the constructor, ensuring that they have a value (line 4). Previously, if a property of a null-object was accessed, a runtime error was generated. With null-safety, static type checking is intended to limit such errors as much as possible. In line 25, a null-safe variable is assigned the value of a null-able variable. It can be done explicitly, using the exclamation mark (`!`) after the variable. Completely dynamic variables can still be *null* (line 26).

Listing 2.38: nullable.dart

```
1  class Foo {
2    late int bar;
3    int baz;
4    Foo([this.baz = 23]);
5  }
6
7  void main() {
8    int x1 = 42;
9    print(x1);
10   int? x2;
11   print(x2);
12
13   List<int> x3 = [1, 2, 3];
14   List<int>? x4;
15   List<int?> x5 = [1, 2, 3, null];
16   List<int?> x6;
```

```
  x6 = [1, 2, 3, null];
  print(x6);

  Foo? f = Foo();
  f.bar = 42;
  //print(f?.bar);
  print(f.bar);
  x2 = 23;
  //x1 = f.baz!;
  var x7;
  print(x7);
}
```

This was a short overview of the special features of the programming language Dart. It is the foundation for programming with the app development framework Flutter, which the rest of the book is about. Additional information can be found in the documentation on the Dart homepage [13].

3 Tools

In this section, we discuss the tools that are necessary and useful for development with Flutter. The installation steps are discussed, and hints for necessary configurations provided. It should be noted that the principle and the interaction of the components is the main focus here. For more detailed instructions, refer to the given sources available online.

3.1 Installation

You need to fulfill a number of requirements to work with Flutter. Thus, the following software is necessary:

- Dart
- Flutter SDK
- Native SDK (Android and/or XCode)
  - for Android
    - Android SDK
    - Java SDK
  - XCode via App-Store
- Flutter
- Code editor
  - Android Studio and / or
  - Visual Studio Code and / or
  - Another editor
- Test devices iOS and / or Android
  - iOS emulator (with XCode) and/or
  - Android device emulator and/or
  - iOS device and/or
  - Android device

https://doi.org/10.1515/9783110721331-003

3.1.1 Additional Software

It is also suggested that at least the following software is available or installed to develop apps:

- Git (Versioning and repository)
- Image editing software
  - Gimp or Photoshop
  - Illustrator or Inkscape
- Sound editing software
  - Adobe Audition or Audacity

Git can be used to manage your own software development, thus to do versioning and backups. It is also very useful to obtain software from the Internet. If not already present on your computer (e.g., Windows), you can find installation instructions here [32].

Since this is about programming graphical user interfaces and games, image editing software is necessary. Installing this is also recommended for the developers, because you often have to make adjustments to images. And, it depends on whether you realize a hobby project or work in a company in a team. However, it is a good idea not to have to hire a designer for every small change. For example, you have to provide icons and banners of different sizes when you publish (deploy) the app. The choice of professional graphic designers here would be Adobe products, including Photoshop and Illustrator, which are available in the Creative Cloud (CC) Software Suite [1]. In addition to the drawing programs, other graphics software such as 3D animation programs or video editing software can also be used. In the professional sector, Premiere and After Effects can be recommended for video editing (also Adobe Creative Cloud). During deployment, preview videos of the app can be uploaded to the store. The task of the professional developer would be to convert the pre-produced video into the appropriate format. A sound editing program would also be useful; an option is Adobe Audition. Suitable prototyping software could be used during the planning of the project, so there is Adobe XD in Adobe CC for the creation of click prototypes. In the 3D area, the software of the company Autodesk is established with 3ds Max and Maya [6].

3.1.2 Free Software, Resources and Licenses

However, you can also (as a developer or graphic designer) fall back on very established open source software. It is worth taking a look at the licenses of the software you use, because there are different terms of use. Some software may only be used for non-commercial projects. This also applies to educational versions of some of the graphic programs mentioned. Certain software creates digital watermarks in the generated images or movies, and special companies search the web for such products for

the purpose of issuing a warning. Therefore, as a developer, whether a hobbyist or professional, you should be aware of the modalities of using your own software. There is a lot of free software and a lot of different open source licenses. The best known and oldest license is the GNU license, which originally had the goal to realize a free Unix platform [33]. However, there is other software under this license that is not directly related to the operating system. For image processing, Gimp is recommended, it offers most of the important functions of Photoshop [31]. For vector graphics, there is a sister program called Inkscape [45]. For sound editing, you can use Audacity [65].

Especially when it comes to sound, the question arises where to get the content from. With all resources that are used, you have to think about whether they can be used and under which conditions. Much creative content, like images, sounds and fonts are made available for free use under a Creative-Commons license [11]. Please note the differences in the licenses. Often you have to mention the name of the author when using the content. In this case, you should list the licenses and the resources used on an extra page in the software (About...). Sometimes it is easier if you create the content all by yourself, like it was done in the example project in chapter 7, which is not easy, especially with sound. You need sound effects for the game. You could record these yourself and use Audacity to modify them. Free background music is not so easy to get. In the example project, the music was created by the synthesizer software Sonic Pi [60]. This allows you to create music algorithmically using program code in the Ruby programming language, which meet the taste of many developers. The results can then be cut with Audacity. The situation is similar with graphic resources. Graphics licensed under Creative Commons that are suitable for Flutter projects can be found on the website Open Game Art [58]. Free fonts can be found on the web- Google Fonts [38] offers high-quality fonts and is comfortable to use.

For projects that are published, you have to pay attention to the licenses. It has to be clarified under which conditions tools, software libraries and content can be used. Often only non-commercial use is allowed. An important software license is GNU [33]. For creative content there is the Creative Commons License [11].

3.1.3 Installation of the Flutter Framework

The easiest way to install Flutter and all the tools you need is to install Android Studio or Visual Studio (the full version, not VS code) and the Flutter plugin for Android Studio and install the device simulators through it.

Flutter itself can be downloaded and installed manually. The instructions can be found at [27]. The problem is that you still need the Android or iOS developer tools. The easiest way to get them is to install Android Studio (or Visual Studio) or XCode on the Mac. A manual installation of the Android Software Development Kit (SDK) is not advisable: You would have to install the complete software stack, first the correct

Java version, the Android SDK, then a device emulator, thus the cell phone simulator. The author made such experiments under Linux and had to get, compile and install additional software libraries, since these are necessary to start the emulator and the SDK. Sometimes there were conflicts with existing libraries. This effort and the danger of making your computer unusable can be avoided.

Fig. 3.1: Android Phone Emulator

Assuming that a native developer SDK has already been installed, Flutter can also be installed independently without an integrated development environment (IDE) by directly obtaining the framework. This can have advantages if you prefer a different editor. You can execute Flutter commands without IDE. For instance, there are programmers who appreciate VI or EMACS or Sublime Text as editors. They do not have to use the default editors. For Mac and Linux, change to the home directory, make sure that Git is installed First. This is where the stable version of Flutter is cloned into:

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

In order to be able to enter the Flutter command in Terminal, the path variable in the Shell environment must be adjusted. However, it depends on which shell you are using. The Bash is the default shell in older macOS operating systems and usually in Linux. Starting with the macOS version Catalina, the default shell is the Z-Shell. You have to open the hidden resource file of the corresponding shell, thus in the HOME directory (~) the file ".zshrc" or ".basrc". Here you have to enter `export PATH="$PATH:~flutter/bin"`. It is important to restart the terminal afterwards.

The HOME directory (~) on Linux and Mac is the directory `/users/<USERNAME>`. There you will find the configuration files for the shell. The hidden files are not visible in the terminal or via the file browser. Under the Mac Finder, you can make them visible by pressing "Shift+CMD+." and then change them with the standard text editor. Under Linux, you can run the Nano terminal editor in superuser mode to do this: `sudo nano ~/.bashrc`.

Then you should be able to run a diagnosis via command line and then create and start a flutter example project.

```
flutter doctor
flutter create example
cd example
flutter run
```

3.1.4 Installation of Android Studio

Depending on the operating system you are developing on, you have to follow a separate installation guide. Under the link `https://flutter.io/get-started/editor/` [27], you can read the instructions. Visual Studio Code is also supported. VS Code is suggested here as it is found to be more user friendly than Android Studio. However, it is recommended to install Android Studio first and Flutter with it, because sometimes (especially with Linux) there can be problems during installation. You can still install VS Code and work with it afterwards. The installation process is described briefly below.

Android Studio and Visual Studio Code are available as development environments for Flutter.

- Install Android Studio (installer at: [18]).
- Install Android SDK: You should also install an Android SDK under Tools ⇒ SDK Manager.
- You also need a virtual device for testing. Under tools ⇒ AVD Manager ⇒ Create Virtual Device you can create a new device.

- Open Android Studio. File ⇛ Settings ⇛
  Plugins Browse repositories..., search Flutter, click install. (Yes for "Dart" plugin; confirm during installation). Then you have to specify the path to the missing Flutter SDK.
- So you have to download it (as described via Git) or by download:
  `https://flutter.io/get-started/install/`
  and unzip it into a folder, e.g., /home folder or programs... (The SDK is the largest download chunk)
- After the installation, you should create a new flutter project in Android Studio, Enter the path of the SDK there. Only when it points to the folder "flutter" in the unzipped folder, you can click "continue."
- You should not delete the project, otherwise the settings will be removed from Android Studio.

3.1.5 Installation of Visual Studio Code

The next step can be to install VS Code via an installer [53]. Afterward, you have to do some configurations to work with Flutter. You can search the extensions and reach them via the symbol with the four squares in the right sidebar (see figure 3.4). There you can search for the extensions. You have to install the Flutter plugin and if necessary the Dart plugin.
With this guide, you install the development environment, the Flutter-SDK and the Android-SDK for the development of Android applications. If you intend to develop applications for iOS, which is an advantage of Flutter, you have to install XCode. In this case, however, you have to use a Mac computer to create the native file for the App Store. It would be possible to program on a non-Mac machine under Windows or Linux and test the Android app via the emulator and then create the `.app` file on another machine under macOS and XCode, which is most effective if you work in a team. This way you could configure a special Mac compile-computer that is shared by several developers.

3.1.6 Creating and Starting Projects with Android Studio

The following is a description of how to create and start projects. The configuration is first explained for Android Studio and an Android app. But the principle is the same, no matter if you work with Android Studio, VS Studio Code or just from the command line. Since Android Studio guides you through the workflow, it will be explained first. After everything is configured, you can create flutter projects: Under New ⇒ New Flutter

Fig. 3.2: Android Studio: Flutter Plugin

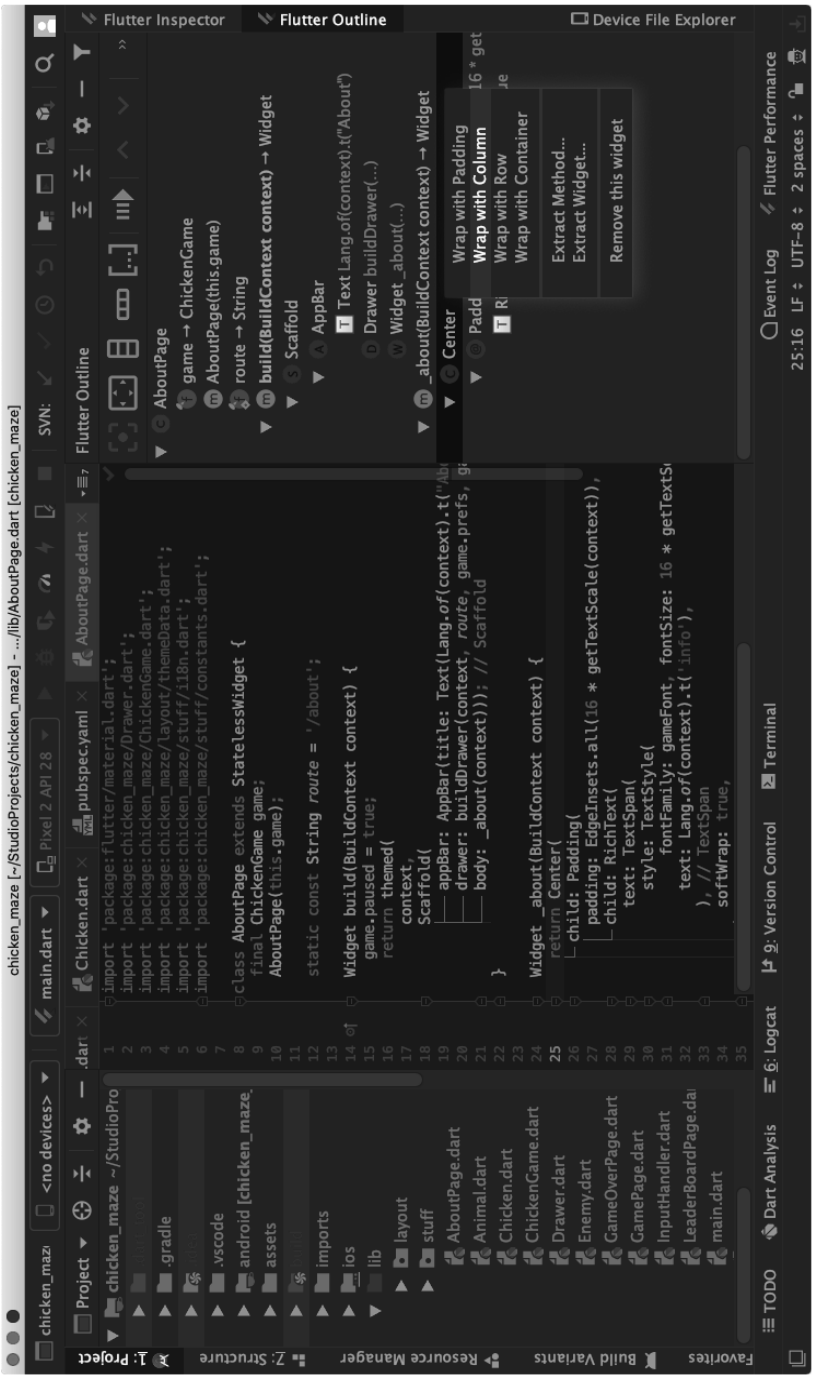


Fig. 3.3: Android Studio: Workspace

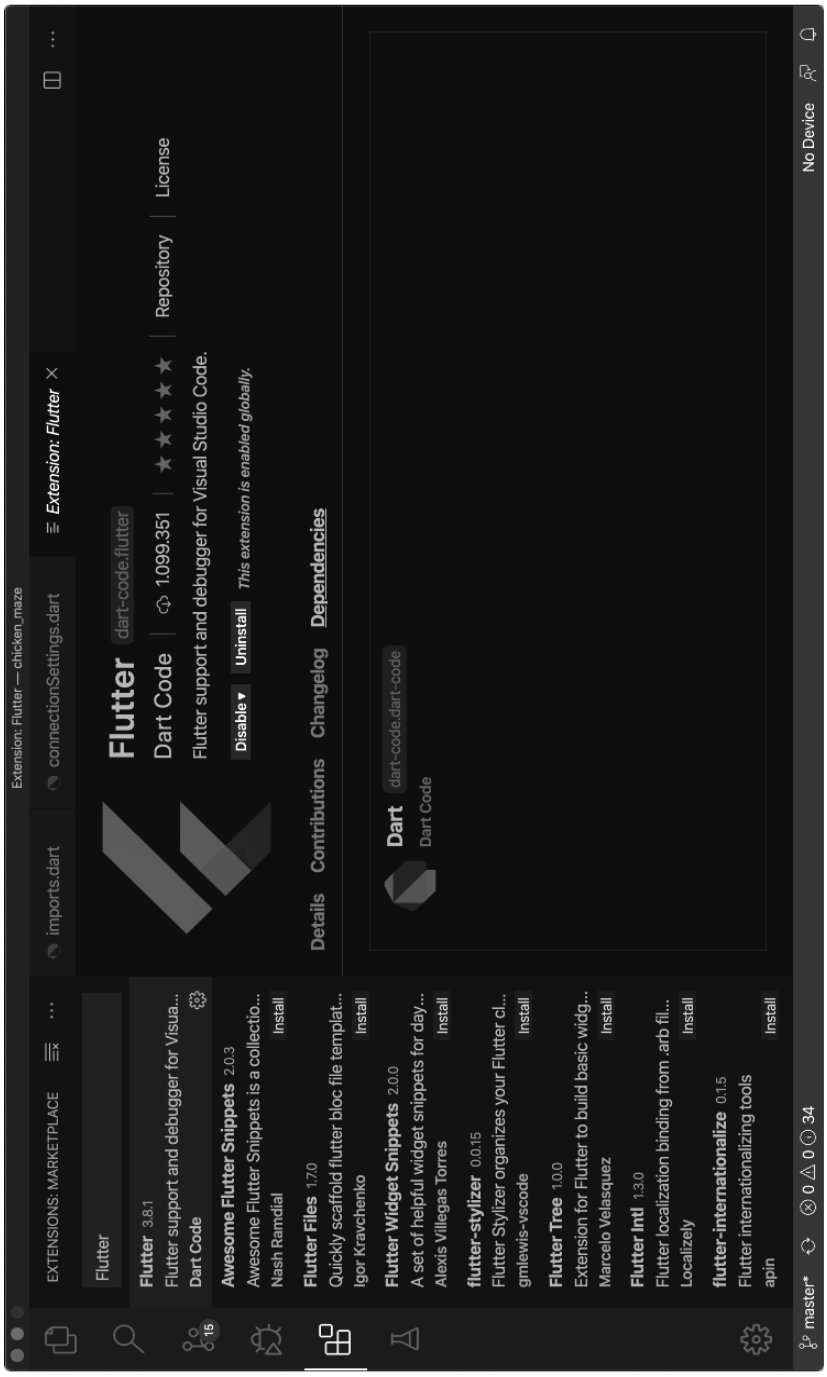


Fig. 3.4: Visual Studio Code: Flutter Plugin

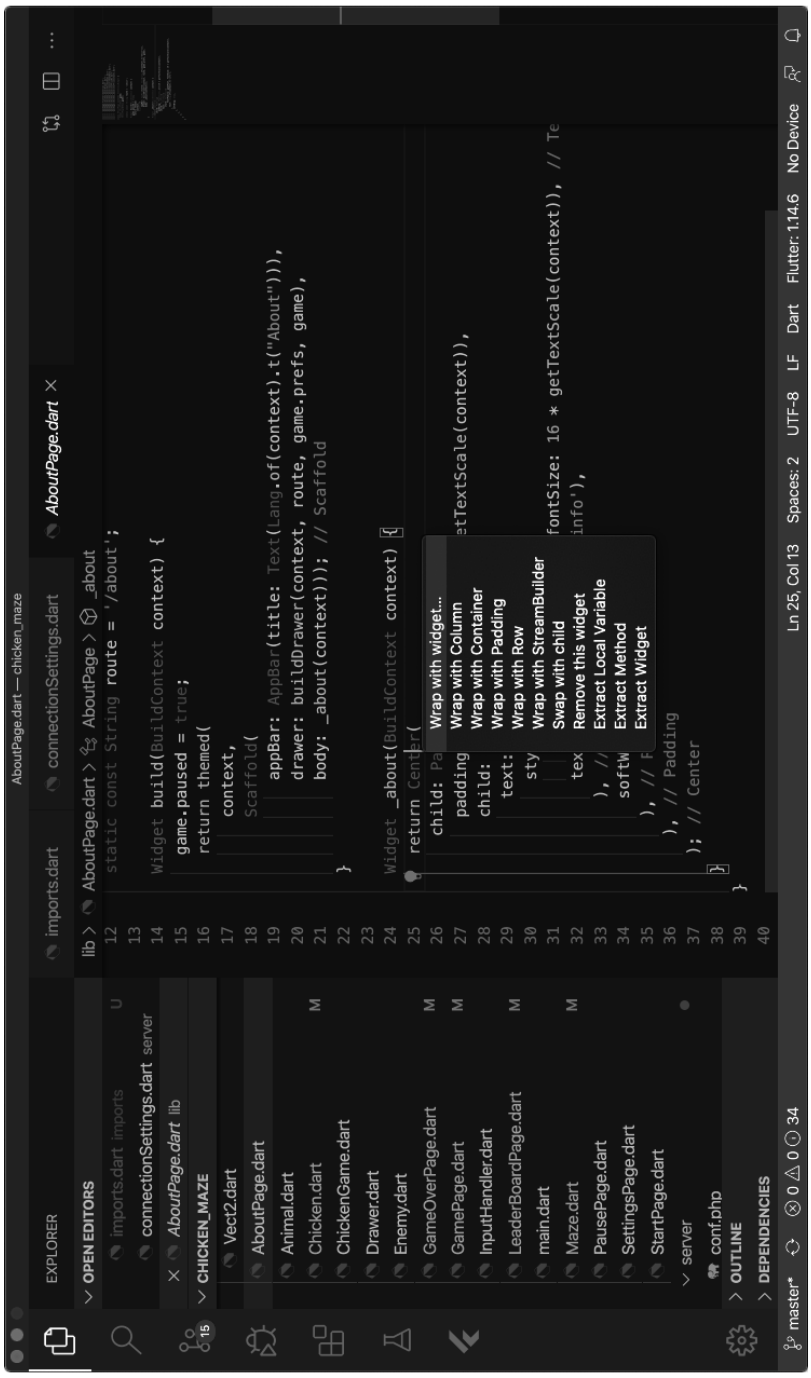


Fig. 3.5: Visual Studio Code: Workspace

Project, you can choose between a Flutter application, a Flutter package, a Flutter plugin, and a Flutter module. To create an app, you should create a Flutter project. The other options are for the programming of components. The section 4.4.5 describes this in more detail. And in the section 4.4, it is introduced how you can program your own packages. The difference is, as you can read on the selection menu page, the following:

A plugin contains native code, thus code in Swift or Objective-C for iOS and/or code in Java, or Kotlin for Android. It is used to implement new features that make special capabilities of the device available to Flutter. With this plugin, you can program an interface to the Dart world. A package is also a flutter component, but it is programmed only in Dart and builds on existing flutter components. More complex GUI elements are made available in a reusable way. Flutter modules are then still components implemented in Flutter, but can be used in the native world, for example in an Android Java project.

If we now create a new Flutter project, a Flutter application, we are first asked for the name of the project. It is important to note that the project name is written in lower case and words are separated by an underscore "_". You can also specify the SDK path again. Android Studio then creates the correct project structure: A folder "ios" where the native iOS project is located, and a folder "android" where the native Android project is located. There is a folder "test" which contains the unit tests. Android Studio has already created a file "widget_test.dart" here. The actual project code is located in the folder "lib." Here you can find the file "main.dart." It is a working example layout. If you use version control (Git), the files that are not checked in are shown in red.

You can start the project by clicking the debug button or the play button. You should either start the device simulator in advance or connect a test device via USB.

Under Windows you still have to install the Google USB driver to access a hardware test device. At Tools ⇒ SDK Manager ⇒ SDK Tools, you have to check the box "Google USB Driver". You can also install the driver manually [19].

If you are using an Android test device, you must first enable the developer mode. This is done by tapping the build number seven times under the settings and the "About the phone" info point. After that, there is another item called Developer Options under the Settings item. Here you have to select "USB-Debugging." If everything works, after starting the debugging, the APK file, which is the compiled and with the resources packed native app, is loaded and installed on the device.

APK files are ZIP-compressed folders that contain the entire project, but the code is compiled.

During further development, you can change the created "main.dart" file as you wish. Please note that the code in the file "widget_test.dart" has to be changed as well or can be deleted at the beginning as the tested components do not exist anymore.

The editor of Android Studio offers all kinds of features, a detailed description would go too far here. Figure 3.3 shows the working environment of the editor. Here the project from section 7 is opened. On the left is the file browser, in the middle the tabs with the opened code files and on the right a structural view of the layout in the current file. At this point, you can also refactor the layout structure (Refactoring): If you right-click on a node, you can enclose it with a layout element, which you can select from a list.

There is no visual layout editor for Flutter. However, there are external programs, such as a desktop application [47] and an online application [64]. These are freely accessible software.

A special and pleasant feature of Flutter is the so-called "Hot Reload": If you change the layout and save the file, the code on the device is updated without noticeable delays while the debug session is running, without the need to stop the debug session, recompile the code and upload it – this can save a lot of time. However, experience has shown that this is only helpful when layouting. Deeper changes to the structure of the code are not updated, so it is not always possible to avoid the mentioned step of recompiling.

3.1.7 Creating and Starting Projects with Visual Studio Code

Flutter projects can also be created with Visual Studio Code, assuming that it is installed correctly. To do this, you can use the command palette of VS Studio. To do so, press CTRL+Shift+P on Windows or Linux and CMD-Shift+P on Mac. An input field appears at the top of the development environment. Here you can enter "Flutter: New Project". While typing, suggestions are displayed, which can be selected with the cursor keys and confirmed with ENTER. After that you have to enter the name of the project. Ensure that you choose the name according to the conventions, thus lower case with underscore as word separator. After confirming with ENTER, select a folder in which the project will be created. Afterwards a sample project is created there, it is the usual "Hello World" project of Flutter, a scaffold with a button that increments a counter.

To start the project, press the F5 key[8]. After that you have to create a new emulator or select an already created emulator, in which the project will be uploaded after compilation. If you have connected a device directly to the computer, you can also select this device. With Shift-F5, you can stop the program. Hot reload works here as well, when saving changes, the app will be updated on the device. When the app is running, VS-Studio Code displays a control bar, which allows you to control the app. Here there is a blue button to display the "DevTools Widget Inspector Page" in the browser (see section 3.1.9).

8 On Mac or other computers FN-F5, because the F5 key may already be assigned differently by the system.

In figure 3.5, you can see the opened project and the upper left corner the open files, below the project folder. Files not checked in to Git are shown in yellow. On the right, there is a mini view of the code, so you can keep track of longer files. In the middle are the tabs with the open files. You can scroll through these files with CMD-Alt -Cursor left or -Cursor right. Visual Studio Code offers many features whose exact description would go too far here. The refactoring is an interesting feature: If the cursor is over a widget, you can press CTRL+Shift+R or right click ⇒ Refactor, then you get a drop-down menu like shown in the picture. You have several options like "Wrap with widget...," which allows you to change the structure of the layout tree. This is manually rather complicated, because you always have to change the closing brackets. This can lead to syntax errors that are not easy to fix.

3.1.8 Creating and Debugging Projects from the Command Line

You can also work well without a development environment with an editor of your choice and the command line. You can display all options of the Flutter command in the console (line 1). Line 2 displays all options of the "create" command. Line 3 creates a flutter project, as the organization you specify a domain, `-a` creates an Android project (`-i` would create an iOS project additionally). Then you can give a description. In line 4, you switch to the project folder. Line 5 starts the emulator, line 6 finally executes the project.

```
1 flutter help
2 flutter create --help
3 flutter create --org de.meillermedien -a java --description 'Test App
      ' my_test_app
4 cd my_test_app
5 flutter emulators --launch Pixel_2_API_28
6 flutter run
```

Debugging and Hot Reload are also available for console-based development: If you press the "r" key while the process is running on the console, the hot reload is performed. The console also outputs a web address of the type `http://Localhost:Port/<id>` at the start of the project, where you can find a profiler and debugger. On the right side of the web page, a link to the debugger and an object hierarchy view can be found. The debugger allows you to debug using commands from a web-based console. For example, you can run the command `break <line number>` to add a breakpoint.

3.1.9 Dev-Tools

The browser-based DevTools only work in the Chrome browser. You cannot use any other browser for it, because only Chrome has a Dart VM. These are started via Android Studio and via VS-Studio Code by clicking the corresponding button in the command palette. They can also be started via the console:

```
flutter pub global activate devtools
flutter pub global run devtools
```

Then the URL is displayed where you can reach the DevTools via a local server. You can view them in the Chrome-Browser (see figure 3.6). Here you can select a widget on the device and display the render tree. You can also view the Performance and log output.

3.2 Tiled-Editor

There is a useful editor to create large game levels as you can handle them later in the flame engine (see section 7.1.1.1) "Tiled" (Figure 3.7), an open source project, see: [48].

The editor has a drawing area in the middle, in the upper right corner, you can create single layers. On the lower right, you can see the "tilesets" with the tiles you can draw. From these templates, you can select single tiles and use them to pave a large map. The tilesets are not single graphics, but images with fixed width and height that are mounted together in a graphic. You have to define these dimensions (figure 3.7: top-left). When designing the graphics, you should make sure that they fit together in different combinations at the edges. The number of tiles of the whole map in horizontal and vertical direction has to be specified as well.

For later work with Flutter and the Flame plugin (see 7.1.1.1), it is important to set the layer format to base64, zlib compressed. Otherwise, the file cannot be loaded (see figure 3.7). Additionally, the tile numbers are not written to the file in CSV format (Comma Separated Values), but compressed, which saves disk space. (See screenshot 3.7 on the left for the setting).

3.3 Rive and Nima

With Rive and Nima, two animation environments from the platform of the same name like the company Rive, [61] graphics can actually be animated as it was done in Adobe Flash [2] via a timeline. Unlike Flash, the animation software runs only in the browser and can be used free of charge as long as you make your project available to the public. Otherwise, you have to use the commercial version. The entire project is also stored in the cloud. You have to decide what kind of graphics you want to animate; vector or pixel graphics. If you choose vector graphics, you have to use Rive, otherwise Nima. You can

then upload your graphics to the project using drag and drop and then animate them using a timeline. The result can be downloaded as a ZIP file and used in the project. You can give each animation a name and use it to control the animation in the program (see example in section 4.4.4).

At the bottom right of figure 3.8, you can see the two animations “shot” and “fly” used in the example. You can add more animations here. At the bottom in the middle is the timeline, where you can create keyframes at a desired time. You can choose between the keyframe (blue circle) and a curve view to control the time transitions between the keyframes with Bezier curves. There is also a symbol with circular arrows; if you activate it, the animation will be played as a loop; even after export. In the upper left corner, you can switch between the animation view and the setup, thus the positioning view. When you completed the process, you can export the animations for Flutter (Export to Engine) by clicking on the export icon in the lower right corner (icon with arrow out of box). The format is “Generic.” The handling is similar to Adobe AfterEffects. Animation and compositing professionals should, therefore, find their way around without much effort.

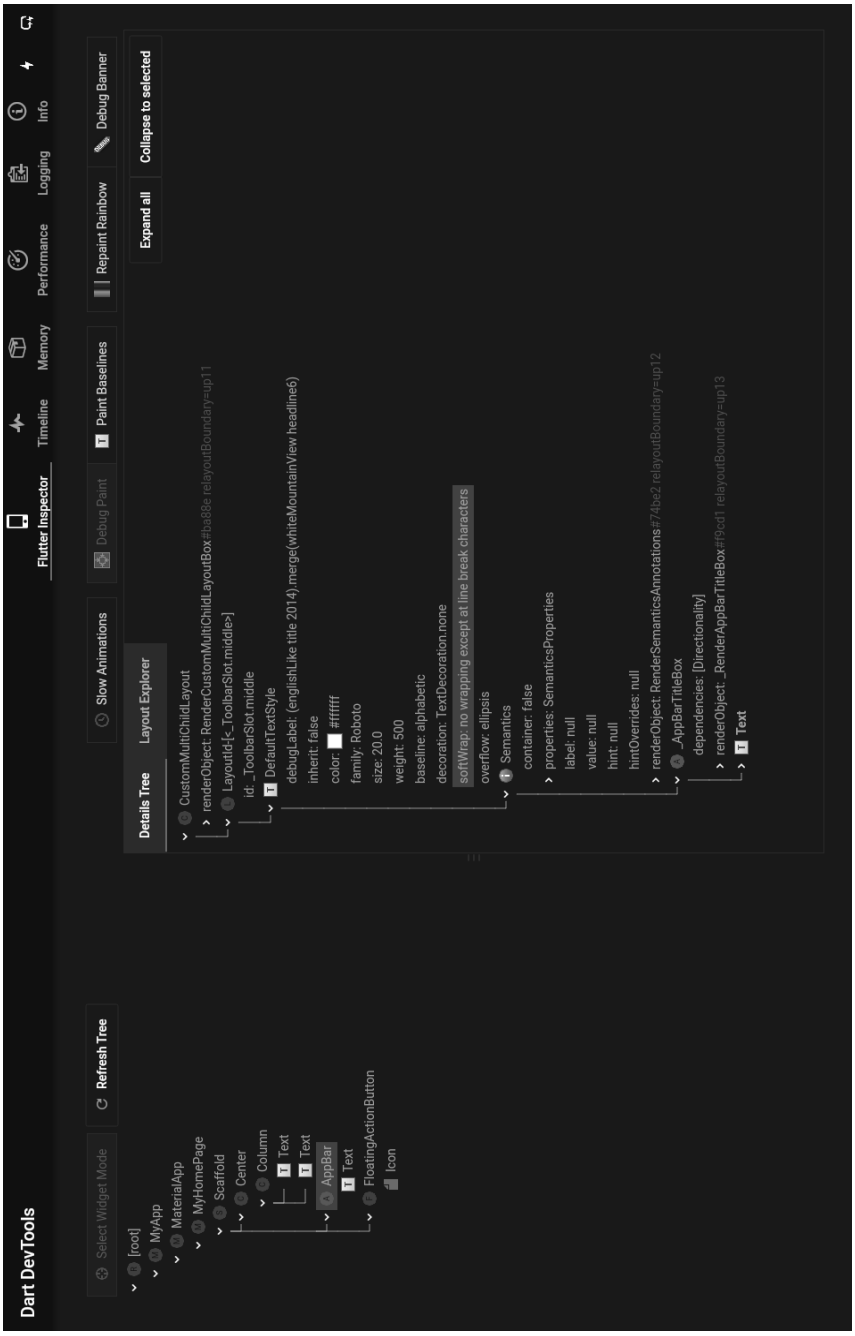


Fig. 3.6: Dart DevTools

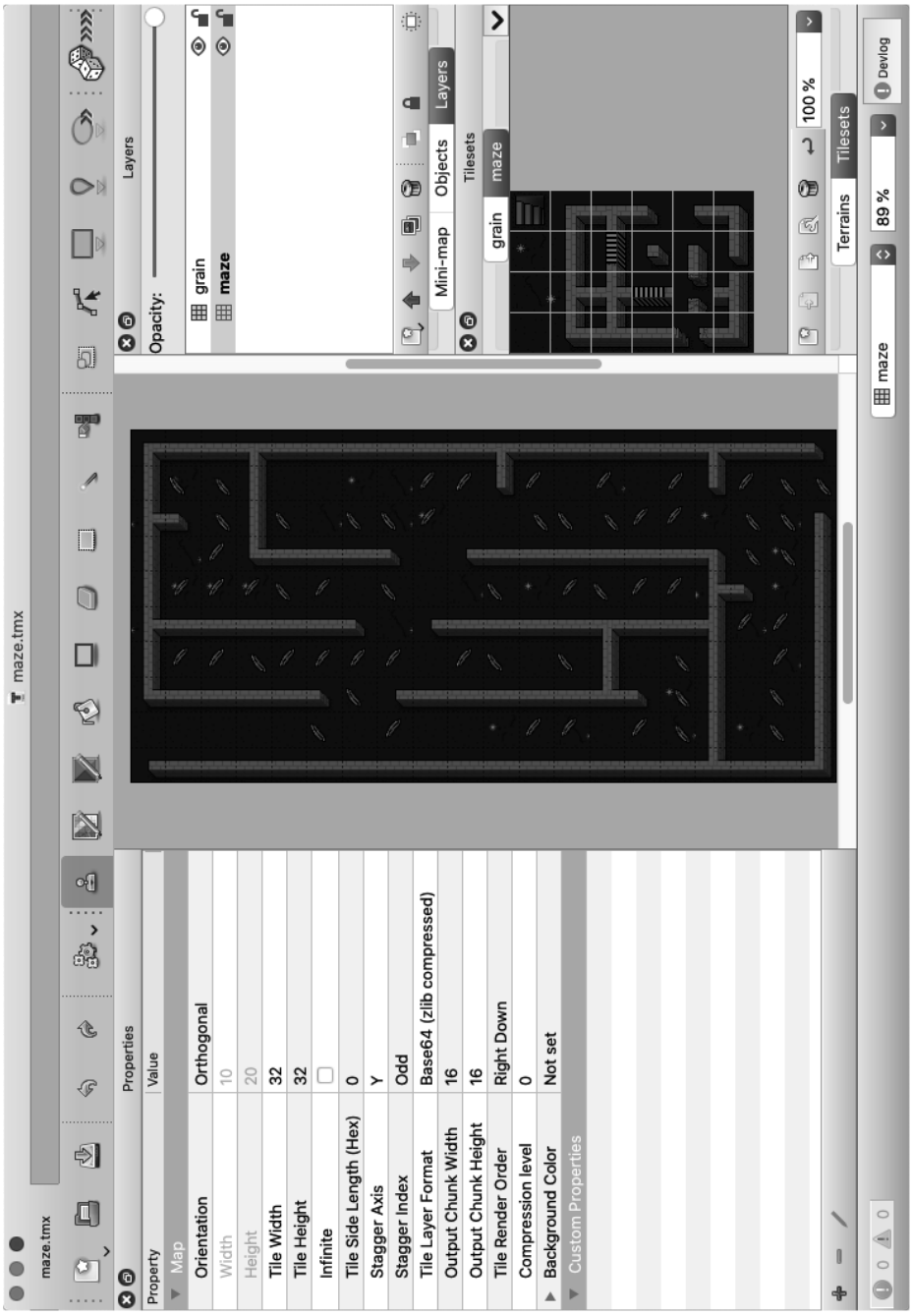


Fig. 3.7: Tiled Editor

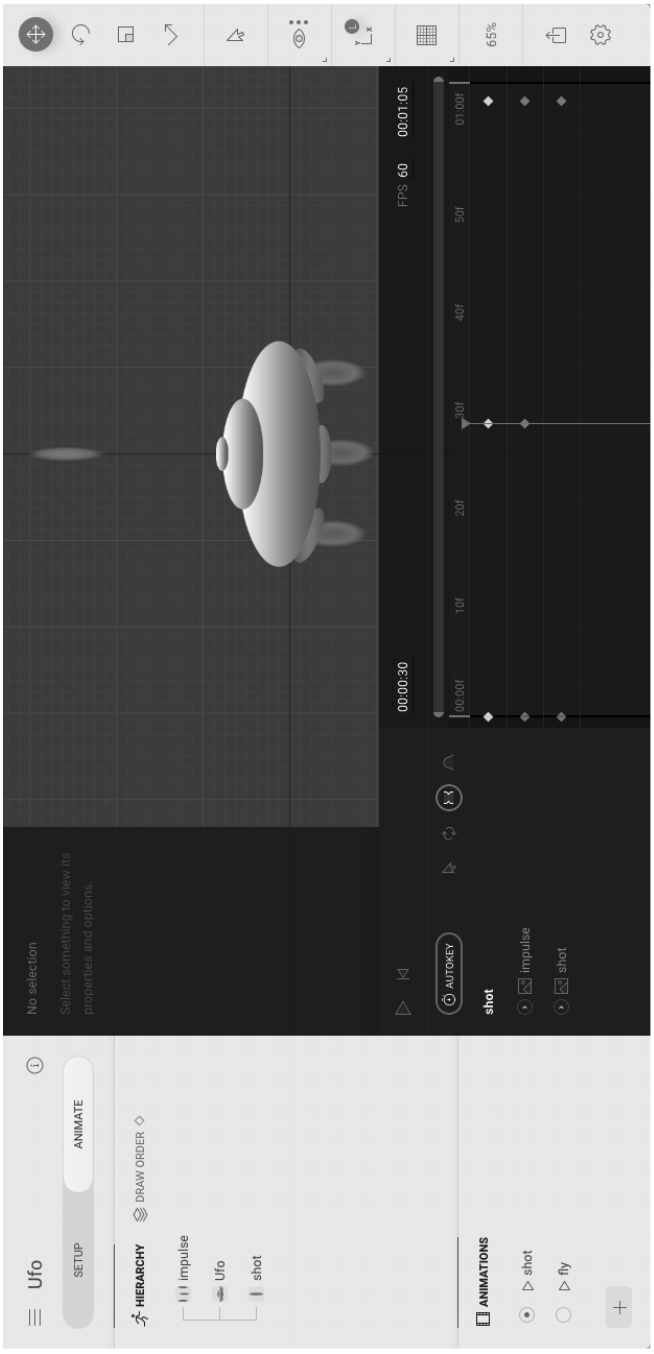


Fig. 3.8: Rive Animations with Rive and Nima

4 Fundamentals of Flutter

The use of the framework “Flutter” is the main topic of the book. It deals with Flutter in version2.0 (stable). Before realizing more complex application examples, this chapter examines the philosophy and basics of Flutter. In chapter 3 the installation of Flutter and the tools were discussed, this is the prerequisite for understanding the examples.

4.1 Why Flutter?

The purpose of Flutter is to develop mobile applications. Target platforms are iOS for Apple devices and Android for the other mobile devices. The focus is on the development for devices with touch-screen, thus cell phones and tablets. However, other devices are also supported, such as desktop computers or Smart TVs. Hardware specific functions are offered as plugins for the respective devices, with a native implementation for each platform. It is possible to program with a single code base for all platforms. The visual appearance is modern, because Flutter is based on Google Material Design. However, you can also use the Cupertino design to have a consistent design for Apple devices. The performance of Flutter is very good, since it is compiled to native ARM code. Flutter also uses its own high-performance rendering engine “Skia.” This makes the framework performant, unlike other platform-independent app frameworks that are often based on web technologies. The programming philosophy is also modern, where Flutter supports the so-called reactive programming.

For Flutter there are realizations for two design languages: Cupertino and Material Design.

In the future, even more platforms are to be supported. There is experimental support for desktop computers and their operating systems Windows, macOS and Linux. Flutter can even be used on Raspberry Pi computers. A web version of Flutter is also available. In addition, smart devices from the IOT area will be supported in the future. When programming on these platforms, it should be noted that the plug-ins available for mobile applications are usually not supported there. Flutter is on the way to become a universal framework. Reminiscences of Java with the slogan “Write once, Run Everywhere” are coming back to older developers. At that time, Java created the claim that applications realized with it could run via the Java Virtual Machine (Java VM) on all devices on which it was available. The difference to Flutter: On the one hand, native code is generated and not byte code that has to be executed via a VM. On the other hand, the formats of the devices changed considerably, smartphones and tablets did not exist in the early stages of Java.

https://doi.org/10.1515/9783110721331-004

4.2 Material Design +

Flutter is a realization of Google Material Design [41]. This is a new design philosophy. Already in the early days of graphical user interfaces, visual effects were used, at first bevel. This was followed by drop shadows, which made windows appear to float. The climax of this development was the macOS X operating system surface, where transparencies, reflections and refractions were added. Buttons looked glassy. Skeuomorphism is the name of this style, which tries to make us believe that something is real (fig. 4.1).

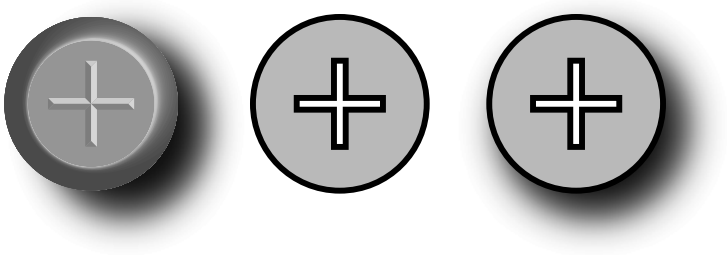


Fig. 4.1: Skeuomorphism, Flat- und Material Design

Many criticized this style and stated that these surfaces embodied the opposite of Apple's physical devices with their minimalist design. Microsoft initially copied this style rather poorly than well. But then, with the arrival of Windows 8, Microsoft radically changed the look of the user interface and created the exact opposite: they focused on pure colored surfaces and an abstract look. This style is called Flat-Design. Apple in turn reacted to this with a similar development and defused the realism of the screen elements. Since then, flat elements with gradients and slight shadows as well as transparencies determine the appearance of Mac surfaces. Google did not have its own special design style until now. As a result, it created the Google Material Design, a corporate design for all Google products and a design philosophy that can be read by all designers and developers on the website [41]. There it states that the metaphor underlying the design refers to physical material. This means colored surfaces such as paper or cardboard elements that lie on top of each other (fig. 4.2). These have a thickness and can float ("elevation"). These elements should behave physically (apart from floating) in a way similar to the real world. Therefore, they should move in and out of the screen area when they appear; exceptions are dialog boxes. They can also change their shape and size as if they were made of flexible material.

Material design means: All elements are to be seen as colored cardboard shapes, they have a thickness and a shadow.

Flutter is an implementation of this design metaphor. There are other implementations like Google Material Design Lite for web projects [42]. But when using Flutter, you are not bound to it. If you want to develop apps that have the design of native iOS apps, you can also use Apple's Cupertino design, which is available in the framework.

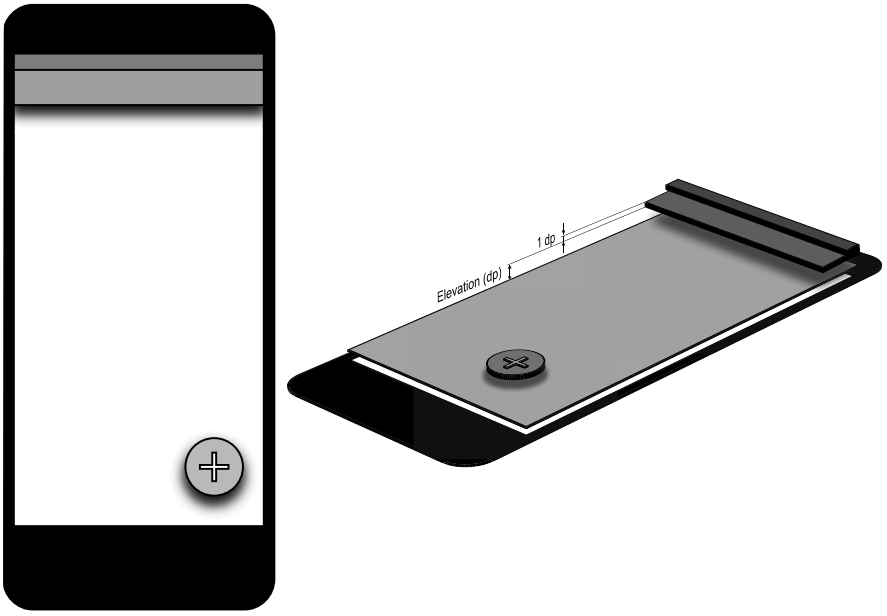


Fig. 4.2: The Philosophy of Material Design

4.3 Flutter Layouts

In Flutter, layouts neither are created in XML, as in Android, nor by using a graphical editor, but are described directly in a dart data structure. Here the strengths of Dart become apparent. It is expressive enough to make this possible. In Listing 4.1, you can see a simple example: A centered text, with a distance to the surrounding elements. Listing 4.2 is a similar example in HTML. The HTML code is less readable because all elements are embedded in common `div` or `span` tags. Another language, CSS, is also required. The dart code, on the other hand, is immediately understandable, "Literature programming" in the best sense. After the dart syntax was introduced in chapter 2, it is

clear how the syntax works. To create objects, you can also write `new Padding(...)`, but you can omit the keyword `new`, so the coding of the layout structure looks more like a declarative description. The constructor parameters are named parameters for the layout elements, which specify further properties, such as `padding` here. The values themselves are mostly objects. In order to build a tree structure, there are two special named parameters, which, however, cannot occur together in one single object. `child:` defines a child object, with `children:` you can specify a list of children. With this simple syntax, you can describe any tree structure and its properties.

Layouts are described as hierarchical dart data structures containing widgets and lists of widgets.

The question why there is no layout editor can be answered, if one has tried to create a complex responsive layout for different devices and in portrait or landscape format by pointing and clicking with a mouse. With a WYSIWYG editor (What you see is what you get), you get what you see, but nothing more. Professional web coders do not work with graphical editors for this reason.

Listing 4.1: layout.dart

```
Padding(
    padding: EdgeInsets.all(8.0),),
    child: Center(child: Text("Hello"), ),
),
```

Listing 4.2: layout.html

```
<div style='padding: 8px'>
    <div style='text-align: center'>
        <span>Hello</span></div>
</div>
```

4.3.1 Everything is a Widget

In Flutter everything is a "widget." Widget is an artificial word made up of "Window" and "Gadget." These window objects describe everything visible like text or buttons, but also layout formatting like centering `Center(...)` or animation. In figure 4.3, you can see a class diagram, that shows important connections. So all widgets inherit from the base class *Widget*. You can see here that there are three types of widgets.

Firstly *RenderObjectWidgets*, thus formatting like alignments. Second, *StatelessWidgets*, thus graphic, immutable elements without a state. Third, *StatefulWidgets*, thus elements that have a state that can change. States are specified separately as a class. These are parameterized classes; the type parameter is the corresponding

implementation of the *StatefulWidgets*. Future examples will bring clarity. In the diagram 4.3, implementations of *StatefulWidget* "*MyPage*" and of *State* "*MyPageState*" are shown (grey), but they are not part of the class library of Flutter. The reason for separating the state from the *StatefulWidget* class is, that the *Widget* superclass is marked with `@immutable`, which means that all widgets are immutable. The mutable state can be created using the `createState()` factory method, which creates an object of type `State<StatefulWidget>`.

There are three types of widgets: *RenderObjectWidgets*, which describe layout properties; *StatelessWidgets*, which stand for immutable visible elements; and *StatefulWidgets*, which have a variable state. Since all widgets are immutable, the state is defined separately.

Important for the construction of apps is the ability to provide several pages with different content and the ability to navigate between pages. Usually, these pages are called "Routes" in Flutter. Any *Widget* can be used for a route. A Navigator can be used to switch between the pages. The design philosophy of Google Material Design is that such routes can be stacked on top of each other and removed from the stack. The transition is usually animated, and new routes are pushed in or out from the page or from the top. The class `Navigator` has the corresponding methods `push(...)` and `pop(...)`.

To create a route, you must generate a route object and pass it to a WidgetBuilder. This is a lambda function that returns a widget, for example, MyPage in the diagram (later, examples will follow to make this clearer).

Routes are individual screens that can be navigated to and from.

4.3.2 Layout Example without State

Now, we show an example of a screen with a list. The result should look like figure 4.4. The hierarchy and nesting of the individual widgets are in the schematic figure 4.5. Listing 4.3 is the corresponding source code. First, the material library is imported, which is the implementation of material design that also provides the widgets and all flutter basic functionality.

The entry point for the program is, as usual, the `main(...)` function. This calls a function `runApp(Widget app)` that expects a widget and displays it. From line 5 onward, this widget is implemented as a class; here as a subclass of the abstract class *StatelessWidget*. You have to override the `build(...)` method, which a widget must have as return value. From line 8, the layout structure is directly coded. As an all-embracing element, there is the MaterialApp, which provides important things like the

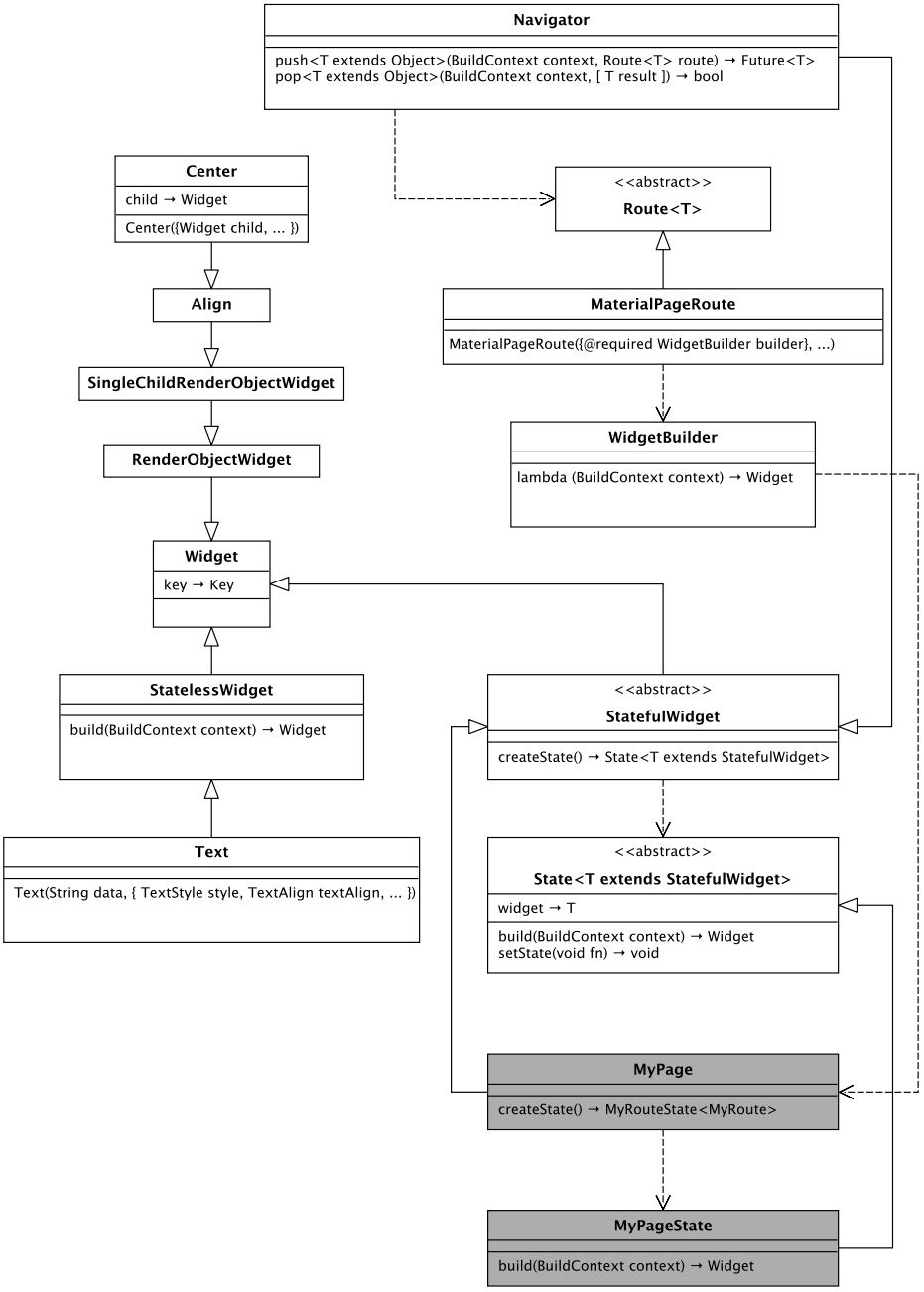


Fig. 4.3: Flutter Classes

Fig. 4.4: List Screenshot

navigator or localization. Everything else is specified via the various named parameters. With `title:` you can simply specify the name of the app with a text string. In `home:`, the page that should appear first is specified; in our example also the only page. Here a scaffold object is created, thus a template for material design pages. There you can specify a title bar `appBar:` and create a corresponding AppBar, again with a title that is then visible in the default blue title bar, preceded by a menu icon. In `body:`, the area below, a list is now created, via a ListView object, which gets as child a list of widgets that define the individual lines (line 16).

Line 17 shows the use of the Python-like syntax for initializing lists (see section 2.2.2). Here a card object is created 30 times. This is just a container, that can be colored and has a certain height (elevation). The widget can have a list of children representing individual elements, thus a preceding icon, a bug (line 25), a larger title and a smaller subtitle. There is also a named parameter `onTap:` which expects a lambda function that determines what happens when you tap the element with your finger (line 28).

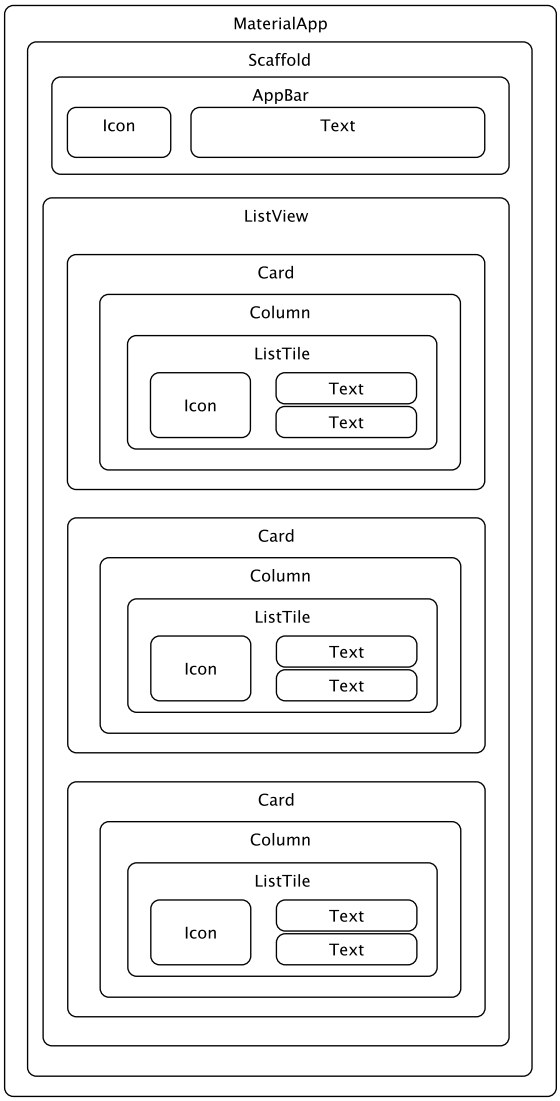


Fig. 4.5: List View

All list and parameter entries can end with a comma, even if no further entry follows (e.g., line 29). This is a convention in Flutter, as it simplifies the insertion of further elements. The IDEs always automatically add such a comma. This example shows that the declarative description of a layout including the automatic generation via loops with dart tools is readable and clear.

Listing 4.3: main.dart

```
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        debugShowCheckedModeBanner: false,
        title: 'List',
        home: Scaffold(
            appBar: AppBar(
              leading: Icon(Icons.menu), title: Text('Bug Report'),
            ),
            body: ListView(
              children: <Widget>[
                for (int i = 1; i <= 30; i++)
                  Card(
                    elevation: 5.0,
                    color: Colors.deepOrange,
                    child: Column(
                      mainAxisSize: MainAxisSize.min,
                      children: <Widget>[
                        ListTile(
                          leading: Icon(Icons.bug_report, size: 50),
                          title: Text("Bug Nr. $i"),
                          subtitle: Text("Lorem Ipsum"),
                          onTap: () => print("Pressed: Nr.: $i"),
                        ),
                      ],
                    ),
                  )
              ],
            )));
  }
}
```

4.3.3 The Navigator

In Listing 4.4, the functionality of the navigator is explained with a simple example. Two screens are created, between which you can switch by pressing a button. In the figures 4.6, you can see the two screens. In line 4, the app is started by creating an environment. In lines 6-15, a widget is provided for this purpose, which contains a *MaterialApp*. The first page to be seen *PageOne* is selected as home page. This is described in lines 17-38: In the "build" method, a scaffold is built, with title (*AppBar*) and body. This contains a text and a button. When this is pressed (line 30), a new route is added to the navigator of the context. This route contains a builder, thus a lambda function that creates page two (*PageTwo*). It should be ensured that the context really contains a Navigator. The top level must, therefore, be the *MaterialApp*, that provides a navigator. Page two is described from line 40 onward. It also contains a counter that counts how many of these pages have already been generated (line 44). Its variable is a class variable that is static and reinitialized as an object variable every time an object is created, but applies to all objects. If the widget is built, this counter is increased by one. As in the other class *PageOne*, the *PageTwo* class generates a scaffold with a title line and a body with text and a button. If this button is pressed, the navigator uses `.pop()` to take the last page from the navigation stack. This is the current page.

---

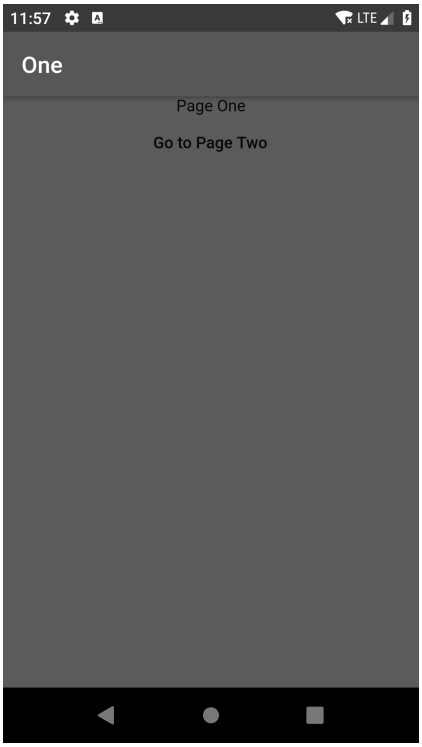
i The Navigator is provided by a *MaterialApp*. Individual routes can be added to and removed from it like cards to a stack with `.push(...)` and `.pop(...)`.

By clicking on the button of the first page a new second page is created. This can be seen in the example by the fact that their color always changes randomly (see line 57). Here a short explanation of the line: A random RGB value is generated, 0-255 (0-FF hexadecimal) possible values for red, green and blue. To make the color visible, the alpha value is set to 255 (FF) by setting the highest byte to 255 using the shift operator. Thus, you have an ARGB value as color value here; A stands for alpha.

Listing 4.4: main.dart

```
import 'package:flutter/material.dart';
import 'dart:math';

void main() => runApp(new RootApp());

class RootApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Flutter Demo',
      home: PageOne(),
      debugShowCheckedModeBanner: false,
    );
  }
}

class PageOne extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: Text("One"),
        ),
        body: Center(
          child: Column(
            children: <Widget>[
              Text("Page One"),
              MaterialButton(
                child: Text("Go to Page Two"),
                onPressed: () => Navigator.of(context).push(
                    MaterialPageRoute(
                    builder: (BuildContext context) => PageTwo())),
              ),
            ],
          ),
        ),
        backgroundColor: Color(0xFFCCCCFF));
  }
}

```

```
class PageTwo extends StatelessWidget {
  static int _counter = 0;
  @override
  Widget build(BuildContext context) {
    _counter++;
    return Scaffold(
        appBar: AppBar(
          title: Text("Two"),
        ),
        body: Center(
          child: Column(
            children: <Widget>[
              Text("Page Two Nr. $_counter"),
              MaterialButton(
                child: Text("Back to Page One"),
                onPressed: () => Navigator.of(context).pop(),
              ), ], ), ),
        backgroundColor: Color(Random().nextInt(0xFFFFFF) | 0xff <<
            24));
  }
}
```



(a) Screenshot Navigator

(b) Screenshot Navigator Back

Fig. 4.6: Two Screens

4.3.4 A Layout with State

The following is an example of a widget that is not static. The code is available online at GitHub [52]. You can download it using Git with

```
git clone https://github.com/meillermedia/world_cup.git
```

on the computer. In contrast to the previous example, the content of the list is retrieved dynamically and can change. The example here loads a list with the soccer results of all matches of the soccer world cup 2018. The idea for the app is that you can look up all current results of a running championship (see fig. 4.7). The list should be updated after every match, so the state should be able to change. The URL of the data source refers to a GitHub repository (line 11, 12), and the file was updated during the championship by dedicated soccer fans. As shown in the diagram 4.3, *StateFulWidgets* must have their own state in the form of an object. This is generated in the method `createState()` which is to be overwritten (see line 16 in Listing 4.5).

The corresponding class `FootballAppState extends State<FootballApp>` has a list as object variable (`_gamesList`). All fields of a state contain the current state. Here the list simply contains all formatted lines as widgets. You can initialize the state in the method `initState()` (line 188). If you want to change the state later, you have to use the method `setState(void Function fn)`, which gets a lambda function as argument, in which you can change the variables (starting from line 27). The help method `_getFootballResults()` works as follows: From line 28 on, first the whole list is deleted and a loading animation is displayed. Then, line 42 reads the file. The `readCachedData()` help method is not explained further, because it does not contribute to the explanation of the state principle (see code online in the repository). In any case, it loads the file. If the Internet connection is not available, the cached file is loaded. After waiting for the file to load, line 44 first clears the list. From line 46 on, the JSON file is evaluated and the individual values are read. In the help method `_addResult(...)`, a complex layout is built up in which individual lines with the individual results are generated and placed in a *Card* (line 128), which stands for a matchday. In line 143, this is added to the list. In line 111, an interaction is added. When a result is pressed, a route is placed on the navigation stack, which displays a page with details. The generation of the detail page is not relevant here, it is a *StatelessWidget*. The navigation and the *AppBar* automatically insert a back arrow, which takes the page back from the stack when activated.

The example implements a responsive design using an *OrientationBuilder* (starting at line 164): There are several such builder classes in Flutter. They react to external influences and create new widgets when changes are made. Here the builder function is called when the orientation of the device changes. A three-column grid is created when the device is aligned in landscape format, and a one-column layout when it is aligned horizontally (Fig. 4.8).

The *OrientationBuilder* is used to distinguish whether the device is in portrait or landscape mode. The orientation can be evaluated and, depending on it, an extra layout can be generated. If the orientation is changed, the layout is re-generated and the Builder is executed.

Listing 4.5: main.dart

```
import 'dart:convert';

import 'package:flutter/material.dart';

import 'details_page.dart';
import 'read_cached_data.dart';

void main() => runApp(
    MaterialApp(debugShowCheckedModeBanner: false, home: new
        FootballApp()));

const String wc_uri = "raw.githubusercontent.com";
const String wc_path = "/openfootball/world-cup.json/master/2018/
    worldcup.json";

class FootballApp extends StatefulWidget {
  @override
  State<StatefulWidget> createState() {
    var fbs = FootballAppState();

    return fbs;
  }
}

class FootballAppState extends State<FootballApp> {
  List<Widget> _gamesList = <Widget>[];

  void _getFootballResults() async {
    setState(() {
      _gamesList = <Widget>[
        Center(
          child: Text("Please wait"),
        ),
        Center(
          child: new Container(
```

```
            alignment: FractionalOffset.center,
            child: CircularProgressIndicator()),
      ),
      Center(
        child: Text("Loading results"),
      ),
    ];
  });
  var response = await readCachedData(name: 'football.json', uri:
      wc_uri, path: wc_path);
  setState(() {
    _gamesList = <Widget>[];
  });
  Map<String, dynamic> wc = json.decode(response);
  for (var r in wc['rounds']) {
    print(r['name']);
    var day = <String>[];
    var teams = <String>[];
    var scores = <String>[];
    for (var m in r['matches']) {
      day.add(m['group'] ?? ' ');
      teams.add("${m['team1']['name']}-${m['team2']['name']}");
      var s1et = m['score1et'] ?? 0;
      var s2et = m['score2et'] ?? 0;
      var score = m['score1'] == null || m['score2'] == null
          ? '? : ?'
          : "${m['score1'] + s1et} : ${m['score2'] + s2et}";
      scores.add(score);
    }
    try {
      _addResult(r['name'], day, teams, scores, r);
    } finally {
      //Nothing
    }
  }
}

void _addResult(String day, List<String> groups, List<String> teams
    ,
    List<String> scores, Map<String, dynamic> details) {
  List<Widget> groupList = <Widget>[];
  for (String s in groups) {
```

```
74        groupList.add(Text(s));
75      }
76      List<Widget> teamList = <Widget>[];
77      for (String s in teams) {
78        teamList.add(Text(s));
79      }
80      List<Widget> scoreList = <Widget>[];
81      for (String s in scores) {
82        scoreList.add(Text(s));
83      }
84      var rows = <Widget>[];
85      for (int i = 0; i < teamList.length; i++) {
86        Row r = Row(
87          // mainAxisAlignment: MainAxisAlignment.spaceBetween,
88          children: <Widget>[
89            Expanded(
90              child: groupList[i],
91              flex: 1,
92            ),
93            Expanded(
94              child: teamList[i],
95              flex: 2,
96            ),
97            Expanded(
98              child: scoreList[i],
99              flex: 1,
100           ),
101         ],
102       );
103       var ink = InkWell(
104         child: Padding(
105           child: r,
106           padding: EdgeInsets.all(10.0),
107         ),
108         highlightColor: Colors.blue,
109         splashColor: Colors.blue,
110         borderRadius: BorderRadius.all(Radius.circular(5.0)),
111         onTap: () {
112           Navigator.of(this.context)
113               .push(MaterialPageRoute(builder: (BuildContext context)
                      {
114             return detailsPage(details, i);
```

```
115           }));
116         },
117       );
118       rows.add(ink);
119     }
120     var col = Column(
121       children: rows,
122     );
123
124     SingleChildScrollView s = SingleChildScrollView(
125       child: col,
126     );
127
128     Card c = Card(
129         child: Column(
130       children: <Widget>[
131         Container(
132           child: Text(day),
133           padding: EdgeInsets.all(15.0),
134           decoration: BoxDecoration(color: Colors.lightBlue),
135           alignment: FractionalOffset.center,
136         ),
137         Expanded(
138           child: s,
139         ),
140       ],
141     ));
142     setState(() {
143       _gamesList.add(c);
144     });
145   }
146
147   @override
148   Widget build(BuildContext context) {
149     return Scaffold(
150       appBar: AppBar(
151         title: Text("Worldcup 2018"),
152       ),
153       body: Center(
154         child: Padding(
155           padding: EdgeInsets.all(5.0),
156           child: Column(children: <Widget>[
```

```
157             MaterialButton(
158               child: InkWell(child: Text("Reload")),
159               highlightColor: Colors.orange,
160               splashColor: Colors.orange,
161               onPressed: _getFootballResults,
162             ),
163             Expanded(
164               child: OrientationBuilder(
165                   builder: (BuildContext context, Orientation
                          orientation) {
166                 var size = MediaQuery.of(context).size;
167                 print(size.height);
168                 if (orientation == Orientation.landscape) {
169                   return GridView.count(
170                       crossAxisCount: 3,
171                       childAspectRatio: size.height / 752.0 * 2.4,
172                       children: _gamesList);
173                 } else {
174                   return GridView.count(
175                       crossAxisCount: 1,
176                       childAspectRatio: size.height / 1232.0 * 4.5,
177                       children: _gamesList);
178                 }
179               }),
180             ),
181           ]),
182         ),
183       ),
184     );
185   }
186
187   @override
188   void initState() {
189     super.initState();
190     _getFootballResults();
191   }
192 }
```

Fig. 4.7: World Cup vertical Layout

4.4 Flutter Packages

A strength of Flutter is its modularity and expandability. In the central repository at *https://pub.dev/flutter* [25], you can find many extensions, all packages for app development are hosted there. It is worth searching the repository if you want to use a desired feature.

Mostly these are functionalities that access special hardware or operating system functions, such as access to the camera or photo album or the accelerometer. These are called "plugins" in the flutter jargon. However, you will also find more complex layout components, such as a map component or special types of buttons. Such platform-independent packages, which are programmed in their entirety in Dart, are called "Flutter-Packages." Packages and plugins can then be included in the *pubspec.yaml* file as described in section 4.4 and downloaded via `flutter pub get` analogous to the use of `pub get` for the pure Dart packages. Next, these packages are stored locally in

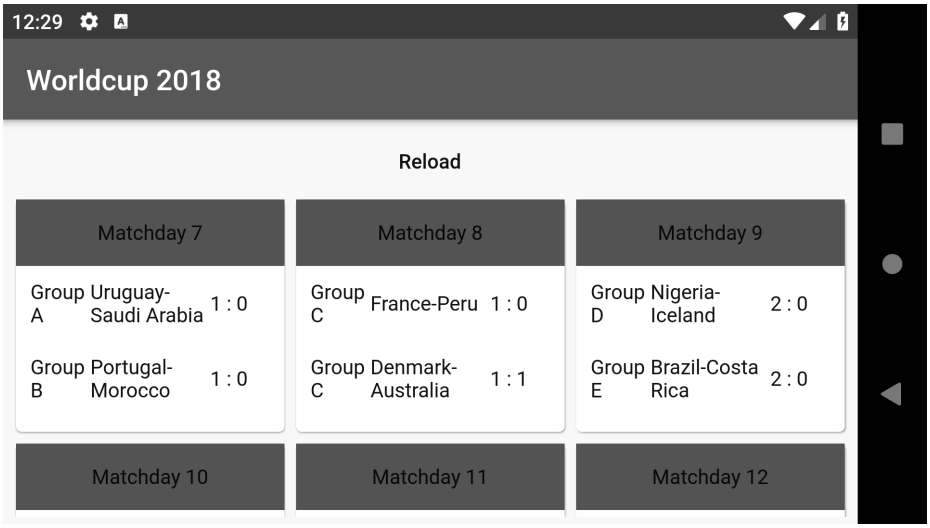


Fig. 4.8: World Cup horizontal Layout

the folder `.pub-cache` in the home directory under Mac, Linux or `\Pub\Cache` under Windows.

For each component listed on the repository web page, there is a mini-homepage with good documentation. Figure 4.9 shows the page of the package for the flame game engine. So you can basically learn what the component does, which versions are available and under which license it was released. For the use in commercial projects, the license plays an important role. You should be clear about the terms and conditions under which you may use the software. Further, it is explained how to install it by adding the appropriate entry to the *pubspec.yaml* that you can copy and paste. The corresponding import instruction can also be found there; also, a simple example code exists. The version can give an indication of how mature the package is. Additionally, it is interesting to know which other libraries it is based on. So, you have a hint about which conflicts can occur. Also, the other direction can be seen, which packages are based on it. There is also an API reference available, that you can search online.

You can view the complete source code via the linked git repository (usually GitHub). There, you will find a folder `lib` with the source code, often with a folder `example` containing more sample code. In the folder `test`, there are usually test cases for unit tests. For components that use special platform-dependent functionalities (plugins), there are two more folders, `ios` and `android`, which contain the native code that accesses the native system libraries of the respective operating system. This is where the platform independent world of Dart ends and where you can find XML, Java, Kotlin or JAR files (Android), or Swift and Objective-C files (iOS). Sometimes, you can

find a bug. If this happens, you should report it on GitHub. This way you can contribute to the development and improvement of open source projects.

At `https://pub.dev/flutter/packages` you can check the global directory to see which packages are available. Here you can distinguish between several types of platforms on which the packages are available: "Android", "iOS", "Web" or others. In some cases, not all platforms are equally well supported.

4.4.1 Map Extensions

To program a map application, for example, you can use a suitable package. The "flutter_map"-Package [62] is a port of the popular leaflet plugin [3] from JavaScript to Dart and Flutter. Figure 4.10 shows the screenshot of the example.

When testing map applications in the simulator, you must first define your own position manually. Otherwise, a position in California is selected by default. For this purpose, you have to set it in the menu of the simulator (Three Points ...): Location ⇒ Search ⇒ Save Point ⇒ Set Location.

First, the package for the maps and the geolocation is included in *pubspec.yaml*, see listing 4.7. Then, these are installed in the terminal by typing `flutter pub get`. Further, for Android, you have to allow Internet access and access to the location sensor in the *AndroidManifest.xml* file, in the *android/app/src/main* folder (listing 4.6). Note that at the time of writing this book, not all modules used were null-safe. Thus, at startup, null-safety must be disabled with the command `flutter run --no-sound-null-safety`.

Listing 4.6: AndroidManifest.xml

```
1 <uses-permission android:name="android.permission.INTERNET"/>
2 <uses-permission android:name="android.permission.
      ACCESS_COARSE_LOCATION" />
3 <uses-permission android:name="android.permission.
      ACCESS_FINE_LOCATION" />
```

In Listing 4.8, the packages are imported first, in the main function in line 9, it is first ensured that everything is initialized. Then you can select your own position with the *Geolocator* and save it for later use. A *MaterialApp* is created with a *StatefulWidget* and a *State* with a *Scaffold*, which contains the *FlutterMap* in `body:`, thus the actual map. In the `options:`, the position at which the map should be displayed is specified in the *MapOptions* as `center:` [9]. Further, you can specify the amount of detail using the

9 Here the position is hard-coded in, and if you have stored the position of the University OTH Amberg in the simulator, you can comment out line 12 and comment in line 13.

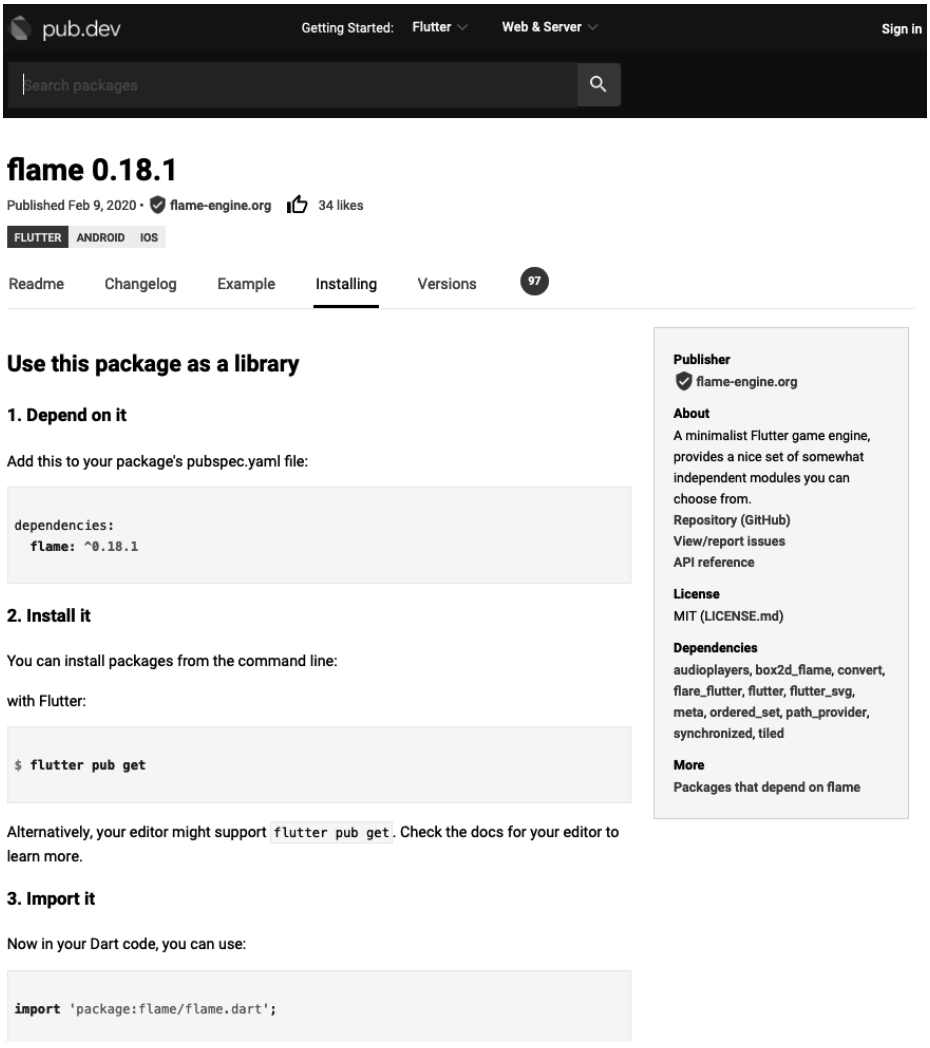


Fig. 4.9: Website for Flutter Packages

zoom levels, followed by several levels. The lowest level contains the graphics which are loaded online via OpenStreetMap. The *TileLayerOptions* specify how the address for each graphic is formed. Depending on the zoom level, location and tile numbers, a new address for a tile is generated using the string in `urlTemplate:`.

A layer with markers follows. These also have a position, a size and a builder that generates a widget. Here you can put any widget on the map; two icons are placed on the map. When you press on these icons, a function is specified for each icon, which is then executed.

Listing 4.7: pubspec.yaml

```
dependencies:
  flutter:
    sdk: flutter
  flutter_map: ^0.12.0
  geolocator: ^7.0.1
  cupertino_icons: ^0.1.2
dev_dependencies:
  flutter_test:
    sdk: flutter
```

Listing 4.8: main.dart

```
import 'package:flutter/material.dart';
import 'package:flutter_map/flutter_map.dart';
import 'package:latlong/latlong.dart';
import 'package:geolocator/geolocator.dart';

late LatLng _position;

void main() {
  WidgetsFlutterBinding.ensureInitialized();
  Geolocator.getLastKnownPosition().then((pos) {
    if (pos != null) {
      //_position = LatLng(pos.latitude, pos.longitude);
      _position = LatLng(49.4448369, 11.8473098);
    } else {
      _position = LatLng(49.4448369, 11.8473098);
    }
    runApp(new MyMapApp());
  });
}


```

```
class MyMapApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      home: new MyMapPage(),
      debugShowCheckedModeBanner: false,
    );
  }
}

class MyMapPage extends StatefulWidget {
  @override
  _MyMapPageState createState() => new _MyMapPageState();
}

class _MyMapPageState extends State<MyMapPage> {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
        appBar: new AppBar(title: new Text('My Locations')),
        body: new FlutterMap(
            options: new MapOptions(
                center: _position, minZoom: 10.0, maxZoom: 18, zoom:
                    17),
            layers: [
              new TileLayerOptions(
                  urlTemplate:
                      "https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.
                          png",
                  subdomains: ['a', 'b', 'c']),
              new MarkerLayerOptions(markers: [
                Marker(
                    width: 45.0,
                    height: 45.0,
                    point: new LatLng(49.4448369, 11.8473098),
                    builder: (context) => new Container(
                         child: IconButton(
                           icon: Icon(Icons.local_bar),
                           color: Colors.red,
                           iconSize: 45.0,
                           onPressed: () {
                             print('Marker tapped');
```

```
                        },
                      ),
                    )),
              Marker(
                  width: 45.0,
                  height: 45.0,
                  point: new LatLng(49.443369, 11.8473098),
                  builder: (context) => new Container(
                       child: IconButton(
                         icon: Icon(Icons.person),
                         color: Colors.red,
                         iconSize: 45.0,
                         onPressed: () {
                           print('Marker tapped');
                         },
                       ),
                     ))
            ])
          ]));
  }
}
```

4.4.2 How to program Flutter Extensions

You can develop your own packages (for detailed instructions see: [24]). These can be uploaded to the cloud using the publish command (`flutter pub publish`) and can be made available on the pub website. For this purpose, however, the license, documentation and implementations should be available for all platforms, as described in the manual.

However, you can also use packages for your own development to make the code more modular or develop modular components for internal projects. These components will then not be managed by an external repository but can be located locally on the computer. Of course, you can then manage them using a different versioning system, such as an internal GitLab server. To use these locally available components, you must also make them known to *pubspec.yaml* by specifying the local path (see listing 4.15).

Using the extensions is very comfortable, but nevertheless, the following problems can occur: With the plug-in packages you cannot automatically assume that the desired function is identical or works equally well on the respective platforms. Sometimes, there are only implementations for one platform, or one implementation works and the other one has a bug. However, if you want to develop for only one platform, this is not

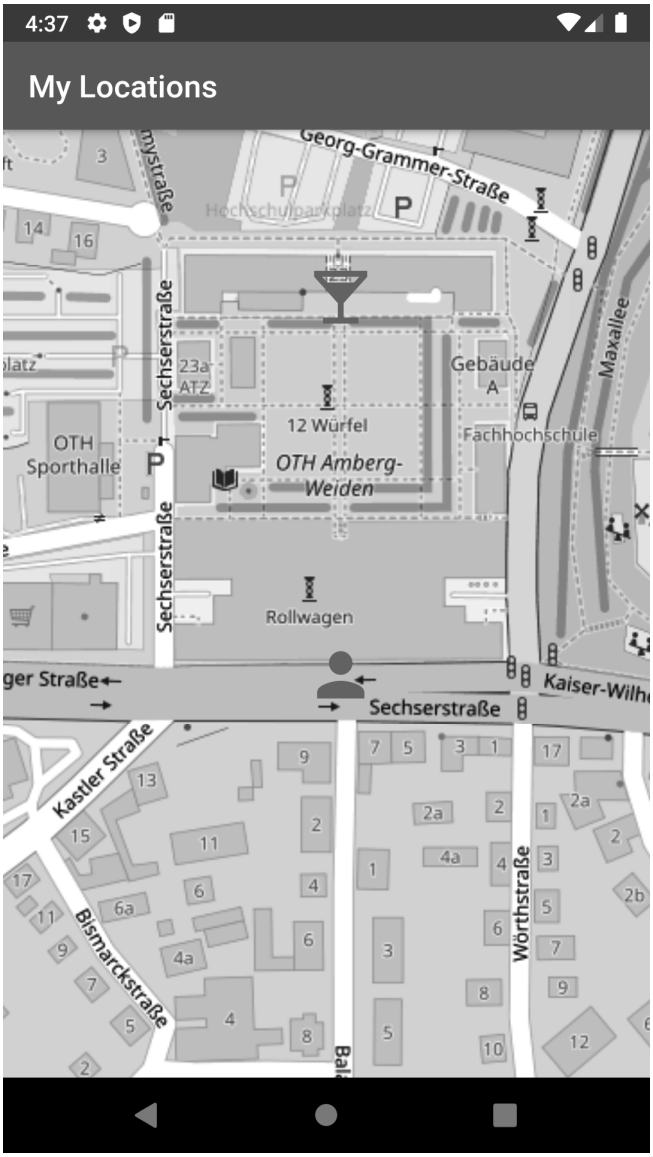


Fig. 4.10: Screenshot Map Application

relevant. Here it is worth looking at the issues on GitHub, where questions, suggestions for improvement and bugs are listed. If you discover a bug, you should get involved and report it there. In general, your own involvement in such open source projects is useful and desired. You usually get a quick feedback and do not have to wait until something works again, because you can take the initiative yourself.

Another problem are indirect dependencies in the extensions. For example, it happens that you use a plugin A, which, in turn, uses a plugin B in a certain version X. Now there is the case that you either use the Plugin B in another version Y > X, or you use another Plugin U, which also uses (directly or indirectly) the Plugin B. Be aware that version conflicts may occur when using the plugin where it can be very time-consuming to resolve such dependencies.

4.4.3 Simple Animations

The following is an example of how to use a simple plugin. A strength of Flutter is the animation of the widgets. However, the concept of programming behind this is not particularly intuitive and difficult to implement. There a Plugin helps, which simplifies the production of animations considerably: "simple_animations"[9]. To import it, you have to specify it like in Listing 4.9 under the `dependencies`. Then you have to call `pub get` via `flutter pub get`. Now you can perform the import in the source code (see line 2 in Listing 4.10). In the example, the setup for a *StatelessWidget* follows. Again, a *MaterialApp* and a *Scaffold* framework are used, in whose body you will find a *MirrorAnimation*, a class from the new plugin. Inside the type parameter you have to specify the animated parameters using an *enum* (see line 2). With the class *MirrorAnimation*, the animation is played forwards and then backwards. Another possibility would be a class *LoopAnimation*, in which the animation would start over and over again. The parameter `curve:` can be used to determine how the speed of the animation behaves, thus whether it is linear or whether it becomes slower or faster. Here, `Curves.easeInOutSine` specifies a soft start and a soft end (see figure 4.11).

Next, the duration of the entire animation is specified: It is possible to specify different time units, here milliseconds, using the *Duration* object. The values, which should change per animation, are given in *Tween* from line 20 on. The word "tween" is derived from "between." There is a start value and an end value, between these values a transition occurs in a given time, and a current value is approximated to the target value per time progress. For more complex animations, the class *MultiTween* is useful. Here you can specify a list of entries that you want to animate. In the method `.add(...)` you enter the parameter to be animated, the fadeing (*Tween*) of the number, which is assigned to the parameter, and the duration of the animation. The parameter types decide whether the width or the height of the object is animated. You could specify even more types, for example for color change. Lines 21, 22 and 23 read like this: At first nothing happens for half a second because the tween is a *ConstantTween*, which

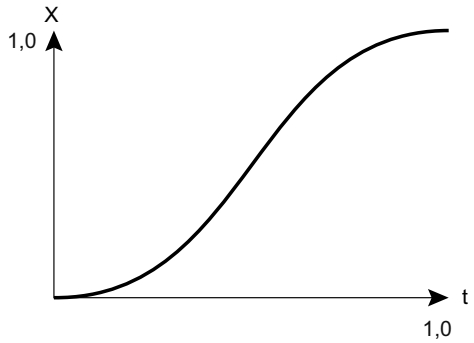


Fig. 4.11: *Curves.easeInOutSine*

means that the number remains constant at 10. Only after this half second, another animation plays, again with half a second length, where the value is interpolated from 10 to 200. In lines 25-28, the same happens for the track for the width, but there the animation starts immediately and then waits half a second. You will see the following in the animation later:

First the object changes its width, then its height. In the *MirrorAnimation*, an additional parameter is specified: `builder:` This is, as usual for the *Builder* classes, a lambda function that takes the context and the animation as parameters. The particular values can be read using the *enum* entires for identification purposes. If the values are changed, the *Builder* creates a new object with different values for the width and the height of the object, which is a blue container. As result, a rectangle is animated; first its width is scaled, then its height. Since *MirrorAnimation* is used, this process is played backwards, then the whole animation starts again. More information about this flutter extension can be found in the GitHub repository, which is also linked in the repository website (pub.dev) [8].

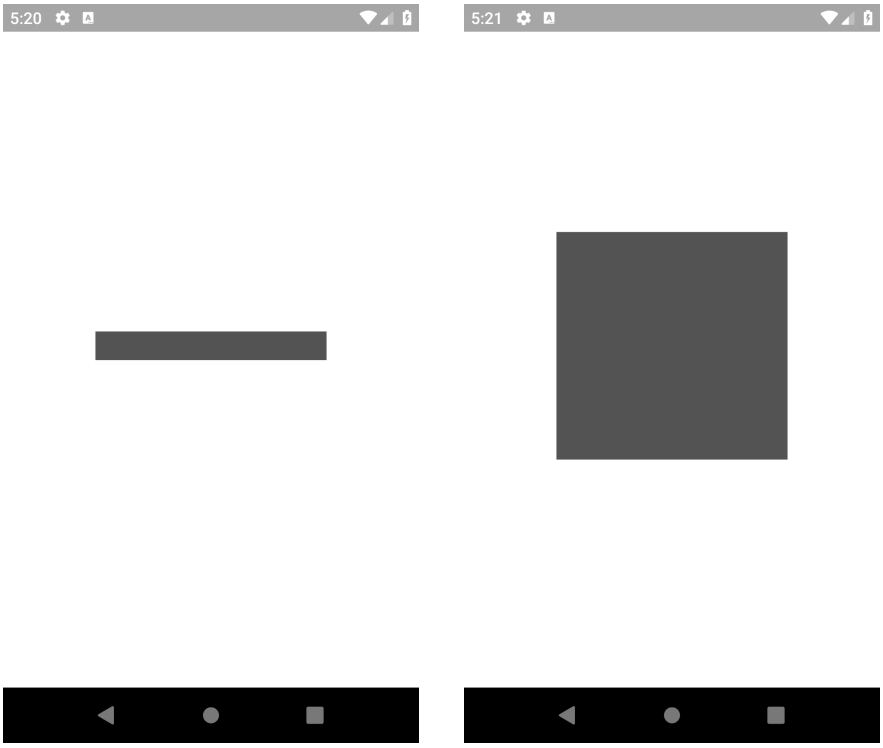
You can also specify programmed animations as widgets: These have a duration and an acceleration function. You can animate all visual properties such as size or color.

Listing 4.9: pubspec.yaml

```
1  dependencies:
2    flutter:
3      sdk: flutter
4    simple_animations: ^3.0.2
```

Listing 4.10: main.dart

```
import 'package:flutter/material.dart';
import 'package:simple_animations/simple_animations.dart';

void main() => runApp(MyApp());
enum AnimatedParams { width, height }

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Animation Demo',
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        backgroundColor: Colors.white,
        body: Center(
          child: MirrorAnimation<MultiTweenValues<AnimatedParams>>(
            curve: Curves.easeInOutSine,
            duration: Duration(milliseconds: 1000),
            tween: MultiTween<AnimatedParams>()
              ..add(AnimatedParams.height,
                  ConstantTween(10.0), Duration(milliseconds: 500))
              ..add(AnimatedParams.height,
                  Tween(begin: 10.0, end: 200.0), Duration(
                      milliseconds: 500))
              ..add(AnimatedParams.width,
                  Tween(begin: 10.0, end: 200.0), Duration(
                      milliseconds: 500))
              ..add(AnimatedParams.width,
                  ConstantTween(200.0), Duration(milliseconds: 500)),
            builder: (context, child, value) {
              return Container(
                  width: value.get( AnimatedParams.width),
                  height: value.get(AnimatedParams.height),
                  color: Colors.lightBlue);
            },),
        ),
      ),);
  }
}
```



(a) Height animated

(b) Width animated

Fig. 4.12: Simple Animation

4.4.4 Animations with Nima and Rive

In the section 3.3 we got to know the animation tool Rive. If you use the Nima plugin, you can embed the animations directly as a widget. If you have animated vector graphics, you have to use the "Flare-Flutter" plugin with the Rive tool.[10] Here is a short example of interactive control of a Nima animation: In Listing 4.11, is the corresponding pubspec file: The package Nima is specified and in `assets:` the position of the exported files from the animation tool is given. Note that at the time of writing this book, the plugin was not null-safe. Thus, at startup, null-safety must be disabled with the command `flutter run --no-sound-null-safety`. In listing 4.12, the code for the animation is given: In line 2, the library for the animation of the Nima widget is imported. Then, starting in line 4, a *MaterialApp* is created, with another stateful widget as homepage. In the state, there is a private method `_ufo()` (line 17) that returns a *NimaActor* widget. Here the ".nma" file, which was exported from the animation tool, is specified. As

10 Rive was previously called Flare, the Flutter plugin is still called that.

named parameter `animation:` the current animation is specified. This is also the state, initially initialized with the string "fly". The parameter `completed:` can be used to specify a lambda function that is called when the animation has finished playing. However, this does not work reliably and is only mentioned here, because the parameter is used in some tutorials. Experiments have shown that, depending on the animation, this event was not always called. The reasons are not clear and future versions might react more reliable.

When the widget is created, (from line 27), a scaffold is created, in the body is the described *NimaActor*, the scaffold also has a button. If this button is pressed, the state is changed to "shot." The widget is recreated and *NimaActor* gets a new animation, the starship, which was designed in section 3.3, shoots. To play the flight animation again after the shot animation has ended, the `completed:` handler is not used as mentioned, but waits a fixed time until the animation is finished: it is 1000 milliseconds in "fly" mode. The `Future.delayed(...)` method provides this functionality. The method also has the advantage that you can control the sequence exactly by programming, independent of the animations. Figure 4.13 shows a screenshot of the animation with the button.

Listing 4.11: pubspec.yaml

```
1  environment:
2    sdk: ">=2.12.0 <3.0.0"
3
4  dependencies:
5    flutter:
6      sdk: flutter
7    nima: ^1.0.5
8
9  dev_dependencies:
10   flutter_test:
11     sdk: flutter
12
13 flutter:
14   uses-material-design: true
15   assets:
16    - assets/Ufo.nma
17    - assets/Ufo.png
18    - assets/Hop.nima
19    - assets/Hop.png
```

Listing 4.12: main.dart

```
import 'package:flutter/material.dart';
import 'package:nima/nima_actor.dart';

void main() => runApp(MaterialApp(
      title: 'Nima',
      debugShowCheckedModeBanner: false,
      home: MyHomePage(),
    ));

class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  String _anim = "fly";
  NimaActor _ufo() {
    return NimaActor("assets/Ufo.nma",
        alignment: Alignment.center,
        fit: BoxFit.scaleDown,
        animation: _anim,
        mixSeconds: 0.01,
        completed: (String animationName) {});
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Nima Example'),
      ),
      body: _ufo(),
      floatingActionButton: new FloatingActionButton(
        onPressed: () {
          setState(() {
            _anim = "shot";
            print("Buff");
            Future.delayed(Duration(milliseconds: 1000))
                .then((value) => setState(() {
                      print("Ready");
```

```
                    _anim = "fly";
                  }));
          });
        },
        child: new Icon(Icons.arrow_upward),
      ),
    );
  }
}
```

Fig. 4.13: Nima Example

4.4.5 An animated Backdrop Component

The following is an example of a self-made component, a so-called "Backdrop." It is a material design component (see: [34]). The component basically consists of two layers, a background and a foreground layer. The foreground layer reaches from below slightly into the screen area and can be moved up over the screen on demand. The backdrop is a good example because it illustrates the material design principles (see section 4.2). In contrast to dialog windows that suddenly appear and disappear, the front card is moved in and out into the visible area. Figure 4.14a shows an expanded component, figure 4.14b a collapsed component. A similar component can be found in the central repository, but our implementation is less complex. With the help of the backdrop component it is explained how animations work.

In Listing 4.14, you can see the *pubspec.yaml* file. There you can see the name of the package, a description, a version and the author.

In Listing 4.13, the code for the component itself can be seen. It is also a widget, with State. In line 14 the constructor is shown. As parameter when creating a new backdrop, we have to specify: a callback function, which is called when the state variable of the component's state changes (line 25: `panelIsOut`). In addition, the constructor contains the title as text, an icon that is pressed to extend and retract, a `base` widget with the rear level and a `content` widget with the front level and the overall height. It does not matter how the two layers are structured here, the main criterion is that they are widgets. Finally, you can optionally provide a key to identify the component for testing purposes (more about this later in section 4.5). In the method `initState()` the state variable `panelIsOut`) is initialized with `true`, because the component is extended first. Furthermore, an animation controller is created, which controls the animations (line 30). Here the duration of the animation with 500 milliseconds is also specified.

This controller must have a so-called *TickerProvider* (parameter `vsync:`), which ticks like a clock, thus allowing time to run out. In line 22, this is added in the state object as a mixin. The *SingleTickerProviderStateMixin* only ticks when the respective component is active. That is why this procedure seems a bit cumbersome. From line 37 on, a *Scaffold* is created as a framework, which gets a header (`appBar:`) with the title passed before. At this line, an icon button is inserted on the left (`leading:`) with the animated icon that is also passed in the constructor. If this *IconButton* is pressed, the method `changePanelState()` is called from line 56 onwards, which then inverts the state variable. Additionally, the callback function passed in the constructor is called. Inside the component, developers can additionally implement actions as side effect on the folding and unfolding. Afterward the animation controller is activated, which, depending on the state of the state variable `panelIsOut`, animates to the value -1 or 1, thus changing the animation slightly every tick.

If you want to use an *AnimationController*, you have to provide a *TickerProvider*. This ensures a synchronized time sequence. You can use this as a mixin in your class.

As body of the structure, a *LayoutBuilder* is created in line 50, which generates the actual layout tree starting at line 65. In the parameter `color:` in the topmost container widget, not a fixed color is defined, but, depending on the color theme, the primary color is used as color. The child is a stack, thus a stack of widgets. Exactly two widgets are specified here, namely the background passed in the constructor (`base`) and an animated surface: This consists of a transition, the *PositionedTransition* (also a widget). This, in turn, contains a rectangle transition: (`rect: RelativeRectTween`), where the start and end rectangle corner coordinates are specified. This is then animated (`.animate(...)`). This method returns the animation that must be specified in `rect:`. In the curve animation passed there, the previously created controller is specified as "parent " (line 75). The *CurvedAnimation* ensures a smooth and not abrupt temporal progression. The child of the transition widget is a material, thus a surface. There the corners are rounded and its child is the `content` passed to the constructor. If the value of `panelIsOut` changes, the controller becomes active and the tween, thus the fade, creates transition values for the specified rectangles, so the impression is that the area moves in and out.

Listing 4.13: backdrop.dart

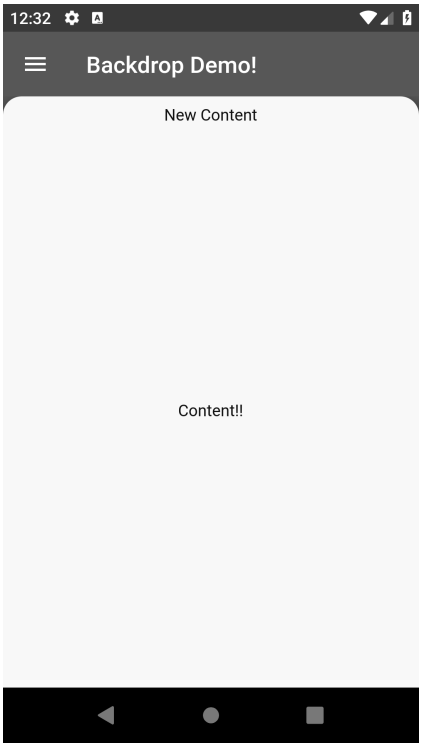
```
1  library oth_backdrop;
2  import 'package:flutter/material.dart';
3
4  class BackDrop extends StatefulWidget {
5
6    Widget content;
7    Widget base;
8    Widget title;
9    AnimatedIconData animatedIcon;
10   double height;
11   Function callback;
12   Key testKey;
13
14   BackDrop(this.callback, this.title, this.animatedIcon, this.base,
         this.content, this.height, [this.testKey = const ValueKey(0)]);
15   @override
16   State<StatefulWidget> createState() {
17     return _BackdropPageState();
18   }
19 }
20
```

```
class _BackdropPageState extends State<BackDrop>
    with SingleTickerProviderStateMixin {
 late BackDrop _backDrop;
  late AnimationController _controller;
  late bool panelIsOut;
  @override
  void initState() {
    super.initState();
    panelIsOut = true;
    _controller = AnimationController(
        duration: Duration(milliseconds: 500), value: 1.0, vsync:
            this);
  }

  @override
  Widget build(BuildContext context) {
    _backDrop = context.widget as BackDrop;
    return Scaffold(
        appBar: AppBar(
          elevation: 0.0,
          title: _backDrop.title,
          leading: IconButton(
            onPressed: changePanelState,
            key: _backDrop.testKey,
            icon: AnimatedIcon(
              icon: _backDrop.animatedIcon,
              progress: _controller.view,
            ),
          ),
        ),
        body: LayoutBuilder(
          builder: _buildStack,
        ),
      );
  }

  void changePanelState() {
    setState(() {
      panelIsOut = ! panelIsOut;
    });
    _backDrop.callback(panelIsOut);
    _controller.animateTo(
```

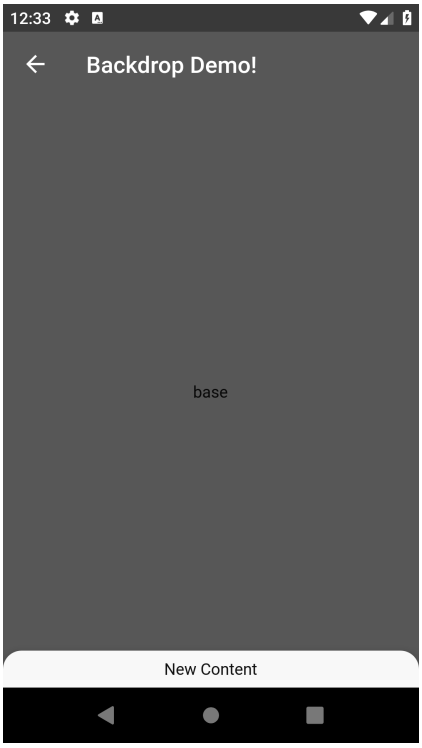
```
62          panelIsOut ? 1.0 : -1.0);
63    }
64
65    Widget _buildStack(BuildContext context, BoxConstraints constraints
          ) {
66      return Container(
67        color: Theme.of(context).primaryColor,
68        child: Stack(
69          children: <Widget>[
70            _backDrop.base,
71            PositionedTransition(
72                rect: RelativeRectTween(
73                begin: RelativeRect.fromLTRB(0.0, constraints.maxHeight
                      - _backDrop.height, 0.0, 0.0),
74                end: RelativeRect.fromLTRB(0.0, 0.0, 0.0, 0.0),
75      ) .animate(CurvedAnimation(parent: _controller, curve: Curves.
            easeInOut)),
76              child: Material(
77                borderRadius: const BorderRadius.only(
78                    topLeft: const Radius.circular(16.0),
79                    topRight: const Radius.circular(16.0)),
80                elevation: 12.0,
81                child: _backDrop.content,
82              ),
83            ),
84          ],
85        ),
86      );
87    }
88    @override
89    void dispose() {
90      _controller.dispose();
91      super.dispose();
92    } }
```

Listing 4.14: pubspec.yaml

```
1  name: oth_backdrop
2  description: A package for a backdrop
3  version: 0.0.1
4  author: Dieter Meiller
```



(a) Backdrop open



(b) Backdrop closed

Fig. 4.14: Backdrop Component

Listing 4.15: pubspec.yaml

```
1    oth_backdrop:
2      path: ../backdrop_package
```

4.5 Automated Tests

Flutter offers its own test framework for automated testing of user interfaces. In contrast to classical unit tests, where functions are tested to see if they deliver the desired values, UI tests simulate user behavior. Input can be generated for individual widgets by pretending that a person presses a button with a finger, for example. Usually the test scripts are located in the `test` folder of the project.

With Android-Studio you have to take care that the automated code in the test folder always contains a Hello World Test when you create a project with the project wizard. If you change your code, the test will usually not work, because it accesses possibly deleted objects. You can also delete the code altogether if you do not want to create tests for experiments.

In Listing 4.16, the backdrop component is tested. The corresponding package `flutter_test` is imported in line 1. First, a sample content for the backdrop window is created (lines 12-18). Here the backdrop protrudes from the screen by 32 pixels and the actual content is centered on the screen (text "Content"). In lines 19-21, the background and a simple text ("base") is generated. In line 22, a function is defined that is called when the component changes. In it a bool variable is negated, which represents the status and is checked later. The function `testWidgets` (line 23) starts the test and gives it a name — here "Test Callback." The Lambda function, which contains the test code, is given a virtual tester (widget tester) that can simulate user input. First a *BackDrop* object is created and started in line 24. Here the method `.pumpWidget(...)` `runApp(...)` is called and waits until everything is initialized. Then the status variable to be tested is cached to be able to compare later whether it has changed. Now the touch of the button that opens and closes the component is simulated. For this purpose, the component is first identified by a key generated in line 10 and passed in the constructor (line 24, 27). The *Find* object provided by the framework offers a corresponding method `find.byKey(Key key)`. Note that you can give all widgets such a key in the constructor so that you can identify all widgets.

The *Find* object can do even more, for example, it can find a specific instance if you have access to it using `find.byWidget(Widget widget)`. Or it can search for text in text-based widgets using `find.text(String text)`. As return, you get a *Finder* object representing the found widget. Here, the found widget is first pressed (line 28), then the component is waited until it is extended or retracted (line 29). Then, in line 30, `expect(...)` checks whether the result of the interaction meets the expectations. Here the old status is compared with the new status. If the test was successful, the name of the test is marked with a check mark in the console, otherwise an exception is thrown. While writing such tests is time-consuming and does not replace manual testing. You can run these tests again and again. For example, you could check if a login mechanism always works. The virtual tester (WidgetTester) can also simulate text

input (`.enterText(Finder finder, String text)`). So, you can enter the username or password in a text field found by the finder.

In Flutter, you can run automated GUI tests by simulating user interactions. The individual widgets can be found through the *Find* object. Widgets can optionally get a self-defined key to identify them.

Listing 4.16: backdrop_test.dart

```
import 'package:flutter_test/flutter_test.dart';
import 'package:flutter/material.dart';
import 'package:oth_backdrop/backdrop.dart';

late bool status;

void main() {
  const _PANEL_HEADER_HEIGHT = 32.0;
  status = true;
  final key = Key('TestKey');
  var title = Text("Backdrop Demo!");
  var content = Column(children: <Widget>[
    Container(
      height: _PANEL_HEADER_HEIGHT,
      child: new Center(child: new Text("Panel")),
    ),
    Expanded(child: Center(child: Text("Content")))
  ]);
  var base = Center(
    child: Text("base"),
  );
    var callback = (var stat) => status = ! status;
    testWidgets('Test Callback', (WidgetTester tester) async {
      final BackDrop app = BackDrop(callback, title, AnimatedIcons.
          menu_close, base, content, _PANEL_HEADER_HEIGHT, key);
      await tester.pumpWidget(MaterialApp(home: app,));
      bool oldstat = status;
      Finder f = find.byKey(key);
      await tester.tap(f);
      await tester.pump(Duration(milliseconds:1500));
      expect(status, !oldstat);
    });
  }
```

Part II: **Practice**

In the second part of the book more complex projects are presented and explained: a cloud-based application, a drawing app for the desktop and finally a game project that includes a web server.

5 Cloud Based Application

Modern web applications requires persistent storage. Tis book discusses various options for storing data. A local memory on the device is important to store data like settings of the app or to cache data from the internet in case the connection is lost or if the service used is currently unavailable. One advantage of apps over web pages is their local availability.

The local memory is used in combination with a non-local memory. There are also different possibilities, two of them are presented here in the book: In the game from chapter 7 (sections 7.5.1 and 7.5.2), a separate server service, implemented in Dart, is presented. However, you can also use a cloud service from a commercial provider.

5.1 Google Firebase

Google Firebase is a cloud service that integrates well with Flutter. We explain step by step what Firebase does, how to set up the Firestore database and how to link it to your own app.

Firebase not only provides cloud storage, it also offers an extensive range of features [37]: It allows you to analyze the app and has a link to Google Analytics. It means that you can analyze the access statistics or program crashes, even A/B-Testing can be realized via Firebase. Furthermore, you can have Google perform calculations on their servers and reduce the load on the app or your own server. And as the services scale very well, so you do not have to expect overload. You can also host files and assets of the app there. Machine Learning features can also be realized via Firebase. So, text or face recognition can be implemented relatively easily. [11].

Here the storage of app data in the real-time database Firestore is explained. It is interesting to know that Firebase was an independent start-up company in the beginning and bought by Google. The first product was the real-time database. Firestore is its successor. The old model might be maintained further, because it is still in use in old products.

5.1.1 Setup

First you need a Google Account. With this you can log in to the Firebase Console (at: https://console.firebase.google.com) and then create a project. This is easy, simply click on the “+” and enter a project name to start a project-“wizard”, questions are asked that you have to answer and click on the “Next” button. As a first step, you will

11 The description of all possibilities would go beyond the scope of this book (see [37])

https://doi.org/10.1515/9783110721331-005

be asked if you want to enable Google Analytics for your project, by default this is recommended. Then you need to select the physical location, the country, and then the project will be created.

After that, you have to register an app in which you want to use Firebase. Four options are offered: You can choose between an iOS, Android, web, or Unity app. Only the iOS and Android versions are relevant for Flutter. The instructions in the example are limited to Android, the iOS version works very similar.

After selecting the Android app, four steps are displayed which you should perform:

- Register App
- Download configuration file
- Add to Firebase SDK
- Run the app to check the installation

Register App: You will be asked for a package name. Here a Java-like package name must be specified. The logic behind the package names is the same as in the URI schema, but vice versa. For example, the name "com.mystartup.myapp" would be a convenient name for the package. Optionally you can define a SHA-1 value of a signature certificate, which is necessary for certain Firebase services. However, this is not mandatory for using the Firestore.

Download configuration file: The file "google-services.json" must be downloaded and moved to a specific project folder. In contrast to the folder structure, specified in the wizard, you have to make sure that you do not move the file to the root directory of the project, but to the "android/app" folder of the project (or ios in the iOS version).

Add Firebase SDK: Now you have to change the file "android/build.gradle": The classpath for the Google services must be added:

```
buildscript {
  ...
  dependencies {
    ...
    classpath 'com.google.gms:google-services:4.3.4'
  }
}
```

Then you have to change the file "android/app/build.gradle," another "build.gradle" file, which is located one level deeper in the "app" folder. Three lines must be added, and the minimum SDK version has to be changed here:

```
...
defaultConfig {
    ...
    minSdkVersion 21
    ...
}
...
apply plugin: 'com.google.gms.google-services'
...
dependencies {
...
    implementation platform('com.google.firebase:firebase-bom:26.1.1')
    implementation 'com.google.firebase:firebase-analytics'
}
```

Run app to check the installation: It waits until you run the app. It connects to the Google cloud and tests if a connection is established. You can skip this step.

There are two build.gradle files in the Android project: one in the folder *android/* and one in the folder *android/app/*. In both files different changes have to be made.

5.1.2 Firestore

After setting up Firebase, you can choose the button "Create database" under the menu item "Database." You will then be asked to decide whether to start Firestore in production or test mode. The test mode is automatically terminated after 30 days. In test mode, all data is accessible from outside without security mechanisms. This should simplify the setup. However, you cannot get around them anyway, and after 30 days you might be busy with other things and would have to change the settings again. You can set the login method to "Anonymous," in the Firebase settings (gear button), then everyone can use the app without login. In this case you would have to get at least a confirmation from the user, that s/he allow to transfer data to the server. Another point in setting up Firestore is to define the location of the database servers, but here in a geographically larger scale than before, you can only choose between the continents. The question is which data is stored on the server previously selected in Firebase and why you have to make such a selection twice.

The selection of the physical location can only be made during setup. You should consider this carefully as you cannot change it afterwards.

After that, you can start entering data in the web interface (see figure 5.1). You have to create at least one collection; you can compare it with the tables of a SQL database and store data in these collections.

The structure of the Firestore database is as follows: There are three subsections. You can create a collection that contains a number of documents. These documents consist of individual data fields. Fields have a name, a data type and a value. The types are primitive data types, there are strings, numbers and boolean values. But there are also maps and arrays as types, so you can build complex JSON-like tree structures. In addition, you can create new collections in documents and repeat this recursively. There is also a reference type, fields can refer to other documents. So Firestore does meet its categorization in the class of NoSQL databases: Documents can have different fields, unlike table entries in SQL database tables. Furthermore, the hierarchy is not flat, the data structures can be tree-like. There is no type for files or binary data. However, files can be stored as Base-64 encoded strings, but there is a limit, single strings may only use a maximum of 1 MB memory. Binary data and files should not be stored here, but you could use the storage solution of Firebase.

---

i The Firebase database has three levels of hierarchy: Collection ⇒ documents ⇒ fields. In documents you can recursively create additional collections.

You cannot compare the Firestore database with SQL databases, but you could say that the collections correspond to the tables and the documents to the entries with the fields, as long as you follow a fixed scheme without different numbers of fields and without substructures.

Interesting is the pricing model of Firestore — 20.000 writes per month are free of charge. Data accesses beyond that cost a small amount, a fraction of a cent, for example, 0.013 dollars per access. So, you can run an app with a small number of users for free. Successful apps with higher access rates will probably generate enough revenue to cover these costs anyway. Firestore is a so-called NoSQL database. The data is hosted on servers managed by Google, but you can choose the location of the server, which may influence the price. Frankfurt is more expensive than Los Angeles and Zurich even more expensive[12]. For sensitive data, this would be worth considering, as the servers are bound to local data protection laws. However, it is not possible to estimate whether the NSA will still have a back door for access.

A collection “messages” is created for the Messenger app described below. The documents contained in this collection represent the messages. They have three fields: A time of type “timestamp,” consisting of date and time (UTC, thus world time), a message of type “string” and a user, which is also a string for convenience.

---

12 Probably because of the more elaborate safety measures .

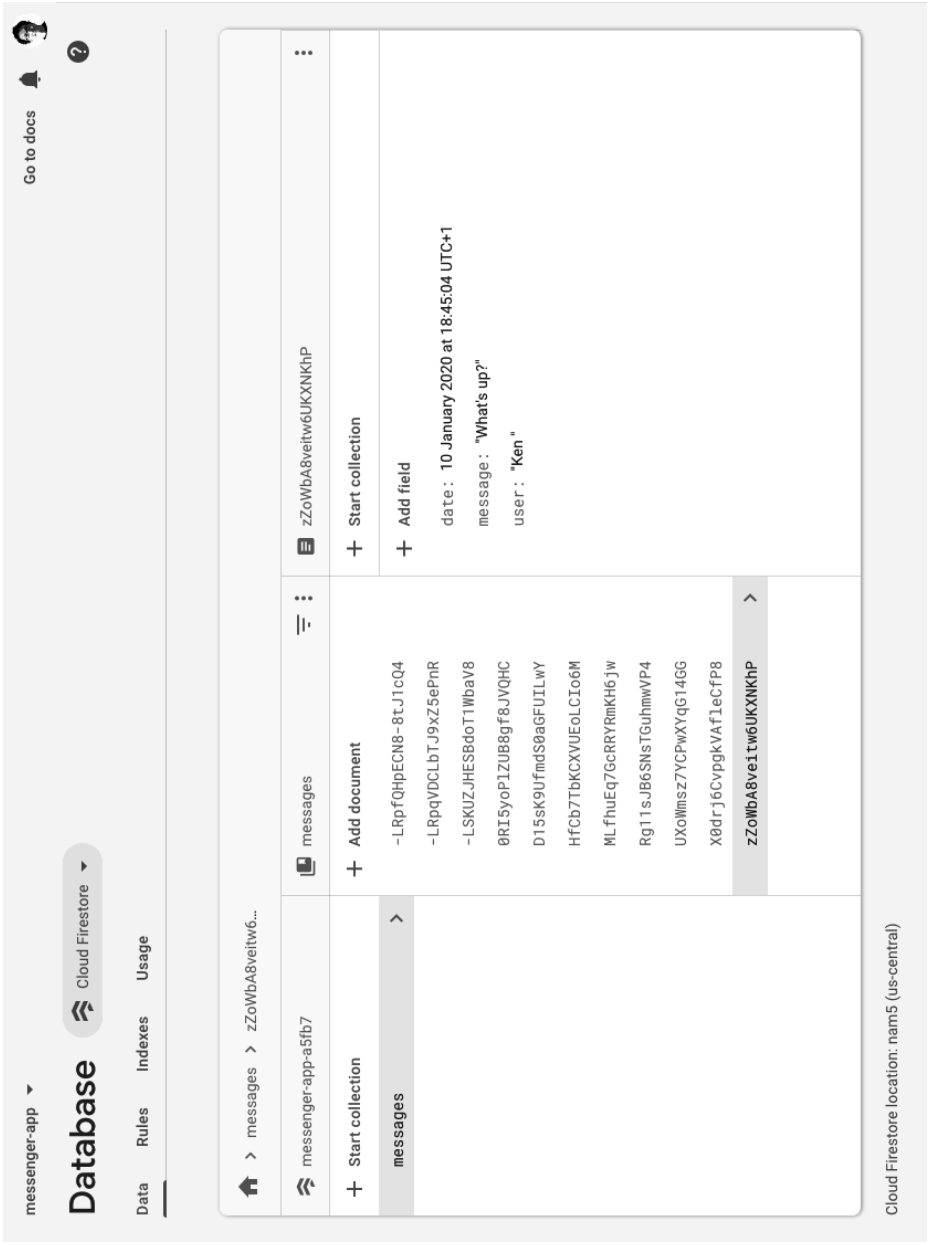


Fig. 5.1: Cloud Firestore Database

5.1.3 Messenger

In Listing 5.1, you can see the code for the Messenger app. Figure 5.2 shows the application. The Android and/or iOS setup described above must be done first for the app to connect to the cloud. In the *pubspec.yaml* of the project, the Firestore module must be loaded.

```
1 dependencies:
2   cloud_firestore: ^1.0.1
```

The code is available online at GitHub [51]. You can download it using Git with
`git clone https://github.com/meillermedia/messenger_app.git`
on the computer.

After that, you can use the library by importing it (see listing 5.1, lines 2,3). Using a *FutureBuilder*, Firebase is initialized and a *MaterialApp* with an app skeleton is created, in which a *StateFulWidget* (*Note*) is placed. Its state consists of three lists that contain the entries of the chat participants and correspond to the fields in the cloud database (lines 36-38). When the app is started (from line 45), the list is initialized empty (from line 47) and then the data is retrieved from the cloud in the method `_loadAll()`. In line 60, a reference to the collection "messages" is retrieved via an instance of a Firebase database object. A query is sent to this object, which is to return the documents sorted by date and time. A Stream<QuerySnapshot> object is provided, which can return several QuerySnapshots as a data stream. These do not already contain the individual documents, but one or more collections of documents. This was probably done because the number of documents in the database can contain very large amounts of data ("Big Data"). Thus, large volumes of data are delivered in separated packages. The documents can then be obtained from the single packages. The expression "Snapshot" (Snapshot) is probably intended to clarify that this is always a snapshot of the state of the database since it is possible that the entries change while the data is being read from the database. Since the database, as usual in this area, follows the ACID principle ("Atomicity, Consistency, Isolation, Durability") and is not locked during the readout, the snapshot might be outdated when you look at it. If a document was delivered by the stream, it is written to the state variables (starting at line 72). However, only the last 200 messages are taken over. In a "real" application you might have to delete the old messages on the server, depending on the use case of the Messenger service.

A Firestore snapshot object, as available in Flutter, is a snapshot of the state of the database. This can change continuously so the snapshot may not be consistent with the true state of the database.

Starting at line 121, the layout structure of the Messenger client is built: This has two columns, the upper column contains the messages, the lower column the input for the messages. The messages are created with a `Listview.builder(...)` call. The `itemBuilder:` parameter is called with an index variable, with all indexes from 0 to the

number of entries in the messages (-1). A `_getColor(String nick)` method (starting at line 111) maps the first letter of the user name to a color so that for most users there is a separate color for the message display. For each message, a new ListTile is created, which gets a CircleAvatar as `leading:` parameter, with the corresponding color and the first letter as text content. The title then contains the actual message. This is packed into a decorated box, with the generated color (only with 50 percent transparency) and three rounded corners. This creates the look and feel of a speech bubble as known from messenger services.

The input area initially contains two text fields, one for the user name and one for the message text. Both are linked to a controller that is called when the text in the field changes. Press the button below to send the message. Its `onPressed:` controller calls the method `sendMsg(String msg)`. However, you can also send the message by tapping ENTER on the software keyboard. The Controller `onSubmitted:` from the text field also calls the mentioned method. Starting at line 101, the system first checks whether any text has been entered. If yes, the message is sent and then scrolled to the top of the list of messages using the scroll controller. Position 0 is the beginning because the parameter `reverse:` was set to true in the ListView Builder. Saving the message is done starting at line 92 in the method `_saveMsg(String msg)`. You simply pass a map to the method `.add(Map<String, dynamic> data)` via the reference to the collection "messages." For the respective key, the string, a corresponding field is created in the new document.

The magic and the data binding of the list in the client to the server happens in the stream from line 60: When a new document is added in the cloud, the lambda function passed in the stream's `forEach(...)` method is automatically called for all documents that the request returns if the document is part of the request. This means that the stream is actually a running data stream that always returns the current query.

The stream of the collection from the Firestore database does not interrupt but delivers new elements asynchronously. This can be linked to the state so that the widget elements are automatically updated. All streams have a *forEach* method (see [26]).

In most cases, for the use of cloud services, one would add an authentication, at this point it is only explained in principle for reasons of comprehensibility. A step-by-step guide for implementing a login process can be found at [54] or [10]. The principle is briefly explained here: As mentioned, you can select the login methods in the Firebase settings. Besides anonymous, you can choose a number of providers, for example, Apple, Microsoft, Facebook, Twitter, over the phone, and others. Additionally, you need a Flutter plugin. For Flutter, there is the Google Sign-In plugin to sign in via Google service. This would be a good choice if you generate an Android app. So you need the plugin and you have to activate the appropriate login method. Furthermore, you need the Firebase-Auth Plugin to enable authentication with Firebase. Using an

asynchronous method of the Google Sign-In Plugin, you can now login. You could enable it at startup of the app or via a login button. A corresponding Google page should be displayed, with a login screen.

When setting up the Firebase app (see 5.1.1), you still have to add a SHA1 fingerprint to be able to test the app during development. You can create this fingerprint yourself using the Java keytool and then enter it into the form. If you publish a release version of the app in the Play Store, you have to create a signed APK version using the generated SHA1 fingerprint. You can also do this online via the Google Play Developer console. When activating the Google login method, you also have to specify a web page with the privacy policy, which you have to create yourself and provide the link there. Thus, you need a webspace and a website for the app (see [20]).

---

i A SHA (Secure Hash Algorithm) is the result of a cryptographic function: A plaintext of arbitrary length results in a string of fixed length. There can be several plaintexts for one hash value. Therefore, it is not possible to determine the plaintext from the hash value. (SHA1: 160 bits, SHA256: 256 bits, SHA512: 512 bits) (See: [67])

5.1.4 Google Play Games Services

As mentioned, Firebase offers many more possibilities. Additionally, Firebase can be linked to other Google services, for example, to the "Google Play Games Services (GPGS)." These allow the provision of achievements for games or a leaderboard, thus a high score list (see [40]). Through the link, you are able to analyze the behavior of the players using Google Analytics for Firebase, another service of Google. The "Google Play Games Services" allow the integration of the game into "Google Play Games", another Google cloud platform that enables multiplayer capabilities for games. This is not be discussed further. In the following example game in section 7, an own server with Dart is implemented. Alternatively, we could have used the play services. Another flutter plugin, "play_games," would be available for this (see: [56]); the exact integration and the login process to the GPGS can be read there in detail. There it also says that the login to the service is stressful, which the author can confirm. Furthermore, there is no guarantee for the smooth functioning of the Flutter plugins, so at the time of writing this book, the plugin did not work correctly. As mentioned before, it is worth reading the "issues" on GitHub.

Listing 5.1: main.dart

```
import 'package:flutter/material.dart';
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:firebase_core/firebase_core.dart';

void main() {
  runApp(NoteApp());
}

class NoteApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return FutureBuilder(
        future: Firebase.initializeApp(),
        builder: (context, snapshot) {
          return MaterialApp(
            home: Scaffold(
              appBar: AppBar(
                title: Text("Messenger App"),
              ),
              body: Note(),
            ),
            debugShowCheckedModeBanner: false,
          );
        });
  }
}

class Note extends StatefulWidget {
  @override
  State<StatefulWidget> createState() {
    return NoteState();
  }
}

class NoteState extends State<Note> {
  final txtController = new TextEditingController();
  final scrollController = new ScrollController();
  final nickController = new TextEditingController();

  var messages;
```

```
41  var users;
42  var dates;
43
44  @override
45  void initState() {
46    super.initState();
47    _clearAll();
48    _loadAll();
49  }
50
51  _clearAll() {
52    setState(() {
53      messages = <String>[];
54      dates = <Timestamp>[];
55      users = <String>[];
56    });
57  }
58
59  _loadAll() {
60    var stream = FirebaseFirestore.instance
61        .collection('messages')
62        .orderBy('date')
63        .snapshots();
64    stream.forEach((QuerySnapshot snap) {
65      snap.docs.forEach((el) {
66        var doc = el.data();
67        if (doc != null) {
68          _addMessage(doc['user'], doc['message'], doc['date']);
69        }
70      });
71    });
72  }
73
74  void _addMessage(String user, String value, Timestamp date) {
75    setState(() {
76      messages.insert(0, value);
77      users.insert(0, user);
78      dates.insert(0, date);
79      if (messages.length > 200) {
80        messages.removeLast();
81        users.removeLast();
82        dates.removeLast();
```

```
    }
  });
}

String _date(Timestamp ts) {
  DateTime d = ts.toDate();
  return "${d.day}.${d.month}. ${d.hour}:${d.minute}.${d.second}";
}

void _saveMsg(String msg) async {
  CollectionReference fire =
      FirebaseFirestore.instance.collection('messages');
  String usern = nickController.text == "" ? "NN" : nickController.
      text;
  await fire
      .add({"message": msg, "user": usern, "date": DateTime.now().
          toLocal()});
}

void sendMsg(String msg) {
  if (msg.length == 0) return;
  _saveMsg(msg);
  scrollController.animateTo(
    0.0,
    duration: Duration(milliseconds: 500),
    curve: ElasticInCurve(),
  );
  txtController.clear();
}

Color _getColor(String nick) {
  if (nick.length > 0) {
    int code = nick.codeUnitAt(0) % 50;
    return Color.fromARGB(255, code * 5, code * 5, 255 - code * 5);
  } else {
    return Colors.brown.shade800;
  }
}

@override
Widget build(BuildContext context) {
  return Column(
```

```
123       children: [
124         Expanded(
125             flex: 4,
126             child: ListView.builder(
127                 reverse: true,
128                 controller: scrollController,
129                 itemCount: messages.length,
130                 itemBuilder: (context, index) {
131                   var col = _getColor(users[index]);
132                   return ListTile(
133                     leading: CircleAvatar(
134                       backgroundColor: col,
135                       child: Text('${users[index].substring(0, 1)}'),
136                     ),
137                     title: Container(
138                       child: Text('${messages[index]}'),
139                       decoration: BoxDecoration(
140                         color: col.withAlpha(50),
141                         borderRadius: BorderRadius.only(
142                           topRight: Radius.circular(10.0),
143                           topLeft: Radius.circular(10.0),
144                           bottomRight: Radius.circular(10.0),
145                         ),
146                       ),
147                       padding: EdgeInsets.all(10),
148                     ),
149                     subtitle: Text(_date(dates[index])),
150                   );
151                 })),
152         Expanded(
153           flex: 1,
154           child: Row(
155             children: [
156               Expanded(
157                   flex: 1,
158                   child: TextField(
159                     controller: nickController,
160                     decoration: InputDecoration(hintText: "Nickname"),
161                   )),
162               Expanded(
163                   flex: 2,
164                   child: TextField(
```

```
165                     controller: txtController,
166                     onSubmitted: (txt) => sendMsg(txtController.text),
167                     decoration: InputDecoration(hintText: "Enter
                            Message"),
168                   )),
169               ],
170             ),
171           ),
172           ElevatedButton(
173             onPressed: () => sendMsg(txtController.text),
174             child: Text("Send!"),
175             style: ButtonStyle(
176                 elevation: MaterialStateProperty.all(4.0),
177                 backgroundColor: MaterialStateProperty.all(Colors.
                        blueGrey)),
178           ),
179         ],
180       );
181     }
182   }
```

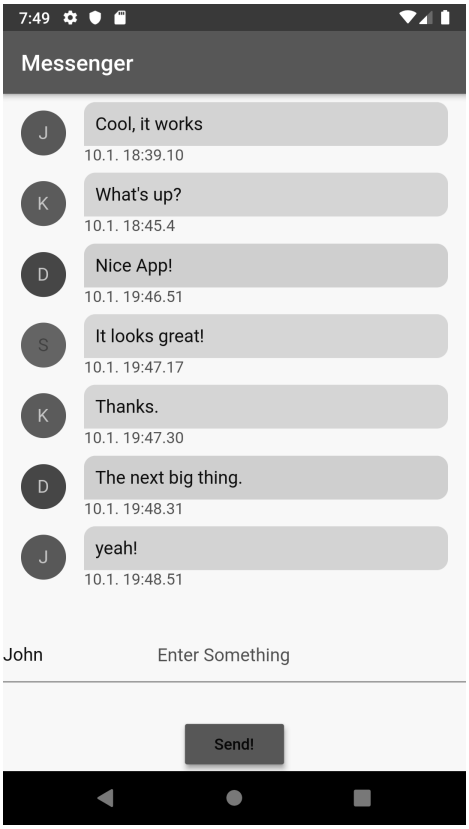


Fig. 5.2: Screenshot Messenger-App

6 Desktop App

In the following, a drawing application is described, which is a good example of state management in Flutter. Moreover, the application is platform independent, it works on mobile devices with iOS and Android as well as on the desktop under macOS, Linux and Windows. The app was tested on iOS, macOS and Linux. An application that can be used to draw must be able to load and save images. Thus, such an application is well suited to test Flutter's platform independence, since file access works differently on all platforms. In Figure 6.1, an image of the drawing app for the Mac is shown. To create native apps for desktop operating systems with Flutter, desktop support [23] has to be enabled first. In later versions this feature will be available normally. In the current version of Flutter 2, it is necessary to switch to the stable channel. Select the appropriate desktop (macOS, Linux or Windows), and then start the program on the target platform, in the example 6.1 on macOS. The target platform is usually identical to the developer platform, since it is necessary to have the appropriate development tools available here. It would be possible to work with virtual operating systems. Note that at the time of writing this book, not all modules used were null-safe. Thus, null-safety must be disabled at startup.

Listing 6.1: Desktop support activation

```
flutter channel dev
flutter upgrade
flutter config --enable-macos-desktop
flutter run --no-sound-null-safety
```

The code for the app is available online at GitHub [50]. You can download it using Git with

```
git clone https://github.com/meillermedia/draw_app.git
```

on the computer. The book omits the imports of the files from each other, the exact code can be obtained from the repository.

6.1 Access to the File System

The plugins File-Picker Cross [66] and Colorpicker [46] are used. File-Picker Cross is a special plugin, because it is one of the first that has desktop support as well. It is also needed to gain access to the file system. On the desktop, it offers a file selection dialog, which delivers the path to the selected document. The plugin also works on mobile devices, where images from the photo app can be selected, for example. It also allows write access. With the app itself, it is possible to draw images with a brush tool. Users can choose the thickness and color of the brush and can zoom or move the drawing surface with the help of additional tools.

https://doi.org/10.1515/9783110721331-006

Fig. 6.1: Screenshot Draw-App

First, the code for creating the images that will be saved and for accessing the file system is explained, as it is an essential part of the application. Listing 6.2 shows the code for reading and writing images. The asynchronous function `getImage()` loads an image from the device's persistent storage. Line 2 waits for a reference to a file. How this works will be explained below, when Listing 6.3 is discussed. The function `instantiateImageCodec(Uint8List list)` from the Dart standard library *ui* decodes the PNG information of the file and returns the image. The function `writeImage(data, width, height)` creates a PNG image and stores it: In line 9, a *PictureRecorder* is created, which has a virtual drawing canvas. On this canvas, exactly the same picture is drawn, which can be seen on the actual drawing surface (lines 10-18). First, a white background is drawn, then a background image, if available, is drawn, and then the particular lines are drawn. This process is Identical to the drawing process on the screen. Line 20 ends the recording of the image. Then the bytes of the image are extracted and stored as an integer list.

In Listing 6.3, the `save(Uint8List data)` function saves this list as a PNG file using the *FilePickerCross* object (line 12). It does all the work. It gets the data to be saved, a default file path with name and the file name extension as arguments. The method call `exportToStorage()` starts a complex process that is not evident from the

code: A dialog is opened and the user can choose a path and a file name under which he wants to store the image. If the dialog is closed with “Save”, the saving process provided by the plugin will be executed. It is worth noting here that this complex operation works on all devices, although it looks very different on mobile devices and desktop operating systems, and also works technically a different way, since access to files and file systems are different. At the end, an optional exception is caught as this operation may fail. The mentioned function `getFilePath()` uses the *FilePickerCross* method `importFromStorage(...)`. Here, line 6 waits until the user has selected a path to an image file via the file selection dialog. The path is returned as *file*. The image can then be read from the file.

Listing 6.2: io/image.dart

```
1  Future<ui.Image> getImage() async {
2    File f = await getFilePath();
3    var bytes = await ui.instantiateImageCodec(f.readAsBytesSync());
4    var frm = await bytes.getNextFrame();
5    return frm.image;
6  }
7
8  void writeImage(data, width, height) async {
9    ui.PictureRecorder recorder = new ui.PictureRecorder();
10   Canvas canvas = new Canvas(recorder);
11   canvas.drawRect(Rect.fromLTWH(0.0, 0.0, width + 0.0, height + 0.0),
12       Paint()..color = Colors.white);
13   if (data.image != null) {
14     canvas.drawImage(data.image, Offset(0, 0), Paint());
15   }
16   for (var shape in data.shapes) {
17     shape.draw(canvas);
18   }
19
20   var pic = recorder.endRecording();
21   var im = await pic.toImage(width, height);
22   im.toByteData(format: ui.ImageByteFormat.png).then((bytes) {
23     if (bytes != null) {
24       var uints = bytes.buffer
25           .asUint8List(bytes.offsetInBytes, bytes.buffer.lengthInBytes
                 );
26       save(uints);
27     }
28   });
29 }
```

Listing 6.3: io/io.dart

```
1  import 'dart:typed_data';
2  import 'dart:io';
3  import 'package:file_picker_cross/file_picker_cross.dart';
4
5  Future<File> getFilePath() async {
6    var fp = await FilePickerCross.importFromStorage(type:
         FileTypeCross.image);
7    return File(fp.path);
8  }
9
10 void save(Uint8List data) {
11   try {
12     var fp = FilePickerCross(data,
13         path: "./image", type: FileTypeCross.image, fileExtension: "
               png");
14     fp.exportToStorage().then((value) => print("Write File: $value"))
           ;
15   } on FileSystemException {
16     print("Error while Saving.");
17   }
18 }
```

6.2 The Data

The actual data that defines a drawing are described in the *DesignerShape* class (see listing 6.4). Essentially, it consists of lines, which are lists of connected points (line 13). The visual appearance of the lines is described via a *Paint* object. The helper function `createPaint(...)` (line 3) can be used to create it, specifying a color and a thickness for the lines. By calling the `draw(...)` method, the polylines are drawn on the passed *Canvas* object.

Listing 6.4: geom/DesignerShape.dart

```
1  import 'dart:ui';
2
3  Paint createPaint(Color color, double thickness) {
4    return Paint()
5      ..strokeWidth = thickness
6      ..color = color
7      ..style = PaintingStyle.stroke
```

```
    ..strokeJoin = StrokeJoin.round
    ..strokeCap = StrokeCap.round;
}

class DesignerShape {
  late List<Offset> _points;
  late Paint _paint;

  DesignerShape(Paint paint) {
    _points = <Offset>[];
    _paint = paint;
  }

  void add(Offset p) {
    _points.add(p);
  }

  void draw(Canvas canvas) {
    if (_points.length > 0) {
      Path p = Path();
      p.moveTo(_points[0].dx, _points[0].dy);
      for (int i = 1; i < _points.length; i++) {
        p.lineTo(_points[i].dx, _points[i].dy);
      }
      canvas.drawPath(p, _paint);
    }
  }
}
```

The data in the application are summarized in the data structure *DesignerData* (see listing 6.5). In addition to an optional background image, there is also a list of lines of type *DesignerShape* here. There is also a second list (*redoShapes*) which is used to implement an undo function. The constructor empties both lists.

Listing 6.5: geom/DesignerData.dart

```
import 'dart:ui' as ui;

class DesignerData {
  ui.Image? image;
  final shapes = <DesignerShape>[];
  final redoShapes = <DesignerShape>[];

```

```
8    DesignerData() {
9      init();
10   }
11   void init() {
12     shapes.clear();
13     redoShapes.clear();
14   }
15 }
```

Listing 6.6 contains a class that draws the mentioned data structure. It is a subclass of the widget *CustomPainter*, an empty container that provides a method `paint(Canvas canvas, Size size)` (line 19), which can be used to draw on the surface of the widget (`Canvas`). Here, the background image, if any, is drawn first. This can be used as a template for the drawing to be made, but in this example it is also used for demonstration purposes for the programming of the file loading process. All line drawings are then drawn above the image.

Listing 6.6: geom/DesignerPainter.dart

```
1  import 'package:flutter/material.dart';
2
3  enum Mode { pan, zoom, draw }
4
5  class DesignerPainter extends CustomPainter {
6    final DesignerData data;
7    late Size currentSize;
8
9    int get width => currentSize.width.floor();
10   int get height => currentSize.height.floor();
11
12   DesignerPainter(
13     this.data,
14   ) {
15     currentSize = Size.zero;
16   }
17
18   @override
19   void paint(Canvas canvas, Size size) {
20     this.currentSize = size;
21     if (data.image != null) {
22       canvas.drawImage(data.image!, Offset(0, 0), Paint());
23     }
24     for (var shape in data.shapes) {
```

```
25        shape.draw(canvas);
26      }
27    }
28
29    @override
30    bool shouldRepaint(covariant CustomPainter oldDelegate) => true;
31  }
```

6.3 The Layout

Listing 6.7 shows the code for the entire graphical layout. Lines 6-12 contain the widget with the root, the *MaterialApp*. Then follows the *StatefulWidget* contained in it and its state, where all the layout, state variables and interaction are coded. In lines 23-30 all state variables can be seen. The meaning of each variable becomes clear when its usage is explained. In the *initState* method (starting at line 33), they are provided with initial values. Changing them through user interactions has a direct effect on the rendering tree. When a value is changed, it is rebuilt by calling the *build* method (starting at line 68) again, creating a new layout. Starting at line 71, a *Scaffold* is built, with a *BottomNavigationBar* that provides all drawing and interaction tools as icons. When an icon is tapped or clicked (on the desktop with the mouse), the lambda function specified in the `onTap:` argument (line 78) is called with the index number of the icon. This solution is a bit messy, because when adding more icons, it is necessary to adjust the numbering. Presumably it was assumed that a *BottomNavigationBar* has only a small number of icons, and the effort is small because of that. The interaction will be explained later, now the layout will be discussed first. The icons themselves can be found starting at line 155. These are individual *BottomNavigationBarItem* widgets with an icon and a text label. The icons are chosen from Flutter's extensive icon collection (class *Icons*). Here you can find almost all material icons of the material design, but not all.

In the `body:` of the scaffold (line 185) the actual drawing area can be found, but wrapped by an *InteractiveViewer*, which allows to zoom in and move the drawing. This has the actual drawing area with its *CustomPainter* as a child. This has a detector as a child, which handles user interaction with the drawing area and is responsible for drawing the lines. Finally, its child is a *Container*, that determines the size of the area on the app.

6.4 The Interaction

After describing the layout, the explanation of the interaction follows, starting with the tools in the bar. When clicking on the first icon, the house icon, the case "0" is executed (line 80). The combination `switch` and `enum` has found to be useful here and in other places. If you program the case selection with enums, you will get a warning from the development environment if you do not consider all cases. Now, when the house icon is pressed, the state variables that are responsible for moving and enlarging the drawing area are reset to their initial values. The `value` property of the *TransformationController* object inside the *InteractiveViewer* widget affects the transformation (translation and scaling). When changing the state variables, there is no need to explicitly cause a redraw of the layout. Flutter does this by itself, since there is a dependency of the layout on the state. When the icons with the indices 1-3 (lines 86-94) are pressed, the drawing mode is changed. This *mode* is defined in listing 6.6. It has an effect on which button is shown as selected. The auxiliary function `getSelectedIndex(Mode mode)` (line 271) returns a different number depending on the mode. The `currentIndex:` parameter of the *BottomNavigationBar* (line 77) specifies the returned index number by calling the function. So, always only one selected button for one of the three modes will be highlighted. The selection of the color is handled in the case "4" starting at line 106, where a dialog box of type *SliderDialog* is opened to select the color. The code is not printed in the book, because it is not essential for the example, you can see it online. The dialog makes use of the Colorpicker plugin, which provides the color selection functionality. Case "5" opens another dialog box with a slider to select the stroke width. The state variables are set to the selected values, after the dialog boxes are closed. The cases "6" and "7" (lines 120-139) control the Undo-Redo functionality. In the "6" undo case, the last stroke is removed from the visible stroke list and added to the *redoShapes* list. In the redo case "6", this is put back again. The view of the buttons is updated in the process, which should indicate whether Undo or Redo is possible, depending on the fill status of the lists. The cases "8" and "9" (lines 140-151) trigger the loading and saving process of the image, which has already been described.

6.4.1 The Drawing Process

The *GestureDetector* monitors the user interaction with the drawing surface. The application should work cross-platform, thus working with a finger on mobile devices as well as on the desktop with a mouse should be possible. It would make sense to use a pen to draw on the desktop or even on the tablet. This is mentioned because the Flutter plugin *InteractiveViewer* automatically takes care of zooming and panning the drawing area but requires multitouch operations for these interactions when using gestures. With a pen or mouse, however, a single touch point on the sensitive surface is assumed. Thus, an additional operations for zoom and pan has to be programmed, that works

with only one touch point. That is why there are the two buttons for zoom (magnifying glass) and pan (hand). So, one can switch between drawing, zoom or pan with the pen or mouse. Multiple finger operation also works additionally, thanks to the built-in functionality of the *InteractiveViewer* widget. For the desktop, it also allows zooming with the mouse wheel. The *GestureDetector* (Line 193) offers three events to handle: `onPanDown:` when touching the area for the first time, `onPanUpdate:` when moving the pointer on the area, `onPanEnd:` when finishing the operation. Depending on the mode, the action must be different. With `onPanDown:` in drawing mode, a new line is added to the lines, with this a first start point added at the pointer position. In zoom mode, the start point of the operation is saved, which will be the center for zooming in or out. With `onPanUpdate:` in drawing mode, a new point is simply added to the last, current line in the list. With pan mode, the matrix of the transform controller is translated. In zoom mode, it is checked whether the new zoom level is within the allowed limits (lines 240 and 241). If so, before zooming is performed (line 245), the area is shifted depending on the memorized center of the operation, so that zooming in or out is performed around this point (lines 242-244). At `onPanEnd:` undo is reset in drawing mode.

Listing 6.7: main.dart

```
1  import 'package:flutter/material.dart';
2  import 'package:designer/io/image.dart';
3
4  void main() => runApp(Painter());
5
6  class Painter extends StatelessWidget {
7    @override
8    Widget build(BuildContext context) {
9      return MaterialApp(
10         debugShowCheckedModeBanner: false, title: "Designer", home:
               Designer());
11   }
12 }
13
14 class Designer extends StatefulWidget {
15   @override
16   _DesignerState createState() => _DesignerState();
17 }
18
19 double _minScale = 1;
20 double _maxScale = 4;
21
22 class _DesignerState extends State<Designer> {
23   late DesignerData data;
```

```
  late DesignerPainter _designerPainter;
  late Color currentColor;
  late Color undoButtonColor, redoButtonColor;
  late double currentWidth;
  late Offset _center;
  late Mode mode;
  final TransformationController _transContr =
      TransformationController();

  @override
  void initState() {
    data = DesignerData();
    currentColor = Colors.blue;
    _setRedoButton(false);
    _setUndoButton(false);
    currentWidth = 2;
    _center = Offset.zero;
    mode = Mode.draw;
    _transContr.value = Matrix4.identity();
    super.initState();
  }

  double _getDialogWidth(BuildContext context) {
    double wh = MediaQuery.of(context).size.width;
    if (wh > 600) wh = 600;
    return wh;
  }

  _setRedoButton(onoff) {
    if (onoff) {
      setState(() => redoButtonColor = Colors.blue);
    } else {
      setState(() => redoButtonColor = Colors.black38);
    }
  }

  _setUndoButton(onoff) {
    if (onoff) {
      setState(() => undoButtonColor = Colors.blue);
    } else {
      setState(() => undoButtonColor = Colors.black38);
    }
```

```
65    }
66
67    @override
68    Widget build(BuildContext context) {
69      var size = MediaQuery.of(context).size;
70      _designerPainter = DesignerPainter(data);
71      return Scaffold(
72        bottomNavigationBar: BottomNavigationBar(
73          selectedItemColor: Colors.black87,
74          unselectedItemColor: Colors.black87,
75          selectedFontSize: 15,
76          unselectedFontSize: 5,
77          currentIndex: getSelectedIndex(mode),
78          onTap: (value) {
79            switch (value) {
80              case 0:
81                setState(() {
82                  _center = Offset.zero;
83                  _transContr.value = Matrix4.identity();
84                });
85                break;
86              case 1:
87                setState(() => mode = Mode.pan);
88                break;
89              case 2:
90                setState(() => mode = Mode.zoom);
91                break;
92              case 3:
93                setState(() => mode = Mode.draw);
94                break;
95              case 4:
96                showDialog(
97                  context: context,
98                  builder: (BuildContext context) {
99                    return ColorPickerDialog(
100                       _getDialogWidth(context),
101                       currentColor,
102                       (color) => setState(() => currentColor = color));
103                 },
104               );
105               break;
106             case 5:
```

```
107               showDialog(
108                 context: context,
109                 builder: (BuildContext context) {
110                   return SliderDialog(
111                       _getDialogWidth(context),
112                       1,
113                       300,
114                       currentWidth,
115                       (width) => setState(() => currentWidth = width));
116                 },
117               );
118               break;
119             // Undo- Redo!
120             case 6:
121               if (data.shapes.isNotEmpty) {
122                 data.redoShapes.add(data.shapes.removeLast());
123               }
124               if (data.shapes.isNotEmpty) {
125                 _setRedoButton(true);
126               } else {
127                 _setUndoButton(false);
128               }
129               break;
130             case 7:
131               if (data.redoShapes.isNotEmpty) {
132                 data.shapes.add(data.redoShapes.removeLast());
133               }
134               if (data.redoShapes.isNotEmpty) {
135                 _setUndoButton(true);
136               } else {
137                 _setRedoButton(false);
138               }
139               break;
140             case 8:
141               getImage().then((value) {
142                 if (value != null) {
143                   setState(() {
144                     data.image = value;
145                   });
146                 }
147               });
148               break;
```

```
149             case 9:
150               writeImage(data, _designerPainter.width,
                      _designerPainter.height);
151               break;
152             default:
153           }
154         },
155         items: [
156           BottomNavigationBarItem(icon: Icon(Icons.home), label: "Home
                  "),
157           BottomNavigationBarItem(
158               icon: Icon(Icons.pan_tool_outlined), label: "Pan Image")
                      ,
159           BottomNavigationBarItem(
160               icon: Icon(Icons.zoom_in_outlined), label: "Zoom Image")
                      ,
161           BottomNavigationBarItem(
162               icon: Icon(Icons.brush_outlined), label: "Draw"),
163           BottomNavigationBarItem(icon: Icon(Icons.color_lens), label:
                   "Color"),
164           BottomNavigationBarItem(
165               icon: Icon(Icons.circle), label: "Brush Size"),
166           BottomNavigationBarItem(
167             icon: Icon(
168               Icons.undo,
169               color: undoButtonColor,
170             ),
171             label: "Undo",
172           ),
173           BottomNavigationBarItem(
174               icon: Icon(
175                 Icons.redo,
176                 color: redoButtonColor,
177               ),
178               label: "Redo"),
179           BottomNavigationBarItem(
180               icon: Icon(Icons.file_upload), label: "Load Image"),
181           BottomNavigationBarItem(
182               icon: Icon(Icons.file_download), label: "Save Image"),
183         ],
184       ),
185       body: InteractiveViewer(
```

```
186         minScale: _minScale,
187         maxScale: _maxScale,
188         scaleEnabled: mode == Mode.zoom,
189         panEnabled: mode == Mode.pan,
190         transformationController: _transContr,
191         child: CustomPaint(
192           painter: _designerPainter,
193           child: GestureDetector(
194             child: Container(
195               width: size.width,
196               height: size.height,
197             ),
198             behavior: HitTestBehavior.translucent,
199              // Single Touch
200             onPanDown: (details) {
201               switch (mode) {
202                 case Mode.draw:
203                   setState(() {
204                     data.shapes.add(
205                         DesignerShape(createPaint(currentColor,
                                currentWidth)));
206                     data.shapes.last.add(details.localPosition);
207                   });
208                   break;
209                 case Mode.pan:
210                   break;
211                 case Mode.zoom:
212                   setState(() {
213                     _center = details.localPosition;
214                   });
215                   break;
216               }
217             },
218             onPanUpdate: (details) {
219               switch (mode) {
220                 case Mode.draw:
221                   setState(() {
222                     data.shapes.last.add(details.localPosition);
223                   });
224                   break;
225                 case Mode.pan:
226                   setState(() {
```

```
227                   var sc = details.delta;
228                   if (sc.distance > 0.2) {
229                     _transContr.value.translate(sc.dx, sc.dy);
230                   }
231                 });
232                 break;
233               case Mode.zoom:
234                 var val = (details.delta.dx + details.delta.dy) /
235                     (_designerPainter.width + _designerPainter.height
                        ) *
236                     2;
237                 setState(() {
238                   var scale = _transContr.value.getMaxScaleOnAxis();
239                   var newScale = scale + val;
240                   if (newScale > scale && newScale < _maxScale ||
241                       newScale < scale && newScale > _minScale) {
242                     var dx = -_center.dx * val;
243                     var dy = -_center.dy * val;
244                     _transContr.value.translate(dx, dy);
245                     _transContr.value.scale(1.0 + val, 1.0 + val);
246                   }
247                 });
248                 break;
249             }
250           },
251           onPanEnd: (details) {
252             switch (mode) {
253               case Mode.draw:
254                 setState(() => data.redoShapes.clear());
255                 _setRedoButton(false);
256                 _setUndoButton(true);
257                 break;
258               case Mode.pan:
259                 break;
260               case Mode.zoom:
261                 break;
262             }
263           },
264         ),
265       ),
266     ),
267   );
```

```
268     }
269   }
270
271   int getSelectedIndex(Mode mode) {
272     int i;
273     switch (mode) {
274       case Mode.pan:
275         i = 1;
276         break;
277       case Mode.zoom:
278         i = 2;
279         break;
280       case Mode.draw:
281         i = 3;
282         break;
283     }
284     return i;
285   }
```

Desktop support allows, with proper programming, to develop native applications, that can run on all major devices and operating systems without modification. Note that all plugins used must provide desktop support.

7 Chicken Maze

Now that it is clear how Flutter works and how to use layouts, whole screens and animations, it is time for a more advanced example.

The example that follows is a game. In order for it to be explained in a book and not to lose the character of an educational example, it should not be too complex. Nevertheless, it should not be too trivial either, as it should be an example of a professional project and contain professional features.

The code for the game is available online at GitHub [49]. You can download it using Git with

```
git clone -b edition_2 https://github.com/meillermedia/chicken_maze.git
```

on the computer. The code described in this book is that of the branch "edition_2".

7.1 Overview

The game is called "Chicken Maze." Here you can navigate a chicken through mazes across the screen. In the labyrinth, you have to avoid enemies. You have to eat as many grains as possible, which increases your score. Additionally, there are power pills, which can be consumed. Equipped with such power, the chicken can tear down brittle walls and destroy enemies. There are several levels, the difficulty and the size of the mazes increases after each level. If your chicken is killed by an enemy, one life is taken away. You can get more lives if you pick up one of the medicine packages. If the lives are spent, the game is over. If you have achieved a new high score, you are placed on the high score list. This list is also available in a database on a server. There the best 100 players are listed. There is a start screen and a drop- down menu. With this you can start the game, view the high score list or change game settings. In the settings, you can also change the player name.

7.1.1 Technical Features

A short overview of the technical features of the game follows. In the further steps, details are explained using the particular code.

7.1.1.1 Flame Game Engine

The game is based on a 2D game engine for Flutter: "Flame," developed by Luan Nico [55]. It is minimalistic, as you can read on the GitHub page. Flame provides an empty page to draw on. The rendering method is called up about every 20 milliseconds, depending on the performance, thus you can create a new drawing per frame and so change the game graphics. The logic is similar to other frameworks, like in Android or

https://doi.org/10.1515/9783110721331-007

Fig. 7.1: Chicken Sprite-Sheet

the processing project [29]. However, this logic differs from the philosophy for animated layouts in Flutter, where animations are described and then executed. However, this concept is not well suited for games.

An important feature of the engine is the display of sprites, thus animated two-dimensional images. All graphics for Flame is stored as sprite sheets in the `assets` folder. You tile all animation sequences to a single image and specify how the animation is to run over time. Figure 7.1 shows the graphic for the game character, which is the chicken. The first two images in the first line on the left are the images for the foot positions when the chicken goes to the right. Then the foot positions left/right follows for the other walking directions. In the bottom line, there are four pictures for the pause animation, where the chicken looks in different directions and closes and opens its eyes.

Flame's ability to display tiled large graphics is especially useful. With these tilemaps, you can create large worlds. These tilemaps can also have different layers. The templates for the graphics for this are provided as tiled images and in the tilemap, only the number of the image at the desired position is given.

Flame has the ability to read ".tmx" files. This XML-based file format can describe such maps, which can have multiple layers. The tiled editor, which can create these files, was already introduced in chapter 3.2. It would be difficult or impossible to create the files using a text editor only. An example for the generated XML can be seen in Listing 7.1. In line 2, the dimensions of the map and the width and height of the tiles in pixels are defined. You can set the tile dimensions in the editor. In the game, a tile has a dimension of 32 pixels squared. Next, the images of the tilesets are defined (lines 3-11), followed by the individual layers with the compressed tile numbers.

The TMX format is a format to describe tilemaps, tiled large graphics. A tilemap is essentially a list of image numbers. The corresponding images are available in the Tilesets. A tileset is a single graphic that is composed of several images of the same size, which are numbered consecutively. There can be several tilesets in a file, as well as several layers with tiles on top of each other. The tilemap has the number of tiles in horizontal and vertical width as dimension. Everything has a fixed dimension of pixels.

Inspiration for creating the game graphics can be found on the website of the OpenGameArt project [58]. There you can also find many free graphics for game tiles and sprites.

Listing 7.1: map1.tmx

```
<?xml version="1.0" encoding="UTF-8"?>
<map version="1.2" tiledversion="1.2.2" orientation="orthogonal"
    renderorder="right-down" width="10" height="10" tilewidth="32"
    tileheight="32" infinite="0" nextlayerid="5" nextobjectid="1">
 <tileset firstgid="1" name="maze" tilewidth="32" tileheight="32"
     tilecount="24" columns="4">
  <image source="tiles.png" width="128" height="192"/>
 </tileset>
 <tileset firstgid="25" name="grain" tilewidth="32" tileheight="32"
     tilecount="10" columns="10">
  <image source="grain.png" width="320" height="32"/>
 </tileset>
 <tileset firstgid="35" name="enemy" tilewidth="32" tileheight="32"
     tilecount="12" columns="4">
  <image source="enemy.png" width="128" height="96"/>
 </tileset>
 <layer id="2" name="maze" width="10" height="10">
  <data encoding="base64" compression="zlib">
   eJyFkFEKwCAMQ53CdLDCQDfvf9Nl0EIoHX48IoWkqSOltIEJ
   KthVmQYEFJBVRX2M0NzyHtVO8/PHH+Udrldz+dXlXeTlvtyv
   uB1Z32NxR1fvHexd3WGeCPubr/sLbsoDFQ==
  </data>
 </layer>
 <layer id="3" name="grain" width="10" height="10">
  <data encoding="base64" compression="zlib">
   eJxjYGBgkGYgHkiiqZeBiimgqSHFPBCQIkEfLvdKo9GyUDbI
   jYpI6mTQaJj96GaTEi4gIMmA8Ic0khixekEAAESeAvE=
  </data>
 </layer>
 <layer id="4" name="enemy" width="10" height="10">
```

```
26    <data encoding="base64" compression="zlib">
27     eJxjYBh+QG2gHUABAAAdeAAn
28    </data>
29   </layer>
30  </map>
```

The ability to play sound files is important to create ambient sound in the background as well as sounds for events that may occur during the game. Since Flutter itself does not provide any capabilities to play locally stored sound files, it is convenient that this is possible with the Flame engine. Further, in Flame the physics engine "box2d" can be included. However, our example uses its own game mechanics. Google Play Games services can also be used with Flame, but there were some bugs at the time of programming the example. In our example, we will use our own server. Also, the project should be designed to run on iOS as well; however, the Play Services only run with Android.

7.1.1.2 Localization and Responsive Design

The game should be able to adapt to all conditions. Therefore, the layout is designed in such a way that it adapts to the size of devices, thus looking good on smaller and larger cell phones and tablets. It is multilingual and supports a number of other languages in addition to English, which is supported by Flutter (see section 7.3.2).

7.1.1.3 Monetization

Although the game is intended to be platform-independent, a cloud service for monetization is still used as this also works for iOS: "Firebase Admob" from Google. Later, we explain in detail how to create the project on the Google service and how to link it to the Flutter app (see section 7.6.8.3).

7.1.1.4 Web Communication

Interesting to know is also how Flutter and Dart can be used to communicate with your own web server. Instead of using Google's Play Games Services, a web server is utilized to communicate with the scores. Cryptological mechanisms for authentication are also used to prevent fraud the best possible. However, the example is rather didactic in nature and should not be used to transmit confidential data (see the sections 7.5.1 for the client implementation and 7.5.2 for the server).

7.2 Conception

Figure 7.2 shows the individual screens and their state transitions. For each screen, there is a dart class with the same name. From the *StartPage* you can start the game with the pull-down menu (*Drawer*) or get to all possible pages, for example the *AboutPage*. Or you can start the game directly by pressing the corresponding button. The prerequisite is that you have chosen a user name, that is saved in the settings. You can do this on the *SettingsPage*. After that you can start the game. The running game can be paused, and you will get to a waiting page (*PausePage*). If you lose a life or change the level, a short message is shown for a second, then an ad is shown. If you close it, you can continue the game as long as you have lives left. As soon as all lives are used up, you get to the *GameOverPage*. If you have a new high score, it will be written into the database (*Scores*) via the Internet. The high score table (*HighScorePage*) can also be selected and displayed via the menu, which then retrieves the global high score list of the most successful players from the Internet.

7.3 The Code in Detail

The individual classes and files of the project are now explained in detail below. The extensive import statements are omitted in the listings. Listing 7.3 contains all imports of the external libraries. These correspond to the packages in the pubspec.yaml file in Listing 7.2 and are listed in the same order. The imports of the individual files from the "chicken_maze" package are omitted for the sake of clarity and are not indicated in the listings.

The following packages are required (see listing 7.3): From the Dart world "ui," which belongs to Flutter and provides types for Flutter. "convert" is needed for web operation, because the UTF-8 conversion is included. Then the Flutter and by the Flame libraries. "flame_tiled" and "flame_audio" belong to Flame as well, but they are wrappers for external modules of the tiled project and the audioplayer project. „flame_tiled" is included via a Git url, as the project had not been migrated to Flutter 2 at the time of writing. "http" is needed for web communication (section 7.5.1). "crypto" is also used for authentication by forming a hash value. "shared_preferences" allows access to the app memory to secure persistent data such as the player name and high score. "admob_flutter" allows the display of Google AdMob ads.

The whole code can be looked up in the GitHub project [49] and downloaded with the assets.

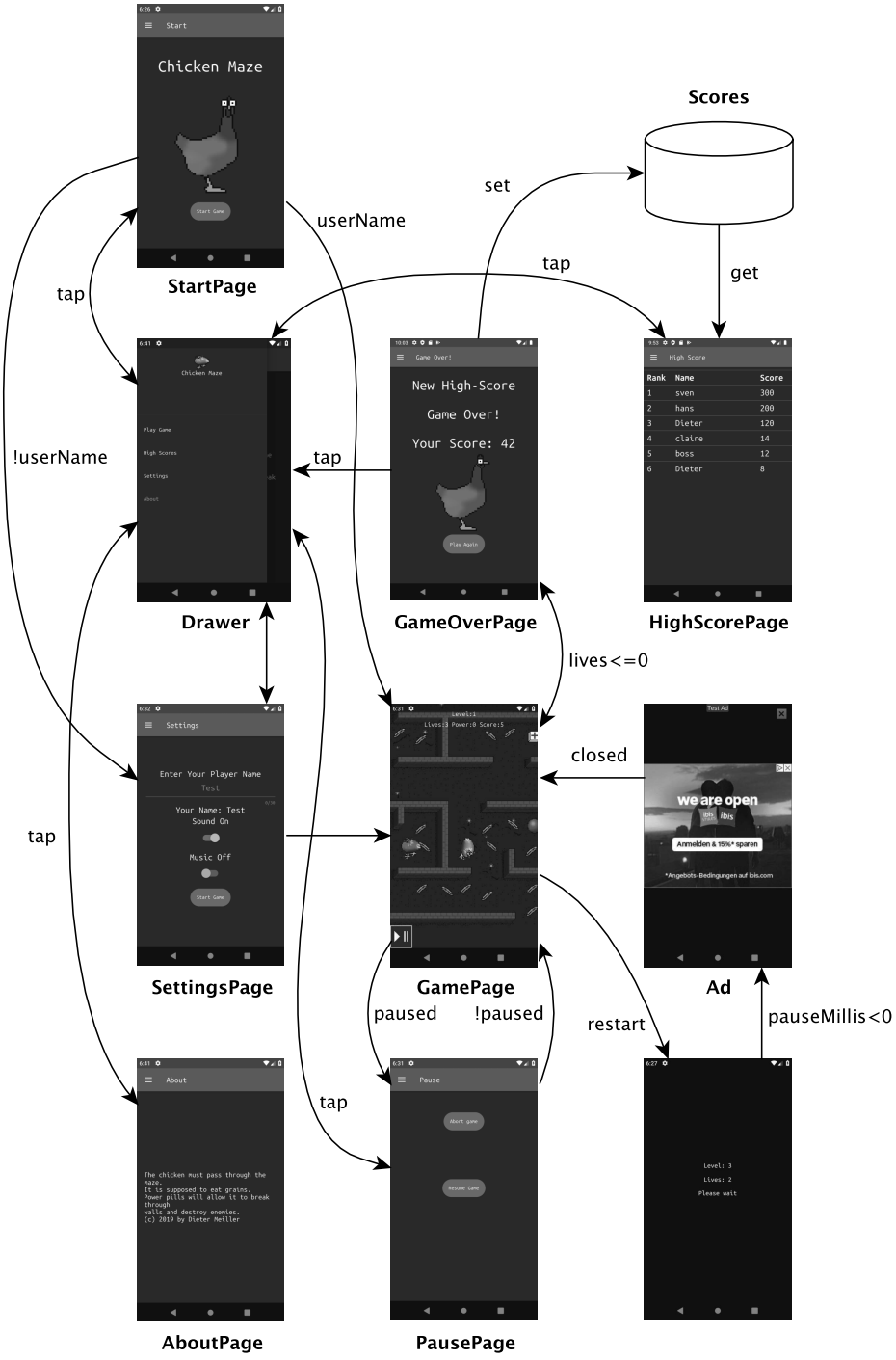


Fig. 7.2: State Diagram

For the packages specified in the 7.2 file, there are probably newer versions available from the time this book was written. You are welcome to look up the new versions in the Flutter repository [25]. However, it often depends on the combination of the individual packages in certain versions. The configuration used here can be compiled without problems.

Listing 7.2: pubspec.yaml

```
name: chicken_maze
description: A game with flutter and flame.
version: 1.1.0
publish_to: none

environment:
  sdk: ">=2.12.0 <3.0.0"
dependencies:
  flutter:
    sdk: flutter
  flutter_localizations:
    sdk: flutter
  flame: ^1.0.0-rc8
  flame_tiled: #^0.1.0
    git:
      url: https://github.com/Schnurber/flame_tiled.git
      ref: for_tiled

  http: ^0.13.1
  crypto: ^3.0.1
  shared_preferences: ^2.0.5
  admob_flutter: ^2.0.0-nullsafety.0
  flame_audio: ^0.1.0-rc5
  cupertino_icons: ^0.1.2

dev_dependencies:
  mysql1: ^0.19.0

  flutter_test:
    sdk: flutter
  flutter_launcher_icons: ^0.9.0

flutter_icons:
  android: "launcher_icon"
```

```
  ios: true
  image_path: "assets/icon/icon.png"

flutter:
  assets:
    - assets/images/logoChicken.png
    - assets/images/chickenIcon.png
    - assets/images/chicken.png
    - assets/images/enemy.png
    - assets/images/tiles.png
    - assets/images/grain.png
    - assets/images/pause.png
    - assets/tiles/map1.tmx
    - assets/tiles/map2.tmx
    - assets/tiles/map3.tmx
    - assets/tiles/map4.tmx
    - assets/tiles/map5.tmx
    - assets/tiles/map6.tmx
    - assets/tiles/map7.tmx
    - assets/tiles/map8.tmx
    - assets/tiles/map9.tmx
    - assets/tiles/map10.tmx
    - assets/audio/chicken.mp3
    - assets/audio/pick.mp3
    - assets/audio/cry.mp3
    - assets/audio/music.mp3

  uses-material-design: true
  fonts:
    - family: UbuntuMono
      fonts:
        - asset: assets/fonts/UbuntuMono-Bold.ttf
        - asset: assets/fonts/UbuntuMono-Regular.ttf
# flutter downgrade v1.22.4
# flutter run --no-sound-null-safety
```

Listing 7.3: stuff/import.dart

```
import 'dart:ui';
import 'dart:core';
import 'dart:convert';
import 'dart:math';
```

```
5
6  import 'package:flutter/material.dart';
7  import 'package:flutter/gestures.dart';
8  import 'package:flutter/foundation.dart' show SynchronousFuture;
9  import 'package:flutter/services.dart';
10 import 'dart:core';
11 import 'package:flutter/material.dart';
12 import 'dart:async';
13 import 'package:flutter/foundation.dart' show SynchronousFuture;
14 import 'package:flutter/services.dart';
15
16 import 'package:flame/flame.dart';
17 import 'package:flame/game.dart';
18 import 'package:flame/extensions.dart';
19 import 'package:flame/sprite.dart';
20 import 'package:flame/components.dart';
21 import 'package:flame/game.dart';
22 import 'package:flame/flame.dart';
23 import 'package:flame/widgets.dart';
24
25 import 'package:flame_tiled/flame_tiled.dart';
26 import 'package:flame_audio/flame_audio.dart';
27
28 import 'package:tiled/tiled.dart';
29
30 import 'package:http/http.dart' as http;
31 import 'package:crypto/crypto.dart';
32 import 'dart:convert';
33 import 'package:shared_preferences/shared_preferences.dart';
34 import 'package:admob_flutter/admob_flutter.dart';
35
36 import 'dart:io';
37 import 'package:mysql1/mysql1.dart';
```

7.3.1 Start and Initializations

The main file 7.7 with the main function is the starting point of the game. In line 5, the orientation is set to portrait using the tools provided by Flame. Then the size of the screen is fetched.

7.3.1.1 Local Settings

From line 7 on, a variable for storing the current high score is created in the app's local memory: If this variable does not exist, it is initialized with 0. The constant `prefHiScore` comes from Listing 7.4. There are more constants defined. The same is done for the user name, which should only have to be defined once later and not every time after the game has started. Further settings, such as whether the sound effects are switched on or off, or the background music is on or off, should also be saved in the settings. The music and the effects are turned on by default. After, the game begins by starting an instance of the widget `ChickenApp`.

---

i You can use a local app store. You can store any value under a key (a string), even complex objects. The database itself can be understood as type Map<String, dynamic>. Example: The line `SharedPreferences.getInstance().then( (prefs) => prefs.setInt('hiScore', 23))` stores the value 23 under the entry "hiScore."

From line 27 onward, the main widget is created. But at first, instances of the game, as well as the single screens of the game, have to be created, and the variables are `final`, because they only have to be instantiated once. Thus, an instance of the game itself (Listing 7.7 line 32) is created, as well as the help screens: The screen that contains the game, the start screen that is shown at the beginning, the info screen that should show information about the game, the screen where you can change the settings of the game, the screen where the high score is shown, the game-over screen, and a screen that is shown during breaks. When the *ChickenGame* object is created in line 27, the media files available locally in the app, thus the sprites and the audio files, are loaded there in the constructor (Listing 7.23, line 49) using the AssetLoader helper class.

Listing 7.5 is an snippet of the *AssetLoader* class. In the method `init(SharedPreferences p)`, lines 24-28, the previously declared static variables are provided with values once. First, the *SharedPreferences* are stored for later use.

The variable `isPlayingEffects` is initialized first. It is always set to true if an effect sound, like the clucking of a chicken, is being played. It serves as a switch that causes only one sound to be played at a time and not several at the same time, which could lead to memory and performance problems, and it could cause acoustic confusion.

Note the use of the conditional assignment operator `??=` for initialization. Afterward, two singleton *AudioCache* objects are created, which can load and play local sound files, one object for the effects and one for the background music respectively.

The method `loadAudio()` (from line 36) then loads the sound effects. `assert(...)` ensures that the mentioned *AudioCache* objects exist. For these, the cache is first cleared, and the sounds are loaded. The log output is also deactivated. However, log output is still generated, but less than if this was not done, which can be confusing. In the help view, you can read a TODO which indicates that the native Android implementation produces log output. By the way, in Android Studio, you can display the documentation by selecting the corresponding method or expression with the cursor

and pressing the F1 key. For VS-Studio code, it is sufficient to move the cursor over it. Then the documentation is shown in a floating window.

There can be several problems when playing sound files, also the behavior can differ between iOS and Android. With Android, looped sounds that exceed a certain length have a small pause at the end. Here the problem was avoided by keeping the music loop short.

Listing 7.4: stuff/constants.dart

```
const gameFont = 'UbuntuMono';
const prefHiScore = 'hiScore';
const prefUserName = 'userName';
const defaultName = 'Unnamed';
const prefSoundEffects = 'soundEffects';
const prefMusic = 'music';
```

Listing 7.5: stuff/AssetLoader.dart

```
import 'package:shared_preferences/shared_preferences.dart';
import 'package:flame_audio/flame_audio.dart';
import 'package:chicken_maze/stuff/constants.dart';
class AssetLoader {
  static const chickenpath = "chicken.png";
  static const enemypath = "enemy.png";
  static const logopath = 'logoChicken.png';
  static const chickenSound = "chicken.mp3";
  static const pausepath = 'pause.png';
  static const pickSound = "pick.mp3";
  static const crySound = "cry.mp3";
  static const music = "music.mp3";

  static var chickenImage;
  static var enemyImage;
  static var logoImage;
  static var pauseImage;

  static var player;
  static var musicPlayer;
  static SharedPreferences? prefs;
  static bool? isPlayingEffects;

  static init(SharedPreferences p) {
```

```
25      prefs = p;
26      isPlayingEffects ??= false;
27      player ??= FlameAudio.audioCache;
28    }
29
30    static Future loadAll() async {
31      chickenImage = await Flame.images.load(chickenpath);
32      enemyImage = await Flame.images.load(enemypath);
33      pauseImage = await Flame.images.load(pausepath);
34    }
35
36    static void loadAudio() {
37      assert(player != null);
38      player?.clearCache();
39      player?.loadAll([chickenSound, pickSound, crySound, music]);
40      player?.disableLog();
41    }
42
43    static SpriteAnimation get logoAnimation {
44      return SpriteAnimation.fromFrameData(
45          logoImage,
46          SpriteAnimationData.variable(
47            amount: 4,
48            texturePosition: Vector2(0, 0),
49            textureSize: Vector2(600, 600),
50            stepTimes: [2.5, 0.5, 2.5, 1],
51            loop: true,
52          ));
53    }
54
55    static Future<SpriteAnimation> get logoAnimationLoaded async {
56      await Flame.images.load(logopath).then((value) => logoImage =
          value);
57      return logoAnimation;
58    }
59
60    static Widget getChickenWidget(double chickenWidth, double
        chickenHeight) {
61      return FutureBuilder<SpriteAnimation>(
62          future: AssetLoader.logoAnimationLoaded,
63          builder:
```

```
64              (BuildContext context, AsyncSnapshot<SpriteAnimation>
                    snapshot) {
65            if (!snapshot.hasData) {
66              return Container(
67                width: chickenWidth,
68                height: chickenHeight,
69              );
70            } else {
71              return Container(
72                width: chickenWidth,
73                height: chickenHeight,
74                child: SpriteAnimationWidget(
75                  anchor: Anchor.topLeft,
76                  animation: logoAnimation,
77                ),
78              );
79            }
80          });
81    }
82  }
```

7.3.2 Localization

Then the *MaterialApp* is created (listing 7.7 line 39), the parameter `localizationsDelegates:` gets a number of translations: a separate reference to a class with the translations of the words occurring in the game: *LangDelegate*, as well as two translation classes existing in the framework. After that, the supported languages are listed individually (see 7.7 line 46-49).

An extra file was created for translation (see listing 7.6). The class *Lang* (line 21) contains the specific translation. From line 30, there is the private variable `_lokalizedValues` of the type *Map*, which contains as keys first the language codes, e.g., `en` for English and as values further maps with the identifiers as keys and as values the translations used in the respective language, for example `'StartGame': 'Starte Spiel'` in German. The method `String t (String what)`, line 60, can be used to get a string in the language set in the device. An identifier is given to this method as a parameter. In the variable `locale` initialized in the constructor, you can read the language code in the property `.languageCode`, e.g., "en." Line 61 then looks in the "en" part of the map under the entry in `what` and returns the English translation. This method can be called within a widget, calling the static method `Lang.of(context).t("StartGame")` returns "Start Game" in English. The *LangDelegate* class is responsible for providing

an instance of the *Lang* class with the appropriate language localization (see lines 12 and 13). The method `.load(Locale locale)` is called by the system with the correct localization and returns a future object containing the *Lang* object.

For Android or iOS development, there is the possibility to enter translations in text or XML files. Flutter does not offer this possibility. In this case, you have to access your own map data structures of the form Map<String, Map<String, String». Detailed instructions for multilingualism can be found under [28].

Listing 7.6: i18n.dart

```
1  import 'package:flutter/material.dart';
2  import 'dart:async';
3  import 'package:flutter/foundation.dart' show SynchronousFuture;
4
5  class LangDelegate extends LocalizationsDelegate<Lang> {
6    const LangDelegate();
7
8    @override
9    bool isSupported(Locale locale) => ['en', 'de', 'es', 'fr'].
         contains(locale.languageCode);
10
11   @override
12   Future<Lang> load(Locale locale) {
13     return SynchronousFuture<Lang>(Lang(locale));
14   }
15
16   @override
17   bool shouldReload(LangDelegate old) => false;
18 }
19
20
21 class Lang {
22   Lang(this.locale);
23
24   final Locale locale;
25
26   static Lang? of(BuildContext context) {
27     return Localizations.of<Lang>(context, Lang);
28   }
29
30   static Map<String, Map<String, String>> _localizedValues = {
31     'en': {
```

```
      'info': """The chicken must pass through the maze.
It is supposed to eat grains.
Power pills will allow it to break through
walls and destroy enemies.
(c) 2021 by Dieter Meiller""",
      'WaitForLoading' : 'Waiting for network connection',
      'ChickenMaze' : 'Chicken Maze',
      'BitteWarten': 'Please wait',
      'Start': 'Start',
      'Pause': 'Pause',
      'StartGame': 'Start Game',
    },
    'de': {
      'info' : """Das Huhn muss durch das Labyrinth.
Dabei soll es Körner Fressen.
Kraft-Pillen ermöglichen es ihm,
Mauern zu durchbgrechen und Feinde
zu zerstörten.
(c) 2021 by Dieter Meiller""",
      'WaitForLoading' : 'Warte auf Netzwerkverbindung',
      'ChickenMaze' : 'Chicken Maze',
      'BitteWarten': 'Bitte Warten',
      'Start': 'Start',
      'Pause': 'Pause',
      'StartGame': 'Starte Spiel',
    },
  };

  String t(String what) {
    return _localizedValues[locale.languageCode]![what]!;
  }
}
```

7.3.3 The Main Function and Routes

At startup in line 4 of listing 7.7 `WidgetsFlutterBinding.ensureInitialized()` is called first, this may be necessary if you use Flutter functionality before calling `runApp(Widget app)`.

The individual screens are transferred to MaterialApp as routes when the app is started: The parameter `routes:` is a map consisting of the keys that identify the pages

and a lambda function that returns the corresponding pre-instanticiated pages (starting at line 57). The keys are stored in the respective classes, as you can see later when they are explained individually. For example, the SettingsPage class has a property `route` with the string `'/settings'` as value.

The parameter `home:` (line 56) is the name of the start screen, which is the first screen that appears at startup.

In Flutter, you can realize "simple" routing with `Navigator.push(...)` or "named" routing, where the routes get a name via `.pushNamed(...)` or `.pushReplaceNamed(...)`. There you have to give a unique identifier as string as second argument (in addition to the context).

Listing 7.7: main.dart

```
1  late SharedPreferences prefs;
2
3  void main() {
4    WidgetsFlutterBinding.ensureInitialized();
5    Flame.device.setOrientation(DeviceOrientation.portraitUp);
6    Flame.device.fullScreen();
7    SharedPreferences.getInstance().then((p) {
8      prefs = p;
9      if (!prefs.containsKey(prefHiScore)) {
10       prefs.setInt(prefHiScore, 0);
11     }
12     if (!prefs.containsKey(prefUserName)) {
13       prefs.setString(prefUserName, defaultName);
14     }
15     if (!prefs.containsKey(prefSoundEffects)) {
16       prefs.setBool(prefSoundEffects, true);
17     }
18     if (!prefs.containsKey(prefMusic)) {
19       prefs.setBool(prefMusic, true);
20     }
21     AssetLoader.init(prefs);
22     AssetLoader.loadAudio();
23     runApp(ChickenApp());
24   });
25 }
26
27 final _chickenGame = ChickenGame(prefs);
28 final gamePage = GamePage(_chickenGame);
29 final startPage = StartPage(gamePage.chickenGame);
```

```
final aboutPage = AboutPage(gamePage.chickenGame);
final settingsPage = SettingsPage(gamePage.chickenGame);
final leaderBoardPage = LeaderBoardPage(gamePage.chickenGame);
final gameOverPage = GameOverPage(gamePage.chickenGame);
final pausePage = PausePage(gamePage.chickenGame);

class ChickenApp extends StatelessWidget {
  Widget build(BuildContext context) {

    return MaterialApp(
      localizationsDelegates: [
        const LangDelegate(),
        GlobalMaterialLocalizations.delegate,
        GlobalWidgetsLocalizations.delegate,
      ],
      supportedLocales: [
        const Locale('en'), // English
        const Locale('de'), // German
        const Locale('es'), // Spanish
        const Locale('fr'), // French
      ],
      debugShowCheckedModeBanner: false,
      checkerboardOffscreenLayers: false,
      checkerboardRasterCacheImages: false,
      debugShowMaterialGrid: false,
      title: 'Chicken Maze',
      home: startPage,
      routes: <String, WidgetBuilder> {
        SettingsPage.route: (context) => settingsPage,
        AboutPage.route: (context) => aboutPage,
        GamePage.route: (context) => gamePage,
        LeaderBoardPage.route: (context) => leaderBoardPage,
        GameOverPage.route: (context) => gameOverPage,
        PausePage.route: (context) => pausePage,
      },
    );
  }
}
```

7.4 The Screens

After setup, the individual screens of the game are explained. In principle, they all have a similar structure. The screens are not explained in the sequence of their appearance, but listed according to their complexity — the simple, static screens first.

7.4.1 The About-Page

The About page (see figure 7.3, is a simple page containing plain text. Since nothing much changes on the page, it is implemented as a *StateLessWidget* (Listing 7.8). A reference to the game is stored in the constructor. Line 5 contains the identifier that is used to select the page. When the widget is built, the game pauses; otherwise, it continues running in the background (line 8). A scaffold is created again. However, it will be given a new theme, thus a different look than the default one. The *Scaffold* gets an *AppBar* with the title of the page, which contains the translation of the info text in the local language (line 12).

The parameter `drawer:` gets a collapse menu, which is explained in the following. This is integrated in all screens except the game itself. The content of the page is integrated into the body, which comes from the function `_about(BuildContext context)`. It is a centered RichText with distance to the border. Unlike normal text, this text has more formatting options and can be wrapped automatically (line 27). The text itself is looked up in the translation, under the entry "info." The size of the text depends on the device and the settings, the function `getTextScale(context)` returns the scaling factor, which is multiplied by the default value of 16 pixels.

---

i You can include the constant string for the named route in the classes of the pages as property "route", then there are no problems with spelling mistakes when you refer to it.

Listing 7.8: AboutPage.dart

```
class AboutPage extends StatelessWidget {
  final ChickenGame game;
  AboutPage(this.game);

  static const String route = '/about';

  Widget build(BuildContext context) {
    game.paused = true;
    return themed(
        context,
        Scaffold(
            appBar: AppBar(title: Text(Lang.of(context)!.t("About"))),
            drawer: buildDrawer(context, route, game.prefs, game),
            body: _about(context)));
  }

  Widget _about(BuildContext context) {
    return Center(
      child: Padding(
        padding: EdgeInsets.all(16 * getTextScale(context)),
        child: RichText(
          text: TextSpan(
            style: TextStyle(
                fontFamily: gameFont, fontSize: 16 * getTextScale(
                    context)),
            text: Lang.of(context)!.t('info'),
          ),
          softWrap: true,
        ),
      ),
    );
  }
}
```

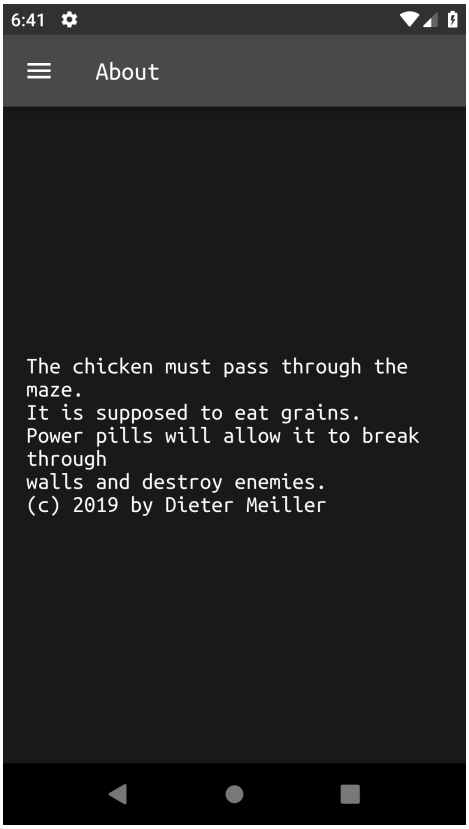
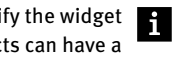


Fig. 7.3: About Page

7.4.2 The Custom Look

The file in Listing 7.9 contains the code that is responsible for giving the app its own look. The scaffold from Listing 7.8 is passed to the function `themed(BuildContext context, Widget child)`, line 16. There, it is integrated as a child into a theme widget, which allows a different look. This is set as *ThemeData* in the parameter `data:`. The private function `_themeData` in line 1 describes this appearance. There the brightness and colors, the font, as well as standard sizes of the text for headlines, title and body are defined. The previously mentioned function for determining the scaling factor for the texts can also be found in the file (in line 25, it is calculated). It is the quotient of the screen width and the standard width of a cell phone in portrait format in density independent pixels: 320 "Dips." This measure is independent of the actual pixel count, since a retina display contains more pixels than an older display. Depending on the actual resolution, several pixels are combined to a virtual pixel, so that on displays of the same size with different resolutions, the number of pixels is the same. The

calculated quotient is multiplied by the desired size. The result: On large displays, the font is displayed larger. This is not desirable for every application, because in a text application such as an editor, for example, one likes to have more space for text. However, in a game, the screen should always look the same on different sizes. In line 28, you can define the text configuration for the actual game. Here another factor is added, as the game area is scaled again, depending on the display.



There is a widget *Theme*. This has two named parameters `child:`, where you must specify the widget to style and `data:`, where you must specify the style as a *ThemeData* object. Such objects can have a lot of layout parameters, such as `fontFamily:` or `primaryColor:`.

Listing 7.9: themeData.dart

```
1  ThemeData _themeData(BuildContext context) {
2    double txtScale = getTextScale(context);
3    return ThemeData(
4      brightness: Brightness.dark,
5      primaryColor: Colors.deepOrange[800],
6      accentColor: Colors.amber[600],
7        fontFamily: gameFont,
8        textTheme: TextTheme(
9        headline5: TextStyle(fontSize: 72.0 * txtScale,
10       fontWeight: FontWeight.bold),
11       headline6: TextStyle(fontSize: 36.0 * txtScale, ),
12       bodyText2: TextStyle(fontSize: 20.0 * txtScale, ),
13       ),
14   );
15 }
16
17 Theme themed(BuildContext context, Widget child) {
18   return Theme(
19     data: _themeData(context),
20     child: child,
21   );
22 }
23
24 double getTextScale(BuildContext context) {
25    return MediaQuery.of(context).size.width / 320.0;
26 }
27
28 TextConfig gameTextConf(BuildContext context, double faktor) {
29   // double fakt = MediaQuery.of(context).textScaleFactor;
```

```
30    double fakt = getTextScale(context) / faktor;
31    return TextConfig(textAlign: TextAlign.left, fontSize: 15 * fakt,
          fontFamily: gameFont, color: Colors.white);
32  }
```

7.4.3 The Foldout Menu

The mentioned *Drawer* (fig. 7.4) is integrated into all pages (except into the game, as mentioned). The listing 7.10 contains the function that creates the drawer. It returns a *Drawer* object in line 5, with a *ListView* as child. This *Drawer* has several children, first a *DrawerHeader*, that contains as child a centered column with a picture of the chicken and the title of the game.

Then from line 16 several ListTiles follow, which contain the individual menu items. Note that not all items are printed in the book. If the identifier of the corresponding screen matches the parameter `currentRoute` passed to the function, the *ListTile* is selected and shown in a different color (e.g., in line 18).

---

i A drawer is a pop-up menu that pops up when you click on the navigation button (e.g. hamburger button). In the material design description you can also find the drawer and get information about its properties: [43]. Scaffolds have a named parameter, `drawer:`. There you can specify the Drawer object. A *Drawer* can have a child (`child:`).

Listing 7.10: Drawer.dart

```
1  Drawer buildDrawer(BuildContext context, String currentRoute,
2      SharedPreferences prefs, ChickenGame game) {
3    double ts = getTextScale(context);
4    double tss = ts * 0.9;
5    return Drawer(
6      child: ListView(
7        children: <Widget>[
8          DrawerHeader(
9            child: Center(
10             child: Column(children: <Widget>[
11               Image(image: AssetImage('assets/images/chickenIcon.png')
                     , width: 32 * ts, height: 32 * ts, fit: BoxFit.fill
                     ,),
12               Text(Lang.of(context)!.t("ChickenMaze") ,
                     textScaleFactor: ts,),
13             ],),
14           ),
```

```
15          ),
16          ListTile(
17            title: Text(Lang.of(context)!.t('PlayGame'), textScaleFactor
                  : tss,),
18            selected: currentRoute == GamePage.route,
19            onTap: () {
20              if (prefs.getString(prefUserName) == defaultName) {
21                Navigator.pushReplacementNamed(context, SettingsPage.
                      route);
22              } else {
23                game.startGame();
24                game.initLevel();
25                game.paused = false;
26                Navigator.pushReplacementNamed(context, GamePage.route);
27              }
28            },
29          ),
30          ListTile(
31            title: Text(Lang.of(context)!.t('HighScores'),
                      textScaleFactor: tss,),
32            selected: currentRoute == LeaderBoardPage.route,
33            onTap: () {
34              Navigator.pushReplacementNamed(context, LeaderBoardPage.
                      route);
35            },
36          ),
37        ],
38      ),
39    );
40  }
```

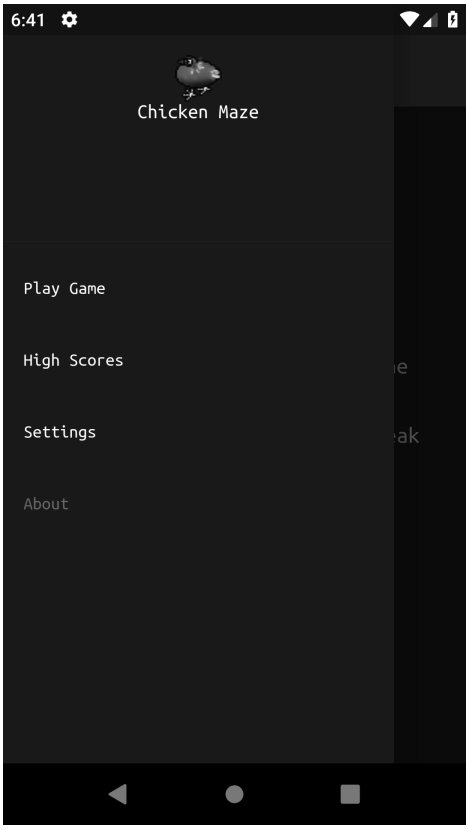


Fig. 7.4: Drawer

7.4.4 The Start Page

The start page (fig. 7.5) is similar in structure to the about page. In the listing 7.11, line 5 is the identifier for the page and the `Widget build(BuildContext context)` is identical. The content is assembled in the `Widget _start(BuildContext context)` function.

First, the chicken logo is inserted as an animation. For this purpose, a help function `AssetLoader.getChickenWidget(...)` is used, which displays a Flame animation at a certain position (lines 18 and listing 7.5, lines 43-64). Normally, animations in Flame are not widgets, but rendered directly on the game's drawing surface. In a column, first the title "Chicken Maze" and then the logo animation (line 29) is displayed. This is enlarged to the maximum.

The *SpriteAnimationWidget* of Flame basically generates a widget with a small Flame game in it. However, it only contains the specified Flame animation.

Below, there is a button (line 33). The button is a self-created widget, which is often used in the game. If it is pressed, the system checks whether a name has already been assigned to the player. If not, the settings page is displayed, where you have to enter a name first (line 36). If the name has already been changed, the game is started (lines 38-39). The text for the button comes from the translation file, as described (see 7.3.2).

Listing 7.11: StartPage.dart

```
1  class StartPage extends StatelessWidget {
2    final ChickenGame game;
3    StartPage(this.game);
4
5    static const String route = '/';
6
7    Widget build(BuildContext context) {
8      return themed(
9          context,
10         Scaffold(
11             appBar: AppBar(title: Text(Lang.of(context)!.t("Start"))),
12             drawer: buildDrawer(context, route, game.prefs, game),
13             body: _start(context)));
14   }
15
16   Widget _start(BuildContext context) {
17     var w = MediaQuery.of(context).size.width / 3 * 2;
18     Widget chicken = AssetLoader.getChickenWidget(w, w);
19
20     return Center(
21       child: Column(children: <Widget>[
22         Container(
23             padding: EdgeInsets.only(top: 50, bottom: 50),
24             child: Text(
25               "Chicken Maze",
26               style: TextStyle(fontSize: 38 * getTextScale(context)),
27             )),
28         Expanded(
29           child: chicken,
30         ),
31         Container(
```

```
32              padding: EdgeInsets.only(top: 50, bottom: 50),
33              child: FineButton(
34                  onPressed: () {
35                    if (game.prefs.getString(prefUserName) ==
                          defaultName) {
36                      Navigator.pushReplacementNamed(context,
                            SettingsPage.route);
37                    } else {
38                      game.paused = false;
39                      Navigator.pushReplacementNamed(context, GamePage.
                            route);
40                    }
41                  },
42                  text: Lang.of(context)!.t("StartGame"))),
43        ]),
44      );
45    }
46  }
```

7.4.5 The Settings Page

The settings page (fig. 7.6) is a *StateFulWidget*, as it does not always have the same state. In the listing 7.12, the state class is defined starting at line 15. The state variables are declared in lines 17-20. In the method `initState()` they are initialized with the values from the locally stored settings. The `Widget build(BuildContext Context)` method in lines 31-36 builds the framework according to the logic already described. The content of the body with the elements for configuration is created in the `Widget _settings()` method in line 39. First comes a text with the request to enter the name, followed by an editable TextField. When activated (line 50), the other GUI elements, the switches for the sounds, are hidden by setting the `areSwitchesVisible` state variable to false. In line 73, there is a *Visibility* widget with the switch and its label as a child, the visibility of the widget is linked there to the named state variable.

The reason for hiding when the text field is activated is that the software keyboard that appears needs additional space and the page becomes confusing. When the input is completed, the button is faded in again (starting at line 51) and the keyboard is hidden by taking the focus off the *TextField* (line 55). In lines 64-70, the contents of the *TextField* are used as text for the new user name. It is trimmed before, thus removing white space that surrounds the text, then the state variable `userName` and the local memory is updated by the variable `prefUserName`. The switch for activating and deactivating the sound effect within the *Visibility* widget is provided with a text

Fig. 7.5: Start-Page

in line 77, which, depending on the state variable `soundEffects`, is a translated text of "Sound on" or "Sound off." In lines 81-85, the state and preference variables are changed accordingly. In the source code there is more code for a switch to turn music on and off, which changes the state variable `music`. This is not printed in the book, because it has essentially the same function. At the bottom there is another button to start the game, if you have already changed the name, or it is deactivated (see line 96).

If widgets no longer fit on the screen due to lack of space, a black and yellow stripe is displayed as a warning, in the place where the layout is not correct. This can happen if inner widgets are larger than the screen dimensions. Especially the software keyboard should be handled with care. It is advisable to test the functionality on different devices, even in portrait or landscape format, if you have not disabled rotation. In our example, elements are hidden to balance out the lack of space. Another approach is to make the contents scrollable. For example, you could move the root widget to a *SingleChildScrollView* as its child.

Listing 7.12: SettingsPage.dart

```
class SettingsPage extends StatefulWidget {

  static const String route = '/settings';
  final ChickenGame game;

  SettingsPage(this.game);

  @override
  State<StatefulWidget> createState() {
    game.paused = true;
    return SettingsPageState();
  }
}

class SettingsPageState extends State<SettingsPage> {

  late String userName;
  late bool soundEffects;
  late bool music;
  late bool areSwitchesVisible;

  @override
  void initState() {
    super.initState();
    userName = this.widget.game.prefs.getString(prefUserName)!;
    soundEffects = this.widget.game.prefs.getBool(prefSoundEffects)!;
    music = this.widget.game.prefs.getBool(prefMusic)!;
    areSwitchesVisible = true;
  }

  Widget build(BuildContext context) {
    return themed(context, Scaffold(
        appBar: AppBar(title: Text(Lang.of(context)!.t("Settings"))),
        drawer: buildDrawer(context, SettingsPage.route, this.widget.
            game.prefs, this.widget.game),
        body: _settings()));
  }


  Widget _settings() {
```

```
40      double ts = getTextScale(context);
41      return Center(
42        child: Padding(
43          padding: EdgeInsets.only(left: 20.0 * ts, right: 20.0 * ts),
44            child: Column(
45                mainAxisAlignment: MainAxisAlignment.center,
46                crossAxisAlignment: CrossAxisAlignment.center,
47                children: <Widget>[
48              Text(Lang.of(context)!.t("EnterYourPlayerName")),
49              TextField(
50                onTap: () => setState(() => areSwitchesVisible = false )
                      ,
51                onEditingComplete: () {
52                  setState(() {
53                    areSwitchesVisible = true;
54                    //Hide Keyboard
55                    FocusScope.of(context).requestFocus(new FocusNode());

56                  });
57                },
58                maxLength: 30,
59                textAlign: TextAlign.center,
60                style: TextStyle(fontSize: 20 * ts),
61                decoration: InputDecoration(
62                  hintText: userName,
63                ),
64                onChanged: (name) {
65                  String nam = name.trim() == "" ? defaultName : name.
                        trim();
66                  setState(() {
67                    userName = nam;
68                  });
69                  this.widget.game.prefs.setString(prefUserName, nam);
70                },
71              ),
72              Text(userName == defaultName ? Lang.of(context)!.t("
                    YouHaveToEnterANameToPlay") : Lang.of(context)!.t("
                    YourName") + ": " + userName),
73              Visibility(visible: areSwitchesVisible, child:
74                Padding (
75                    padding: EdgeInsets.only(top: 5.0 * ts, ),
76                    child: Column(children: <Widget>[
```

```
77                      Text(Lang.of(context)!.t("Sound") + " " + (
                            soundEffects ? Lang.of(context)!.t("On") :
                            Lang.of(context)!.t("Off")),),
78                      Padding(padding: EdgeInsets.only(top: 5.0 * ts,
                            bottom: 5.0) * ts, child:
79                      Transform.scale( scale: ts , child:
80                      Switch(value: soundEffects,
81                      onChanged: (onoff) {
82                        this.widget.game.prefs.setBool(prefSoundEffects
                              , onoff);
83                        setState(() {
84                          soundEffects = onoff;
85                        });
86                      },))),
87                  ]),
88                ),
89              ),
90              Container(padding: EdgeInsets.only(top: 5.0), child:
                    FineButton(
91                  onPressed: () {
92                      this.widget.game.paused = false;
93                      Navigator.pushReplacementNamed(context, GamePage.
                            route);
94                  },
95                  text: Lang.of(context)!.t("StartGame"),
96                  enabled: userName != defaultName,
97                  ),
98              ),
99              ]
100           ),
101         ),
102     );
103   }
104 }
```

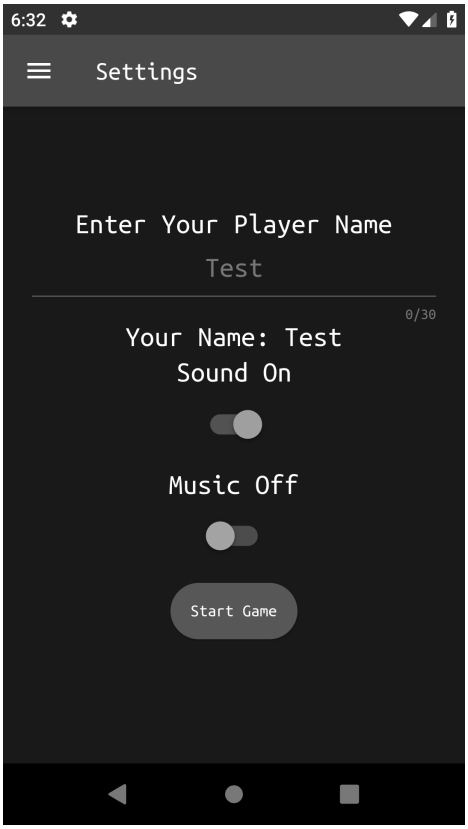


Fig. 7.6: Settings-Page

7.4.6 The Pause Page

Before explaining the game in greater detail, the screens displayed within the game are explained below. The pause page (fig. 7.7) is a *StateLessWidget* that consists of only two buttons. It is displayed after pressing the pause button in the game. In the listing 7.13, they are generated in the function `_start(BuildContext context)` (starting at line 15), which represents the body of the page. The first button cancels the running game and reinitializes it. The second button resumes the paused game: after setting the variable `paused` of the game to `false`, the navigator replaces the screen with the game screen.

Listing 7.13: PausePage.dart

```
class PausePage extends StatelessWidget {

  static const String route = '/pause';
  final ChickenGame game;

  PausePage(this.game);

  Widget build(BuildContext context) {
    return themed(context, Scaffold(
        appBar: AppBar(title: Text(Lang.of(context)!.t("Pause"))),
        drawer: buildDrawer(context, route, game.prefs, game),
        body: _start(context)));
  }

  Widget _start(BuildContext context) {
    return Center(
      child: Column(children: <Widget>[
        Container(padding: EdgeInsets.only(top: 50, bottom: 50),
            child: FineButton(
            onPressed: () {
              game.startGame();
              game.initLevel();
              game.paused = true;
              Navigator.pushReplacementNamed(context, StartPage.route)
                  ;
            },
            text: Lang.of(context)!.t("AbortGame"))),
        Container(padding: EdgeInsets.only(top: 50, bottom: 50),
            child: FineButton(
            onPressed: () {
              game.paused = false;
              Navigator.pushReplacementNamed(context, GamePage.route);
            },
            text: Lang.of(context)!.t("ResumeGame"))),
      ]),
    );
  }
}
```

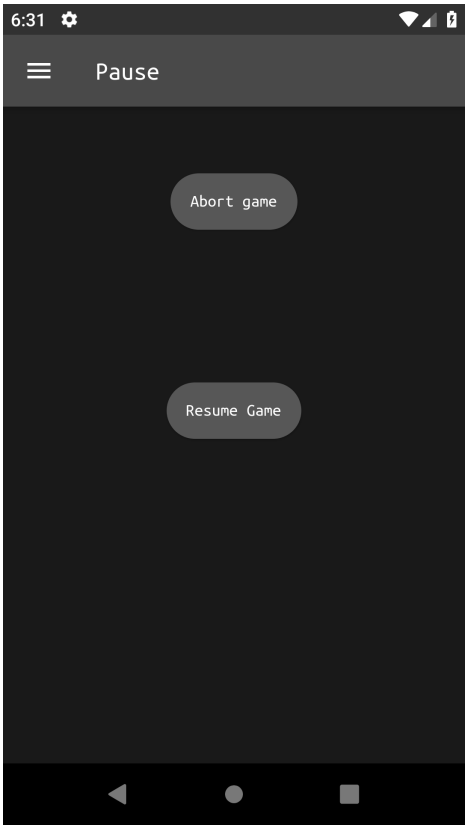


Fig. 7.7: Pause Page

7.4.7 The Game Over Page

The game over page (fig. 7.8) is a *StateLessWidget* which is displayed after the end of the game when all lives are used up. In the listing 7.14 from line 16 on, the body is created in the function `_gameOver()` which returns a *FutureBuilder<bool>* object. This is a *StateFulWidget* whose structure depends on the result of an asynchronous operation. Here, after the current score has been entered in the high score table, the text "New high score" should appear, depending on whether it is a new high score. This is decided in the function `setHiScore(in score)` from Listing 7.15, line 3, which returns a *Future<bool>* object.

The current high score from the *SharedPreferences* is waited for, which is updated from line 7 if necessary. Subsequently, the function returns true or false as a future, depending on whether it is a new high score or not. The mentioned *SharedPreferences* are delivered asynchronously via a future object, which has to be waited for (line 4).

This makes the whole action also asynchronous and leads to the result that the layout of the game over page with the mentioned text has to be generated via a *FutureBuilder*.

---

i A *FutureBuilder* widget waits for the end of an asynchronous operation `future:`. Its result is reported to `builder:`, which can build the widget differently depending on it.

This is a bit cumbersome. But this procedure illustrates the single-threaded philosophy in Dart, where there are no blocking operations. What seems complicated here is actually a simplification, because you are not required to program parallel running program parts (threads), which have to be synchronized, like it is the case in, for example, Java. So, no inconsistencies occur. Deadlocks, thus mutually blocking program parts, are also avoided. All in all, the software seems to be more fluent because there are no states in which the app freezes waiting for the result of an operation. In line 6 in Listing 7.15 the command
`await LeaderBoard.setScore(score)` updates the high score on the server (more about this later).

Listing 7.14: GameOverPage.dart

```
1  class GameOverPage extends StatelessWidget {
2    static final String route = '/game_over';
3
4    final ChickenGame game;
5    GameOverPage(this.game);
6
7    Widget build(BuildContext context) {
8      return themed(
9          context,
10         Scaffold(
11             appBar: AppBar(title: Text(Lang.of(context)!.t("GameOver")
                   )),
12             drawer: buildDrawer(context, GameOverPage.route, game.
                   prefs, game),
13             body: _gameOver()));
14   }
15
16   FutureBuilder<bool> _gameOver() {
17     var score = this.game.score;
18     return FutureBuilder<bool>(
19         future: HiScore.setHiScore(score),
20         builder: (BuildContext context, AsyncSnapshot<bool> hi) {
21           var sc = getTextScale(context);
22           var w = 170 * sc;
```

```
23          Widget chicken = AssetLoader.getChickenWidget(w, w);
24
25          TextStyle ts = TextStyle(fontSize: 32 * sc);
26          var pd = EdgeInsets.only(top: 30, bottom: 10);
27          var list = <Widget>[];
28
29          /// If it is hiscore add new text
30          if (hi.hasData && hi.data == true) {
31            list.add(Container(
32                padding: pd,
33                child: Text(
34                  Lang.of(context)!.t("NewHighScore"),
35                  style: ts,
36                )));
37          }
38
39          list.add(
40            Container(
41                padding: pd,
42                child: Text(
43                  Lang.of(context)!.t("GameOver"),
44                  style: ts,
45                )),
46          );
47          list.add(Container(
48              padding: pd,
49              child: Text(
50                Lang.of(context)!.t("YourScore") + " $score",
51                style: ts,
52              )));
53          list.add(Expanded(child: chicken));
54          list.add(Container(
55              padding: EdgeInsets.only(top: 10 * sc, bottom: 50 * sc),
56              child: FineButton(
57                  onPressed: () {
58                    game.startGame();
59                    game.initLevel();
60                    game.paused = false;
61                    Navigator.pushReplacementNamed(context, GamePage.
                          route);
62                  },
63                  text: Lang.of(context)!.t("PlayAgain"))));
```

```
64
65            return Center(
66              child: Column(children: list),
67            );
68          });
69    }
70  }
```

Listing 7.15: HiScore.dart

```
1   class HiScore {
2
3     static Future<bool> setHiScore(int score) async {
4       final SharedPreferences prefs = await SharedPreferences.
            getInstance();
5       int myHiScore = prefs.getInt(prefHiScore) ?? 0;
6       await LeaderBoard.setScore(score);
7       if (score > myHiScore) {
8         await prefs.setInt(prefHiScore, score);
9         return true;
10      } else {
11        return false;
12      }
13    }
14  }
```

7.4.8 The High Score Page

The stateful widget *LeaderBoardPage* (listing 7.16) shows the best players. The data from this is fetched from a server. From line 17 on, the layout widget is built, the content of the body is created from line 25 on as a *FutureBuilder* widget. The future, thus the data, is fetched as a two-dimensional string array from the auxiliary class *LeaderBoard* (see listing 7.18, line 3). If no data is available or could be loaded (line 31), the following text or corresponding translation is displayed: “Wait for Loading.”

In lines 36-38, lists are created for the columns that hold the ranking, the list for the scores and the names of the players. In lines 40-42, the column header entries are written. Then all entries of the list from the server are numbered and entered in a loop. Afterward, a list is generated from line 53 on, by a *ListView.builder* call. The length of the list `itemCount:` is the length of the server entries. The `itemBuilder:` is a lambda function is always given the respective index, starting from 0. A row with the corresponding list entries is generated for the current index. The entries are split using

Fig. 7.8: Game Over Page

the `flex:` parameter of the Expanded widget: 1 to 3 of the total width (lines 61-63) (see figure 7.9).

You can control the distribution of column layouts using the `flex:` factor of the *Expanded* Widget. For each *Expanded* in a row you can specify how much space it can get. If you do not specify any parameters, the columns are distributed evenly.

Listing 7.16: LeaderBoardPage.dart

```
class LeaderBoardPage extends StatefulWidget {

  static const String route = '/high_score';
  final ChickenGame game;

  LeaderBoardPage(this.game);
  @override
  State<StatefulWidget> createState() {
    return LeaderBoardPageState();
  }
}

class LeaderBoardPageState extends State<LeaderBoardPage> {

  late List<List<String>> scores;

  Widget build(BuildContext context) {
     this.widget.game.paused = true;
    return themed(context, Scaffold(
        appBar: AppBar(title: Text(Lang.of(context)!.t("HighScore"))),

        drawer: buildDrawer(context, LeaderBoardPage.route, this.
            widget.game.prefs, this.widget.game),
        body: _leaderBoard()));
  }

  FutureBuilder<List<List<String>>> _leaderBoard() {

    return FutureBuilder<List<List<String>>>(
      future: LeaderBoard.getHiScore(),
      builder: (BuildContext context, AsyncSnapshot<List<List<String
          >>> snapshot) {
        scores = snapshot.data!;
        if (scores.length == 0) {
          return Container(
            child: Center( child: Text(Lang.of(context)!.t("
                WaitForLoading"), ), ),
          );
        }
        List<String> rankList = <String>[];
```

```
37          List<String> scoreList = <String>[];
38          List<String> nameList = <String>[];
39          int i = 1;
40          rankList.add(Lang.of(context)!.t("Rank"));
41          scoreList.add(Lang.of(context)!.t("Score"));
42          nameList.add(Lang.of(context)!.t("Name"));
43          scores.forEach((r) {
44            rankList.add("$i");
45            i++;
46            scoreList.add(r[0]);
47            nameList.add(r[1]);
48          });
49          var headerStyle = TextStyle(
50            fontWeight: FontWeight.bold,
51          );
52          return Center(
53            child: ListView.builder(itemCount: nameList.length,
54                itemBuilder: (BuildContext context, int index) {
55                  return Column(children: <Widget>[
56                    Divider(color: Colors.white),
57                  Padding(
58                  padding: const EdgeInsets.only(left: 10.0, right:
                        10.0),
59                  child: Row(
60                    children: <Widget>[
61                        Expanded(child:Text(rankList[index], style:
                            index == 0 ? headerStyle : null), flex: 1,
                            ),
62                        Expanded(child: Text(nameList[index], style:
                            index == 0 ? headerStyle : null), flex: 3,)
                            ,
63                        Expanded(child: Text(scoreList[index], style:
                            index == 0 ? headerStyle : null), flex: 1,)
                            ,
64                    ],
65                  ),
66                  ),
67                  ],);
68                }),
69          );
70        },
71      );
```

```
72      }
73  }
```

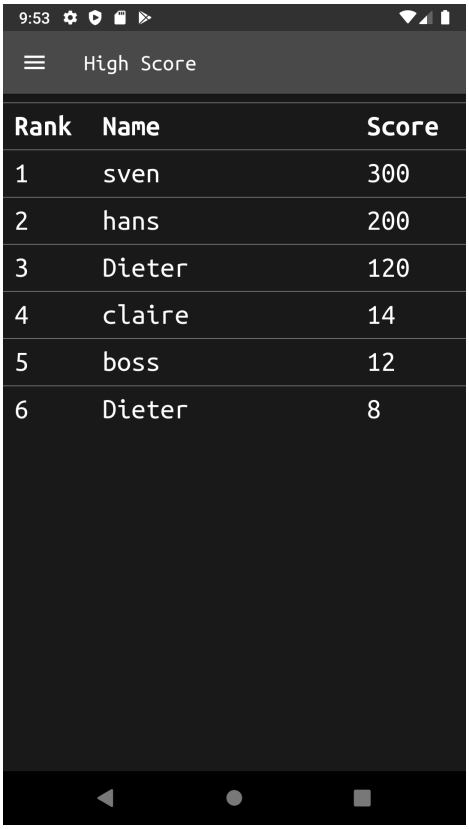


Fig. 7.9: High Score Page

7.5 Server Query

On the server, the top 100 of the best of all players are managed. How the server is implemented or manages the data is not relevant at first.

7.5.1 Client Side

The method `getHiScore()` in listing 7.18 asks the server for the high score by calling a script. This script responds to the HTTP get request (lines 6-8) and returns a result with entries separated by HTML line breaks `<br>` with the name and the high score; each separated by a comma. If the server query is not possible, an exception is thrown (line 19), which then returns the empty list.

When sending the high score data to the server, an additional authentication mechanism is implemented. The method `setScore(int score)` first fetches the username which is transmitted together with the high score (lines 26-27). Then, in lines 30-33, a hash value is formed from the name, the score, and a secret key (line 31) known to both the client and the server. If the procedure is unclear: As already mentioned, a hash value formed by the Sha-256 algorithm from a string of any length always has the same length, so that you always get the same hash value from identical strings, but you cannot calculate back the string from the hash value. The reason is that several strings can result in the same hash value. In lines 35-37, the name, score and hash value are transmitted to the server as "get" variables. There, the hash value is re-calculated and compared to see if the values match. If so, the request to enter the new score probably comes from the client who knows the secret hash value. Subsequently, the score and the name are written into the database. This procedure does not offer a safe protection against manipulation, because attackers could always manipulate the game directly and include so-called cheats, which, for example, influence the number of lives. This approach offers some protection against manipulation of the data during transmission, at least if one assumes that attackers did not extract the secret key from the app. For Android, you have to allow Internet access in the *AndroidManifest.xml* file, in the *android/app/src/main* folder (add lines 1 and 3 from listing 7.17).

Listing 7.17: AndroidManifest.xml

```
1     <uses-permission android:name="android.permission.INTERNET" />
2     <application
3         android:usesCleartextTraffic="true"
4         ...
5         >
```

HMAC (Hash-based Message Authentication Code) is a type of authentication using a hash value: A hash value is formed by a hashing procedure (here: SHA256), from a message to be transmitted and a secret key that is present on both sides of the transmission. It is transmitted together with the message. There, the hash value is formed again from the message and the secret key. If the hash values match, the message is transmitted from a location that also owns the secret key.

Listing 7.18: LeaderBoard.dart

```
class LeaderBoard {

  static Future<List<List<String>>> getHiScore() async {
    List<List<String>> scores = [];
    try {
      var httpClient = http.Client();
      var uri = Uri(host: hiScoreServer, scheme: "http", port:
          hiScoreServerPort, query: "q=get");
      var result = await httpClient.get(uri);
      print(result.body);
      List<String> lines = result.body.split("<br>");
      lines.forEach((s) {
        print(s);
        List<String> entry = s.split(",");
        if (entry.length == 2) {
          scores.add([entry[1], entry[0]]);
        }
      });
      return scores;
    } catch (ex) {
      print("Excepion: $ex");
      return scores;
    }
  }

  static Future<bool> setScore(int score) async {
    final SharedPreferences prefs = await SharedPreferences.
        getInstance();
    String userName = prefs.getString('userName') ?? "unnamed";
    try {
      var httpClient = http.Client();
      var key = utf8.encode(leaderBoardSecret);
      var bytes = utf8.encode('$userName$score');
      var hmacSha256 = new Hmac(sha256, key);
```

```
33        var hash = hmacSha256.convert(bytes).toString();
34
35        Map<String, String> params = {"q":"set", "name":userName, "
              score":"$score", "hash": hash};
36         var uri = Uri(host: hiScoreServer, scheme: "http", port:
               hiScoreServerPort, queryParameters: params);
37        await httpClient.get(uri);
38        return true;
39      } catch (ex) {
40        print("Excepion: $ex");
41        return false;
42      }
43
44    }
45  }
```

7.5.2 Server Side

Since the communication with the server occurs via the HTTP protocol, the implementation of server scripts is independent of the technology used. You could use a LAMP stack, with a Linux operating system, an Apache web server (or NGINX), with MySQL (or MariaDB) as database and PHP as scripting language. But you could also use Dart as scripting language, which is described here. Listing 7.21 is the implementation of the dart server. It consists of two functions, a function which returns the high score list and a function that creates a new high score entry. For this purpose, the script establishes a connection to a SQL database. A running database server is, therefore, needed. In Listing 7.19, you see the SQL-Script that creates the database table. The table has only three columns: one column ID, which is incremented with each entry, one column with strings for the names of the players and one entry for their score. The connections data is read from the file 7.20. The function `getConnectionSettings()` returns a ConnectionSettings object. Here you enter all information for the SQL database connection, thus its domain or IP address, the port under which the service is available, username/-password, and the database you want to use. In the configuration file you should also configure the dart server: First of all, you have to define the type of internet address the dart server should be reachable through. The package `dart:io` can handle IPV4 or V6 addresses. The port through which the HTTP server can be reached is specified. It is followed by the name of the table with the high scores and a secret key. This must match the secret key from the app, thus the game.

You can start the server with the command `dart server.dart`. Before, you have to ensure that you have installed the Mysql1 package and the crypto package of Dart. On

the development machine, you can easily do this via the global *pubspec.yaml* file by selecting the `dependencies:` or the `dev_dependencies:` the package `mysql1: ^0.19.0` and `crypto: ^3.0.1` and then install it with `flutter pub get`. On the high score server, you do not need to install Flutter. There, it would make sense to create a separate *pubspec.yaml* file with the mentioned entry. Then you can use `pub get` to install the library with the standard package manager of Dart.

The main function in Listing 7.21 creates a web server. The address at which it can be reached is then the address assigned to the host and the specified port. With a console command, like `ipconfig getifaddr en0` under macOS or Linux, you can find out the IP address that the computer and the server on the LAN has. From line 3, all requests to the server are handled. Depending on the value of the GET parameter "q" a distinction is made. If values shall be read or written, then the corresponding functions are called. The function `_foo(HttpRequest request)` is only a test function, which outputs a message as HTML text. It is used to test the HTTP server. From line 18, the function `_get(HttpRequest request)` , which generates the high-score list, is in the source code. This is simple: First the ContentType of the response is set to HTML text with Unicode character. Next, a connection to the SQL-Server is established with the described connection data. Line 21 waits for the response of an SQL query. This is a list of names and scores, arranged in descending order (DESC), according to the scores. The entries are written into the response line by line, separated by commas, and provided with an HTML line break `<br>`. The connection to the database is closed and the request is sent.

The function `_set(HttpRequest request)` has the task to write the scores into the database. As response, only a "ok" is returned. The HTML output of the functions allows you to call the HTML interface from your browser and get a response. Additional GET parameters are passed to the function call: "name," "score" and "hash." The request looks as follows:
`http://192.168.178.42:8888?q=get&name=John&score=500&hash=...` As you can see, the parameters in lines 31-33 are extracted from the request. In lines 34-37, a hash value is generated from the name, score and secret key using the Sha-256 algorithm. This is then compared in line 38 with the hash value supplied by the client. If the values do not match, the function is terminated, and the score is not transferred to the database. If an attacker from outside the client tries to submit a score, he must be able to form the correct hash value with knowing the secret key. In line 43, all characters except letters and spaces are removed from the name. Line 46 then checks whether the name and the score exist in the database. If so, the function is terminated so that no duplicate entries are present. Otherwise, line 50 is used to write the data into the database. From line 51, the number of entries is checked. If there are more than 100 entries, the excess worst score entries are removed from the database in line 55.

To test the script locally, you have to install the “mysql1” and the “crypto” package for Dart. Since the server is the local machine and these packages are already in the project’s pubspec.yaml file, they should already be installed. Next, you need the local IP address of the development machine. Otherwise, the emulator or the test device cannot connect to the development machine. Localhost “127.0.0.1” does not work. It should be an address of the kind “192.168.178.42.” Next, an SQL server should be installed on the development computer. During the creation of the example project, the XAMPP package [30] was used; the native version and not the version with the virtual Linux environment (VM), recognizable by the ending “vm.dmg” during download. So, you can easily install the database table (Listing 7.19) via PHPMyAdmin. You only have to create (send) a database and upload the SQL with the import function. Please note that the SQL version of XAMPP does not have a password set, but you have to set it when you connect. Thus, it is best to set a password for the user (root-root for the test system and the example). Of course, you can use all other possible variants of server environments for development.

Listing 7.19: db.sql

```
CREATE TABLE `highscores` (
  `id` int(11) NOT NULL,
  `username` varchar(30) NOT NULL,
  `score` int(11) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

ALTER TABLE `highscores`
  ADD PRIMARY KEY (`id`);

ALTER TABLE `highscores`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=166;
```

Listing 7.20: connectionSettings.dart

```
import 'dart:io';
import 'package:mysql1/mysql1.dart';

ConnectionSettings getConnectionSettings() {
  return new ConnectionSettings(
    host: '127.0.0.1',
    port: 8889,
    user: 'root',
    password: 'root',
    db: 'chicken',
  );
}

```

```
final InternetAddress internetAddress = InternetAddress.anyIPv4;
const port = 8888;
const highScoreTable = 'highscores';
const secret = 'secret';
```

Listing 7.21: scores.dart

```
Future main() async {
  var server = await HttpServer.bind(internetAddress, port);
  await for (HttpRequest request in server) {
    switch (request.uri.queryParameters['q']) {
      case 'get': _get(request); break;
      case 'set': _set(request); break;
      case 'foo': _foo(request); break;
    }
  }
}

void _foo(HttpRequest request) async {
    request.response.headers.contentType = new ContentType("text", "
        html", charset: "utf-8");
    request.response.write('It works!');
    await request.response.close();
}

void _get(HttpRequest request) async {
    request.response.headers.contentType = new ContentType("text", "
        html", charset: "utf-8");
    final mySqlConn = await MySqlConnection.connect(
        getConnectionSettings());
    var results = await mySqlConn.query('select username, score from
        $highScoreTable order by score DESC');
    for (var row in results) {
      request.response.write('${row[0]},${row[1]}<br>');
    }
    await mySqlConn.close();
    await request.response.close();
}

void _set(HttpRequest request) async {
    request.response.headers.contentType = new ContentType("text", "
        html", charset: "utf-8");
```

```
31      String name = request.uri.queryParameters['name']!;
32      String score = request.uri.queryParameters['score']!;
33      String hashv = request.uri.queryParameters['hash']!;
34      var key = utf8.encode(secret);
35      var bytes = utf8.encode('$name$score');
36      var hmacSha256 = new Hmac(sha256, key);
37      var testHash = hmacSha256.convert(bytes).toString();
38      if (testHash != hashv) {
39        request.response.write('err');
40        await request.response.close();
41        return;
42      }
43      name = name.replaceAll(new RegExp("[^A-Za-z ]+"), "");
44      var tstr = "select * from $highScoreTable where username=? and
            score=? limit 1";
45      final mySqlConn = await MySqlConnection.connect(
            getConnectionSettings());
46      var result = await mySqlConn.query(tstr, [name, score]);
47      if (result.length > 0) return;
48
49          var qstr = "insert into $highScoreTable values (NULL,? ,? )";

50          await mySqlConn.query(qstr, [name, score]);
51      result = await mySqlConn.query("select count(*) from
            $highScoreTable");
52          var toDelete = result.length - 100;
53
54          if (toDelete > 0) {
55              await mySqlConn.query("delete from $highScoreTable order
                    by score asc limit ?", [toDelete]);
56          }
57
58      request.response.write('ok');
59      await mySqlConn.close();
60      await request.response.close();
61  }
```

7.6 The Game Logic

So far, all pages of the app have been explained, except the actual page of the game. This is divided into classes for the game elements. The programming logic was implemented with Flutter and Dart, but such a game logic could be realized with many other languages and frameworks. Nevertheless, there are some special Dart and Flutter features. Furthermore, the project is a good example of how to realize more complex projects with these technologies. Two screenshots from two different levels can be seen in figure 7.10.

7.6.1 The Games Page

First, there is a widget that contains the actual game (see the code in Listing 7.22). The page is a stateless widget and has a string for the route (line 3). The constructor is given an instance (object) for the game from the main method. There is only one instance for the game. Here it is initialized. Finally, the widget of the game is returned by the build method.

Listing 7.22: GamePage.dart

```
1  class GamePage extends StatelessWidget {
2    static late Size dimensions;
3    static const String route = '/game';
4    final ChickenGame chickenGame;
5
6    GamePage(this.chickenGame);
7
8    Widget build(BuildContext context) {
9      dimensions = MediaQuery.of(context).size;
10     chickenGame.initialize(dimensions, context);
11     return GameWidget(game: chickenGame);
12   }
13 }
```

7.6.2 The Chicken Game

In Listing 7.23, you can see the extensive implementation for the actual game. The game uses a *TapDetector* mixin to enable the game to respond to finger touches. First, the instance variables that are needed are shown. There is one variable for the chicken, thus the game character itself, which is controlled by the user. Also, there is a variable for the labyrinth, where everything takes place.

(a) Level 1

(b) Level 3

Fig. 7.10: Game Levels

Then there is a variable for the image, which serves as a graphic for the pause button. Two variables that are necessary to realize the function of a pause because of the change of level or because of the loss of a life. Here the pause is fixed at 800 milliseconds. Then the logic to pause the game follows: A setter (line 12) not only sets a bool variable to the appropriate value but also stops or starts the background music. The getter returns the pause state. Such pauses occur when advertisements are displayed, or the app is running in the background of the operating system. Flutter also provides support for reacting to a lifecycle change as usual in Android app programming (see [7]). The Mixin *WidgetsBindingObserver* makes this possible. It is already integrated in a superclass of the game, and in line 40 the method `lifecycleStateChange(AppLifecycleState state)` is overwritten. If the global state of the app is changed, it gets activated. There the setter paused is set to false if the game is not running.

In the following, declarations are added to the variables, for example, a variable *direction*, which indicates the global direction of the main character. Then a variable *inputHandler* that creates an InputHandler object, representing the instance of an

auxiliary class, which triggers an action when inputting with the finger. There the game logic is implemented (see section 7.6.3).

This is followed in line 30 by a variable with the list of enemy chickens. Next follows a variable *screenTileDimensions*, which contains the number of tiles that fit on the actual screen. There ist the variable for the score (*score*). The tile position where the chicken reappears after death is noted in the variable *spawnPos*. The next lines are: a variable for the level number *level*, the flutter context (*context*), the local app memory *prefs*, and the scale factor, depending on the device *scaleFactor*.

The variables are initialized in the constructor from line 44 onwards: The constructor is given the dimensions of the screen and the context. The Google Ads are initialized. The game is paused first, as long as it has not been started, the images are loaded, the pause button and the level number are initialized.

The Lifecycle of the app must be managed separately:

The method `lifecycleStateChange(AppLifecycleState)` of the *WidgetsBindingObserver* mixin is responsible for this. You have to make sure that the app does not continue running in the background when the user(s) switches to the app. This could lead to performance problems and also simply continue to run a game without control. Also, music would continue to run in the background. There are, of course, applications where it makes sense to do things in the background, such as geo-tracking.

When initialized (line 57), the scaling factor is determined (an empirical value); an optimal resolution for vintage computers of 320 horizontal pixels is assumed and calculated with the current width. This also results in good scaling for tablets. Then the value for the *screenTileDimensions* is determined. The calculation includes the mentioned scaling, the screen size and the raster width of a tile.

The method `startGame()` (line 69) starts the game from the beginning, the level is set to 1 and the score to 0. A new game character is created. The direction of movement is set to no movement and the pause button is switched off.

The method `initLevel()` (line 76) sets the first reset position of the chicken to the position x:1 and y:0, which is always the start position in every level. A labyrinth is created, and, starting from the labyrinth of the current level, all enemies appear that are provided at corresponding positions in the level. The *InputHandler* is created, then the music begins.

`restartLevel()` (line 87) restarts the level if a chicken life has been lost. The current direction of movement of the chicken is set to *none*, the pause page is shown because paused is set to true. Then a timer is set, which shows an advertisement after the pause time has expired.

In line 99, the game loop starts. For this purpose, *BaseGame* provides the method `render(Canvas canvas)`. First, the drawing area is adjusted to the scaling adapted for the device. Then, it is checked if the game was paused because of the level change or the loss of life. If so, the pause display is shown, and the render loop is terminated.

After that, it is checked if an advertisement should be shown or if the maze is not ready yet, in this case, the method is also finished immediately.

In line 112, the labyrinth is moved to the position in the area where the chicken is located. The movement of the game character is a mixture of movement across the screen and a movement of the background. Then the labyrinth is rendered. From line 114 on, it is dealing with the enemies: it is compared if each enemy is still alive and if it is on the same position as the chicken. If the chicken has power points, the enemy is killed, and one power point is taken away from the chicken. If not, one life is taken from the chicken. If it has remaining lives, the chicken and its enemies are reset to their initial positions and the level is restarted. The enemies are moved by calling the method `update()` and then drawn. After iteration of all enemies, it is checked if the chicken has arrived at the exit of the maze. The exit is not determined specifically for each level, but is always at the bottom right, where there is always a wall to the right of the chicken (see line 138). Then the number of the level is increased. If all levels have already been played, the game starts from the beginning, but the score and lives are not reset. Then the origin of the coordinates is set back to zero using `canvas.restore()`, which is important to draw the scoreboard on the screen and in the next pass of the game loop everything will be relocated from zero again. Furthermore, it should give the impression that the chicken is moving through the maze by moving the background with the maze and all objects in it, including the enemies. The chicken itself should never leave the screen area, so it is always visible on the screen. So only here the chicken is updated and drawn.

---

i You can define absolute positions on the canvas drawing area, but you can also move the drawing area with `.translate(...)` and scale it with `.scale(...)`. You can then restore the original state with `.restore()`.

Starting at line 152, the scoreboard is created. Two *TextPainter* objects with the global text settings, the correct scaling and translation of the text and the current values for level, life and power are drawn in the center of the top.

In the method `showPause(Canvas canvas)` from line 168, only the extensive drawing of the textual information of the game pause state is outsourced. And only the level and the lives are shown. It must be noted that there is the pause state, which occurs when the level changes and a life is lost, which is drawn here. The pause page as a widget is the extra page that is displayed when the pause button is pressed (see section 7.4.6).

The method `onTapDown(TapDownDetails evt)` is called is called by the *TapDetector* mixin when the screen is touched. If the touch is within the range of the image for the pause button, the pause page (the widget) is displayed. Otherwise the coordinates of

the touch point are passed on to the *InputHandler*, which takes care of the handling of the event in the running game (see listing 7.24).

At the end of the class, there is the method `update(double t)` that is overwritten (from BaseGame). This method is called automatically in every frame. It is checked if the game is over. In such a case, the game is paused and then the "Game Over" page is displayed.

Listing 7.23: ChickenGame.dart

```
class ChickenGame extends BaseGame with TapDetector {

  late Size _dimensions;
  late Chicken chicken;
  late Maze maze;
  late var pauseImage;
  late Timer _pauseTimer;
  late bool _timerPaused;
  bool _loaded;
  static const pauseMillis = 800;

  set paused(bool p) {
    if (p != _paused) {
      if (p) {
        AssetLoader.stopMusic();
      } else {
        AssetLoader.startMusic();
      }
    }
    _paused = p;
  }

  bool get paused {
    return _paused;
  }

  bool _paused = true;
  late Direction direction;
  late InputHandler inputHandler;
  late List<Enemy> enemies;
  late Vect2<int> screenTileDimensions;
  late int score;
  late Vect2<int> spawnPos;
  late int level;
```

```
35  late BuildContext context;
36  late SharedPreferences prefs;
37  late double scaleFactor;
38
39  @override
40  void lifecycleStateChange(AppLifecycleState state) {
41    paused = state.index != AppLifecycleState.resumed.index;
42  }
43
44  ChickenGame(this.prefs) : _loaded = false, _timerPaused = false {
45    Ads.init(this);
46    AssetLoader.initMusic();
47    this.paused = true;
48    AssetLoader.loadAll().then((value) {
49      pauseImage = AssetLoader.pauseImage;
50      level = 1;
51      score = 0;
52      chicken = Chicken(this);
53      _loaded = true;
54    });
55  }
56
57  void initialize(Size dimensions, BuildContext context) {
58    _dimensions = dimensions;
59    this.context = context;
60    scaleFactor = this._dimensions.width / 320.0 * 1.5;
61    screenTileDimensions = Vect2<int>(
62        (this._dimensions.width / (raster * scaleFactor)).floor(),
63        (this._dimensions.height / (raster * scaleFactor)).floor());
64    initLevel();
65    direction = Direction.none;
66    paused = false;
67  }
68
69  void startGame() {
70    chicken.lives = 3;
71    chicken.canKill = 0;
72    direction = Direction.none;
73    paused = false;
74  }
75
76  void initLevel() {
```

```
77      spawnPos = Vect2<int>(1, 0);
78      chicken.initPos(spawnPos.x, spawnPos.y);
79      maze = Maze(this, screenTileDimensions);
80      enemies = <Enemy>[];
81      maze.getEnemyPositions.then((epos) {
82        epos.forEach((p) => enemies.add(Enemy(this, p.x, p.y)));
83      });
84      inputHandler = InputHandler(this, maze, chicken);
85    }
86
87    void restartLevel() {
88      paused = true;
89      direction = Direction.none;
90      _timerPaused = true;
91      _pauseTimer = Timer(Duration(milliseconds: pauseMillis), () {
92        _timerPaused = false;
93        Ads.ad();
94      });
95    }
96
97    /// Game loop
98    @override
99    void render(Canvas canvas) {
100     super.render(canvas);
101     if (!_loaded) return;
102     canvas.scale(scaleFactor);
103     if (paused) {
104       showPause(canvas);
105       return;
106     }
107     if (!maze.initialized || _timerPaused) {
108       return;
109     }
110     var col = Paint()..color = Color(0xff004800);
111     canvas.drawPaint(col);
112     canvas.translate(maze.bgrPos.x, maze.bgrPos.y);
113     maze.render(canvas);
114     for (var e in enemies) {
115       if (!e.isKilled && e.mapPos == chicken.mapPos) {
116         if (chicken.canKill > 0) {
117           //Kill enemy!
118           e.isKilled = true;
```

```
119           chicken.canKill--;
120           continue;
121         }
122         // Hit by enemy
123         AssetLoader.cry();
124         chicken.lives--;
125         // More lives left?
126         if (chicken.lives > 0) {
127           //Reset chicken and enemies to pos
128           chicken.beamToPos(spawnPos.x, spawnPos.y);
129           enemies.forEach((e) => e.initPos(e.initialPos.x, e.
                initialPos.y));
130           restartLevel();
131         }
132         break;
133       }
134       e.update();
135       e.render(canvas);
136     }
137     //Check if next level?
138     if (maze.tileDimensions.x - 2 == chicken.mapPos.x &&
139         maze.tileDimensions.y - 1 == chicken.mapPos.y) {
140       level++;
141       // If end then repeat
142       if (level > maxLevel) level = 1;
143       initLevel();
144       restartLevel();
145     }
146     canvas.restore();
147     canvas.scale(scaleFactor);
148     chicken.update(direction);
149     chicken.render(canvas);
150
151     //Render Text and Button
152     TextPainter ltxt = gameTextConf(context, scaleFactor)
153         .toTextPainter("${Lang.of(this.context)!.t('Level')}:${this.
                level}");
154     ltxt.paint(
155         canvas,
156         Offset((_dimensions.width / scaleFactor) / 2 - ltxt.width / 2,
157             10.0)); // position
```

```
158     TextPainter txt = gameTextConf(context, scaleFactor).
            toTextPainter(
159         "${Lang.of(this.context)!.t('Lives')}:${chicken.lives} ${Lang.
                of(this.context)!.t('Power')}:${chicken.canKill} ${Lang.
                of(this.context)!.t('Score')}:$score");
160     txt.paint(
161         canvas,
162         Offset((_dimensions.width / scaleFactor) / 2 - txt.width / 2,
163             10.0 + ltxt.height * 1.5)); // position
164     canvas.drawImage(pauseImage,
165         Offset(0.0, this._dimensions.height / scaleFactor - raster),
                Paint());
166   }
167
168   void showPause(Canvas canvas) {
169     TextPainter ltxt = gameTextConf(context, scaleFactor)
170         .toTextPainter("${Lang.of(this.context)!.t('Level')}: ${this.
                level}");
171     TextPainter ctxt = gameTextConf(context, scaleFactor).
            toTextPainter(
172         "${Lang.of(this.context)!.t('Lives')}: ${this.chicken.lives}")
                ;
173     TextPainter txt = gameTextConf(context, scaleFactor)
174         .toTextPainter("${Lang.of(this.context)!.t('BitteWarten')}");
175     ltxt.paint(
176         canvas,
177         Offset(
178             (_dimensions.width / scaleFactor) / 2 - ltxt.width / 2,
179             (_dimensions.height / scaleFactor) / 2 -
180                 ltxt.height / 2 -
181                 ctxt.height * 2)); // position
182     ctxt.paint(
183         canvas,
184         Offset(
185             (_dimensions.width / scaleFactor) / 2 - ctxt.width / 2,
186             (_dimensions.height / scaleFactor) / 2 -
187                 ctxt.height / 2)); // position
188     txt.paint(
189         canvas,
190         Offset(
191             (_dimensions.width / scaleFactor) / 2 - txt.width / 2,
192             (_dimensions.height / scaleFactor) / 2 -
```

```
193                 txt.height / 2 +
194                 ctxt.height * 2)); // position
195     }
196
197     @override
198     void onTapDown(TapDownDetails evt) {
199       if (!_loaded) return;
200       var xp = evt.globalPosition.dx;
201       var yp = evt.globalPosition.dy;
202       if (paused || !maze.initialized) return;
203       if (xp < raster * scaleFactor &&
204           yp > this._dimensions.height - raster * scaleFactor) {
205         // Pause
206         this.paused = true;
207         Navigator.pushReplacementNamed(context, PausePage.route);
208       } else {
209         inputHandler.touched(xp, yp);
210         print("Chicken: ${chicken.mapPos.x}, ${chicken.mapPos.y}");
211       }
212     }
213
214     @override
215     void update(double t) {
216       super.update(t);
217       if (!_loaded) return;
218       if (!paused && chicken.lives <= 0) {
219         paused = true;
220         this._timerPaused = true;
221         this._pauseTimer = Timer(Duration(milliseconds: pauseMillis),
                () {
222           _timerPaused = false;
223           Navigator.of(context).pushReplacementNamed(GameOverPage.route)
                ;
224         });
225       }
226     }
227   }
```

7.6.3 The Inputhandler

Listing 7.24 contains the code for the *InputHandler*, which processes the touch events for the running game. The method `touched(double xp, double yp)` from line 103 on takes the position of the finger on the screen. First, the actual coordinates of the chicken on the screen are calculated and then the distance to the finger position is determined. If the absolute amount of the horizontal distance is greater than the vertical distance, the finger is more in a horizontal position to the chicken. Now we have to decide if the distance is positive or negative, thus if the finger was on the right or left side of the chicken. The protected method `_moveDir(Direction dir)` is called with the value Direction.left or Direction.right. The vertical directions are determined in the same way.

In the mentioned method from line 9 onward, a test is run to see if there is already movement. If so, the execution of the method is aborted. Otherwise, for each direction of movement starting with *Direction.up*, it is determined whether the next position is an obstacle. If yes, it is checked if the chicken still has force pills (*canKill*) and if the obstacle is a wall that can be opened. If this is true, the wall is opened by increasing the tile-id by one, which represents the tile with the image of the open wall. The chicken clucks and the method is finished. If there is no obstacle at the next position, the animation of the chicken is changed by one phase and the direction of movement of the whole game is set to the corresponding direction. Then, it is still decided whether the chicken or the background should move: To make the game look more dynamic, the chicken should not only be centered in the middle of the screen but move across the screen.

Only when it reaches the border, the background moves instead of the chicken. If you navigate the chicken in the opposite direction, it should move to the opposite side of the screen and only then the background moves in the opposite direction. Additionally, you have to consider that you have to tap in front of the chicken with your finger to move it in the desired direction. But if the chicken is at the border, you cannot navigate it in the direction of the border. Therefore, when the chicken is three or two tiles in front of the border, the background is moved instead of the chicken. So, one finger still fits between chicken and border. Since the mobile devices are longer than they are broad (in portrait mode), this happens three tiles earlier for vertical movements and two tiles earlier for horizontal movements (see lines 26-30).

Listing 7.24: InputHandler.dart

```
class InputHandler {
  ChickenGame game;
  Maze maze;
  Chicken chicken;

  InputHandler(this.game, this.maze, this.chicken);

  void _moveDir(Direction dir) {
    if (game.direction != Direction.none) return; // is already
        moving
    if (dir == Direction.up) {
      if (maze.isObstacle(chicken.mapPos.x, chicken.mapPos.y - 1)) {
        //Make a hole?
        Tile tile =
            maze.getTileFromLayer(0, chicken.mapPos.x, chicken.mapPos.
                y - 1);
        if (chicken.canKill > 0 && tile.tileId == passageClosed[0]) {
          chicken.canKill--;
          tile.tileId = passageOpened[0];
          tile.gid = tile.tileId + 1;
        }
        chicken.sound();
        return;
      }
      chicken.currentAnimation.update(1);
      game.direction = Direction.up;
      // is chicken not at top edge?
      if (chicken.screenPos.y > 3) {
        chicken.move(Direction.up);
      } else if (maze.bgrTilePos.y < 3) {
        maze.moveTileMap(Direction.down);
      }
    }
    if (dir == Direction.down) {
      if (maze.isObstacle(chicken.mapPos.x, chicken.mapPos.y + 1)) {
        //Make a hole?
        Tile tile =
            maze.getTileFromLayer(0, chicken.mapPos.x, chicken.mapPos.
                y + 1);
        if (chicken.canKill > 0 && tile.tileId == passageClosed[0]) {
```

```
38          chicken.canKill--;
39          tile.tileId = passageOpened[0];
40          tile.gid = tile.tileId + 1; //id!= gid
41        }
42        chicken.sound();
43        return;
44      }
45      chicken.currentAnimation.update(1);
46      // is chicken not at bottom edge?
47      game.direction = Direction.down;
48      if (chicken.screenPos.y < maze.screenTileDimensions.y - 4) {
49        chicken.move(Direction.down);
50      } else if (maze.bgrTilePos.y >
51          -maze.tileDimensions.y - 3 + maze.screenTileDimensions.y) {
52        maze.moveTileMap(Direction.up);
53      }
54    }
55    if (dir == Direction.left) {
56      if (maze.isObstacle(chicken.mapPos.x - 1, chicken.mapPos.y)) {
57        //Make a hole?
58        Tile tile =
59            maze.getTileFromLayer(0, chicken.mapPos.x - 1, chicken.
                mapPos.y);
60        if (chicken.canKill > 0 && tile.tileId == passageClosed[1]) {
61          chicken.canKill--;
62          tile.tileId = passageOpened[1];
63          tile.gid = tile.tileId + 1;
64        }
65        chicken.sound();
66        return;
67      }
68      chicken.currentAnimation.update(1);
69      game.direction = Direction.left;
70      // is chicken not at left edge?
71      if (chicken.screenPos.x > 2) {
72        chicken.move(Direction.left);
73      } else if (maze.bgrTilePos.x < 2) {
74        maze.moveTileMap(Direction.right);
75      }
76    }
77
78    if (dir == Direction.right) {
```

```
79      if (maze.isObstacle(chicken.mapPos.x + 1, chicken.mapPos.y)) {
80        //Make a hole?
81        Tile tile =
82            maze.getTileFromLayer(0, chicken.mapPos.x + 1, chicken.
                  mapPos.y);
83        if (chicken.canKill > 0 && tile.tileId == passageClosed[1]) {
84          chicken.canKill--;
85          tile.tileId = passageOpened[1];
86          tile.gid = tile.tileId + 1;
87        }
88        chicken.sound();
89        return;
90      }
91      chicken.currentAnimation.update(1);
92      game.direction = Direction.right;
93      // is chicken not at right edge?
94      if (chicken.screenPos.x < maze.screenTileDimensions.x - 3) {
95        chicken.move(Direction.right);
96      } else if (maze.bgrTilePos.x >
97          -maze.tileDimensions.x - 2 + maze.screenTileDimensions.x) {
98        maze.moveTileMap(Direction.left);
99      }
100   }
101 }
102
103 void touched(double xp, double yp) {
104   var centerX = chicken.screenPos.x * raster * game.scaleFactor +
105       raster * game.scaleFactor / 2;
106   var centerY = chicken.screenPos.y * raster * game.scaleFactor +
107       raster * game.scaleFactor / 2;
108   var dX = xp - centerX;
109   var dY = yp - centerY;
110   if (dX.abs() > dY.abs()) {
111     //Horiz
112     if (dX > 0) {
113       // R
114       _moveDir(Direction.right);
115     } else {
116       // L
117       _moveDir(Direction.left);
118     }
119   } else {
```

```
120         // Verti
121         if (dY > 0) {
122           // D
123           _moveDir(Direction.down);
124         } else {
125           // U
126           _moveDir(Direction.up);
127         }
128       }
129       maze.tiles.generate(); // Update Map
130     }
131   }
```

7.6.4 The Game Characters

In addition to the labyrinth with its obstacles and objects, there are the creatures that move. These are the chicken and its enemies. Both inherit from an abstract superclass *Animal*. This class describes the abilities and characteristics that both types of creatures have in common. You can read about them in Listing 7.25: Both creatures have animations for the movements in the four directions and an animation for standing. They have a method `initPos(int x, int y)`, where you can set the starting position in the labyrinth.

There, several variables for positions are initialized: *mapPos*, a vector with integers that defines the tile position on the map, *screenPos*, also an integer vector for the tile position on the screen, *pos*, a vector for the actual position on the screen in pixels, and *targetPos*, a vector for the target position in pixels. These variables are necessary to guarantee be a smooth transition between two tile positions. Both have a method `render(Canvas canvas)` to draw the sprites of the current frame of the currently active animation. The common method `move(Direction dir)` sets the new current positions and the target positions (tiles and pixels on screen and map). How exactly the movement in the subclasses works has to be planned separately in the (here abstract) method `void update([Direction direction])` in order for its implementation. You can optionally specify a direction here. This is necessary for the chicken, because it is controlled by the player but not for the enemy. The enemy should decide by itself where to turn. Finally, there is an abstract method `void sound()`, which gives both characters the possibility to make a sound.

Listing 7.25: Animal.dart

```
abstract class Animal {
  SpriteAnimation animationRight;
  SpriteAnimation animationLeft;
  SpriteAnimation animationUp;
  SpriteAnimation animationDown;
  SpriteAnimation animationIdle;
  late SpriteAnimation currentAnimation;

  late Vect2<double> pos;
  late Vect2<int> screenPos;
  late Vect2<double> targetPos;
  late Vect2<int> mapPos;
  ChickenGame game;

  Animal(this.game,
      {required this.mapPos,
      required this.animationLeft,
      required this.animationRight,
      required this.animationUp,
      required this.animationDown,
      required this.animationIdle}) {
    initPos(1, 0);
    currentAnimation = animationIdle;
  }

  void initPos(int x, int y) {
    mapPos = Vect2<int>(x, y);
    targetPos = Vect2<double>(raster * mapPos.x, raster * mapPos.y);
    screenPos = Vect2<int>(mapPos.x, mapPos.y);
    pos = Vect2<double>(targetPos.x, targetPos.y);
  }

  void render(Canvas canvas) {
    //Render Animal
    if (currentAnimation == animationIdle) {
      currentAnimation.update(0.1);
    }
    currentAnimation
        .getSprite()
        .render(canvas, position: Vector2(pos.x, pos.y));
```

```
41    }
42
43    void move(Direction dir) {
44      int xp = 0;
45      int yp = 0;
46      switch (dir) {
47        case Direction.left:
48          xp = -1;
49          break;
50        case Direction.right:
51          xp = 1;
52          break;
53        case Direction.up:
54          yp = -1;
55          break;
56        case Direction.down:
57          yp = 1;
58          break;
59        case Direction.none:
60          xp = 0;
61          yp = 0;
62          break;
63      }
64      targetPos.x += xp * raster;
65      targetPos.y += yp * raster;
66      screenPos.add(xp, yp);
67      mapPos.add(xp, yp);
68    }
69
70    // Abstract
71
72    void update([Direction direction]);
73
74    void sound();
75  }
```

7.6.5 The Chicken

The class *Chicken* is a subclass of *Animal*, see listing 7.26. In the constructor, all animations are initialized first, the lives are set to three at the beginning of the game and the superpower to the value 0 (*canKill*). The method`beamToPos(int x, int y)` sets the chicken to any position. This is important if the chicken has passed a so-called "respawn" point. Then the chicken can reappear at this point if it was killed and still has lives. First, the center of the screen tiles is determined, rounded to an integer. This position serves as the center point on the screen. Then the movement is set to *none*. The position in the maze is then moved by the negated desired position of the parameter variables (line 26).

The extensive method `update([Direction direction = Direction.none])` (at line 33) is the control logic of the chicken. Here the following is done for the respective direction: the corresponding direction animation (with foot left/right) is taken as the current animation. If the chicken has not yet arrived at the target position, the movement in this direction follows with a constant speed. Also, the concerning background, the labyrinth is moved in the opposite direction if it still has to be moved. However, when the chicken and the background have arrived at the positions, the direction of movement is set to `none`, the idle-animation (`idle()`, line 81) is started in two seconds from the end of the movement and tested if there is a grain at the position. In the case there is, the chicken eats it. This is done for all four directions. Note that the global movement direction is set by the *InputHandler* (7.6.3) and is only changed if a movement has been completed and the global game direction (`game.direction`) is *none*. The method itself (`update(...)`) is called in the game loop (see section 7.6.2) with the global game direction. This way, a soft animation is always performed between the tile positions until the target is reached.

Listing 7.26: Chicken.dart

```
1  class Chicken extends Animal {
2    Chicken(ChickenGame game)
3        : super(game,
4              mapPos: Vect2<int>(1, 0),
5              animationLeft: AssetLoader.chickenAnimationLeft,
6              animationRight: AssetLoader.chickenAnimationRight,
7              animationUp: AssetLoader.chickenAnimationUp,
8              animationDown: AssetLoader.chickenAnimationDown,
9              animationIdle: AssetLoader.chickenAnimationIdle) {
10     lives = 3;
11     canKill = 0;
12   }
13
14   Timer? timer;
```

```
  late int lives;
  late int canKill;

  void beamToPos(int x, int y) {
    mapPos = Vect2<int>(x, y);
    int px = (game.maze.screenTileDimensions.x / 2).floor();
    int py = (game.maze.screenTileDimensions.y / 2).floor();
    screenPos = Vect2<int>(px, py);
    targetPos = Vect2<double>(raster * px, raster * py);
    pos = Vect2<double>(targetPos.x, targetPos.y);
    game.direction = Direction.none;
    game.maze.bgrTilePos = Vect2<int>(-x + px, -y + py);
    game.maze.bgrPos = Vector2(
        raster * game.maze.bgrTilePos.x, raster * game.maze.
            bgrTilePos.y);
    game.maze.bgrTargetPos = Vector2(game.maze.bgrPos.x, game.maze.
        bgrPos.y);
  }

  @override
  void update([Direction direction = Direction.none]) {
    if (direction == Direction.right) {
      currentAnimation = animationRight;
      if (pos.x < targetPos.x) pos.x += chickenSpeed;
      if (game.maze.bgrPos.x > game.maze.bgrTargetPos.x)
        game.maze.bgrPos.x -= chickenSpeed;
      if (pos.x >= targetPos.x &&
          game.maze.bgrPos.x <= game.maze.bgrTargetPos.x) {
        game.direction = Direction.none;
        idle();
        game.maze.checkGrain(mapPos.x, mapPos.y);
      }
    } else if (direction == Direction.left) {
      currentAnimation = animationLeft;
      if (pos.x > targetPos.x) pos.x -= chickenSpeed;
      if (game.maze.bgrPos.x < game.maze.bgrTargetPos.x)
        game.maze.bgrPos.x += chickenSpeed;
      if (pos.x <= targetPos.x &&
          game.maze.bgrPos.x >= game.maze.bgrTargetPos.x) {
        game.direction = Direction.none;
        idle();
        game.maze.checkGrain(mapPos.x, mapPos.y);
```

```
55        }
56      } else if (direction == Direction.down) {
57        currentAnimation = animationDown;
58        if (pos.y < targetPos.y) pos.y += chickenSpeed;
59        if (game.maze.bgrPos.y > game.maze.bgrTargetPos.y)
60          game.maze.bgrPos.y -= chickenSpeed;
61        if (pos.y >= targetPos.y &&
62            game.maze.bgrPos.y <= game.maze.bgrTargetPos.y) {
63          game.direction = Direction.none;
64          idle();
65          game.maze.checkGrain(mapPos.x, mapPos.y);
66        }
67      } else if (direction == Direction.up) {
68        currentAnimation = animationUp;
69        if (pos.y > targetPos.y) pos.y -= chickenSpeed;
70        if (game.maze.bgrPos.y < game.maze.bgrTargetPos.y)
71          game.maze.bgrPos.y += chickenSpeed;
72        if (pos.y <= targetPos.y &&
73            game.maze.bgrPos.y >= game.maze.bgrTargetPos.y) {
74          game.direction = Direction.none;
75          idle();
76          game.maze.checkGrain(mapPos.x, mapPos.y);
77        }
78      }
79    }
80
81    void idle() {
82      timer?.stop();
83      timer = Timer(2000, callback: () {
84        if (game.direction == Direction.none) {
85          currentAnimation = animationIdle;
86        }
87      });
88    }
89
90    @override
91    void sound() {
92      AssetLoader.cluck();
93    }
94  }
```

7.6.6 The Enemies

The realization of the enemies can be found in Listing 7.27. The enemies, unlike the chicken, have an individual direction of movement, a fixed speed, a random component that can change the movement spontaneously, and they are either dead or alive. In the constructor, the animations are assigned, and the variables are initialized. The logic of the enemies, the AI, as one might say, can be found mainly in the method `nextPossibleDirection()`. At first it is checked, if the enemy sees the chicken. For this purpose, the method `Direction chickenSpotted()` is called, which does the following: Here it is checked whether the chicken is to be found in the same horizontal or vertical position, then it is checked whether there is an obstacle between the enemy and the chicken. If not, the enemy can spot the chicken in one direction. The enemy then immediately runs in that direction. In line 41, a spontaneous change of direction is made with a probability of 0.2. The next step is to check if there is an obstacle or another enemy in the next position to be approached. If so, the direction is changed. The method `Direction getPossibleDirection()` gives suggestions for possible positions. A direction is chosen randomly, or *none* if no movement is possible, if an enemy is trapped between other enemies and obstacles randomly. In the implementation of the `update()` method, the first step is to check if the enemy is still alive. If so, it is examined in which direction the enemy is running. For all directions, the corresponding animation is then activated. Then the enemy is moved in the direction of the target position. If the enemy has reached the target position, a new direction is set by calling the method `nextPossibleDirection()`. The method `move()` starts the new movement.

Listing 7.27: Enemy.dart

```
1  class Enemy extends Animal {
2
3    late Direction direction;
4    double speed = 1.0;
5    late Random rnd;
6    late Vect2<int> initialPos;
7    late bool isKilled;
8
9    Enemy(ChickenGame game, int x, int y) : super(
10       game, mapPos: Vect2<int>(1, 0),
11       animationLeft: AssetLoader.enemyAnimationLeft,
12       animationRight: AssetLoader.enemyAnimationRight,
13       animationUp: AssetLoader.enemyAnimationUp,
14       animationDown: AssetLoader.enemyAnimationDown,
15       animationIdle: AssetLoader.enemyAnimationIdle) {
16       initialPos = Vect2<int>(x, y);
17       isKilled = false;
```

```
18      initPos(x, y);
19      rnd = Random();
20      direction = getPossibleDirection();
21      move(direction);
22    }
23
24    Direction getOppositeDirection(Direction dir) {
25      switch(dir) {
26        case Direction.left: return Direction.right;
27        case Direction.right: return Direction.left;
28        case Direction.up: return Direction.down;
29        case Direction.down: return Direction.up;
30        default: return Direction.none;
31      }
32    }
33
34    void nextPossibleDirection() {
35      Direction chickenSpot = chickenSpotted();
36      if (chickenSpot != Direction.none) {
37        direction = chickenSpot;
38        return;
39      }
40      // Switch sometimes direction or if not moving
41      if (direction == Direction.none || rnd.nextInt(10)<= 2) {
42        direction = getPossibleDirection();
43      }
44
45      if(isOther(getNextPosition(this.direction, mapPos)) ||
46          game.maze.obstacle(getNextPosition(this.direction, this.
                mapPos))) {
47        direction = getPossibleDirection();
48      }
49    }
50
51    /// Is there the chicken?
52    Direction chickenSpotted() {
53      Direction dir = Direction.none;
54      if (game.chicken.mapPos.x == mapPos.x && game.chicken.mapPos.y ==
              mapPos.y) return dir;
55        bool obstacle = false;
56      if (game.chicken.mapPos.x == mapPos.x) { // same horizontal layer
57        if (game.chicken.mapPos.y < mapPos.y) {
```

```
58          dir = Direction.up;
59          for (int y=game.chicken.mapPos.y+1; y < mapPos.y; y++) {
60            if (game.maze.isObstacle(mapPos.x, y) || isOther(Vect2<int>(
                  mapPos.x, y))) {
61              obstacle = true;
62            }
63          }
64        } else {
65          dir = Direction.down;
66          for (int y=game.chicken.mapPos.y-1; y > mapPos.y; y--) {
67            if (game.maze.isObstacle(mapPos.x, y) || isOther(Vect2<int>(
                  mapPos.x, y))) {
68              obstacle = true;
69            }
70          }
71        }
72      }
73      if (game.chicken.mapPos.y == mapPos.y) { // same vertical layer
74        if (game.chicken.mapPos.x < mapPos.x) {
75          dir = Direction.left;
76          for (int x=game.chicken.mapPos.x; x < mapPos.x ; x++) {
77            if (game.maze.isObstacle(x, mapPos.y) || isOther(Vect2<int>(
                  x, mapPos.y))) {
78              obstacle = true;
79            }
80          }
81        } else {
82          dir = Direction.right;
83          for (int x=game.chicken.mapPos.x - 1 ; x > mapPos.x; x--) {
84            if (game.maze.isObstacle(x, mapPos.y) || isOther(Vect2<int>(
                  x, mapPos.y))) {
85              obstacle = true;
86            }
87          }
88        }
89      }
90      return obstacle == true ? Direction.none : dir;
91    }
92
93    /// looks for all possible directions and take one random or idle
94    Direction getPossibleDirection() {
95      List<Direction> directions = Direction.values;
```

```
    List<Direction> possibleDirections = [];
    directions.forEach((d) {
      if (!game.maze.obstacle(getNextPosition(d, mapPos)) &&
          !isOther(getNextPosition(d, mapPos))) {
        possibleDirections.add(d);
      }
    });
    var newDir = possibleDirections.length == 1 ? possibleDirections
        [0] :
           possibleDirections[rnd.nextInt(possibleDirections.length -
                1) + 1];
    return newDir;
  }

  bool isOther(Vect2<int> pos) {
    for (Enemy e in game.enemies) {
      if (e != this && e.mapPos == pos) {
        return true;
      }
    }
    return false;
  }

  @override
  void sound() { // No Sound
  }

  @override
  void update([Direction _ = Direction.none]) {
    if (isKilled) return;
    if (direction == Direction.right) {
      currentAnimation = animationRight;
      if (pos.x < targetPos.x) pos.x += speed;
      if (pos.x >= targetPos.x) {
        nextPossibleDirection();
        move(direction);
      }
    } else if (direction == Direction.left) {
      currentAnimation = animationLeft;
      if (pos.x > targetPos.x) pos.x -= speed;
      if (pos.x <= targetPos.x) {
        nextPossibleDirection();
```

```
136         move(direction);
137       }
138     } else if (direction == Direction.down) {
139       currentAnimation = animationDown;
140       if (pos.y < targetPos.y) pos.y += speed;
141       if (pos.y >= targetPos.y) {
142         nextPossibleDirection();
143         move(direction);
144       }
145     } else if (direction == Direction.up) {
146       currentAnimation = animationUp;
147       if (pos.y > targetPos.y) pos.y -= speed;
148       if (pos.y <= targetPos.y) {
149         nextPossibleDirection();
150         move(direction);
151       }
152     } else if (direction == Direction.none) {
153       //currentAnimation = animationIdle;
154         nextPossibleDirection();
155     }
156   }
157   void move(Direction dir) {
158     currentAnimation.update(1);
159     super.move(dir);
160   }
161
162   @override
163   void render(Canvas canvas) {
164     if (isKilled) return;
165     super.render(canvas);
166   }
167
168   Vect2<int> getNextPosition(Direction d, Vect2<int> p) {
169     switch(d) {
170       case Direction.left: return p + Vect2<int>(-1 , 0);
171       case Direction.right: return p + Vect2<int>(1 , 0);
172       case Direction.up: return p + Vect2<int>(0 , -1);
173       case Direction.down: return p + Vect2<int>(0 , 1);
174       default: return p;
175     }
176   }
177 }
```

7.6.7 The Labyrinth

One important game object is still missing in the list: the labyrinth thus the background, or level. In Listing 7.28, you can see the code. The graphic is stored in the variable *tiles*. Further, there are the screen dimensions in tiles, the size of the whole map, the tile size of the map, the position on the screen and a target position on the screen for the animation to this position, and a tile position of the map on the screen. Furthermore, there is a "getter", which indicates whether the map is ready for use, thus loaded. In the constructor, the map for the level is then loaded from the assets file, which is a ".tmx" file. Only after this file has been loaded, the dependent variables are initialized, and the maze is marked as ready. Important methods follow: `moveTileMap(Direction dir)` moves the maze across the screen. The target position in pixels and the tile position are changed. In addition, the position of the chicken is updated, since the map has moved under it. The method `bool isObstacle(int x, int y)` tests if there is an obstacle at the given tile coordinates.

The boundaries of the maze are also obstacles so that you cannot run out of the level. The first three tiles are firmly reserved for the floor, everything else is a wall or a door that can be opened. A useful utility `Tile getTileFromLayer(int num, int x, int y)` returns the tile id at the given coordinates and level. The *TiledComponent* objects can have different layers. The objects of the game are on layer 1. The method `bool checkGrain(int x, int y)` checks if there is a food grain or other things from the level at a position. These are then deleted as a side effect of this method. If something is found, it is tested if it is a spawn point where the chicken can reappear. This is then saved. Or if it is a medi-pack? - Then an additional life is granted. In the case of a strength pill, the strength of the chicken is increased. Otherwise, with food, the score is increased. The method `render(Canvas canvas)` only draws the tiles. The asynchronous getter `get getEnemyPositions` returns all positions for the enemies on level 2, but only after the map is initialized.

Listing 7.28: Maze.dart

```
1  class Maze {
2    late Tiled tiles;
3    late Vect2<int> screenTileDimensions;
4    late Size mapDimensions;
5    late Vect2<int> tileDimensions;
6    late Vector2 bgrPos;
7    late Vector2 bgrTargetPos;
8    late Vect2<int> bgrTilePos;
9    late ChickenGame game;
10   late bool _initialized;
11   bool get initialized => _initialized;
12
```

```
13    Maze(this.game, this.screenTileDimensions) {
14      tiles = new Tiled("map${game.level}.tmx", Size(raster, raster));
15      _initialized = false;
16      tiles.future!.then((t) {
17        _initialized = true;
18        tileDimensions =
19            Vect2<int>(tiles.map.layers[0].width, tiles.map.layers[0].
                  height);
20        print("Maze: ${tileDimensions.x}, ${tileDimensions.y}");
21        mapDimensions =
22            Size((tileDimensions.x) * raster, (tileDimensions.y) *
                  raster);
23        tiles.map.layers[2].visible = false;
24        tiles.generate();
25      });
26      bgrPos = Vector2(0, 0);
27      bgrTargetPos = Vector2(0, 0);
28      bgrTilePos = Vect2<int>(0, 0);
29    }
30
31    void moveTileMap(Direction dir) {
32      int xp = 0;
33      int yp = 0;
34      switch (dir) {
35        case Direction.left:
36          xp = -1;
37          break;
38        case Direction.right:
39          xp = 1;
40          break;
41        case Direction.up:
42          yp = -1;
43          break;
44        case Direction.down:
45          yp = 1;
46          break;
47        case Direction.none:
48          xp = 0;
49          yp = 0;
50          break;
51      }
52      bgrTargetPos.x += xp * raster;
```

```
53      bgrTargetPos.y += yp * raster;
54      bgrTilePos.add(xp, yp);
55      game.chicken.mapPos.add(-xp, -yp);
56    }
57
58    bool isObstacle(int x, int y) {
59      // Edges
60      if (x < 0 || x >= tileDimensions.x) return true;
61      if (y < 0 || y >= tileDimensions.y) return true;
62      // ID of Tile ... Collision
63      int id = getTileFromLayer(0, x, y).tileId;
64      return id > 3 &&
65          !passageOpened.contains(id); //A Hole form chicken-killing-
                power
66    }
67
68    Tile getTileFromLayer(int num, int x, int y) {
69      return tiles.map.layers[num].tiles[y][x];
70    }
71
72    bool obstacle(Vect2<int> o) {
73      return isObstacle(o.x, o.y);
74    }
75
76    bool checkGrain(int x, int y) {
77      //Corners
78      if (x < 0 || x >= tileDimensions.x) return true;
79      if (y < 0 || y >= tileDimensions.y) return true;
80      // ID of Tile ... Collision
81      var tl = getTileFromLayer(1, x, y);
82      var id = tl.tileId;
83      var gid = tl.gid;
84      tl.gid = 0;
85      bool found = gid > 0;
86      if (found) {
87        //What?
88        if (spawn.contains(id)) {
89          game.spawnPos = Vect2<int>(x, y);
90        } else if (mediPack.contains(id)) {
91          game.chicken.lives++;
92        } else if (opener.contains(id)) {
93          game.chicken.canKill++;
```

```
94        } else {
95          game.score += 1;
96        }
97        AssetLoader.pick();
98      }
99      return found;
100   }
101
102   void render(Canvas canvas) {
103     tiles.render(canvas);
104   }
105
106   Future<List<Vect2<int>>> get getEnemyPositions async {
107     await tiles.future;
108     List<Vect2<int>> list = <Vect2<int>>[];
109
110     var matrix = tiles.map.layers[2].tiles;
111     for (int i = 0; i < matrix.length; i++) {
112       for (int j = 0; j < matrix[i].length; j++) {
113         if (matrix[i][j].tileId > 0) {
114           list.add(Vect2<int>(i, j));
115         }
116       }
117     }
118     return list;
119   }
120 }
```

7.6.8 Auxiliary Classes

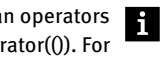
Here auxiliary classes are described that are used in the project. These are not only listed for the sake of completeness, but also because of the special features of dart and flutter, which can be found there.

7.6.8.1 Vectors

Although there is a *Vector2* class in Dart and a *Position* and *Direction* class in Flame, which also represent a 2D vector, the game still needs a class to represent integer pairs so that the tile positions can be processed with it. This would not have been necessary because you could work with lists or single variables, but such a class expresses the purpose better and simplifies the programming. Furthermore, the following implemen-

tation is a demonstration of what else you can do with Dart. Listing 7.29 shows the implementation of the *Vect2* class. As you can see, there are generics in Dart, similar to Java. You can use type parameters. Here it is still required that the type T is a subclass of *num*, the abstract class, which is the superclass of *int* and *double*. You can thus implement integer and decimal vectors using *Vect2* objects, whereas only *Vect2<int>* objects are required in the program. The method `add(T x, T y)` adds numbers to the vector components *x* and *y*. In line 11, you can see that in Dart operators can be overloaded, or even can be defined. Here the operator "+" is defined. Here another *Vect2<T>* is added to the current object and the comparison operator is overwritten. The objects are equal if the components are equal.

If you overwrite the ==-operator, you still have to overwrite the *hashCode*-getter; otherwise a warning (in the IDE) is issued. The reason is not quite clear, because then you would not have to overwrite the ==-operator at all as the hash code is compared when comparing objects. But if you do not overwrite the ==-operator, a warning will be displayed. Objects are only considered equal if the ==-operator returns true and the hashcode is equal. The correct way in Dart to overwrite the ==-operator is to do both.



You can override operators in dart-classes: All arithmetic operators (+, -, *, /, ...), boolean operators (==, !=, <, >, <=, >=, >=, ...) but also the list access operator ([]) and the function call operator(()). For example, the == operator is overwritten as follows: `bool operator ==(var otherObject) {...};` *this* gives you access to the own object in the body and *otherObject* gives you access to the object to be compared to return a boolean value.

Listing 7.29: Vect2.dart

```
1  class Vect2<T extends num> {
2    num _x;
3    num _y;
4    Vect2(this._x, this._y);
5
6    void add(T x, T y) {
7      this._x += x;
8      this._y += y;
9    }
10
11   Vect2<T> operator +(Vect2<T> v) => Vect2<T>(this.x + v.x, this.y +
         v.y);
12
13   @override
14   bool operator ==(v) => v is Vect2<T> && v.x == this.x && v.y ==
         this.y;
15
```

```
  @override
  int get hashCode => this._x.hashCode ^ this._y.hashCode;

  T get x => this._x as T;
  T get y => this._y as T;
  set x(T val) => _x = val;
  set y(T val) => _y = val;
}
```

7.6.8.2 The Control

To check the touch of the display with your finger, you would normally create a *TapGestureRecognizer* object in Flutter. The problem with this object is that it always waits a short time before triggering the event. The purpose is to distinguish between a real and a false touch with the finger. Only if you leave your finger on the display for short time, it counts as a control gesture. For our game, this is not convenient, because you want to let the chicken jump over the screen by quick sequences of touches. Therefore, Flame offers a method of gesture control that is better integrated into the game engine. The mixin *TapDetector*, provides various methods to react to touches. You only have to override them if the mixin is included (see listing 7.30).

Flutter's *TapGestureRecognizer* is normally used to respond to finger gestures. For games it is better to use the *TapDetector* mixin from Flame.

Listing 7.30: gestures.dart

```
// Basic touch detectors
mixin TapDetector on Game {
  void onTap() {}
  void onTapCancel() {}
  void onTapDown(TapDownDetails details) {}
  void onTapUp(TapUpDetails details) {}
}

mixin SecondaryTapDetector on Game {
  void onSecondaryTapDown(TapDownDetails details) {}
  void onSecondaryTapUp(TapUpDetails details) {}
  void onSecondaryTapCancel() {}
}

mixin DoubleTapDetector on Game {
  void onDoubleTap() {}
}
```

7.6.8.3 Monetization

The game is supposed to show advertisements in order to finance it. Therefore, the technology Google AdMob [36] is used to generate ad campaigns and link them to the game. To do this, you need to work through the following points: You have to add an app to AdMob. First, you have to register, then you can click on add apps on the start page or click the corresponding button (see figure 7.11) if you are in the overview of all apps. Next, you get an app ID, that you can copy to the clipboard by clicking the button next to the ID. In the file *secret.dart*, it is stored as constant *myAppId*. Further you have to enter the app ID for Android in the file *AndroidManifest.xml* in the folder *android/app/src/main*:

```
1 <meta-data
2 android:name="com.google.android.gms.ads.APPLICATION_ID"
3 android:value="ca-app-pub-myAppiD"/>
```

Then you have to generate an ad block by going to the menu item "Ad units" under "Apps" and clicking on "Add ad unit." Here you can choose different types, "Banner," "Interstitial," "With bonus" and "Extended native ads." The simple type "Interstitial" has been chosen, these are full-page ads that can be displayed, for example, when changing levels. You get an ID again; but now for the ad itself, which must also be stored as a constant in *secret.dart* under the name *myInterstitialId*.

Now you can start to create and display advertising pages. If the app is not yet available in the play store, dummy ads are generated and displayed. Listing 7.31 contains the code to manage the ads. First the *testDeviceIds* are passed to the *AdMob* API in the method `init()`, which is called in the constructor of *ChickenGame*. The method `AdmobInterstitial ad()` then returns an ad. To be able to test even better on hardware devices, you should add a Test-Device ID. With emulators, this step is not required. The process works as follows: The list *testDeviceIds* is empty at first because the constant *testDevice* in "secret.dart" has not yet been set. The first time the app is started on the hardware, a message of the following type is shown:

```
I/Ads: Use AdRequest.Builder. addTestDevice("<Test- Device Id>")
to get test ads on this device.
```

Now, you can write the ID to the constant *testDevice*. In line 11, the *AdmobInterstitial* object is created. Here, the campaign ID and a listener are passed to the constructor. The listener pauses the game when the ad has been started and displayed, or when the game is exited via the ad or when the ad has been opened. In the *ChickenGame*, the method `ad()` is called in the method `restartLevel()` (listing7.23, line 93). Inside `ad()`, the method of the supplied *AdmobInterstitial* object `load()` is called. The result is a full-page display that covers the game as with magic. Hence, you do not have to create and place the ad as a widget there; it is done automatically. Each display also has a

close button, which closes it again. Then the listener is called, and the event object is *AdmobEvent.closed*. The game is continued (`else`-branch).

The listener of the *AdmobInterstitial* class has to be considered: If the advertisement appears, the game has to pause, which lies below the full-page advertisement, and to resume when it is closed. It is also possible that if you click on the advertisement, the device opens a web page and switches to the browser, so the game app is in the background and the game has to pause.

Listing 7.31: ads.dart

```
class Ads {
  static ChickenGame? game;

  static init(ChickenGame g) {
    Admob.initialize(testDeviceIds: [testDevice]);
    game = g;
  }

  static void ad() {
    AdmobInterstitial? myInterstitial;
    myInterstitial = AdmobInterstitial(
        adUnitId: myInterstitialId,
        listener: (AdmobAdEvent event, Map<String, dynamic>? args) {
          if (event == AdmobAdEvent.started ||
              event == AdmobAdEvent.leftApplication ||
              event == AdmobAdEvent.opened) {
            game?.paused = true;
          } else if (event == AdmobAdEvent.loaded) {
            game?.paused = true;
            myInterstitial?.show();
          } else {
            game?.paused = false;
          }
        });
    myInterstitial.load();
  }
}
```

7.6.8.4 A Customized Button

To give the game an individual look, a new button is designed as a widget. This button should be used on the individual screens and have a uniform look. It should provide different labels and be animated: When you press the button, it should be scaled in an animated way, where it first becomes larger and then smaller again. From the dynamic point of view, this should give the impression that there is a spring inside the button. It should be possible to assign a callback function to it so that you can react to the keystroke with a customized action.

In Listing 7.32 is the code of the button. First, a function type is defined for the callback, namely a function without parameters and return value. In the constructor, the three object variables are initialized, the named callback function, the caption and as an optional argument whether the button is activated or not; by default, it is. Note that all variables of widgets are final, since widgets are all unchangeable. Nevertheless, you can activate and deactivate the button, because the value you pass in the constructor depends on a state variable of a "state" (compare Listing 7.12, line 96), where the value of `enabled:` is made dependent on the state variable `userName` of the state `SettingsPageState`. If it changes, a new *FineButton* object is created, with a different constructor content. In this technique, dynamic surfaces can be created despite the use of immutable objects. The State *FineButtonState* of the button is described from line 17.

The animation technique is the same as described in section 4.4.5. Again, there is an animation controller that controls the padding and the font size by means of a listener (from line 37). In addition, there is a *StatusListener* (line 43), which, when the animation has finished playing, ensures that it runs backwards. The *then* ensures that the specified lambda function is executed afterward, which calls the callback function. Starting at line 53, the layout for the button is assembled. Please note that the button dimensions and shapes depend on the global text scaling and on the other state variables that are changed via the listeners.

Listing 7.32: FineButton.dart

```
1  typedef CallBack = void Function();
2
3  class FineButton extends StatefulWidget {
4
5    final CallBack onPressed;
6    final String text;
7    final bool enabled;
8
9    FineButton({required this.onPressed, required this.text, this.
         enabled = true});
10
11   @override
12   FineButtonState createState() {
```

```
    return new FineButtonState();
  }
}

class FineButtonState extends State<FineButton>
    with SingleTickerProviderStateMixin {

  late Animation<double>? _animation;
  late AnimationController? _animationController;
  late double fontsize;
  late double myPadding;
  late double ts;

  @override
  void initState() {
    super.initState();
    fontsize = 12.0;
    myPadding = 30.0;
    _animationController = AnimationController(
        vsync: this,
        duration: Duration(
            milliseconds: 500));
    _animation = Tween<double>(begin: 1.0, end: 2.0).animate(
        new CurvedAnimation(parent: _animationController!, curve:
            Curves.elasticIn));
    _animation!.addListener(() {
      setState(() {
        fontsize = 12.0 * _animation!.value;
        myPadding = 30 * ts - (36.0 * ts * _animation!.value - 36.0 *
             ts) / 2;
      });
    });
    _animation!.addStatusListener((status) {
      if (status == AnimationStatus.completed) {
        _animationController!.reverse().then((x) {
            widget.onPressed();
        });
      }
    });
  }

  @override
```

```
53    Widget build(BuildContext context) {
54     ts = getTextScale(context);
55      Text txt = Text(widget.text, style: new TextStyle( fontSize:
            fontsize * ts, color: Colors.white));
56      return Container(height: 56.0 * ts, child: Stack(
57        clipBehavior: Clip.none, fit: StackFit.loose,
58        children: <Widget>[
59            ButtonTheme(
60              minWidth: 88.0 * ts *
61                  _animation!
62                      .value,
63              height: 36.0 * ts *
64                  _animation!
65                      .value,
66              child: ElevatedButton(style: ButtonStyle(
67                elevation: MaterialStateProperty.all(1.0),
68                padding: MaterialStateProperty.all(EdgeInsets.all(15 *
                      ts)),
69                shape: MaterialStateProperty.all(RoundedRectangleBorder(
                      borderRadius: BorderRadius.circular(30.0 * ts)),),),
70                child: txt,
71                onPressed: this.widget.enabled ? () =>
                      _animationController!.forward() : () {}),
72            )
73          ],
74      ),
75      );
76    }
77  }
```

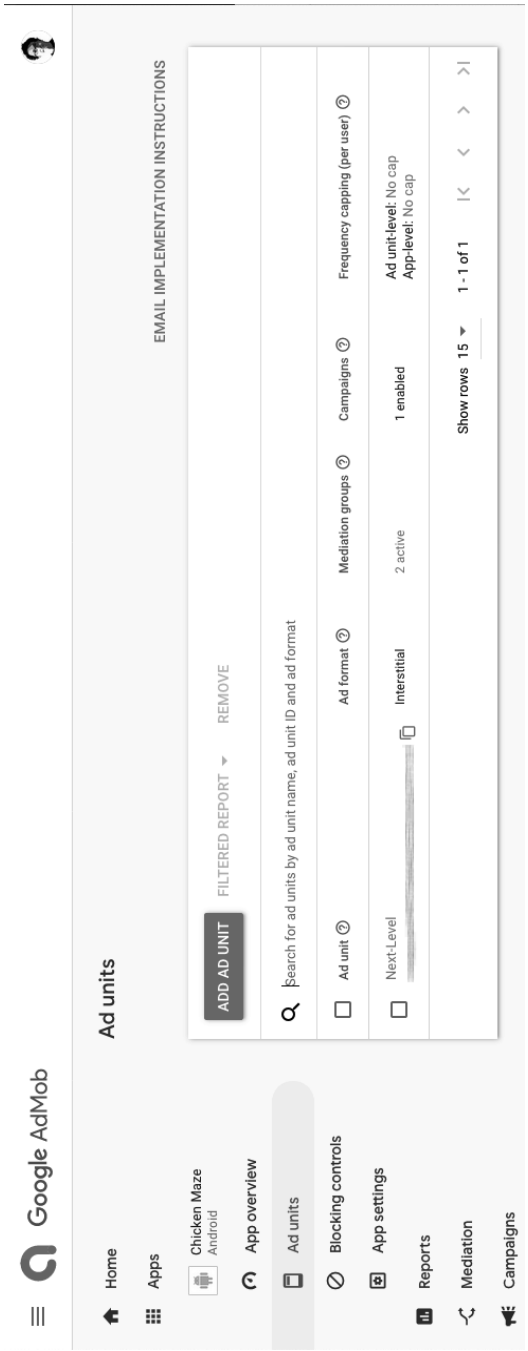


Fig. 7.11: Screenshot Google AdMob Ad Unit

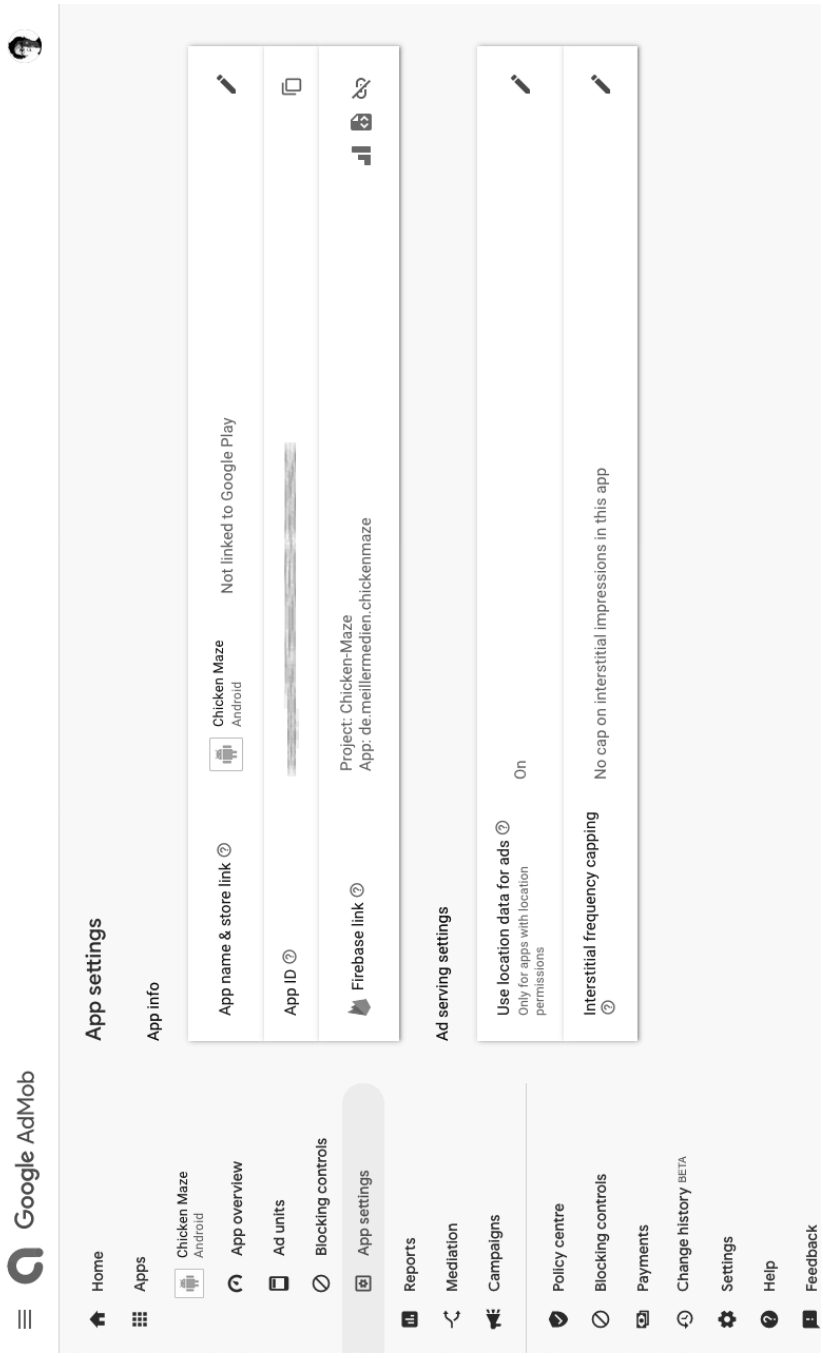


Fig. 7.12: Screenshot Google AdMob App Settings

8 Deployment

During development, the app is tested in debug mode on the emulators or available test devices. Now it is time to publish an app. To do this, you have to go to the tier of the desired platform, iOS or Android. Here you still have to distinguish between working in the cloud and at the project level. This means that you leave the Flutter world and work in the platform-specific cloud backend and in the respective platform folders. For Android, this is the folder "android," and iOS the corresponding folder "ios" in the project folder. Detailed instructions are available for Android at [21], for iOS at [22]. The description below is mainly dealing with the Android world. It does not present all the details, because a book is not the right place for this.

8.1 Releases

The work on iOS and Android is conceptually similar and can be found at the references given. Please note that for iOS you have to install and configure XCode. From a workflow point of view, the following steps must be taken to prepare the app for the respective stores:

- Add icons
- Add signature
- Create a release version
- Create additional media for the store
- Upload app

8.1.1 Create Icons

You still need an icon for the app. This is first of all a question of design. It is a good idea to have a look at the Material-Design Guidelines, which comes with a description of a material-design icon [44]. For Apple, there are analogous Human Interface Design Guidelines for the Cupertino Design [5]. The most important is to design the graphics for the icon in a way that users can see them easily, because they are only quite small on the device display. Additionally, the graphic should have a clear recognition value. You should consider the contrast of the colors. You can also use transparencies and shadows as design options.

The icon must be provided in different resolutions, as different device types and operating systems require different resolutions. For this work, there is a Flutter-based command line tool that can be used to generate corresponding icons from a source graphic [59]. To use it, you have to include the tool in the *pubspec.yaml* file. In Listing 7.2, the passage is added in lines 24-29. Here you can also enter the path to

https://doi.org/10.1515/9783110721331-008

the source graphic under a new section `flutter_icons:`. At the `dev_dependencies:`, the dependencies necessary only for development, the tool is specified. After the installation with `flutter pub get`, you can start the generation process with the command `flutter pub run flutter_launcher_icons:main`. For Android, a number of PNG graphics of different sizes are then created starting from the folder `android/app/src/main/res`. In addition to generating the icons, the icon entry in the AndroidManifest.xml file is changed. For iOS, the icon assets are also changed.

8.1.2 Signing for Android

After you have created the icons and entered them into the *AndroidManifest.xml* file, you have to prepare the signature of the app. To do this, you use a Java technology called the keystore. This is a store for keys according to the public key procedure. You can store your private keys there. First, you have to create a new keystore with the Java program *keytool*. In the home directory, you have to execute the following command to generate a file, *key.jks*.

```
keytool -genkey -v -keystore key.jks -keyalg RSA -keysize 2048 -
    validity 10000 -alias key
```

Then you have to specify passwords for the keystore and the key (named "key"). These have to be written to the *android/key.properties* file that has to be created. Furthermore, this file has to be referenced in the gradle file *android/app/build.gradle*, where further configurations are required. The details are quite extensive and can be read online [21].

Gradle files are the build files on Android that determine the configurations of the versions. Gradle code is written in Groovy, a standalone programming language for the Java platform. Groovy has a certain similarity to Dart.

8.1.3 Build Process for Android

When all configurations are complete, you can continue to create a release version. There are three possibilities:

- Fat APK (.apk)
- APKs for more ABIs (.apk)
- App bundle (.abb)

The Fat APK is a file (practically a ZIP file) containing the compiled code for several processor architectures, namely ARM 32-bit, ARM 64-bit and x86 64-bit. Additionally, all resources are included. The advantage of this file type is that it is executable on all

devices. The disadvantage is the size of the file: Among the three possible files, such a file is the largest, consumes the most memory on the device, takes the longest time, and requires the largest data volume for the download. Fat APKs are created with the command `flutter build apk`, which must be executed in the project folder.

You can also create several APKs for the respective architectures:

```
flutter build apk --split-per-abi
```

These are then uploaded together into the play store and delivered separately for the respective devices.

However, the recommended way is to use an App-Bundle File, that can be created with the command `flutter build appbundle`. This also contains all architectures and resources. The difference to the Fat APK is: The app-bundle file is not the executable app like an APK file. The store checks which device wants to install the app and generates an APK optimized for the device. Thus, the code for the processor architecture is selected. In addition, the optimized graphics for the device are selected, depending on the type, whether mobile phone, tablet or smart-TV.

There are always problems with the App-Bundles, such as reported device crashes. Furthermore, it is not possible to distribute an app-bundle, because it is not an executable file. Since Flutter works with vectors for the layout anyway, often no resources are wasted for graphics, so the variant to offer ABKs for several ABIs is a good option.

8.2 Google Play Management

The Google Play Console [39] (see figure 8.1) is a Google website where you can upload the app. There are two areas: the release management and the app presence in the play store. In the release management, you have the following areas where you can upload app versions:

- Internal test track
- Closed track (Alpha)
- Open track (Beta)
- Production track (the final versions)

The internal test track can be used to test during development. However, you have to register the tester by e-mail (Google-Mail Account) under the heading "Internal App Release." Also, those who are allowed to upload an app for testing must be registered there. In order to test the app correctly, there is the possibility to get a certificate in this section to use APIs that require a certificate.

In the closed track (Alpha) you can then make the app available to a larger group of testers. The tests can also take place in different countries.

In the open track (Beta), the app is then already visible in the Play Store. Interested users can voluntarily participate in the test program.

In the production track, the final versions are then uploaded. All versions, including the test versions, must have a unique version number, that can be entered in the *pubspec.yaml* file (see Listing 7.2, line 3). The version name here becomes 1.0.1 and the version code becomes 3. All new versions should get a new name and code.

8.2.1 Signing in the Play Store

Figure 8.2 shows the area with the signatures. You can only install signed apps on Android systems. The signing of the apps should guarantee that the app is really from the developer and has not been changed afterward. Otherwise, fraudsters could modify an app and use it for criminal purposes such as spying.

It is important to know that there are two signatures. The developer signs the app before uploading it so that the Play Store ensure sure that the developer is always the same. In the cloud, the app is then signed again, using a key that is kept secret by Google. However, you can create and upload this key yourself with the *keytool* (see section 8.1.2) so that the app's digital fingerprint from the store is the same as that of a self-created and self-installed app. However, this option must be activated first. This is necessary for app bundles. The hash values of the certificates can be downloaded from the signature page if you have to specify them in APIs for plugins.

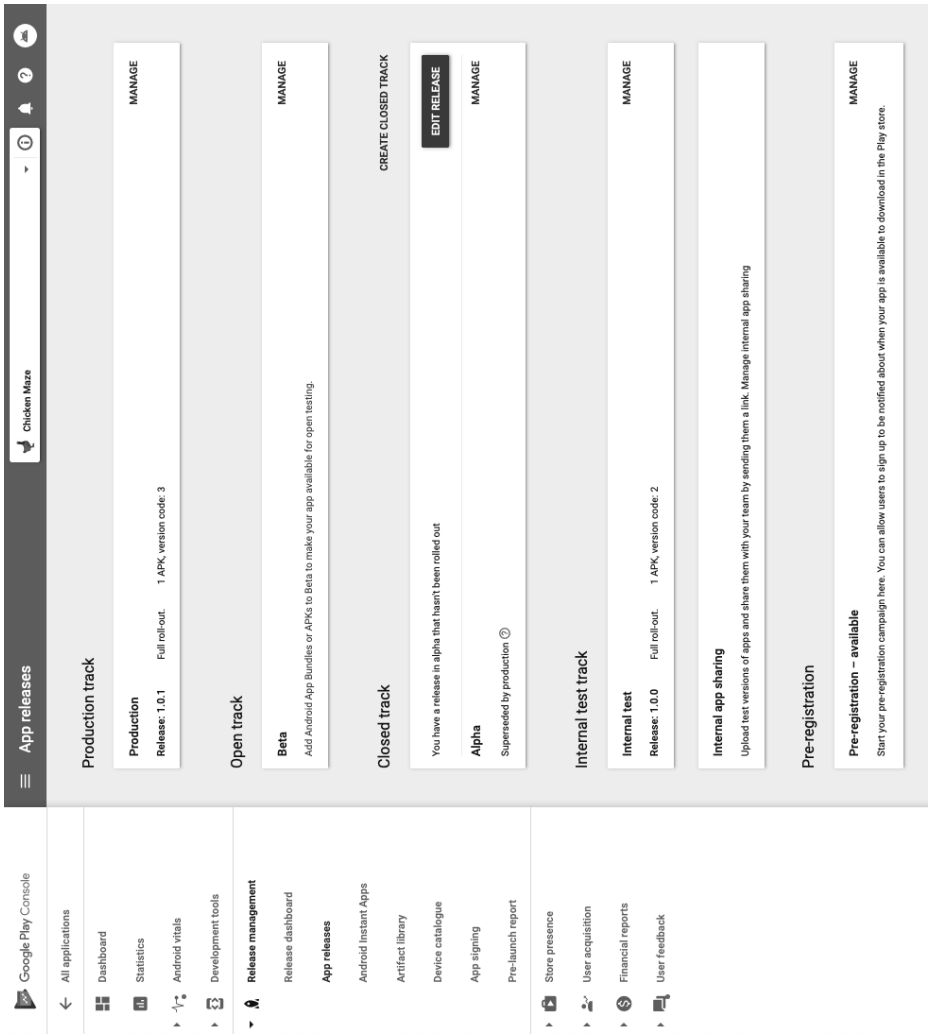


Fig. 8.1: Play Store console: release management

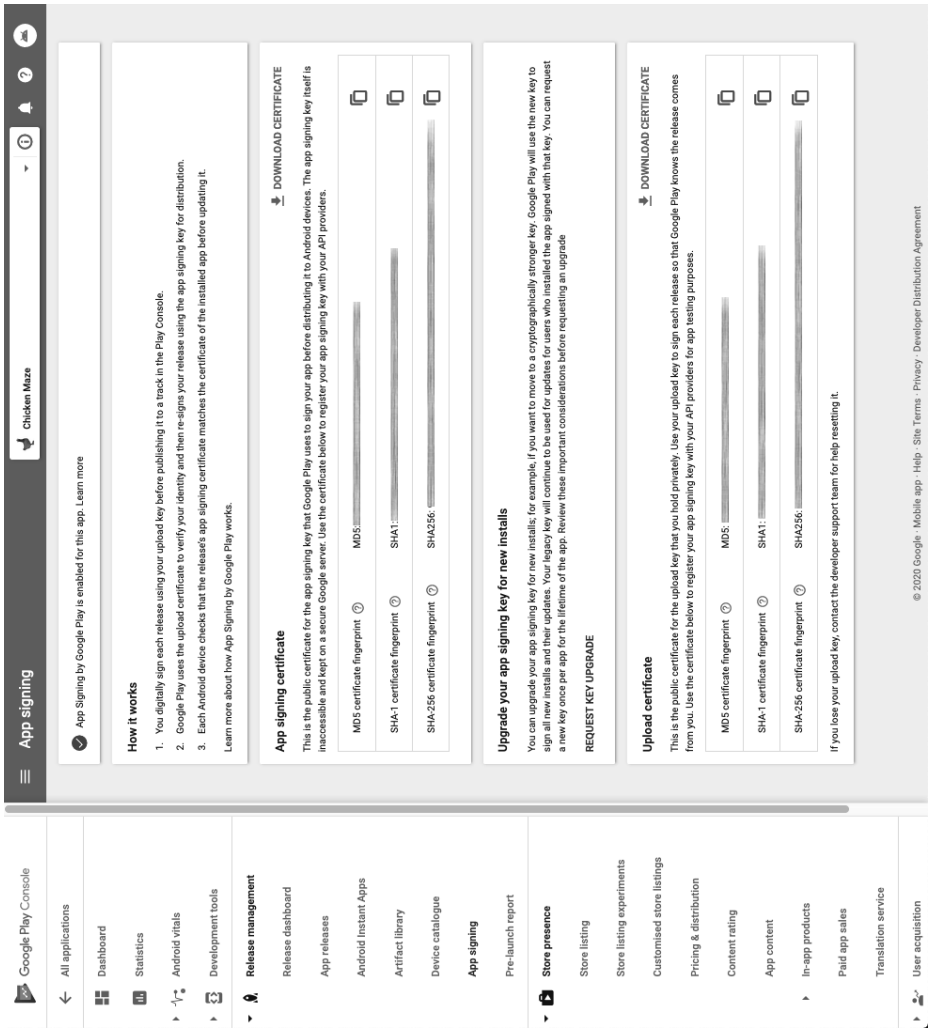


Fig. 8.2: Play Store Console: App Signature

8.2.2 Store Entry

The app presence in the Play Store is managed under the corresponding menu item (see figure 8.3). You have to provide various graphics and text to promote the app. A functional graphic, a web graphic and a TV banner in certain resolutions and graphic formats should be uploaded here. There is an option to upload a stereoscopic 360-degree image for Daydream to be used in VR systems to place advertising. Also, advertising videos can be added via Youtube link.

Additionally, you should add screenshots to give an impression of the running app. Screenshots are possible for different devices with the following categories being offered: Phone, Tablet, Android TV, and Wear OS. The app must then be put into a category: Is the app a game or something else? Which game category does the app belong to, Arcade or board game?

The content must also be classified, as it ensures that the content is suitable for certain age groups. This classification is checked and may take a while. In any case, the app is checked by Google. It is also possible that an app may be removed from the store if policies have changed.

It is still important to add descriptive texts and their translations so that you can add new languages. You can select the countries where the app should be offered in the menu “Pricing and Distribution.” For pricing, you can choose between paid and free. You can also create in-app products, hence functions that you can buy, and which will be activated upon purchase.

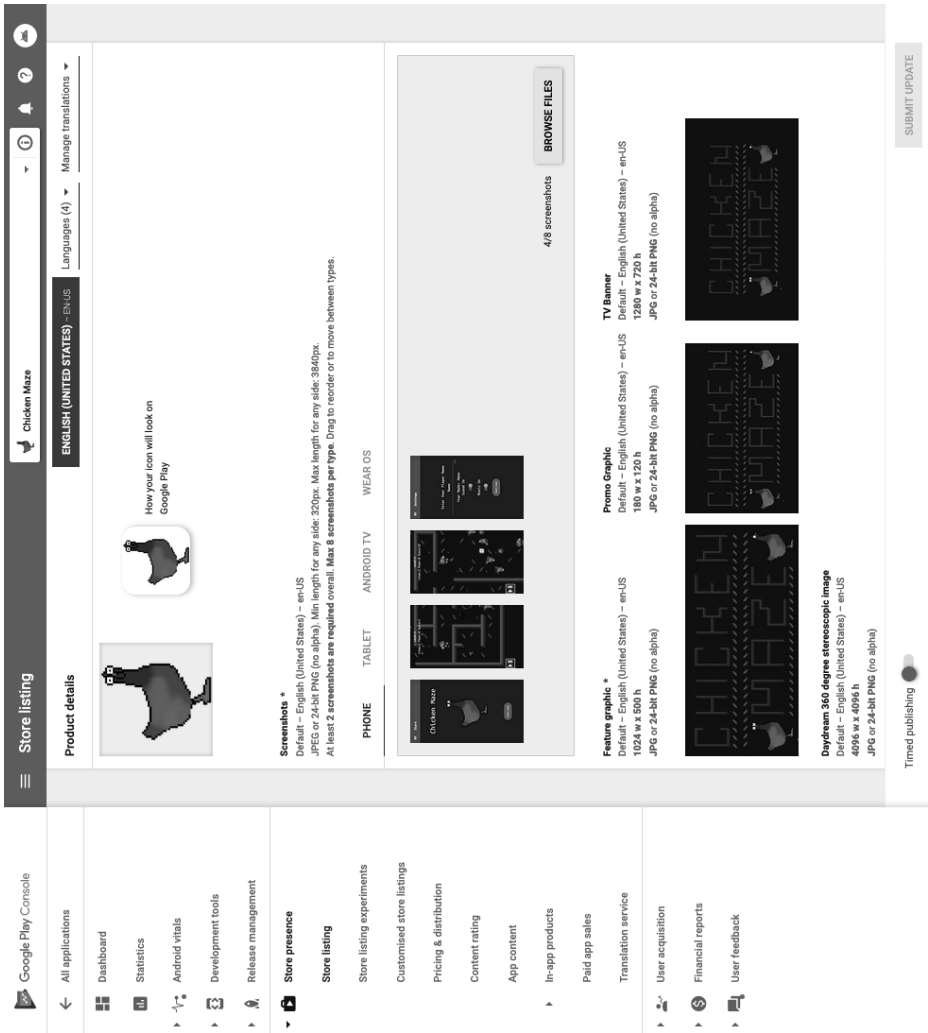


Fig. 8.3: Play Store Console: App Presence in the Play Store

9 Summary

This book explains the basics of the programming language Dart and the basics of the Flame library. It is followed by a practical part where a chat app, a desktop app for drawing and then an extensive game project is presented. The usual problems that can occur during development and that the author himself encountered during development are part of the discussion, too.

9.1 Outlook

The language Dart as well as the Flutter framework are constantly evolving. With this book, you have the state of the art of the Dart language in version 2.12 and Flutter in version V.1.22.4 (channel stable). So far, the technologies covered are backward compatible, which makes the content of the book a treasure for long years to come.

---

i On the "Insights" page at StackOverflow [63] trends can be searched. Readers could repeat this with the tags "Flutter" and "Dart."

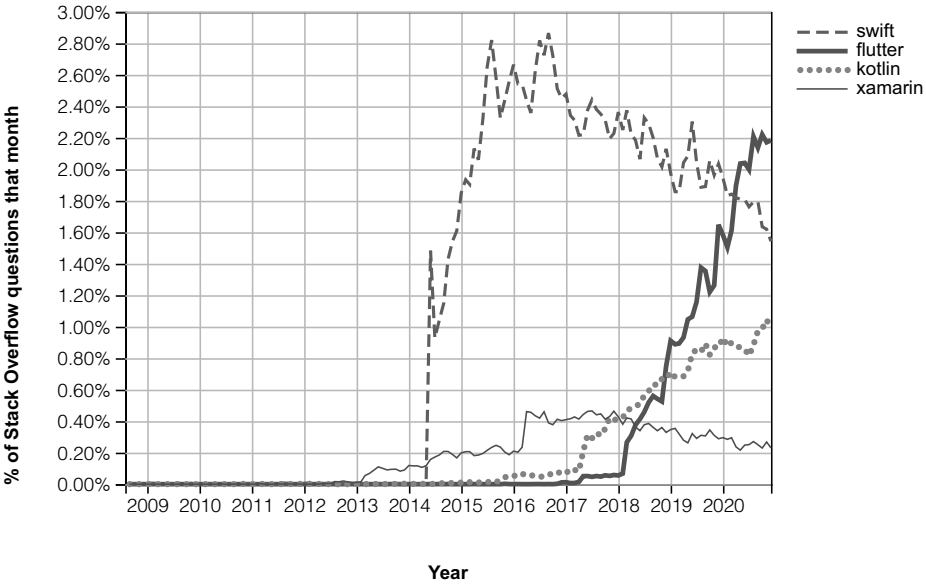


Fig. 9.1: StackOverflow Trends (see [63])

https://doi.org/10.1515/9783110721331-009

At the time this book was written, there was a strong increase in interest in the Dart language on various statistical pages. Also, when looking at the trends on StackOverflow, a rapid increase in demand for Flutter technology could be observed. Kotlin, for comparison, which is a language that can be used for Android programming, slowed down its increase. Swift, the language used in iOS development, slowed down.

Based on these statistics, it is assumed that the interest in this relatively new technology will continue, as it offers some advantages over other technologies: You can program apps for different target platforms with one language and one framework. The code is fast, easy to read and maintain. In addition, the look is modern as it implements the material design. The documentation is always up to date and very detailed. Additionally, there is a very active community that writes tutorials and articles. The open source community has also put numerous plug-ins and open source projects online. Google, as a big player, has often stated that they intend to use the Flutter technology as a graphical framework for various platforms of the future.

Flutter is a young technology, so the experience with Flutter is not a rich one yet. Therefore I, ask for your feedback and also welcome possible suggestions that will help improve the efficiency in app development for the entire community in the future.

With this book and the examples, I hope to contribute to the further spreading of the knowledge about Dart and Flutter and wish all readers much success and joy while reading and developing.

Bibliography

[1] Adobe. *Adobe Creative Cloud*. https://www.adobe.com/de/creativecloud.html. Access: 02-23-2020.
[2] Adobe. *Adobe Flash*. https://de.wikipedia.org/wiki/Adobe_Flash. Access: 02-14-2020.
[3] Vladimir Agafonkin. *Leaflet*. https://leafletjs.com. Access: 03-02-2020.
[4] AngularDart. *AngularDart*. https://angulardart.dev. Access: 08.06.2019.
[5] Apple. *App Icon*. https://developer.apple.com/design/human-interface-guidelines/ios/icons-and-images/app-icon/. Access: 02-28-2020.
[6] Autodesk. *List of all Products*. https://www.autodesk.de/products/. Access: 02-23-2020.
[7] Vladimir Babenko. *Flutter App Lifecycle*. https://medium.com/pharos-production/flutter-app-lifecycle-4b0ab4a4211a. Access: 12-12-2019.
[8] Felix Blaschke. *Articles about Simple Animations*. https://github.com/felixblaschke/simple_animations/blob/master/documentation/ARTICLES.md. Access: 10-02-2019.
[9] Felix Blaschke. *Simple Animations*. https://pub.dev/packages/simple_animations. Access: 09-26-2019.
[10] David Cheah. *Flutter : How to do user login with Firebase*. https://medium.com/flutterpub/flutter-how-to-do-user-login-with-firebase-a6af760b14d5. Access: 01-18-2020.
[11] Creative Commons. *Creative Commons*. https://creativecommons.org. Access: 02-26-2020.
[12] Douglas Crockford. *JSON*. https://www.json.org/. Access: 08-09.2019.
[13] Dart. *Dart Homepage*. https://dart.dev/guides. Access: 08-09-2019.
[14] Dart. *Dart Platforms*. https://dart.dev/platforms. Access: 08-06-2019.
[15] Dart. *Get Dart*. https://dart.dev/get-dart. Access: 06.06.2019.
[16] Dart. *Migrating to null safety*. https://dart.dev/null-safety/migration-guide. Access: 01-26-2021.
[17] Dart. *Sound null safety*. https://dart.dev/null-safety. Access: 01-26-2021.
[18] Google Developers. *Android Studio*. https://developer.android.com/studio. Access: 02-14-2020.
[19] Google Developers. *Get the Google USB Driver*. https://developer.android.com/studio/run/win-usb. Access: 02-16-2020.
[20] Fabcoding. *How to obtain SHA1 Keys for debug and release – Android Studio (MAC)*. http://fabcoding.com/how-to-obtain-sha1-keys-for-debug-and-release-android-studio-mac. Access: 01-18-2020.
[21] Flutter. *Build and release an Android app*. https://flutter.dev/docs/deployment/android. 02-28-2020.
[22] Flutter. *Build and release an iOS app*. https://flutter.dev/docs/deployment/ios. Access: 02-28-2020.
[23] Flutter. *Desktop support for Flutter*. https://flutter.dev/desktop. Access: 02-11-2021.
[24] Flutter. *Developing Packages & Plugins*. https://flutter.dev/docs/development/packages-and-plugins/developing-packages. Access: 09-15-2019.
[25] Flutter. *Flutter packages*. https://pub.dev/flutter. Access: 09-19-2019.
[26] Flutter. *Flutter SDK*. https://api.flutter.dev/flutter/dart-async/Stream/forEach.html. Access: 02-07-2020.
[27] Flutter. *Install*. https://flutter.dev/docs/get-started/install. Access: 02-13-2020.
[28] Flutter. *Internationalizing Flutter apps*. https://flutter.dev/docs/development/accessibility-and-localization/internationalization. Access: 02-10-2020.
[29] Processing Foundation. *Processing*. https://processing.org. Access: 02-09-2020.
[30] Apache Friends. *XAMPP*. https://www.apachefriends.org. Access: 02-10-2020.

https://doi.org/10.1515/9783110721331-010

[31] Gimp. *GIMP - GNU Image Manipulation Program*. https://www.gimp.org. Access: 02-23-2020.
[32] Git. *Getting Started - Installing Git*. https://git-scm.com/book/en/v2/Getting-Started-Installing-Git. Access: 02-13-2020.
[33] GNU. *What is free software?* https://www.gnu.org/philosophy/free-sw.html. Access: 02-26-2020.
[34] Google. *Backdrop*. https://material.io/components/backdrop. Access: 09-15-2019.
[35] Google. *Fuchsia Operating System*. https://fuchsia.dev. Access: 02-18-2021.
[36] Google. *Google AdMop*. https://admob.google.com/home/get-started. Access: 12-30-2019.
[37] Google. *Google Firebase*. https://firebase.google.com/. Access: 01-12-2020.
[38] Google. *Google Fonts*. https://fonts.google.com. Access: 02-26-2020.
[39] Google. *Google Play Console*. https://play.google.com/apps/publish. Access: 02-27-2020.
[40] Google. *Google Play Games Services*. https://developers.google.com/games/services. Access: 01-20-2020.
[41] Google. *Homepage Material Design*. https://material.io. Access: 09-11-2019.
[42] Google. *Homepage Material Design Lite*. https://getmdl.io. Access: 09-11-2019.
[43] Google. *Navigation Drawer*. https://material.io/components/navigation-drawer. Access: 02-10-2020.
[44] Google. *Product icons*. https://material.io/design/iconography. Access: 02-28-2020.
[45] Inkscape. *Draw Freely - Inkscape*. https://inkscape.org. Access: 02-23-2020.
[46] Rostyslav Ivankiv. *Flutter Colorpicker*. https://pub.dev/packages/flutter_colorpicker. Access: 02-11-2021.
[47] Norbert Kozsir. *Widget Maker*. https://www.widgetmaker.dev. Access: 02-16-2020.
[48] Thorbjørn Lindeijer. *Tiled Map Editor - A flexible level editor*. https://www.mapeditor.org. Access: 12-29-2019.
[49] Dieter Meiller. *Chicken Maze on GitHub*. https://github.com/meillermedia/chicken_maze. Access: 02-06-2021.
[50] Dieter Meiller. *Draw App on GitHub*. https://github.com/meillermedia/draw_app. Access: 02-11-2021.
[51] Dieter Meiller. *Messenger App on GitHub*. https://github.com/meillermedia/messenger_app. Access: 02-06-2021.
[52] Dieter Meiller. *World Cup on GitHub*. https://github.com/meillermedia/world_cup. Access: 02-06-2021.
[53] Microsoft. *Visual Studio Code*. https://code.visualstudio.com. Access: 02-14-2020.
[54] Nevercode. *Firebase authentication & Google sign in using Flutter*. https://blog.codemagic.io/firebase-authentication-google-sign-in-using-flutter. Access: 01-18-2020.
[55] Luan Nico. *Flame-Engine*. https://github.com/flame-engine/flame. Access: 10-12-2019.
[56] Luan Nico. *Play Games*. https://github.com/flame-engine/play_games. Access: 01-20-2020.
[57] J. Nowak. *Fortgeschrittene Programmierung mit Java 5: generics, annotations, concurrency und reflection ; mit allen wesentlichen Neuerungen des J2SE 5.0*. dpunkt-Verlag, 2005. ISBN: 9783898643061.
[58] OpenGameArt. *OpenGameArt.org*. https://opengameart.org. Access: 12-29-2019.
[59] Mark O'Sullivan. *Flutter Launcher Icons*. https://pub.dev/packages/flutter_launcher_icons. Access: 02-28-2020.
[60] Sonic Pi. *Sonic Pi*. https://sonic-pi.net. Access: 02-26-2020.
[61] Rive. *Rive*. https://rive.app. Access: 02-13-2020.
[62] John Ryan. *Flutter Map*. https://pub.dev/packages/flutter_map. Access: 03-02-2020.
[63] StackOverflow. *StackOverflow Trends*. https://insights.stackoverflow.com/trends?tags=flutter. Access: 02-12-2020.
[64] Flutter Studio. *Flutter Studio*. https://flutterstudio.app. Access: 02-16-2020.

[65] Audacity Team. *Audacity*. https://www.audacityteam.org. Access: 02-23-2020.
[66] testapp.schule. *File Picker Cross*. https://pub.dev/packages/file_picker_cross. Access: 02-11-2021.
[67] Wikipedia. *Secure Hash Algorithm*. https://de.wikipedia.org/wiki/Secure_Hash_Algorithm. Access: 02-07-2020.

List of Figures

https://doi.org/10.1515/9783110721331-011

Index

https://doi.org/10.1515/9783110721331-012