7. Introduction to Flutter

Introduction

The purpose of this chapter is to give the reader a quick introduction to Flutter before installing it and starting to use it.

What is Flutter?

Flutter is not a language (like JavaScript, for example). Flutter uses Dart for its language.

Flutter is Google's mobile SDK / UI framework that enables developers to build native apps that run on Android and iOS devices. Developers write code in a single codebase that works on both platforms.

High Productivity

Flutter was written for high productivity, to get apps out fast.

- You can change your code and hot reload the changes, without any kind of delay.

- Flutter includes the UI Widgets you need.
- Flutter works with most IDEs.

High Quality

The included Flutter UI Widgets work seamlessly and conventionally with the target platform. Scrolling, navigation, icons and fonts match the target system.

- When you write an Android app with the Flutter Widgets it looks like a normal Android app.
- When you write an iOS app with the Flutter Widgets, it looks like a normal iOS app.

High Performance

The code you write in Flutter runs natively so it flies!

It is Free and Open.

Flutter is free and Open Source.

Fuschsia

Fuschsia is Google's next Operating System for mobile devices. All of the apps for Fuschsia are being developed by Google in Flutter.

Flutter Source Code

Google Flutter is open source and it comprises of several repositories hosted on GitHub here:

https://github.com/flutter/flutter

- Flutter
    - Main repository.
- Samples
    - Sample code repository.
- Plugins
    - This repository contains the source code for plugins developed by the core Flutter team to enable access to platform-specific APIs.
- Engine
    - The Flutter runtime, written in C++.
- Flutter Intellij
    - The Flutter plugin for IntelliJ.
- Flutter Website
    - Flutter.io code.

Flutter SDK

The Flutter SDK contains all the elements you need to perform Flutter development:

Dart Platform

We covered the Dart platform in the previous chapter.

Flutter Engine

The Flutter Engine is the runtime for on which Flutter applications run. It provides graphic rendering support, as well as providing an interface (through the Flutter core libraries) to the Android or iOS SDK layer below.

It was written in C++ and is on github here: https://github.com/flutter/engine

Foundation Library

The Foundation library, written in Dart, provides basic classes and functions which are used to construct applications using Flutter, such as APIs to communicate with the engine.

It was written in Dart and is on github here: https://github.com/flutter/flutter/tree/master/packages/fl

Flutter Release Process

When developing day-to-day with Flutter, Google's engineers work in the 'dev' branch.
Every month, the 'dev' branch is rolled into the 'beta'

branch.
Every quarter, the ‘beta’ branch is rolled into the ‘stable’ branch.
https://github.com/flutter/flutter/wiki/Release-process

Flutter Channels

When working with Flutter, you can decide on what version of Flutter you wish to work on. Most people work on the ‘stable’ channel, which provides the developer with the ‘stable’ version of Flutter.

Available Channels

You can use the ‘flutter channel’ command to work on a more ‘bleeding-edge’ version of the framework, such as ‘dev’. The following channels are available:

master	Bleeding-edge build.
dev	Latest fully-tested build.
beta	Selected, most stable dev build (of the month).
stable	Selected, most stable beta build (of the quarter).

View Current Channel

```
flutter channel
```

Change Channel

The following command list the available channels and highlights the current channel.

```
$ flutter channel

Flutter channels:
* stable
  beta
  dev
  master flutter channel dev
```

The following command selects a channel.

```
flutter channel dev
```

After selects another channel, you should use the following command to update the Flutter library:

```
flutter upgrade
```

Further Reading:

https://github.com/flutter/flutter/wiki/Flutter-build-release-channels#how-to-change-channels

8. Installing Flutter & Editor

Introduction

The purpose of this chapter is to help the reader install Flutter and an editor.

Note: Developing on a PC for iOS

Introduction

You can develop Flutter applications on a PC, and you will have no problems at all until you want to run your code on an Apple iOS device, like an iPhone or an iPad. Apple has made compiling of iOS applications exclusively available to macOS using their XCode tool.

It's Not as Bad as it Seems

Flutter really works well from a cross-platform point of view and you can do 90% of the development on a PC even if you are planning to deploy to iOS. You really can develop on one platform then run it on another and trust that it will almost completely work on the other.

When you get to testing and deployment you will have some options:

- Buy, borrow or rent a Mac.
- Install a Mac virtual machine on your PC using software like VMWare or Virtual Box.
- Rent a Mac on the cloud for $20 a month using a service like www.macincloud.com.

Install Flutter

Introduction

I am not going to go into every detail about Flutter installation because there are plenty of better sources of information about this:

- Youtube – there are lots of videos on this.
- Official Flutter website: https://flutter.io/docs/get-started/install

It's not a terribly difficult process but I am going to cover the basic process, which is similar on all of the environments.

Step 1: Software Pre-Requisites

Git

One thing I noticed from installing Flutter was that nobody mentioned that Git was a pre-requisite for

installing Flutter. So, ensure you have git installed before doing anything else.

Brew

If you are planning on installing Flutter on a Mac, it's a good idea to install Brew first as the Flutter Doctor will ask you to use brew to install additional software when required.

XCode Command-Line Tools

If you are planning on installing Flutter on a Mac, you are definitely going to need these.

Step 2: Download the Flutter SDK

We mentioned the Flutter SDK earlier, how it has all the tools you need to perform basic Flutter development. However, it also has a very useful tool called Flutter Doctor that is used to setup your Flutter Development environment. Download this SDK and copy it into a folder.
Note that the Flutter SDK also contains the Dart SDK.

- Mac: https://flutter.dev/docs/get-started/install/macos
- PC: https://flutter.dev/docs/get-started/install/windows

- Unix: https://flutter.dev/docs/get-started/install/linux

Step 3: Setup Your Path

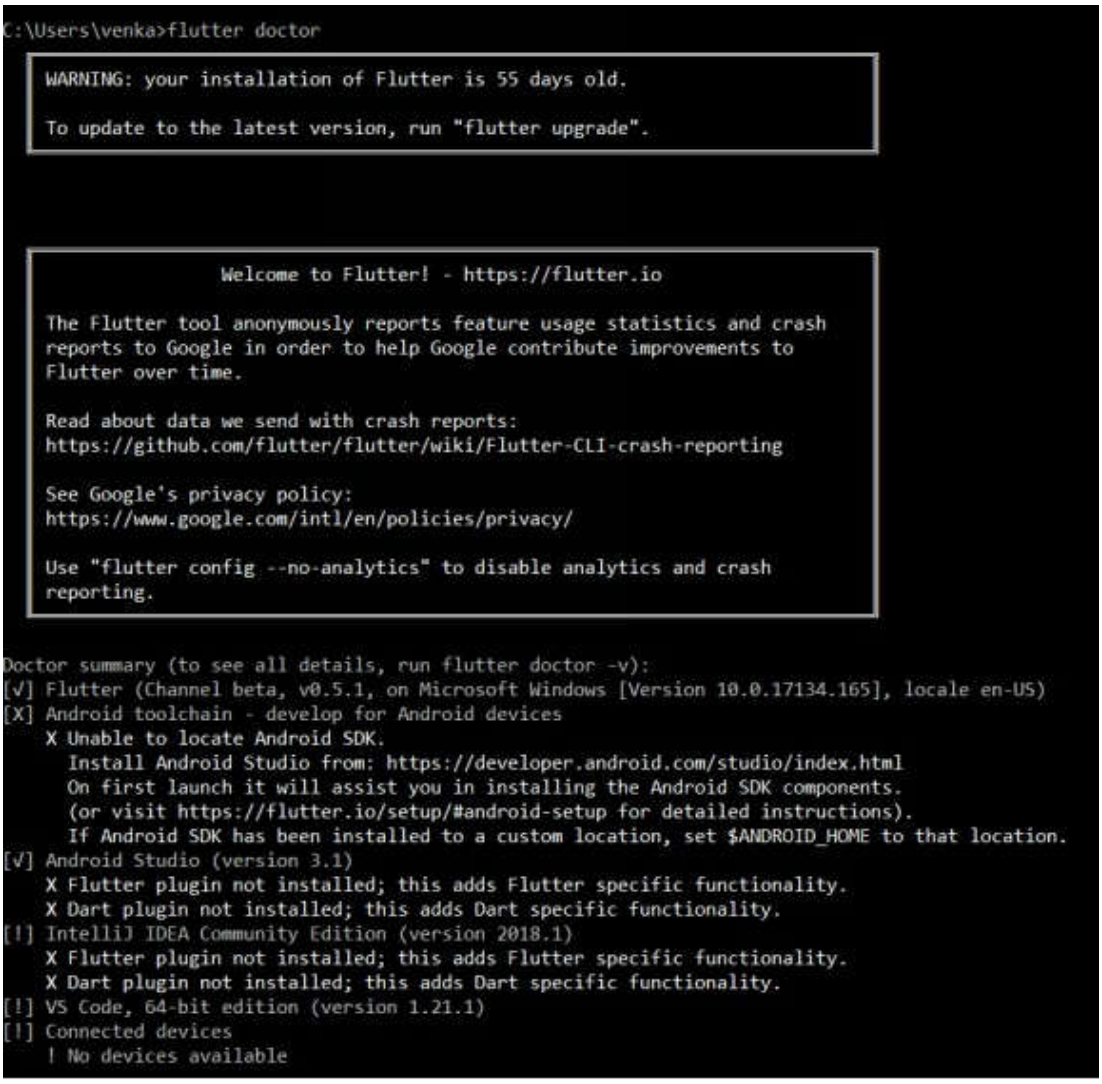
The Flutter SDK has command-line tools, including Flutter Doctor that need to be run from the command-line. These command-line tools reside in the 'bin' folder of the Flutter SDK. You need to include the bin folder (within the flutter SDK) in your computers path so that you can run the command-line tools from the command-line.

Step 4: Run Flutter Doctor

You will need to run the command below:

```
flutter doctor
```

This will checkout your environment and diagnose (like your doctor) what is good and bad about your flutter development environment. It will provide you with a summary, complete with instructions on what you need to do.



Here are my notes from running the install on 3 platforms:

- Just follow the instructions. Some of them are very simple, like saying ‘yes’ to licenses. Some are more involved.
- You might get a message about installing the missing Android SDK. This can be remedied

by installing Android Studio then running it, as the first thing it will do is setup the Android SDK.
    - At this point I usually download Android Studio because:
        - It is free.
        - It sets up the sdk for you.
        - It is great for setting up the Emulators.
        - You can still use Visual Studio Code for most of the work and leave Android Studio closed.
- If you are installing Flutter on a Mac (or Unix) rather than on a PC then there are many more dependencies (for iOS compilation, deployment etc.) and it can take much longer.

Flutter Commands

I know this chapter is about installing Flutter. However, I also thought that we should introduce the Flutter SDK commands:

Command	Description
flutter --help	Lists flutter commands.
flutter analyze	Analyze the project's Dart code.

flutter attach	Attach to a running application.
flutter bash-completion	Output command line shell completion setup scripts.
flutter build	Flutter build commands.
flutter channel	List or switch flutter channels.
flutter clean	Delete the build/ and .dart_tool/ directories.
flutter config	Configure Flutter settings.
flutter create	Create a new Flutter project.
flutter devices	List all connected devices.
flutter doctor	Show information about the installed tooling.
flutter drive	Runs Flutter Driver tests for the current project.
flutter emulators	List, launch and create emulators.
flutter format	Format one or more dart files.
flutter install	Install a Flutter app on an attached device.

flutter logs	Show log output for running Flutter apps.
flutter make-host-app-editable	Moves host apps from generated directories to non-generated directories so that they can be edited by developers.
flutter precache	Populates the Flutter tool's cache of binary artifacts.
flutter pub	Commands for managing Flutter packages.
flutter run	Run your Flutter app on an attached device.
flutter screenshot	Take a screenshot from a connected device.
flutter test	Run Flutter unit tests for the current project.
flutter upgrade	Upgrade your copy of Flutter.
flutter version	List or switch flutter versions.

Install Editor

Once you are done with the Flutter Doctor, you should install your editor.

Editors & UI Builders

None of the Flutter tools currently support a UI builder, where you can just drag and drop to build your UI. You have to code your Flutter UI's 'by hand', which is not difficult anyway. However, these editors help the developers in many ways, offering Code Completion, Error Highlighting, Linting and Debugging.

Flutter Doctor Tells You to Install Android Studio

Flutter Doctor tells you to install Android Studio. This is what the official Flutter website says:

> Note: Flutter relies on a full installation of Android Studio to supply its Android platform dependencies.

So, you should already have the Android Studio editor installed by the time you have got past the flutter doctor.

However, this does not stop you from using another editor.

Android Studio

Introduction

Android Studio is the official IDE for android application development as it provides a very comprehensive, well-supported (by Google) solution:

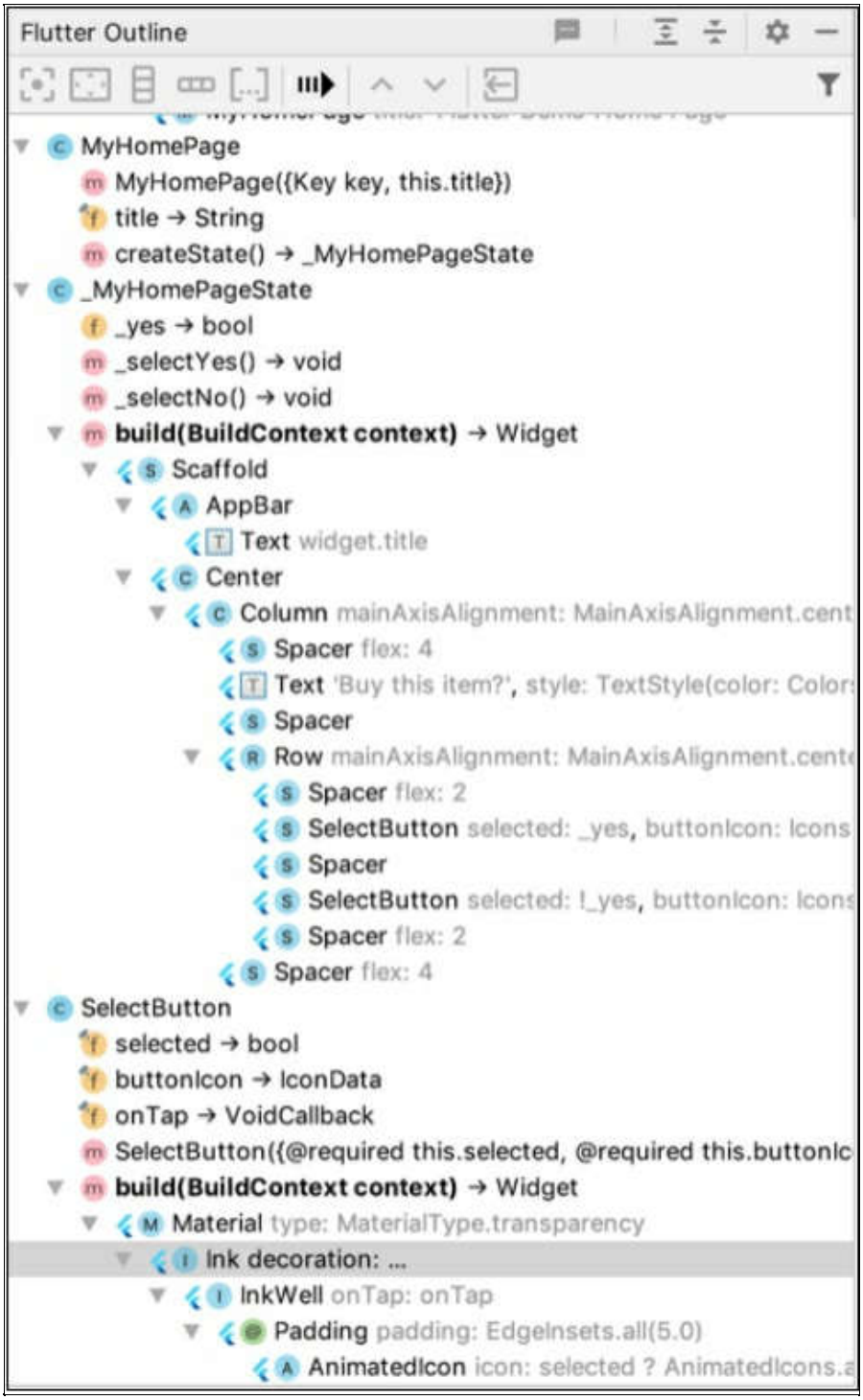
- It is a superb editor.
- It is also free to use.
- It also works (very well) for developing iOS applications in Flutter.
- It was based on IntelliJ IDEA, so it works in a very similar manner.

Thus, the easiest way to get going with an editor is to install the Flutter plugins into Android Studio. Installing the Flutter plugins takes all of five minutes:

1. Start Android Studio.
2. Open plugin preferences (**Preferences** > **Plugins** on macOS, **File** > **Settings** > **Plugins** on Windows & Linux).
3. Select **Browse repositories**, select the Flutter plugin and click **Install**.
4. Click **Yes** when prompted to install the Dart plugin.
5. Click **Restart** when prompted.

Flutter Outline

One of the great things about the Android Studio is the Flutter Outline. When you are editing a file, it shows you the Widgets defined in that file, their variables, their code and their structure.
It also lets you select Widgets in your ‘build’ methods and add Centering, Padding, Rows, Columns etc.



IntelliJ (Android Studio, IntelliJ)

If you are already using Intellij and you don’t want to use Android Studio, then you can simply add the Flutter plugin in the same manner as it is installed into Android Studio (see above):

- Installing the plugin takes 5 minutes.
- You can use the free IntelliJ IDEA Community Edition or buy a License to use the ‘full-fat’ version. I use IntelliJ IDEA all the time and I pay $20 a month, well worth it as I use it for my regular job every day.

Further Reading: https://flutter.io/docs/get-started/editor?tab=androidstudio

Visual Studio Code

Visual Studio code is a great alternative to using Android Studio and it is a little more ‘lightweight’

(runs faster, uses less memory):

- It is a superb editor.
- It is also free to use.

- Installing the Flutter extension takes 5 minutes.

I had never used Visual Studio code with Flutter before writing this book. I had used Flutter for months using Android Studio. I was really surprised and impressed how well the Flutter Extension works with Visual Studio Code. It is comprehensive, easy to use and fast. I definitely recommend it as an alternative to Android Studio Code.

Further Reading: https://flutter.io/docs/get-started/editor?tab=vscode

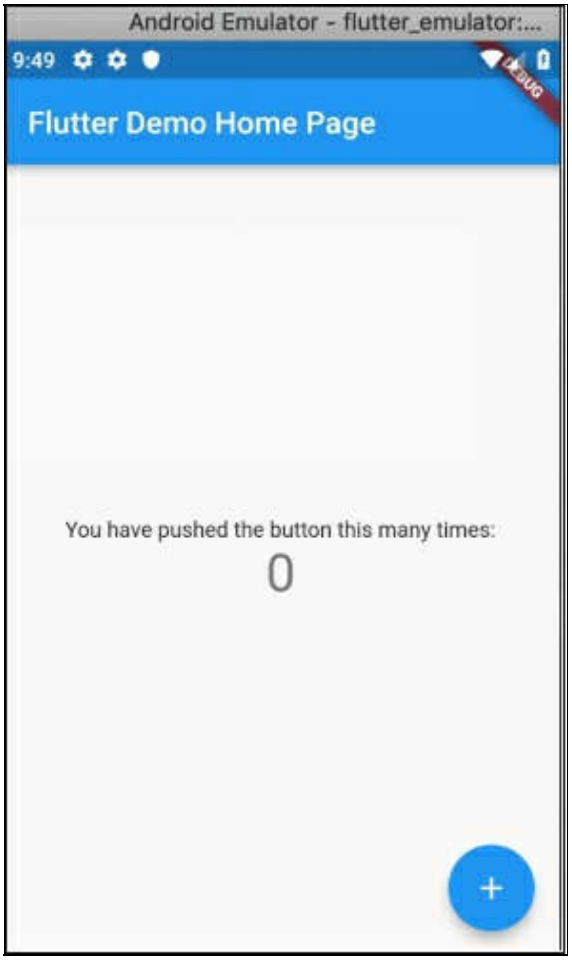
9. Create Default Flutter Project

Introduction

The purpose of this chapter is to get the reader generate his or her first Flutter project.
However, this chapter does not include running it yet! ☺

Default Flutter App

When you create a new Flutter project, it creates a default ‘counter’ app that displays a counter in the middle of the screen. This app is the same, whether you generate it in Android Studio, Visual Studio Code or the Command-Line. It allows the user to click on a ‘+’ round button on the bottom-right to increment the counter (this is a floating button).



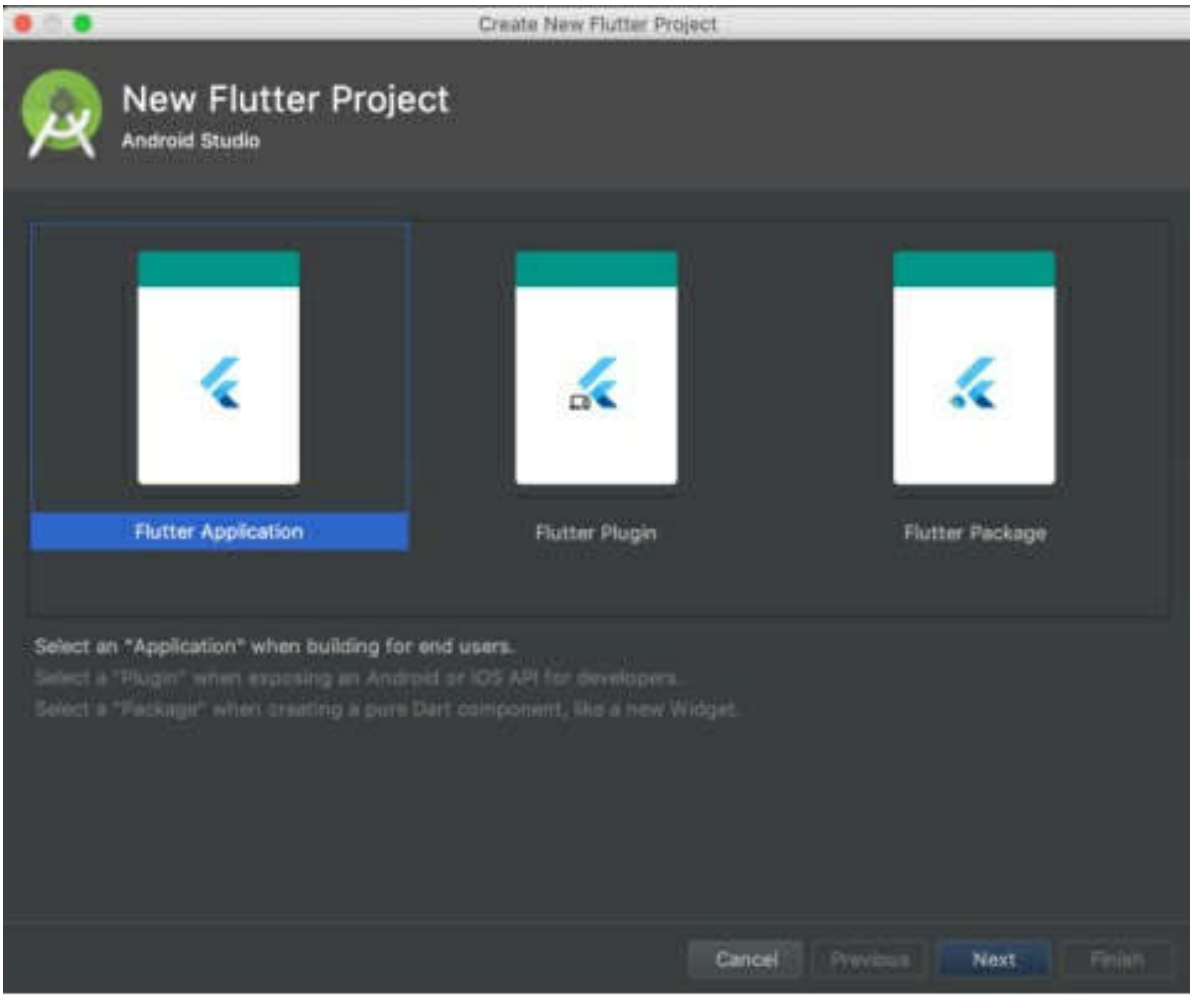
Create Project

You can create a new Flutter Project from your editor or the command-line:

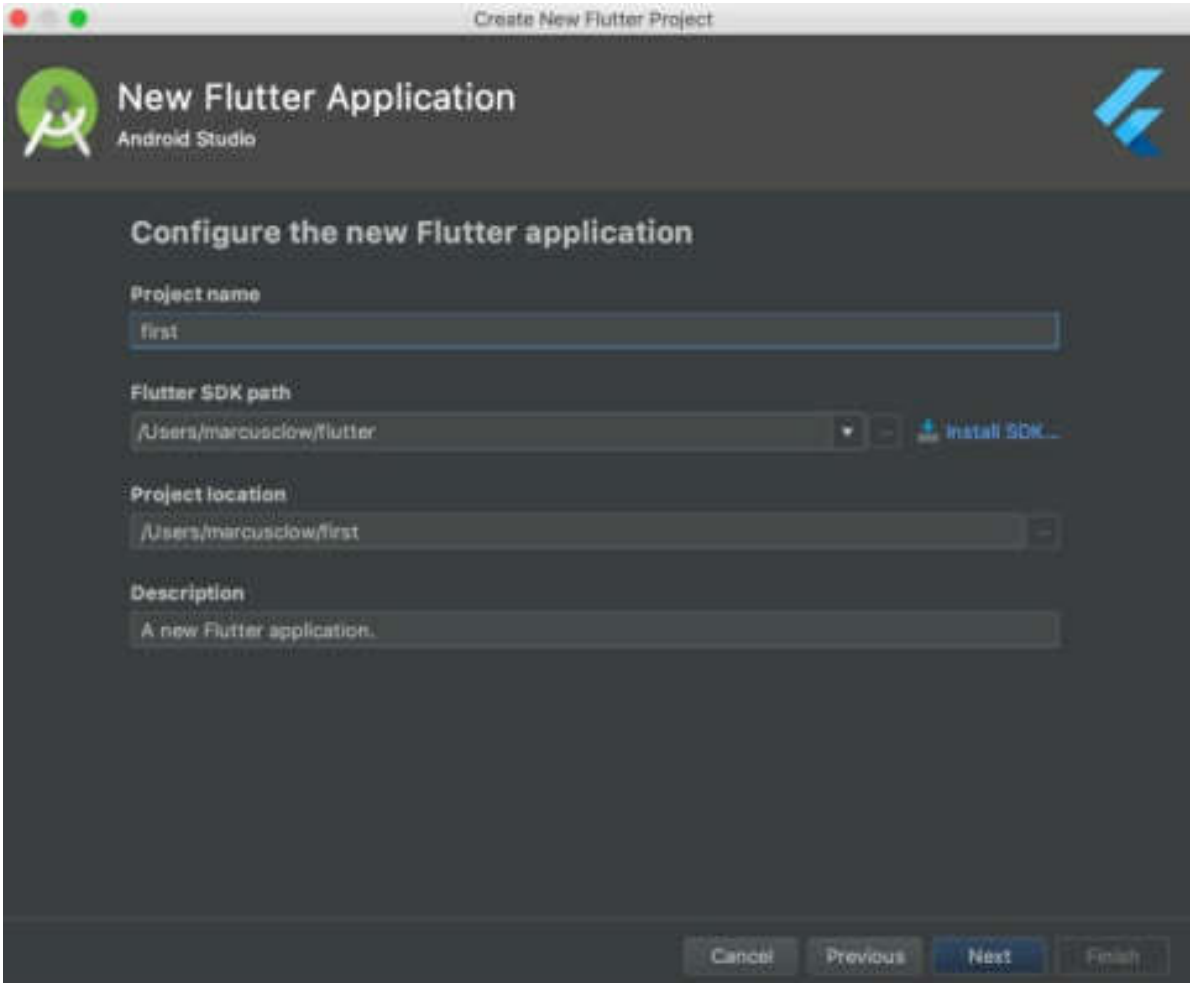
- Android Studio
- Visual Studio Code
- Command Line

Android Studio

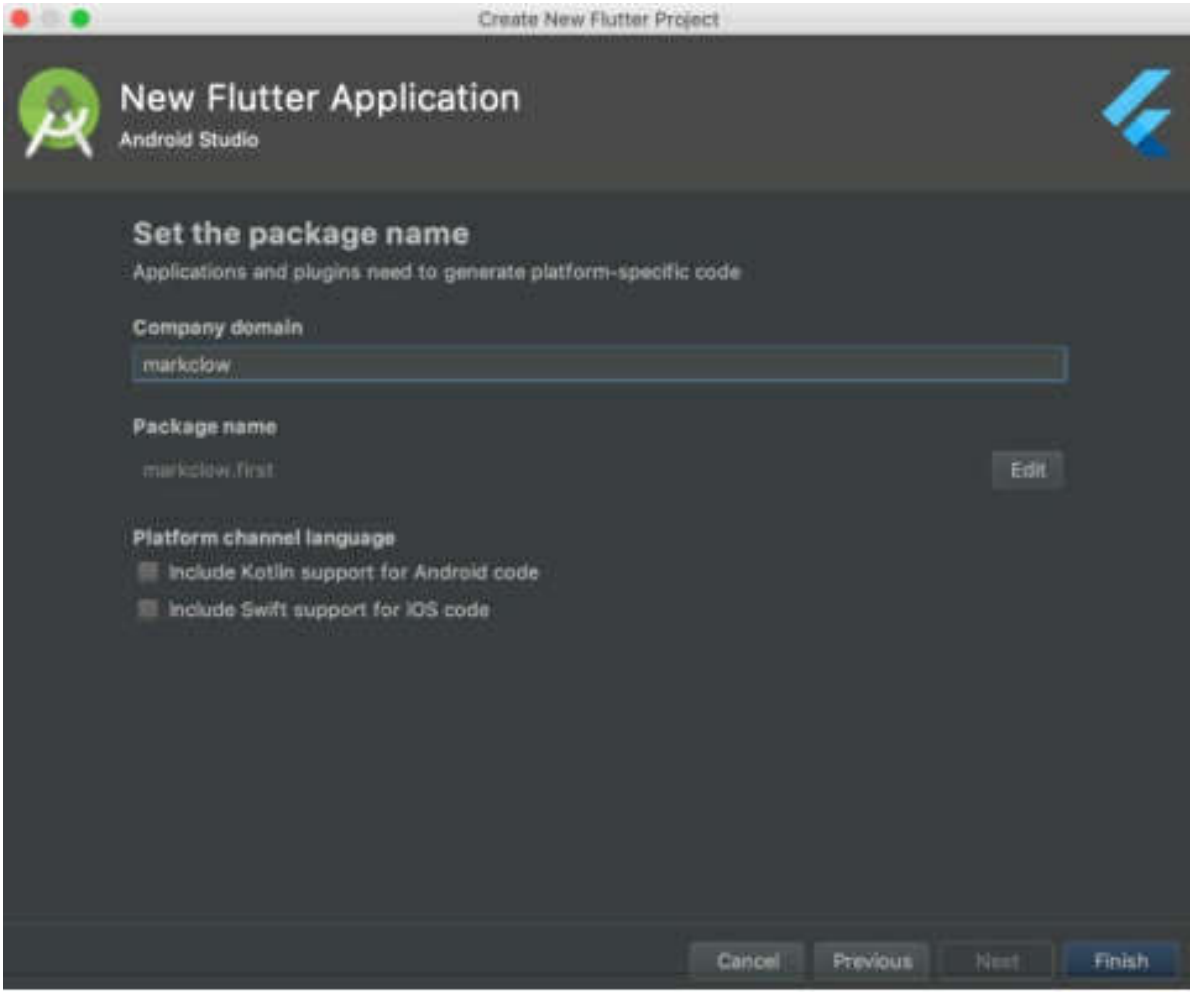
1. Select the following menu option: File > New > New Flutter Project. This will open a wizard.
2. Select ‘Flutter Application’ then hit next.



3. Enter the project name (whatever you want to call it) and hit next.



4. Enter the company domain (whatever you want) and hit finish.

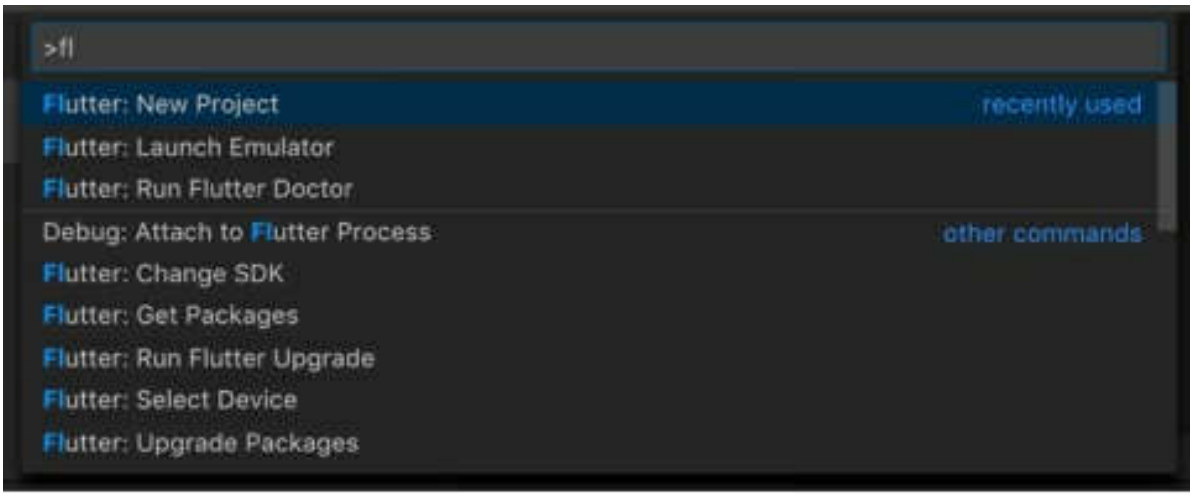


5. The editor will take a couple of minutes to setup the files in the project.
6. That’s it!

Visual Studio Code

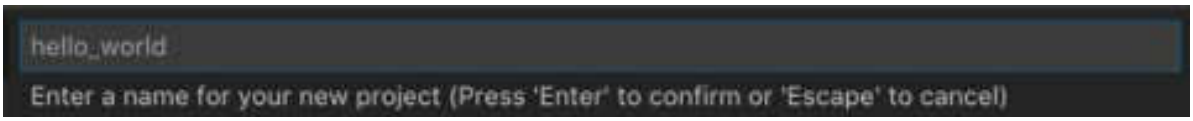
1. Ensure that you have installed the Flutter Extension into Visual Studio Code before doing this.

2. Open the command palette using the keyboard shortcut Ctrl+Shift+P (Command+Shift+P on the Mac) and you will see a list of the available commands. If you start to type ‘Flutter’ in this box, then you will see a list of Flutter commands:

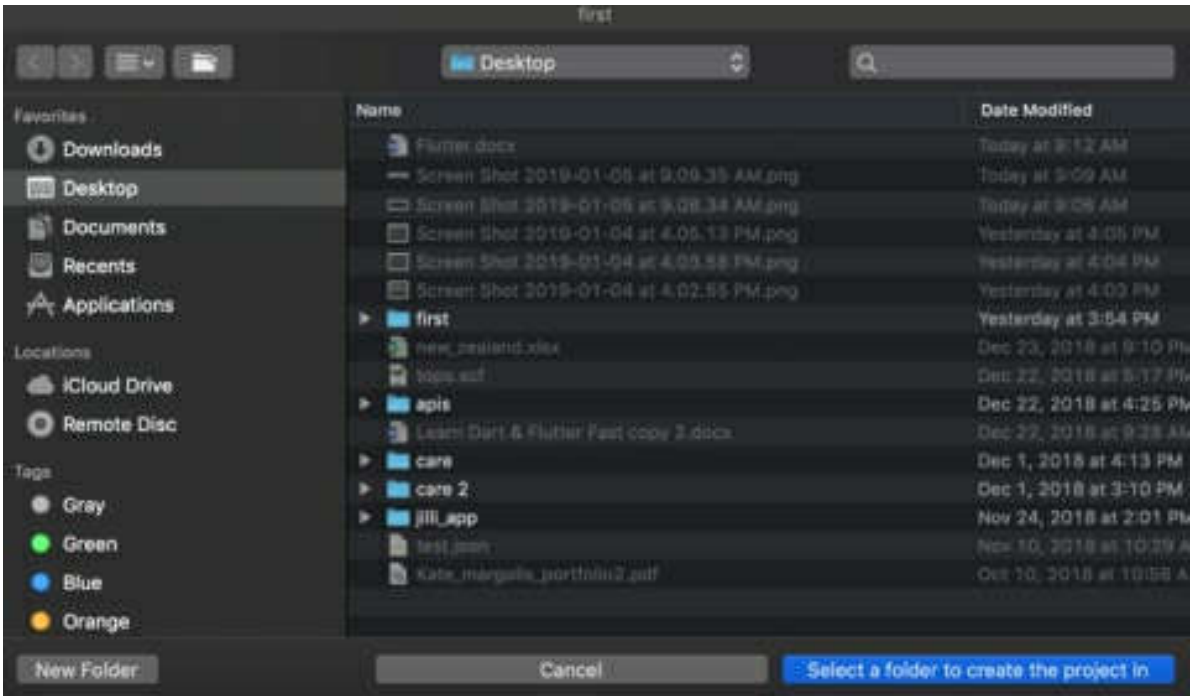


3. Select the command ‘Flutter: New Project’.

4. Enter the name of the new project:



5. Select a folder to create the project in.



7. The editor will take a couple of minutes to setup the files in the project.

8. That’s it!

Command-Line

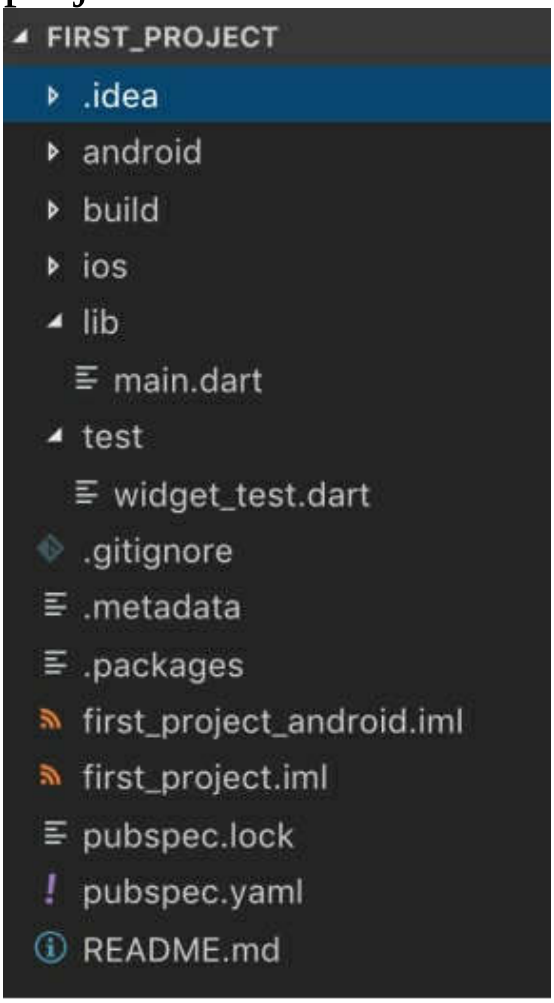
- Ensure that you have installed the Flutter SDK and it is on the path before doing this.
- Enter the command:

```
flutter create <project name>
```

- The command will take a couple of minutes to setup the files in the project.
- That’s it!

Project Created

Congratulations - you have created your first Flutter project! Now let's take a look at it.



Project Folders

The default Flutter project is organized into several folders.

Name	Description
[root]	Root folder.

	This usually contains configuration files. The most important of these configuration files is the ‘pubspec.yaml’ file, which declares the project dependencies and resources. We will cover this file in detail later.
android	As the name suggests, the folder contains all the Android-related files and code(s) for an Android project. This is where Android-specific settings and code resides. When building for Android, Flutter uses Gradle as the dependency manager. When you build your Flutter project, your Flutter code is generated into native code and injected into this folder. You very rarely change things in this folder.
build	This folder is created and used by gradle when you build the project. Generated and managed by the Flutter SDK. Don’t change things in this folder.
ios	Similar to the ‘android’ folder, this folder contains the iOS related files and code(s)

	for an XCode project. This is where iOS-specific settings and generated code resides. When building for iOS, Flutter uses Cocoapods as the dependency manager. Like the ‘android folder’, when you build your Flutter project, your Flutter code is generated into native code and injected into this folder. You very rarely change things in this folder.
lib	This is where the application code resides. You should see a file ‘main.dart’, the entry point for the Flutter application. This is the file you select and run. You will add code, files and subfolders into this folder. This is the folder in which you will work 90% of the time.
test	This is where the unit testing code resides. You may add more files and subfolders into this folder.

Project Files

The default Flutter project contains several important

files.

Name	Description
.metadata	Flutter SDK uses this file to save project info. Don’t change this file.
.packages	Generated file used to manage packages. See Dependencies and Packages. Don’t change this file.
[project_name].iml	Flutter SDK uses this file to save dependency info & settings. Don’t change this file.
pubspec.lock	Generated file used to list the specific versions of each dependency (immediate and transitive) that your project uses. Don’t change this file.
pubspec.yaml	Used to specify your projects dependencies – what 3rd party packages your project may use. You can also specify additional project resources in this file:

	fonts, images etc. See Dependencies and Packages. You will be able to change this file.
readme.md	Automatically generated readme file. You can add additional information about the project here.

Project Application Code File

As mentioned earlier, when you create a new Flutter project, it creates a default ‘counter’ app that displays a counter in the middle of the screen.

The code for this default application resides in a single file: ‘main.dart’ in the ‘lib’ folder.

Note

- Dart lets you declare multiple objects within a single file!
- This ‘main.dart’ acts as the entry point of the application.
    - If you are using an Editor, this is the file

you will launch to run your app.

Widgets & Composition

Introduction

Now we are looking at your project, we need to introduce some of the Flutter concepts, like Widgets and Composition.

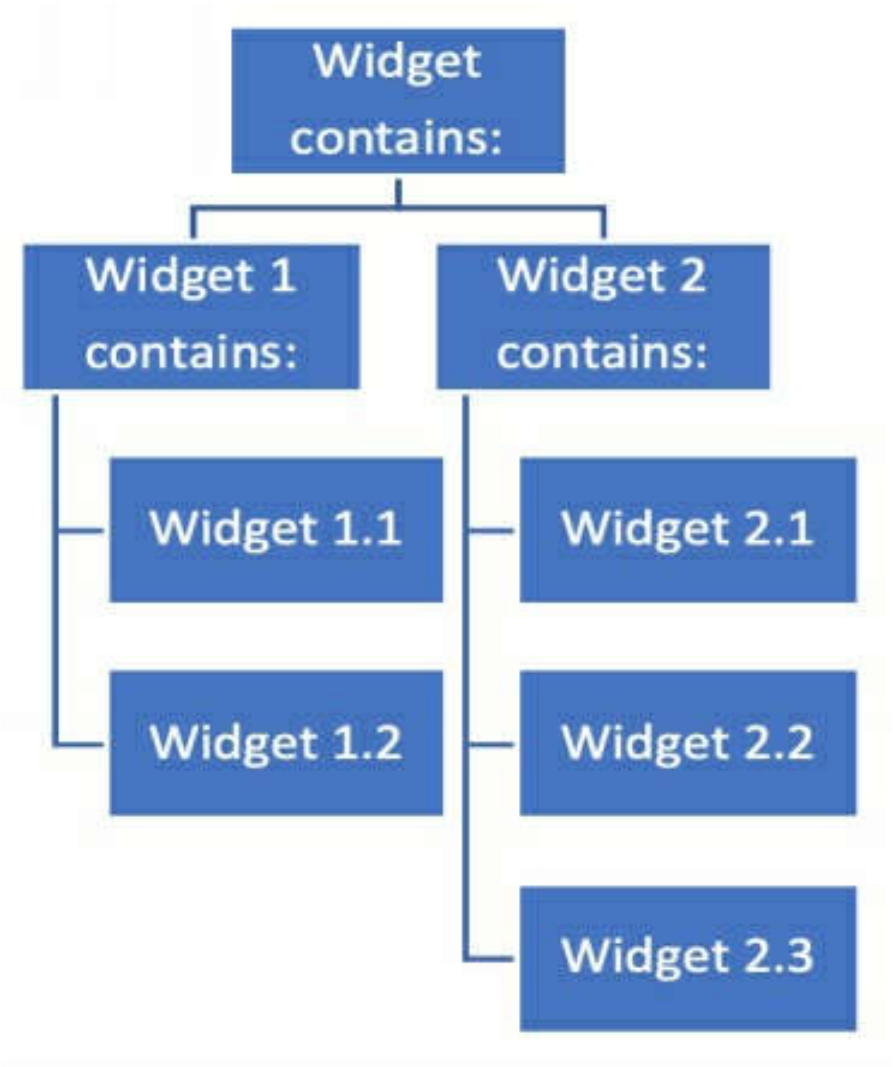
Widgets are the Building Blocks of your UI

Whenever we build a user interface in Flutter, it is composed of Widgets.
Putting your widgets together is called Composition.

Think of a user interface as a jigsaw. The jigsaw is composed of pieces (Widgets):

Widget Tree

Widgets can contain other widgets, in a tree structure, a hierarchy. This is often called a Widget Tree.



Project UI

First of all, lets explain how developers compose UIs using Flutter and explain how they write UIs as code.

UI As Code

Different Development Tools Offer Different Ways to Compose A UI

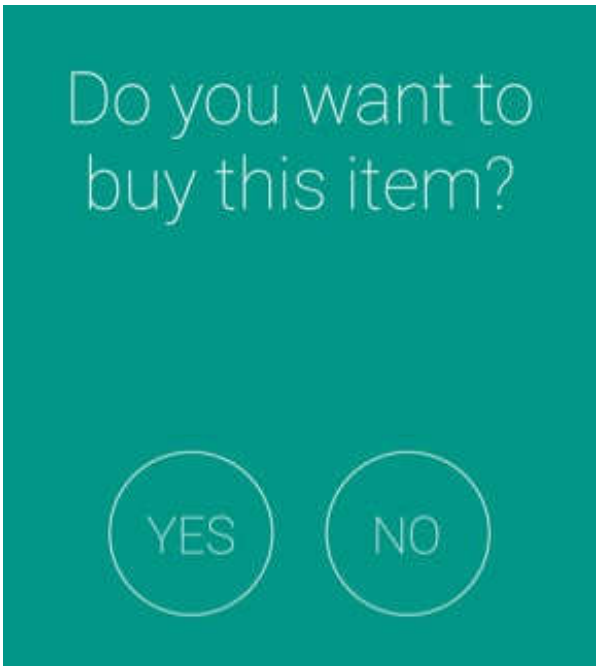
- Angular developers compose UIs using templates, which can be separate files or within the same file as the source code.
- React developers compose UIs using tags in the source code.

Flutter

- Flutter developers compose UIs using code, not editors or templates. You write a method that returns a Widget object, which consists of all of the required sub-Widgets, thus composing the UI.
- You (usually) use the same code for all platforms. Your Android and iOS apps will use the same source code.

Example:

The Dart code below is used to compose a Flutter UI for a Yes / No Dialog:

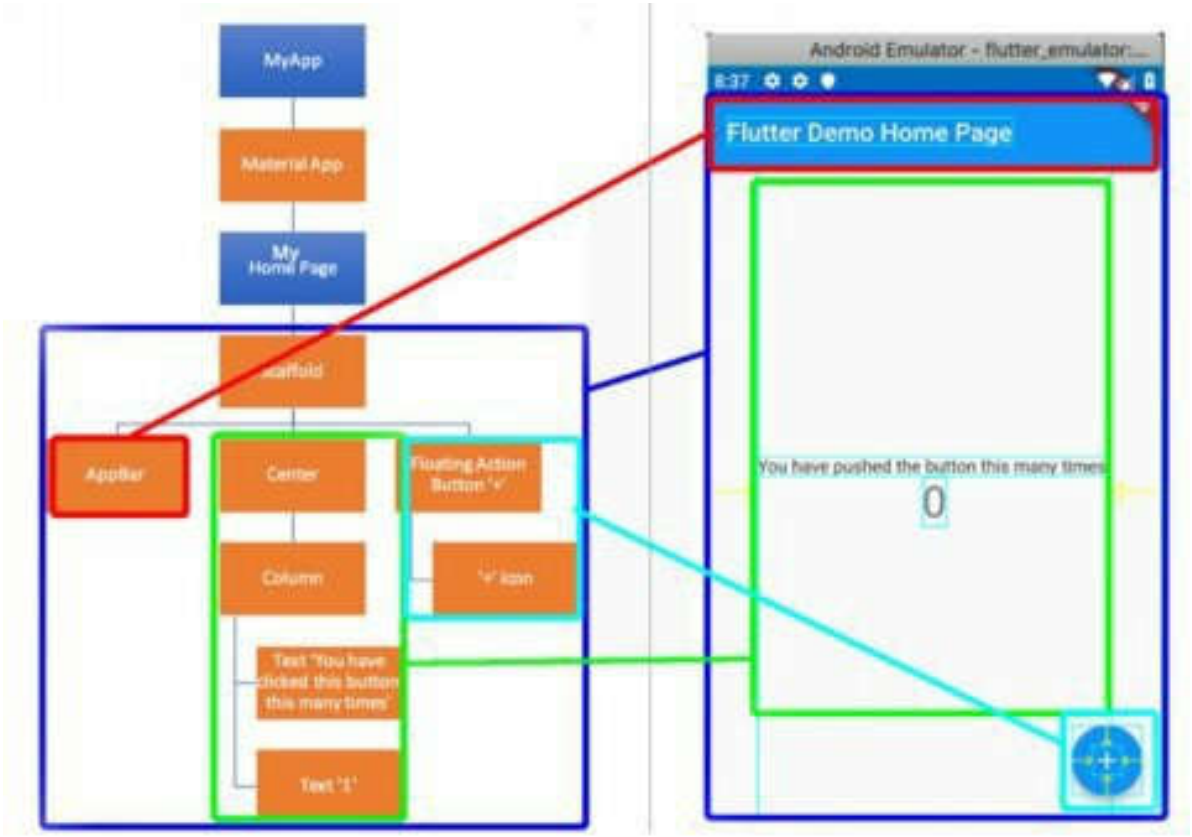



```
@override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Column(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          crossAxisAlignment: CrossAxisAlignment.center,
          children: <Widget>[
            Text("Do you want to\nbuy this item?",
                textAlign: TextAlign.center,
                style: TextStyle(
                    color: Colors.white,
                    fontSize: 40.0,
```

```
            fontWeight: FontWeight.w200)),
        Row(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Spacer(flex: 2),
            SelectButton(text: "YES", onTap: _yesOnTap),
            Spacer(),
            SelectButton(text: "NO", onTap: _noOnTap),
            Spacer(flex: 2),
          ],
        )
      ]),
  );
}
```

Widget Tree

If you look at the next diagram, the project's Widget tree is on the left and the project's UI is on the right. Note how the Widget Tree and UI correspond to each other.



Custom Widgets

Although the default application contains many widgets, only two custom widgets were required to make it work:

MyApp

It is a custom widget for the entire application.

MyHomePage

It is a custom widget that contains the layout for the application, plus the application state (the counter). It

covers the main area of the screen.

Example Code

You will see this Widget used in most of the Examples. This Widget is probably the best place to start when examining Example Code

Flutter Widgets

The rest of the widgets were used from the Flutter widget library, already built for us.

Project Code

Now we have some kind of idea of how the Widgets compose the UI in this app, now let's look at the code.

Entry Point

Every Dart app must start with a main function as a starting point. In this case the main function creates an instance of the MyApp object, a StatelessWidget. The method 'runApp' accepts an instance of a widget (in this case an instance of MyApp) and uses it as the root Widget of the App, rendering it to fit the screen, taking up all the available space.

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

MyApp Widget

The MyApp object is a StatelessWidget. It sets up a Material App that contains a MyHomePage widget. The MaterialApp widget is a built-in Flutter widget that serves as the container for your whole app and its Widgets. It provides services that child Widgets may use, such as navigation, sizing, themes etc.

```
class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Flutter Demo',
      theme: new ThemeData(
        // This is the theme of your application.
        //
        // Try running your application with "flutter run". You'll see the
        // application has a blue toolbar. Then, without quitting the app, try
        // changing the primarySwatch below to Colors.green and then invoke
        // "hot reload" (press "r" in the console where you ran "flutter run",
        // or press Run > Flutter Hot Reload in IntelliJ). Notice that the
        // counter didn't reset back to zero; the application is not
```

```
restarted.
        primarySwatch: Colors.blue,
      ),
      home: new MyHomePage(title: 'Flutter Demo Home
Page'),
    );
  }
}
```

MyHomePage Widget

This is a stateful widget, more on these later. This widget holds the count as State (data) and it sets up the child objects in the UI:

- The center widget, which contains a column object, which contains 2 text objects:
    - ‘You have pushed the button this many times:’
    - ‘0’
- The floating action button widget, which contains a ‘+’ icon.
    - When the user clicks on the floating action button, this increments the instance variable ‘_counter’ inside a the ‘setState’ method.
    - Making a call to the ‘setState’ method

tells Flutter that something has changed and the UI needs to be rebuilt, so it invokes the ‘build’ method in this widget, which redraws itself with the new counter value.

```
class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);

  // This widget is the home page of your application. It is stateful, meaning
  // that it has a State object (defined below) that contains fields that affect
  // how it looks.

  // This class is the configuration for the state. It holds the values (in this
  // case the title) provided by the parent (in this case the App widget) and
  // used by the build method of the State. Fields in a Widget subclass are
  // always marked "final".

  final String title;

  @override
  _MyHomePageState createState() => new _MyHomePageState();
}
```

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

  void _incrementCounter() {
    setState(() {
      // This call to setState tells the Flutter framework that something has
      // changed in this State, which causes it to rerun the build method below
      // so that the display can reflect the updated values. If we changed
      // _counter without calling setState(), then the build method would not be
      // called again, and so nothing would appear to happen.
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    // This method is rerun every time setState is called, for instance as done
    // by the _incrementCounter method above.
    //
    // The Flutter framework has been optimized to make rerunning build methods
    // fast, so that you can just rebuild anything that needs updating rather
    // than having to individually change instances of widgets.
```

```
    return new Scaffold(
      appBar: new AppBar(
        // Here we take the value from the MyHomePage object that was created by
        // the App.build method, and use it to set our appbar title.
        title: new Text(widget.title),
      ),
      body: new Center(
        // Center is a layout widget. It takes a single child and positions it
        // in the middle of the parent.
        child: new Column(
          // Column is also layout widget. It takes a list of children and
          // arranges them vertically. By default, it sizes itself to fit its
          // children horizontally, and tries to be as tall as its parent.
          //
          // Invoke "debug paint" (press "p" in the console where you ran
          // "flutter run", or select "Toggle Debug Paint" from the Flutter tool
          // window in IntelliJ) to see the wireframe for each widget.
          //
          // Column has various properties to control how it sizes itself and
```

```
        // how it positions its children. Here we use
mainAxisAlignment to
        // center the children vertically; the main axis here is
the vertical
        // axis because Columns are vertical (the cross axis
would be
        // horizontal).
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          new Text(
            'You have pushed the button this many times:',
          ),
          new Text(
            '$_counter',
            style: Theme.of(context).textTheme.display1,
          ),
        ],
      ),
    ),
    floatingActionButton: new FloatingActionButton(
      onPressed: _incrementCounter,
      tooltip: 'Increment',
      child: new Icon(Icons.add),
    ), // This trailing comma makes auto-formatting nicer for
build methods.
  );
 }
```

}

10. Setup Android Emulator & Run Project

Introduction

The purpose of this chapter is to setup an Android emulator, open it from your editor and run the project. Ensure that you have your newly-created project open in your editor before continuing.

Emulators

These are great for developers, enabling them to develop their code to run on multiple devices, see how they look on each device. Later on, you can use the real hardware for final pre-release testing.

You can only run your Flutter code on emulators in Debug Mode. Release Mode is not supported. This is not a big problem because you should be testing release code on the real hardware anyway!

Android Emulator

The Android Emulator simulates Android devices on your computer so that you can test your application on a variety of devices and Android API levels without needing to have each physical device.

The emulator provides almost all of the capabilities of a real Android device and it comes with predefined configurations for various Android phone, tablet, Wear OS, and Android TV devices. An AVD is a virtual device that you setup to run in the Emulator.

Setting Up the Android Emulator

Options

There are multiple ways you can setup the Android emulator:

- Use Android Studio.
  - The AVD Manager in Android Studio gives you the most control, it allows you to setup Android Virtual Devices for all kinds of hardware and versions of Android.
- Use Visual Studio Code.
  - If you just want a generic emulator, Visual Studio Code lets you set one up very easily.
- Use Command Line.
  - I would not setup the emulators from the command-line as it is far more difficult.

Mix

You can mix-and-match. You can setup your emulator using one editor but edit your code in another.

Example

You can create your emulators using the AVD Manager in Android Studio (or Visual Studio Code) then control them later from the command line.

Setup Android Emulator in Android Studio

AVD Manager

The AVD Manager in Android Studio is a dialog you can launch from Android Studio that helps you create and manage AVDs.

To open the AVD Manager, do one of the following:

- Select Tools > AVD Manager.
- Click AVD Manager AVD Manager icon in the toolbar.

Using the toolbar is the quickest way to open the AVD Manager:

AVD Manager Dialog

Main Area

The main part of the dialog lists the available AVDs. Note that each AVD has a play button and a pencil button.

Play Button

The play button launches the AVD.

Pencil Button

The pencil button opens a list of commands to do with the AVD to enable the user to perform various tasks:

- Duplicate AVD

- Wipe AVD Data
- Reboot AVD
- Show On Disk
- View Details
- Delete
- Stop (If Launched)

Bottom

The bottom part of the dialog has various buttons, the most important being the button ‘Create Virtual Device’, which allows the user to download the files for and setup an AVD. Downloading the files can take a while but it is much easier than using the command line!

Setup Android Emulator in Visual Studio Code

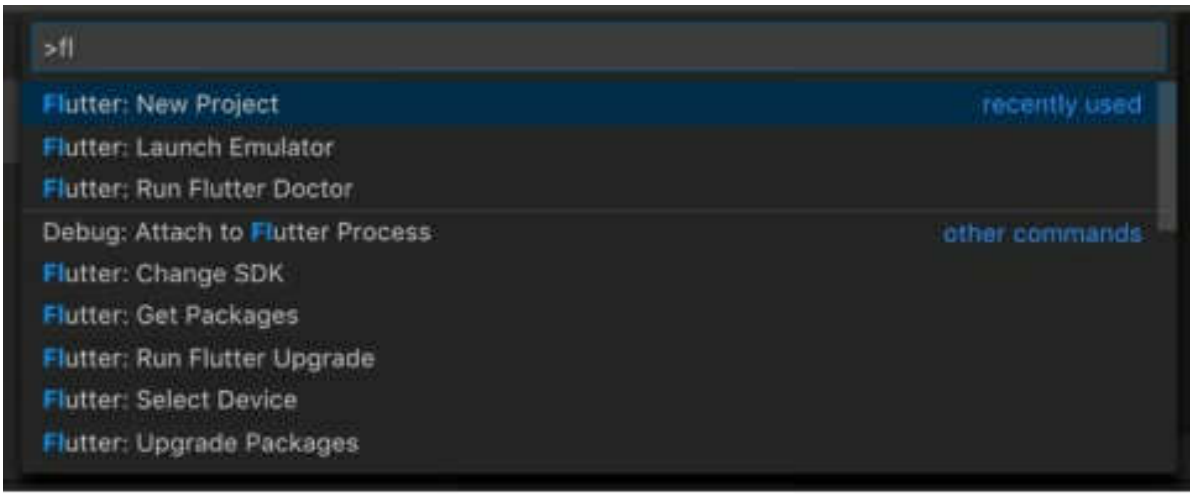
Before You Start

Ensure that you have installed the Flutter Extension into Code before doing this.

View Flutter Commands

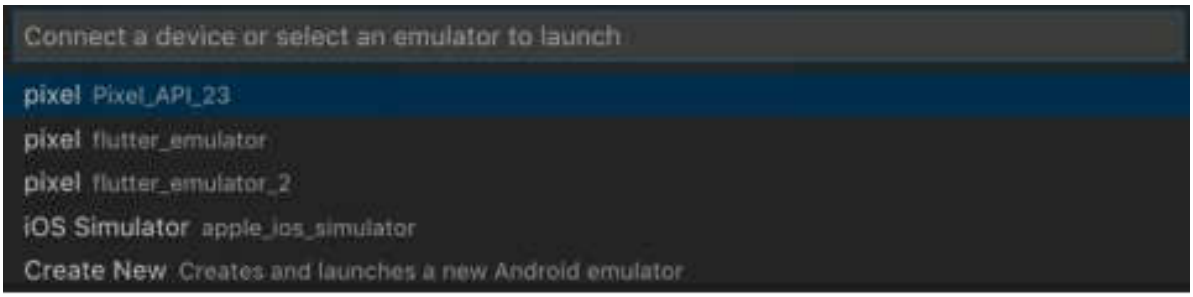
Open the command palette using the keyboard shortcut Ctrl+Shift+P (Command+Shift+P on the Mac) and you

will see a list of the available commands. If you start to type ‘Flutter’ in this box, then you will see a list of Flutter commands:



Launch Emulator

If you select the ‘Flutter: Launch Emulator’ command, you will list the installed emulators and you can select one to launch it.



Create New Emulator

If you don’t have an emulator installed, select the ‘Create New’ command and Visual Studio Code will

create a generic emulator called ‘flutter_emulator’, which you can then launch.

Setup Android Emulator in Command Line

Introduction

There is a lot of information here: https://developer.android.com/studio/run/emulator-commandline

Android SDK Path

Remember that these command-line tools are part of the Android SDK and need to be setup on your path. Your path should include the following Android SDK folders for these commands to work:

```
/Android/sdk/tools
/Android/sdk/platform-tools
```

To Create a New AVD to Test On:

```
android create avd -n <name> -t <targetID>
```

<name> is the name
<targetID> is the required API level

To List Your Available AVDs:

When you use this option, it displays a list of AVD names from your Android home directory. Note that you can override the default home directory by setting the ANDROID_SDK_HOME environment variable: the root of the user-specific directory where all configuration and AVD content is stored.

```
emulator -list-avds
```

To Launch an Available AVD:

Use the emulator command to start the emulator, as an alternative to running your project or starting it through the AVD Manager.

```
emulator -avd <name>
```

Run Project

Android Studio

Note that on the main toolbar there are two dropdowns then a play button. The first dropdown is used to select the device/emulator to run against. The second dropdown is the run configuration.

Run Configuration

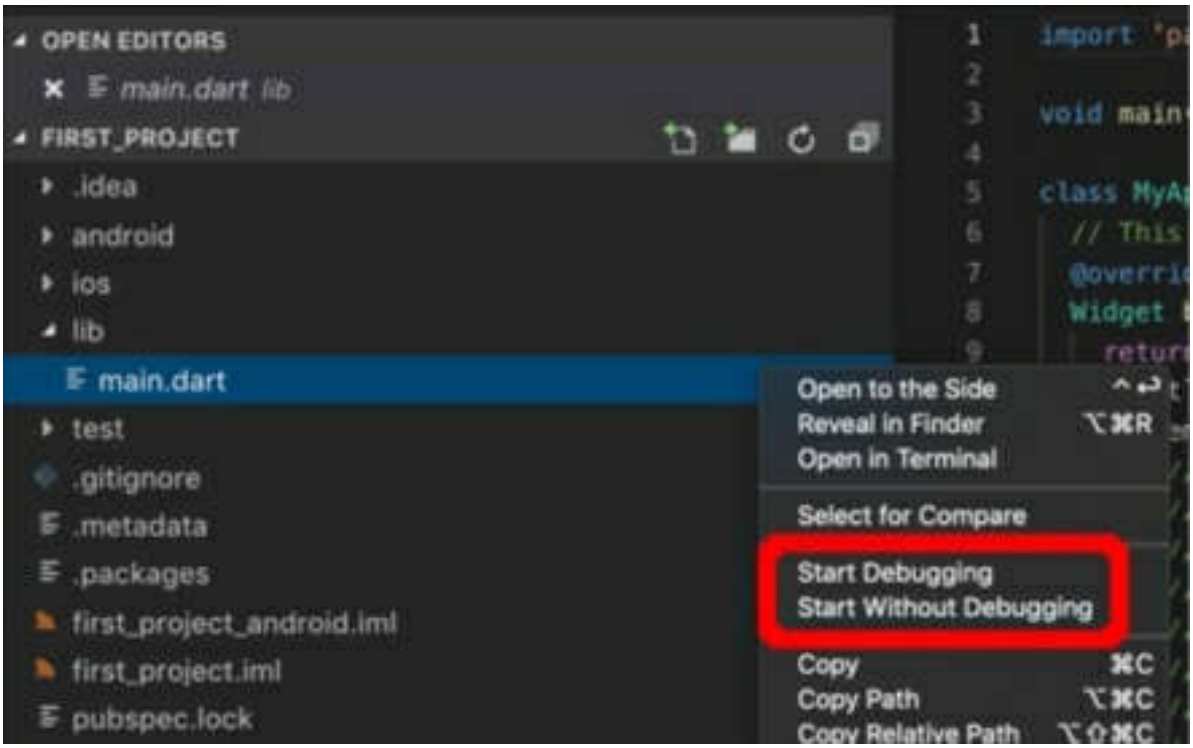
The run configuration is used to provide information

about which Dart class is used as the application starting point, as well as run parameters, options. Clicking on the play button invokes the run configuration on the device/emulator.

Visual Studio Code

Steps

1. Go back to the editor and view the file list (explorer) on the left.
2. Click on the ‘lib’ folder to open it up.
3. Right-mouse click on ‘main.dart’ and you should see a popup menu containing the commands to run or debug. Select run or debug to install and run the app on the emulator. This will run the project on the emulator.



Command-Line

Steps

1. Ensure that you are in the root folder of the project.
2. Enter the ‘flutter run’ command.

```
flutter run
```

11. Setup iOS Emulator & Run Project

Introduction

The purpose of this chapter is to setup an iOS emulator, open it from your editor and run the project. Ensure that you have your newly-created project open in your editor before continuing.

PCs

Obviously, this is not going to work on a PC. So, get your Mac ready (or virtual Mac ready)! You can use the iOS emulator for most of your development and then find a device to test on when you're nearly done.

XCode

XCode is Apple's integrated development environment (IDE) that you use to build apps for Apple products including the iPad, iPhone, Apple Watch, and Mac. XCode provides tools to manage your entire development workflow—from creating your app, to testing, optimizing, and submitting it to the App Store. You don't need to have XCode running to use the Emulator, but you can launch the Emulator from XCode.

iOS Emulator

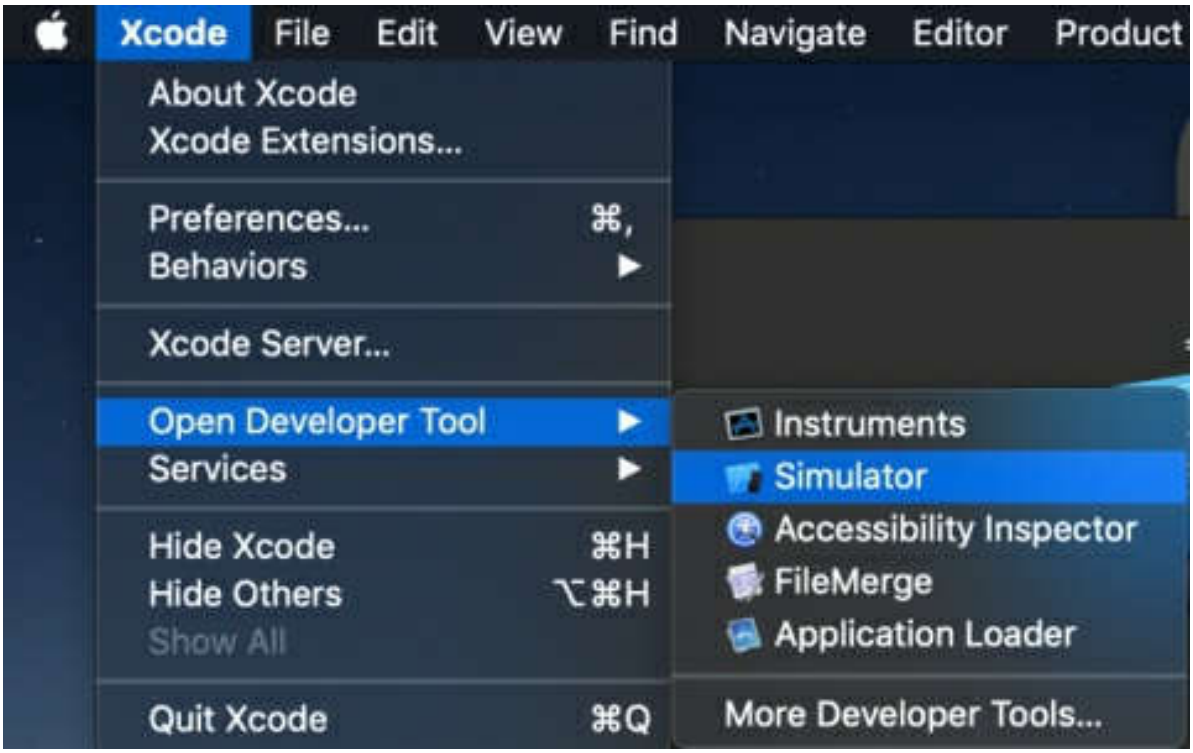
XCode ships with an iOS simulator/emulator.
In addition to running code, the simulator enables you to test the following with virtual iOS devices:

- Device rotation
- Simulating various GPS coordinates
- Device shake
- Simulating low memory scenarios

Open iOS Simulator/Emulator

Open from Xcode

Select the ‘Xcode’ menu then ‘Open Developer Tool then ‘Simulator’:



Opening from Command-Line

Use the following command in your terminal.

```
open -a Simulator
```

Run Project

Android Studio

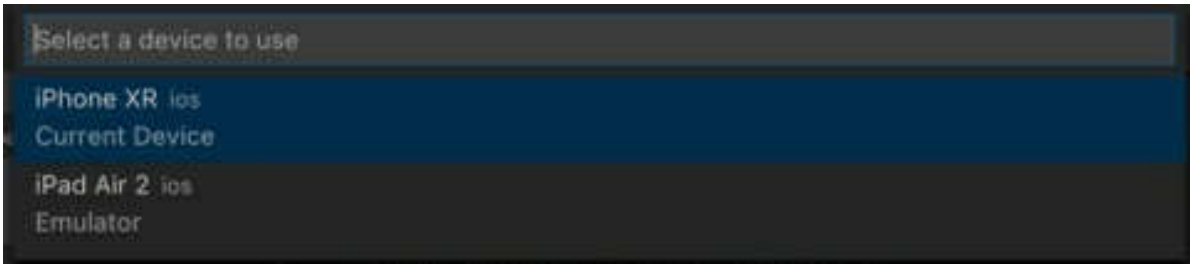
When you have the Simulator(s) open (running), the simulator(s) become available on the main toolbar in the first dropdown, to the left of the run configuration dropdown & run/debug buttons:

If you have more than one simulator running, they will all be displayed in the dropdown on the main toolbar.

Visual Studio Code

When you have the Simulator(s) running, the currently-selected simulator is displayed on the toolbar on the bottom right. The current simulator is the one that will be used when the user selects ‘Start Debugging’ or ‘Start Without Debugging’ on the popup menu for a runnable file.

If you have multiple simulators running, you can select between simulators by clicking on the simulator displayed on the toolbar on the bottom right. This opens a menu at the top to enable the user to select in between them:



Command-Line

You start your iOS simulator(s) as required then you use the ‘flutter run’ command to run your app with an open device or emulator.

No Simulators/Devices Open?

If you invoke ‘flutter run’ without any devices or emulators, you get a message similar to that below:

```
flutter run

No connected devices.


Run 'flutter emulators' to list and start any available device emulators.


If you expected your device to be detected, please run "flutter doctor" to diagnose

potential issues, or visit https://flutter.io/setup/ for troubleshooting tips.
```

Multiple Simulators/Devices Open?

If you invoke ‘flutter run’ with multiple devices or emulators open, you get a message similar to that below:

```
flutter run

More than one device connected; please specify a device with the '-d <deviceId>' flag, or use '-d all' to act on all devices.


iPhone 6s • 34B92793-1355-4E13-857B-D5E7A3FB4F4F • ios
```

```
• iOS 12.1 (simulator)
iPhone XR • D49E45DA-7D58-473A-B0FA-29E3C4E88455 • ios • iOS 12.1 (simulator)
```

The command below runs the app on the iPhone 6s:

```
flutter run -d 34B92793-1355-4E13-857B-D5E7A3FB4F4F
```

One Simulator/Device Open?

Remember that if you only have one device or emulator open, you just do a ‘flutter run’:

```
flutter run
```

12. Setup Device & Run Project

Introduction

The purpose of this chapter is to setup a real device, connect it to your computer and run the project. Ensure that you have your newly-created project open in your editor before continuing.

Android Device

Introduction

This is relatively straightforward. You basically do the following:

- Use the device settings app (on your device i.e. your phone) to set yourself up as a developer and enable USB debugging.
- Setup the computer to detect the device. This involves setting up ADB, which is the Android Debug Bridge. This enables an Android app to be debugged on an emulator or actual Android device.
- Connect the device to the computer.
    - Your connected device should become

visible in the IDE in the same way as it would for an emulator (see ‘Open Android Emulator & Run Your First App’).
    - Run the app in the same way as you would for an emulator.

Further Reading / Instructions

Full instructions here:
https://developer.android.com/studio/run/device

iOS Device

Check Your Hardware First

- Computer.
    - As per the iOS emulator, this is not going to work on a PC. So, get your Mac computer ready (or virtual Mac ready)!
- Device.
    - Your Flutter app won’t just work on any old iOS device. You will need to have an iOS device that is capable of running iOS8 or later. Otherwise you will get an error like this:

> The iOS deployment target is set to 5, but the range of

> supported deployment target versions for this platform is 8.0 to 12.1. (in target 'Runner')

Setup Your XCode Project

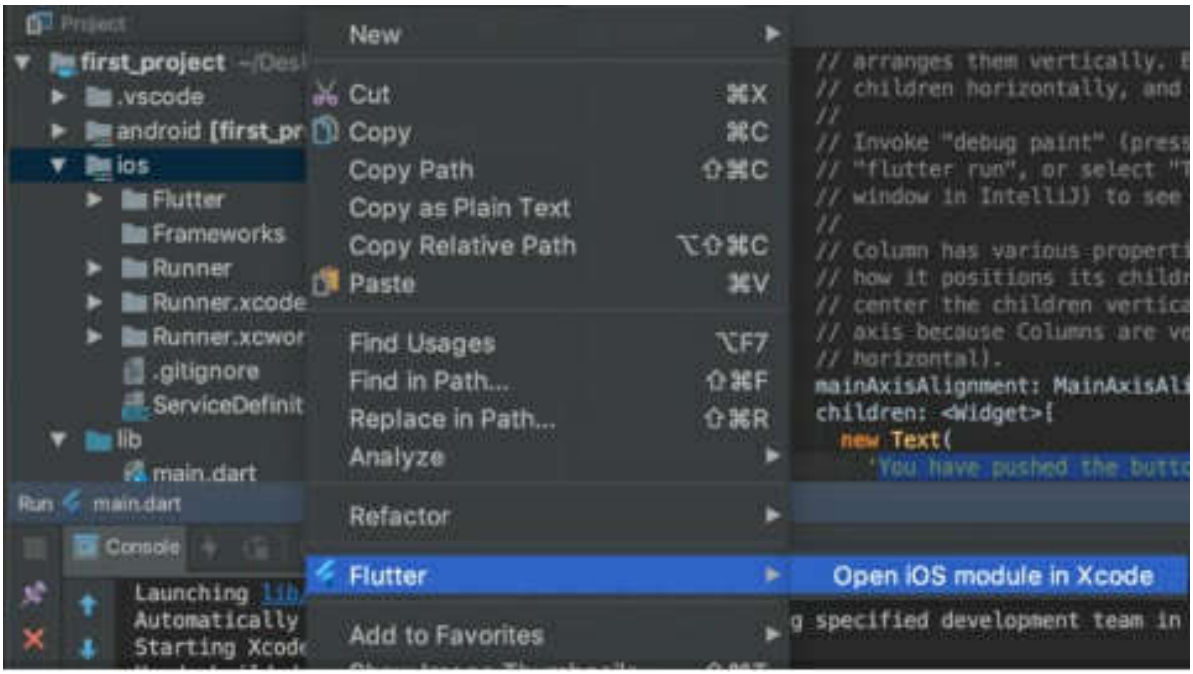
Your XCode project resides within the ‘ios’ folder of your Flutter project.
You will need to open it and set it up:

- Open your XCode project.
- Create a signing team.
- Setup a unique bundle identifier for the project.

Open XCode Project

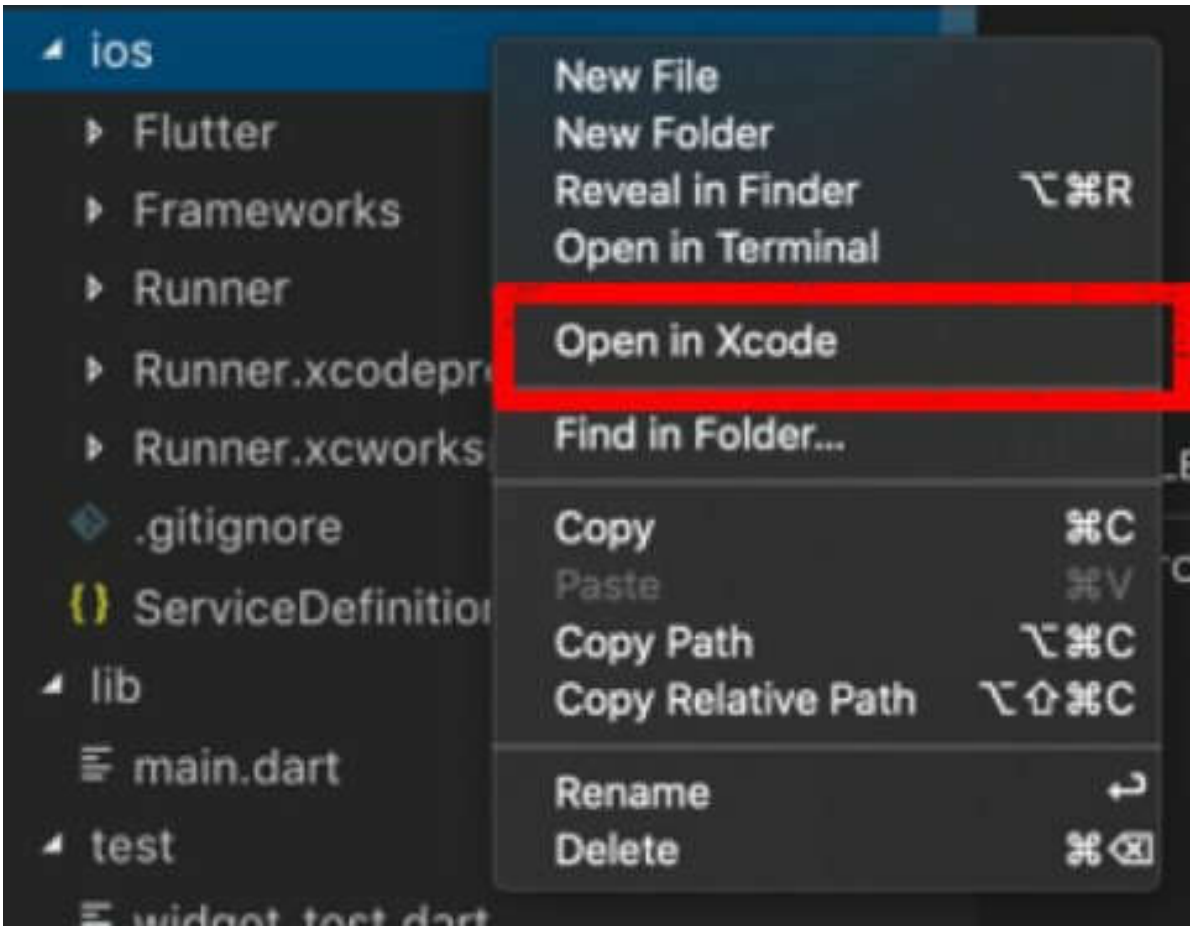
Android Studio

- Right-click on iOS folder in project.
- Select ‘Flutter’ in popup menu.
- Select ‘Open iOS module in XCode’ in popup menu.



Visual Studio Code

- Right-click on iOS folder in project.
- Select ‘Open in XCode’ in popup menu



Command-Line

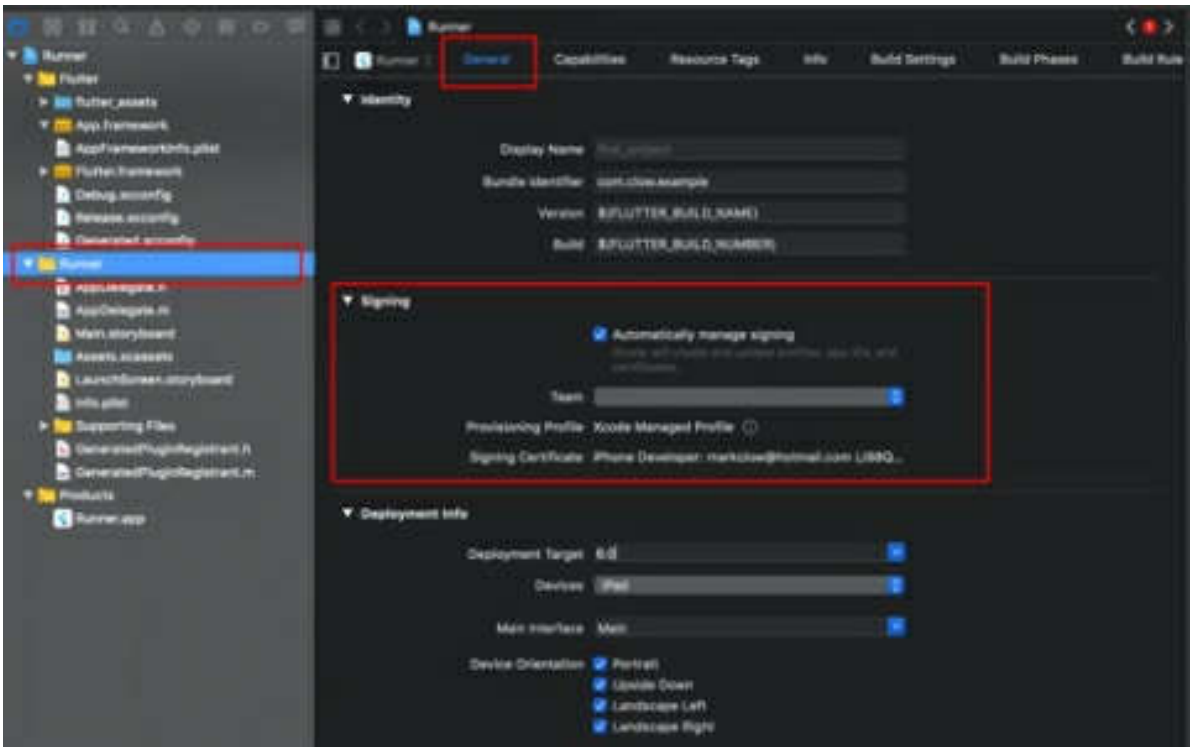
- Open terminal and navigate to the root folder of your project.
- Enter the following command.

```
open ios/Runner.xcworkspace
```

- This command should open the XCode project.

Create Signing Team

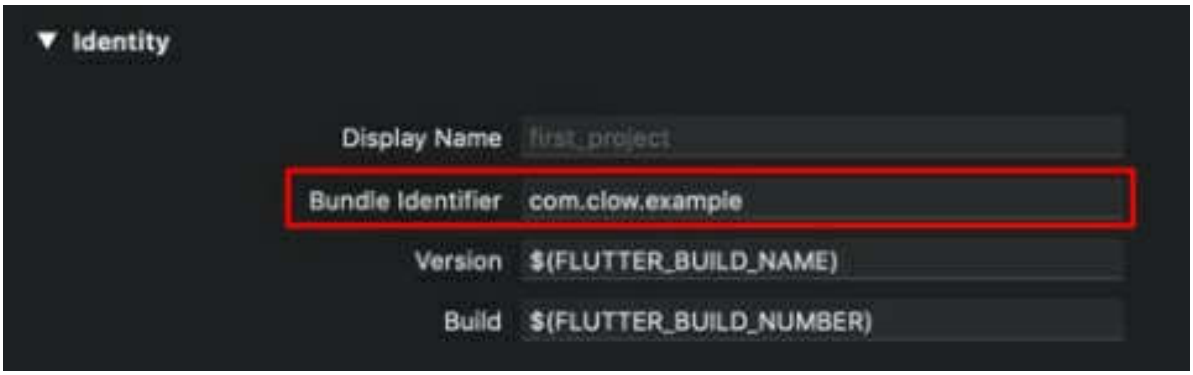
- You will need to login to XCode using your Apple ID and setup a signing team.
- In XCode, navigate to the Runner target settings page, then General > Signing > Team.
- The signing workflow is detailed here: https://help.apple.com/xcode/mac/current/ -/dev60b6fbbc7



Set Bundle Identifier

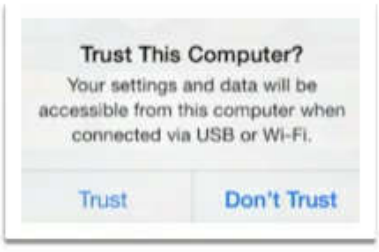
- In XCode, navigate to the Runner target settings page, then General > Identity > Bundle Identifier.
- The Bundle Identifier needs to be unique for your

project and is quite restrictive in terms of characters, so make sure all of the characters are only alphanumeric (A-Z,a-z,0-9), hypen (-), or period (.).



Connect the Device to The Mac & Run

- The first time you connect, you will need to trust both your Mac and the Development Certificate on that device. Select Trust in the dialog prompt (on the iOS device) when first connecting the iOS device to your Mac.



- Your connected device should become visible in

the IDE in the same way as it would for an emulator (see ‘Run Your App on the iOS Simulator’).

- If you see the ‘Untrusted Developer’ error on the iOS device, then go to Settings and search for Device Management. You can then trust the developer there.
- Run the app in the same way as you would for an emulator.

Further Reading / Instructions

Full instructions here: https://flutter.io/docs/get-started/install/macos

13. Hot Restarting & Reloading

Introduction

This is a very short chapter, but it contains valuable information that you will use all the time. When are running a Flutter app and you make code changes, you can tell your editor to reload them. That is the subject of this chapter.

Hot Reloads

In fact, one of the great things about Dart is its ability to hot reload code.

The Official Documentation Says:

Flutter’s hot reload feature helps you quickly and easily experiment, build UIs, add features, and fix bugs. Hot reload works by injecting updated source code files into the running Dart Virtual Machine (VM). After the VM updates classes with the new versions of fields and functions, the Flutter framework automatically rebuilds the widget tree, allowing you to quickly view the effects of your changes.

Two Options

After you have made your code changes, you have two

options in regard to reloading:

- Hot restarting.
- Hot reloading.

Hot Restarting

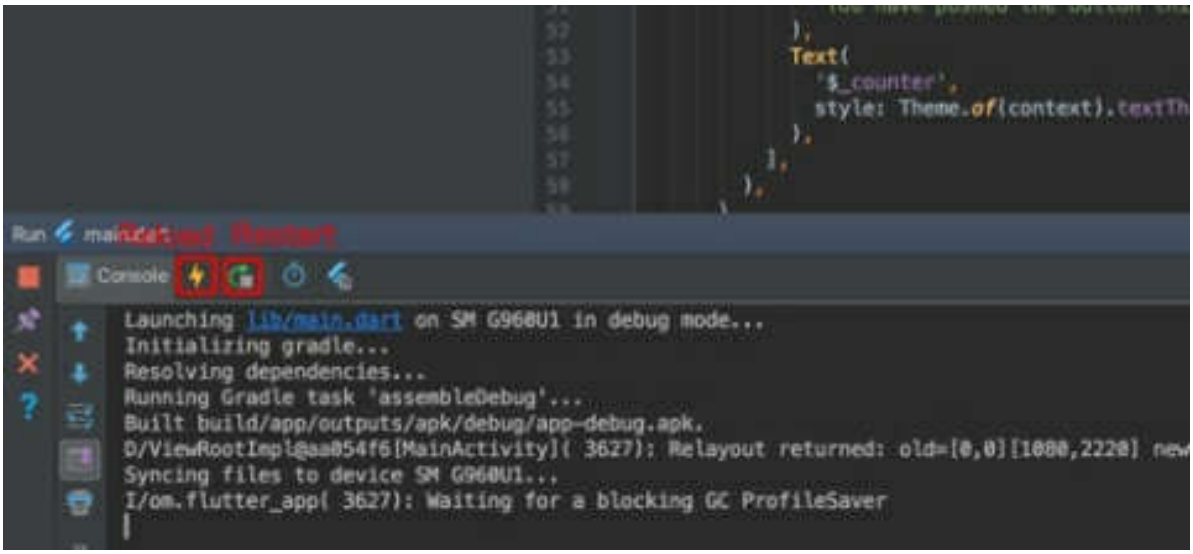
This loads your changed code into the Dart VM and restarts the application. This is the safest thing to do and doesn’t take long.

Hot Reloading

If you want to load your changed code into the Dart VM but you don’t want to restart the application or change its state, you can do this. The result might be different behavior vs a hot restart. Just remember your code changes may not work with the existing state.

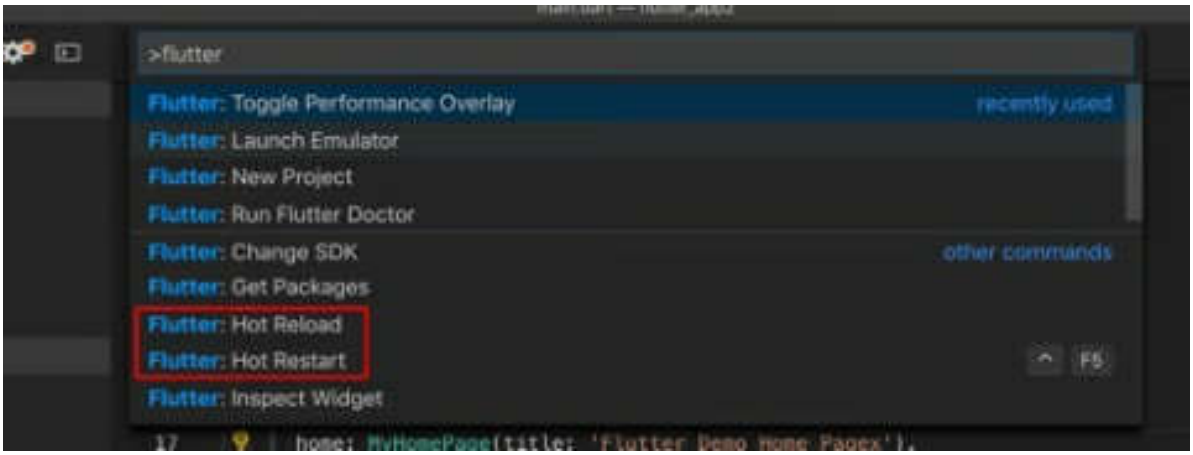
Android Studio

Both hot restart and hot reload are available in the run/debug tool windows.



Visual Studio Code

Both hot restart and hot reload are available in the Command Palette.



Command-Line

If you are using ‘flutter’ run to run the app from the command line, you can use the key ‘R’ to hot restart and the key ‘r’ to hot reload.

14. Dependencies & Packages

Introduction

In Dart, you don’t have to develop everything from scratch. There is a packaging system where developers can develop packages and publish them. Other people can then use these packages.

The purpose of this chapter is to outline how to use this packaging system.

Website

When someone writes a package and it is published to the https://pub.dartlang.org/ site, developers can declare a dependency to that project and pull it into their project as a dependency. Then the user can add imports at the top the files to import code and use it.

Note that Dart and Flutter packages follow semantic versioning rules.

Core Packages

Flutter comes with many packages by default. These are called Core Packages and you don’t need to declare any kind of external dependency to use them.

Non-Core Packages

You could call these ‘External Packages’. These are packages that are not setup by default. You need to declare these dependencies and pull them into your project to use them.

Most Useful Non-Core Packages

These are the packages that I have used the most. This may be very different for other Flutter developers.

Name	Description
http	For HTTP communication.
rxdart	Reactive functional programming library.
datetime_picker_formfield	Date / time picker.
image_picker	Image picker. Very useful apps where you take pictures or upload photos.
zoomable_image	For panning and zooming images by touch
shared_preferences	For saving local settings and data in

	your app.
cached_network_image	A flutter library to show images from the internet and keep them in the cache directory. This helps speed things up. It also lets you display an image placeholder while the image loads.

How to Use an External Package

Declare Dependency in Project

- Open the pubspec.yaml file in the root of your project and add a dependency. For example, the code below declares dependencies to the flutter sdk, cupertino icons and scoped_model. Note how some dependencies specify the version, some don't:

```
flutter:
  sdk: flutter
cupertino_icons: ^0.1.2
scoped_model: ^1.0.1
```

Import Packages

- Once your pubspec.yaml file is setup, you need to install the packages by pulling them from https://pub.dartlang.org/ . Normally your editor will assist you with this.
    - Android Studio
        - Click ‘Packages Get’ in the action ribbon at the top of pubspec.yaml
    - Visual Studio Code
        - Click ‘Get Packages’ located in right side of the action ribbon at the top of pubspec.yaml
    - Command-Line
        - Run the command ‘flutter packages get’.

Import & Use Package Code

You import the package code in the usual manner using the ‘import’ statement at the top of your code. For example, the code imports the flutter material package and the scoped model package.

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

Restart Your App

You will probably need to restart your app if it is running.
That's it!

Package Version Numbers

Some dependencies specify the version, some don't.

- Version specifiers:
  - 'any' – any version
  - '1.2.3' – only version 1.2.3
  - '>1.8.3' – any version higher than 1.8.3
  - '>=1.8.3' – any version 1.8.3 or higher
  - '<1.8.3' – any version lower than 1.8.3
  - '<=1.8.3' – any version 1.8.3 or lower
- Carat syntax.
  - The '^' means - "the range of all versions guaranteed to be backwards compatible with the specified version".
  - '^1.1.1' is equivalent to versions '>=1.1.1 <2.0.0'
  - '^0.1.2' is equivalent to versions '>=0.1.2 <0.2.0'

Project Files

.packages

This file gets generated when you do a ‘packages get’. This file contains a list of dependencies used by your application.

pubspec.lock

Also known as ‘package lock file’.
The first time you get a new dependency for your package, pub downloads the latest version of it that’s compatible with your other dependencies. It then locks your package to always use that version by creating a lockfile. This is a file named pubspec.lock that pub creates and stores next to your pubspec. It lists the specific versions of each dependency (immediate and transitive) that your package uses.

How to Clean & Reimport Your Packages

You may find the sequence of commands below useful if you encounter strange dependency issues in your project:

```
flutter clean
rm -rf pubspec.lock .packages .flutter-plugins
flutter pub pub cache repair
flutter packages get
```

How to Publish Your Own Packages

Introduction

You can easily write your own packages and share them with the rest of the world.
You can publish to kinds of packages:

- Dart Packages
    - These are packages written in dart.
    - Some of these packages are designed for dart only, others are designed for flutter.
    - We are going to cover these.

- Plugin Packages
    - These are packages written in dart that include platform-specific code, for example Android-specific or iOS-specific.
    - These are beyond the scope of the book.

Setting Up a Dart Package

The Flutter SDK has a command line tool that enables you to quickly setup a dart package:

```
flutter create --template=package <name>
```

This doesn't create a large project, in fact it creates a project with two files:

Folder	Description
[root]	Root folder. Contains pubspec.yaml file, readme file.
android	As the name suggests, the folder contains all the Android-related files and code(s) for the package. This is where Android-specific settings and code resides. When building for Android, Flutter uses Gradle as the dependency manager.
ios	Similar to the ‘android’ folder, this folder contains the iOS related files and code(s) for the package. This is where iOS-specific settings and generated code resides. When building for iOS, Flutter uses Cocoapods as the dependency manager.
lib	This is where the application code resides. You should see a file ‘main.dart’, the entry point for the Flutter application. This is the file you select and run. You will add more files and subfolders into this folder.
test	This is where the unit testing code resides. You may add more files and subfolders

	into this folder.

Now you need to implement the code in your package, including writing unit tests.
Once you have completed code implementation, you need to add documentation.

Adding Documentation

- Add text to the README.md file.
    - This is the first place developers will look.
- Every time you make a change to the package, add text to the CHANGELOG.md file.
- Use the dart documentation tool to generate api documentation.
    - Change directory to the location of your package:

```
cd ~/dev/mypackage
```

    - Add an environment variable to tell the Tell the documentation tool where the Flutter SDK is (change to reflect where you placed it):
        - Mac/Unix

```
export FLUTTER_ROOT=~/dev/flutter
```

        - Windows

```
set FLUTTER_ROOT=~/dev/flutter (on Windows)
```

- Run the dartdoc tool (comes as part of the Flutter SDK):
  - Mac/Unix

```
$FLUTTER_ROOT/bin/cache/dart-sdk/bin/dartdoc
```

  - Windows

```
%FLUTTER_ROOT%\bin\cache\dart-sdk\bin\dartdoc
```

Final Review

Review the publishing specification file pubspec.yaml.
Review the documentation, make sure it’s all ready.

Do a Publish Dry-Run

This is good preparation for the real thing.
Running this command will check all the publishing pre-requisites without actually publishing.

```
flutter packages pub publish --dry-run
```

Publish

If everything went well in the publishing dry-run then do the actual publishing.

```
flutter packages pub publish
```

Further Reading

A lot of this information for this chapter came from here:
https://flutter.io/docs/development/packages-and-plugins/developing-packages