Guide to Flutter

This guide is made for new Flutter developers to get a head start on developing application through the Flutter framework. Flutter is an open-source framework designed by Google, allowing developers to create and distribute applications to multiple platforms using a single code base[1].

Content

[1] https://flutter.dev/ (2020-12-23)

How to get started

In this chapter we will install the Flutter framework together with the necessary dependencies as well explain the differences of using a virtual vs physical device.

Installation

The installation is simple and does not have a lot of requirements. The most common dependency is that Git[2] is installed on your machine and that you are using a 64-bit machine.

In Table 1 Machine Requirements you find a summary of the requirements needed by the most common operating systems. The table is based on Flutters own documentation flutter.dev/install (2020-12-23), remember to always check back on the provided documentations for future changes.

Table 1 Machine Requirements

Operating System	Disk Space	Tools	Shared Libraries
Windows 7 SP1 or later (64-bit), x86-64 based	1.32 GB	Windows PowerShell 5.0 (or later), git 2.x	-
macOS (64-bit)	2.80 GB	git or Xcode	-
Linux (64-bit)	600 MB	bash, curl, file, git 2.x, mkdir, rm, unzip, which, xz-utils, zip,	libGLU.so.1

At flutter.dev you find the newest most stable version of Flutter ready to be downloaded for your specific operating system (OS). Depending on your OS you might have to update the paths to allow Flutter to be executable.

To develop application in Flutter you must have the Android SDK installed, an easy way is to use Android Studio[3] to install it. Android Studio can also be used later if you choose it as your IDE. For IOS application you need a Mac with Xcode[4] installed.

Flutter doctor is your hero! Whenever you have a problem with the framework itself use the command $flutter\ doctor$ to check for what is wrong. The command executes a series of checks to determine what is wrong, it could be an old version of Flutter, the path to Android SDK or that your device is not connected correctly.

[2] https://git-scm.com/download/ (2020-12-23)
[3] https://developer.android.com/studio (2020-12-23)
[4] https://developer.apple.com/xcode/ (2020-12-23)

Virtual and physical device

Depending on which OS you are running, and for which target group your application is intendent to you may need different hardware to execute debug mode. You can execute your application on IOS and Android from MacBook with MacOS installed, however on Windows and Linux it is only possible on to execute on Android.

A virtual device is a mimic of a phone, either an Iphone or an Android running on your computer. This is good for instances like testing how different phone sizes affect the application or if you do not have a that specific hardware by yourself.

A physical device is perhaps your own mobile phone. This is good for instance when testing functionality like camera or GPS but also see how the application behave in the real-life environment.

The core of Flutter

In this chapter we will learn about how the Flutter framework is working behind the scenes.

Flutter is built using Widget trees, almost everything is a Widget! Each Widget is built using other Widgets and can use multiple attributes. Like colouring, styling and much more.

Flutter is a framework for creating native applications but with one single code base. Meaning that while you develop an application for Android you will also be able to build an IOS edition.

Programming language

The language used in Flutter is Dart[5]. Dart is like Java but a bit more friendly. One key feature that distinguish these two is that Dart does not have strict typing. Meaning that you can write a variable without a type or return type of a function. You also do not use encapsulation in the same way, for instance instead of “private” you use “_” in Dart (see Figure 1 Code Example).

```
class HelloWord{
 String _phrase = "Hello";
 HelloWord();
 get phrase => _phrase;
}
void main() {
 HelloWord helloWord = HelloWord();
 print(helloWord.phrase);
}
```

Figure 1 Code Example

[5] https://dart.dev/ (2021-02-03)

Your First APP

First time creating an application you must run the command “flutter create <project_name>”. This will create all files necessary with a template. But how does it work?

The template is a small application with two main Widgets, some logic behind them and data binding. The first Widget is the FloatingActionButton and the second one is a Text Widget, well not really... The Text Widget is inside a Center Widget which position the Text to the center of the screen. Well, to be honest everything is actually inside one big Widget called MaterialApp.

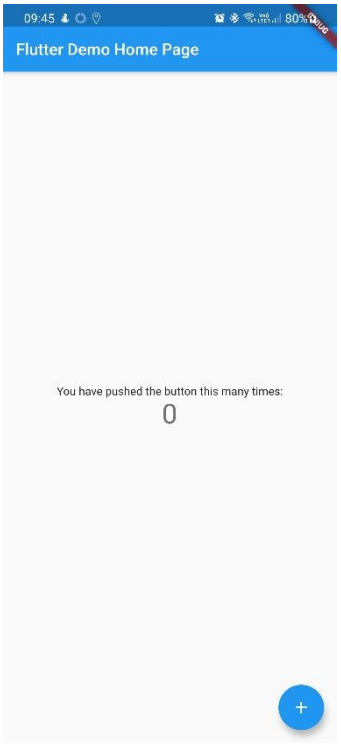

Figure 2 Example App Material

Figure 3 Example App Cupertino

What is MaterialApp? – It is a template Widget given by the Material Design Studio allowing you to develop applications with the Material Design standards. Without it you would have to design everything by your own! Flutter has provided two main Widgets MaterialApp (see Figure 2 Example App Material) and CupertinoApp (see Figure 3 Example App Cupertino), the designs are based on Android or IOS. However, you can use which ever you want, it does not depend on the hardware. One important note is that by using either of these you also get a prebuild navigation handler.

Style

When using either MaterialApp or CupertinoApp you can set a predefined colour schematic and themes for different kinds of Widgets (see Figure 4 Example of styling). See it as the CSS file for an HTML webpage. All Material Design Widgets will automatically refer to this schematic unless you override it, the same goes for Cupertino Widgets.

```
MaterialApp(
  theme: ThemeData(
    primaryColor: Colors.blue,
    accentColor: Colors.green,
    textTheme: TextTheme(bodyText2: TextStyle(color: Colors.purple)),
  ),
```

Figure 4 Example of styling

Most Widgets can override the design, like the Text Widget, you might want different fonts on different pages, different sizes or even want a specific word in bold. Normally you just set the style property, however in some Widgets you have to override the decoration property. Which is a more advanced method for styling like if you want a border around the container.

Widgets

There are two types of Widgets, stateless and stateful, easy to remember right? – It always easier to convert a stateless Widget to a stateful Widget. But when should you use these? – If you ever must redraw something on the screen with new material use a stateful Widget.

The Widget tree is the structure for creating Widgets, you have the top Widget then you can combine multiple Widgets however you want (see Figure 5 Code Snippet).

```
return MaterialApp(
  home: Center(
    child: Text("Hello"),
  ),
);
```

Figure 5 Code Snippet

There are different kinds of Widgets, those which does not draw anything and those who draws something. For instance, a Container Widget does not show up on the screen, however it set constraints for other Widgets, like height and width. The same goes for the Column and Row Widgets which consist of multiple Widgets but in horizontal or vertical direction. These are useful when you want a clean and responsive layout instead of coding each position by yourself.

Data binding

One easy way to send data between Widgets is to send references as properties. Data can be either values or functions. Values are good when you want to show something in your child Widget, while functions are good for updating values (see Figure 6 Example of data binding).

```
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  Widget showCounter(int value) {
    return Text(
      '$_counter',
    );
  }

  Widget addToCounter(Function addFunction) {
    return CupertinoButton(
      child: Text("Add"),
      onPressed: addFunction,
    );
  }

  @override
  Widget build(BuildContext context) {
    return CupertinoPageScaffold(
      child: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text('You have pushed the button this many times:'),
            showCounter(_counter),
            addToCounter(_incrementCounter),
          ],
        ),
      ), // This trailing comma makes auto-formatting nicer for build methods.
    );
  }
}
```

Figure 6 Example of data binding

Navigation

As mentioned by using MaterialApp or CupertinoApp you get a built-in navigation handler. The navigator can be reached by the Navigator Widget. If you want to create a new screen on the fly then

you can use Navigator.push(context, PageRoute). However, if you have a pre-set of routes than you can assigned them on the App with the property routes (see Figure 7 Example of pre-set routes). It is like a real webpage where subpages are defined as /path. When using pre-set routes, you instead use Navigator.pushNamed(context, 'path').

```
MaterialApp(
  // Start the app with the "/" named route. In this case, the app starts
  // on the FirstScreen widget.
  initialRoute: '/',
  routes: {
    // When navigating to the "/" route, build the FirstScreen widget.
    '/': (context) => FirstScreen(),
    // When navigating to the "/second" route, build the SecondScreen
widget.
    '/second': (context) => SecondScreen(),
  },
);
```

Figure 7 Example of pre-set routes

One last note on this part is that if you want to replace the current screen with a new one, you just call Navigator.pushReplacement or Navigator.pushreplacementNamed. Otherwise, you will always be able to go back to the early page. The Navigator keeps a stack on all pages therefore, it is important that you keep it clean, and do not push to much on it.

Packages

A package or plugin is a premade solution normally applicable for any kind of application. They are most commonly adaptive and can be designed by your applications style. A package can be published or private. Either way you can add them by changing your pubspec.yaml file (see Figure 8 Example of a pubspec file). There is a large amount of packages on pub.dev for both Dart and Flutter.

```
name: Project
description: A Password Strenght Meter and wizard/stepper project app
version: 1.0.0+1

environment:
  sdk: ">=2.7.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter
  custom_password_validator: ^0.0.1+1
  multi_wizard: ^0.0.1+1
```

Figure 8 Example of a pubspec file