FLUTTER: THE GUIDE FOR GOING FROM ZERO TO PRO

Daniele Cambi

Flutter: The guide for going from zero to pro

Who I am

Hi, I'm Daniele Cambi, dancamdev on pretty much every major social network, I started as a native Android developer and then switched to Flutter once I realized how powerful it was and the benefits it brought. I have been working on Flutter since the beta in 2018 and I've build many apps, both for my businesses and for clients.

I never looked back to native development, and honestly, I don't miss it. I'm currently working actively on two companies using Flutter:

App and Up, the first company I co-founded, for which I develop Moodify;

More recently, I teamed up with Edoardo to grow the Dancamdev business and work on products specifically tailored to what my audience needs and asks me.

The one I wrote is a complete Flutter roadmap, that will get you from zero to being able of working on full fledged Flutter projects and as a reference if you get stuck.

The one Edoardo wrote, will guide you through the business aspects behind an app, that are often underrated by devs, but that are crucial and just as important as the technical side of things, for turning your app into a business.

Finally, we both document our job, what we learn and our progress on instagram, so if you haven't already, make sure you follow us both there too. I'm @dancamdev, and he's @edoardomartelli23.

Why I'm writing this guide

I created this guide to help you getting started with Flutter and working your way towards making high quality apps.

Through the months I've been on instagram and social media, I received countless messages asking "From where should I start to learn flutter?". Well, that's my attempt to give you a great start to learning and becoming an expert in my framework of choice when it comes cross platform apps, hoping that it will become yours too.

With the 2.0 release flutter now supports web, windows, macOS and Linux too, I'm personally still experimenting with those, but the cool things is that the differences between building a mobile app or building an app for any other platform are really, really minor.

Before we begin

I'm giving for granted that if you're here you know what flutter is about and you want to put in the effort required to learn it.

If you happened to get this guide just out of curiosity and you don't know what Flutter really is. Here's a quick link to the Flutter home page, so you can learn more!

https://flutter.dev/

The purpose of this ebook is not to teach you every aspect of flutter development. Its purpose is to guide you through all the topics you need to learn in a structured way, making sure you don't get lost in the many topics and resources available on the web.

I don't want to digress, therefore, on unneeded introductions. Let's get started with the roadmap!

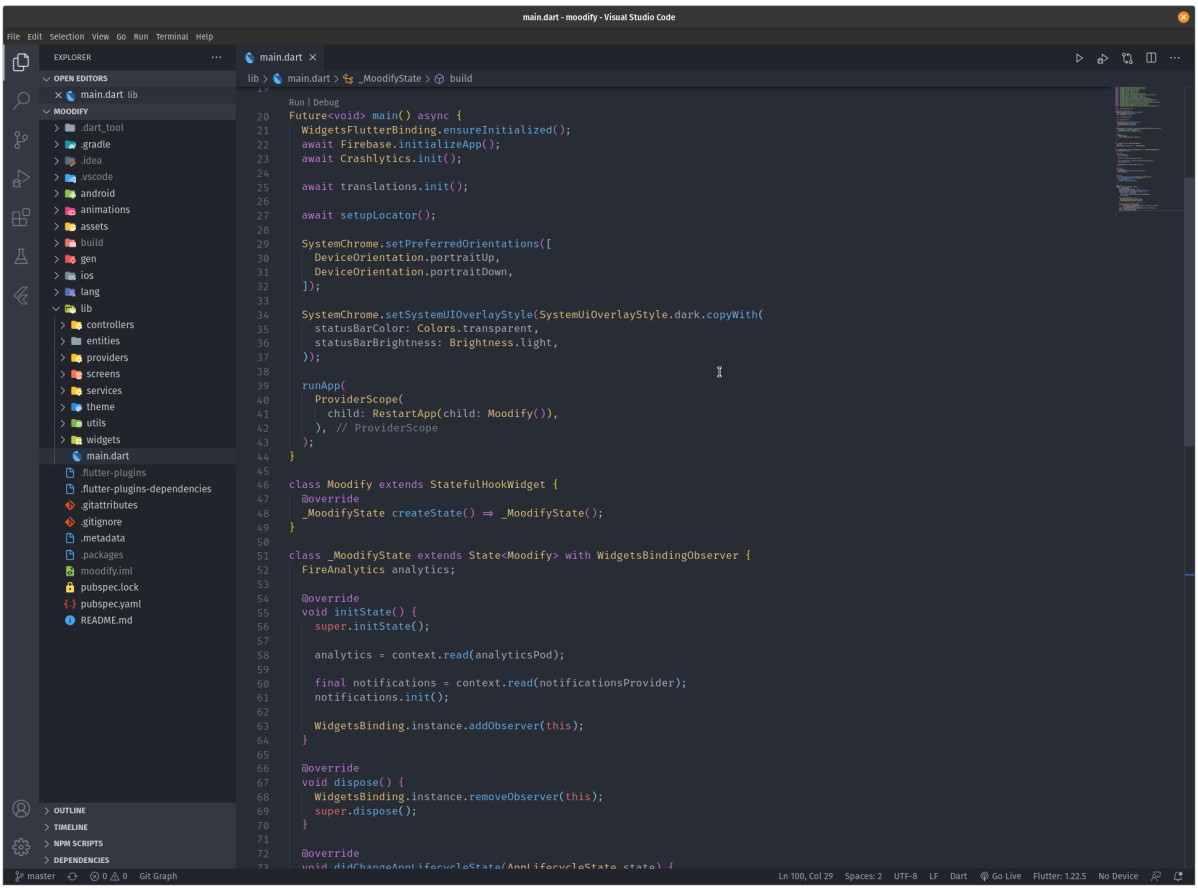
Chapter 0: Prerequisites

In order to get started writing flutter apps you'll need, of course, to install it and have an IDE (Integrated development environment) in which to write your Dart code.

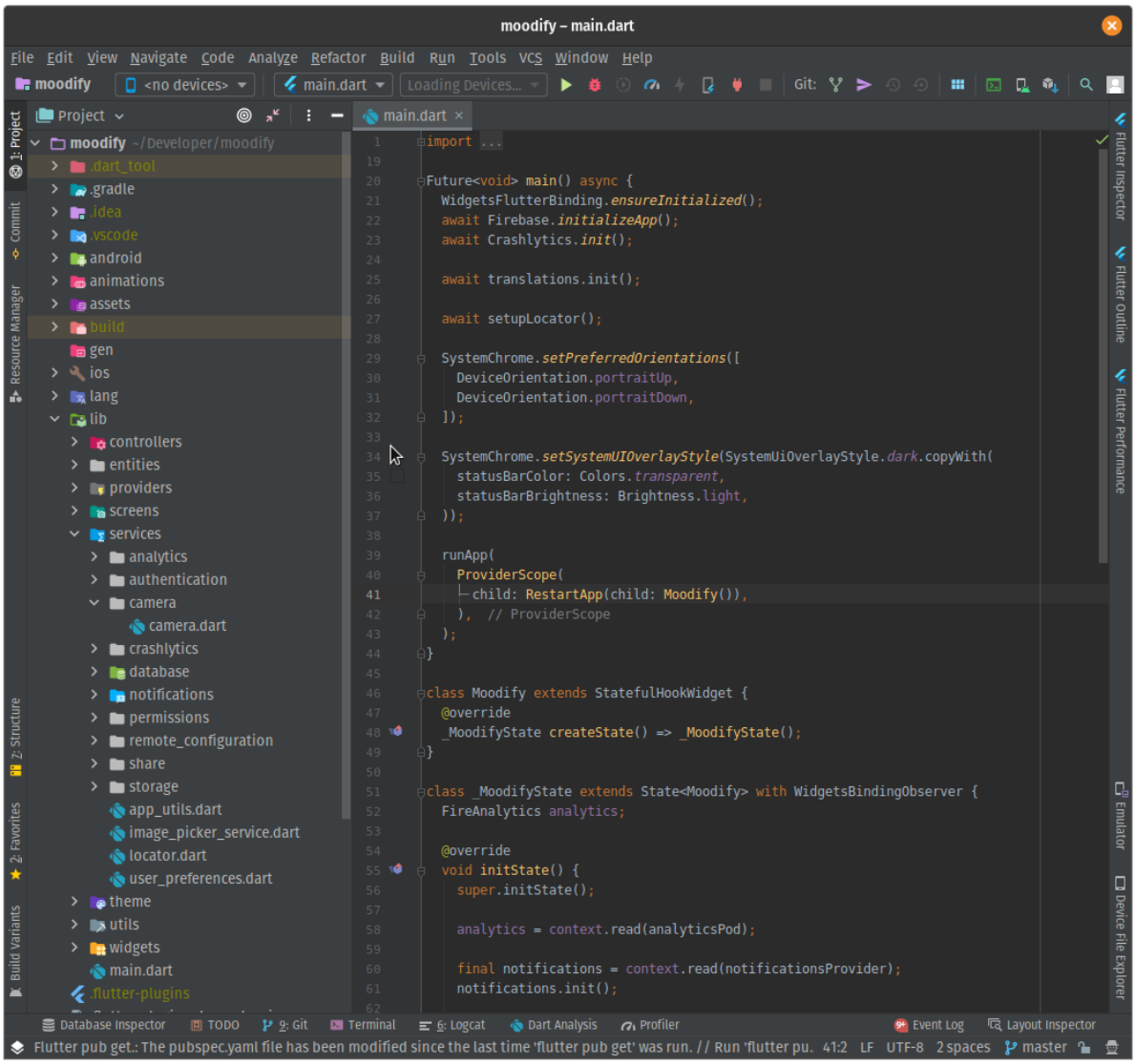
Choosing the IDE

Flutter officially supports three different IDEs.

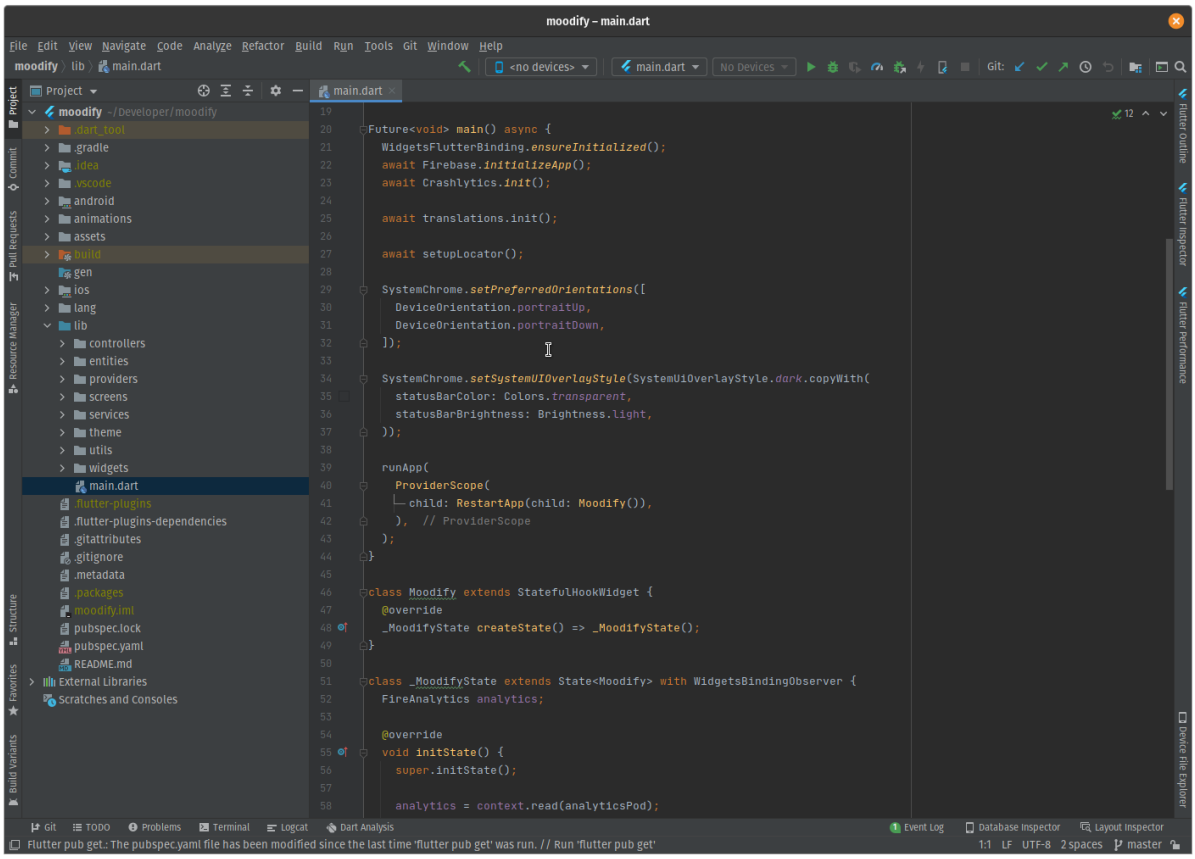
- Visual Studio Code - https://code.visualstudio.com/



- Android Studio - https://developer.android.com/studio



- Intellij IDEA - https://www.jetbrains.com/idea/



Which one you choose doesn't matter too much, all three do the same things. Visual Studio Code is my IDE of choice and it's a bit more lightweight than the other two. Android Studio is based on Intellij, that's why they are so similar.

> **Note**: For Visual Studio Code to work with Flutter you'll need to install the Dart and Flutter extension.

> **Note**: For Android Studio and Intellij IDEA you'll need to install the Flutter Plugin.

Install Flutter

Full installation instructions are available on the official website:

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

If you want to build flutter apps on Android and iOS, you'll need to install the corresponding SDKs, as well as the Android emulator and iOS emulator. The installation guide above explains how to do this step by step.

> Note: You can only develop Flutter apps for iOS on a macOS system. If you are on windows you can only build apps for Android. There are services like macincloud to build iOS apps remotely.

Once you're done with the installation process, make sure you run `flutter doctor` in a terminal to make sure that everything is installed and set up correctly!

Chapter Bonus: Visual Studio Code extensions for professional flutter development

If you chose to go with Visual Studio Code, just as I do, I thought I'd share with you my favorite vs code extension I'd recommend any flutter developer to install.

- Atom Dark One - This is my theme of choice. It looks so good!
- Material Design Icons - My icons of choice. They give vscode a more colored and professional look.
- Bracket Pair Colorizer 2
- Error Lens
- Pubspec Assist
- Git graph
- Todo tree

Chapter 1: The Dart programming language

Flutter is a framework, which means you can't formally write flutter code. The programming language that flutter uses is called Dart.

It is actually a fairly easy language to pick up and there are plenty of resources to learn from.

The Dart language tour, for example, is the official introduction, and it's perfect if you're already familiar with JavaScript, Kotlin, or Swift.

Another great way, more practical, to learn the Dart language is through the Codelabs

Another guide, which is definitely worth a read and to be kept as a reference. is the effective Dart guide, It is a very very broad and thorough guide on writing clean and effective Dart code. You should read it once you have understood the basics of Dart.

Finally, a quick way of writing and playing with Dart code it's DartPad, a free online editor that will allow you to experiment with Dart, without installing anything on your device.

Null safety in Dart

If you know Kotlin or Swift you probably know what null safety is. In Flutter 2.0 and Dart 2.12, Sound null safety has been introduced.

Here's the docs to learn null safety in Dart

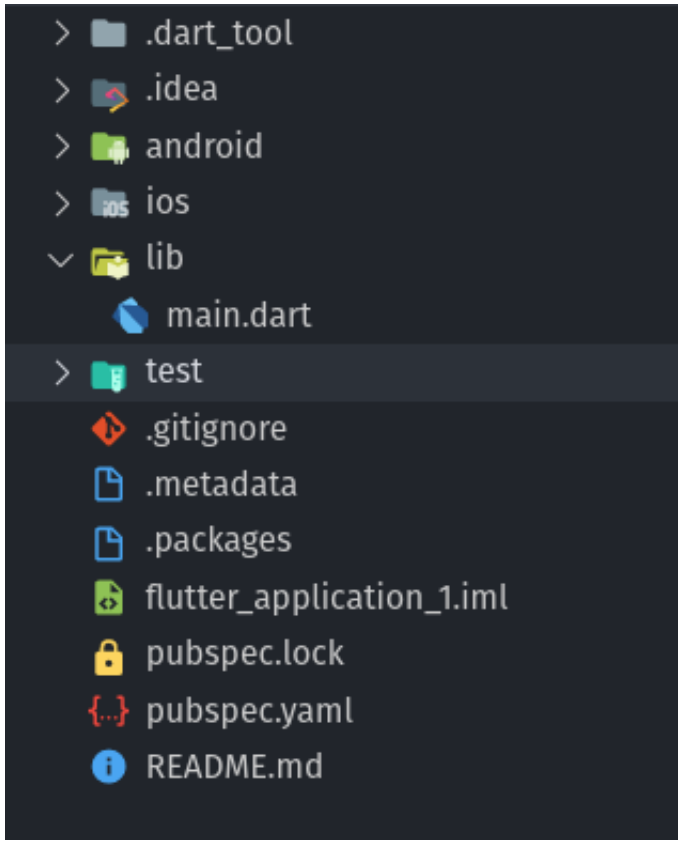
While if you happen to already have some Dart code here is the migration guide

I know, it is a lot of content! But, you don't need to learn everything and have everything in mind. Keep these links and guides as useful resources to get back to when in need!

Chapter 2: Anatomy of a new Flutter Project

You know by now how to create a flutter project. But have you wondered what all those files it generates for you are? Well, let's break it down.

Your newly created flutter project should look something like this:



The most important file you see here is the `pubspec.yaml`. You may wonder why, it's not a dart file, and that's true, but the pubspec.yaml file is used to to specify the project dependencies and assets. It's a vital one. You can read more in the pubspec file docs.

Than there's the `android` and `ios` folders that contain the native app projects, yes, in Kotlin and Swift respectively.

You also see a `test` folder. That's there so you can conveniently split tests from your production code. Tests must be written within this folder. We'll talk about testing flutter apps later.

Finally, the `lib` folder, this contains our actual flutter code! Here you are going to write most of your flutter app.

Chapter 3: Flutter UI

Let's get started with actual Flutter. It controls every pixel drawn on the screen, so you can pretty much do anything you can think about design-wise.

The flutter framework offers a huge amount of UI Components called "Widgets" that by default match the Android material UI guidelines and iOS guidelines, they can also be customized in endless ways allowing for rich and original designs.

Flutter and declarative programming

UI in flutter is mostly written in a declarative programming model (There are a few exceptions that we won't cover here).

Basically, you declare the Widget UI by overriding the `build()` method and the Widgets are held in a Widget tree. That expresses a function of the state. Therefore:

$$UI = f(state)$$

Read more on the start thinking declarative documentation.

Flutter widgets

There are basically three types of Widgets in flutter. Stateless, Stateful and Inherited. It's key that you learn the different at this point.

- StatelessWidget: Stateless widget are useful when the part of the user interface you are describing does not depend on anything other than the

configuration information in the object itself and the BuildContext in which the widget is inflated. Here's a video on how to create one.

- StatefulWidget: Stateful widgets are useful when the part of the user interface you are describing can change dynamically, e.g. due to having an internal clock-driven state, or depending on some system state. Here's a video on when they should be used.
- InheritedWidget: Base class for widgets that efficiently propagate information down the tree. Inherited widgets, when referenced in this way, will cause the consumer to rebuild when the inherited widget itself changes state. Here's a video of the flutter team explaining it.

You can read more about Widgets here (https://flutter.dev/docs/development/ui/widgets-intro)

Assets

Assets define all static data, images, fonts, video and audio files within your app. They are bundled and deployed with your app, so make sure to use them wisely, app size can increase dramatically.

You can learn more about Assets in the docs.

Building layouts in Flutter

The very core of Flutter's layout mechanism is widgets. In Flutter (almost) everything is a widget. All images, icons, and text you see in a flutter app are widgets. And also things you don't see are widgets, such as rows, columns and grids that arrange, constrain and align the widgets you see.

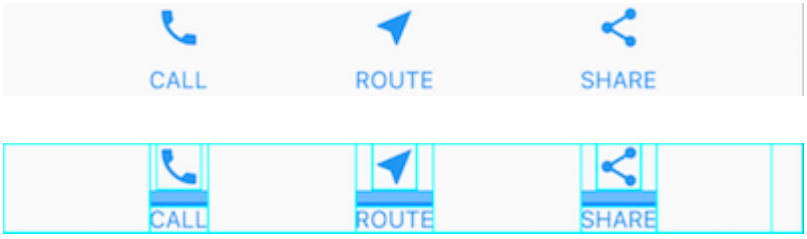
You create a layout by composing widgets in unlimited ways!

Remember this simple rule. It will get you really far:

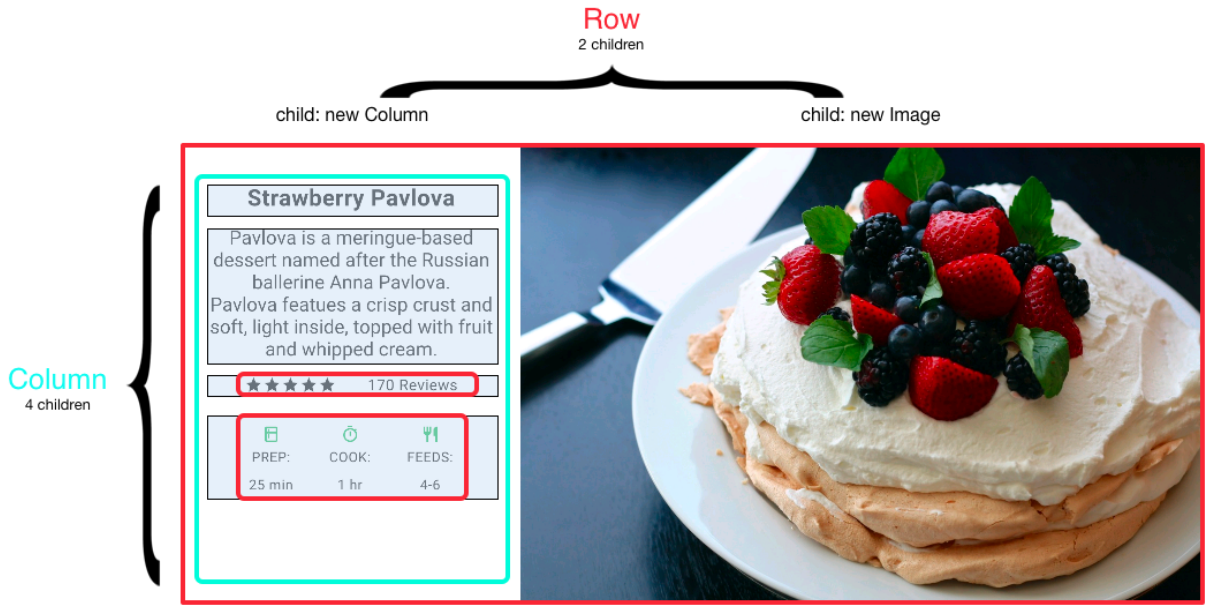
> Constraints go down, Sizes go up, Parent sets position.

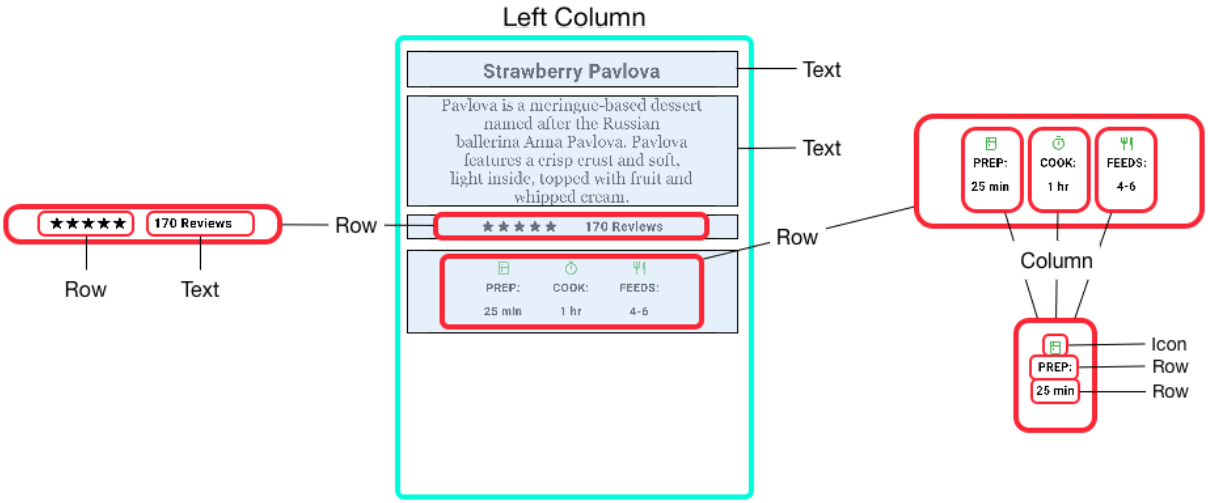
You can read more about this quote in the Understanding Constraints documentation

In the following image you can see the result above and the different widgets with blue borders in the image below.



Here is a more complex example.





I strongly suggest you read the Layouts in Flutter docs. It's really great and thorough.

Chapter 4: Animations in Flutter

When it comes to creating a good user experience for your app, adding well made and fluid animations is crucial. They work as a visual feedback for the user's actions and can also provide meaning of interaction and reassurance.

Flutter has some really well designed and conceived animation APIs that will allow you to take your app to the next level. It will be relatively easy to implement really complex animations.

The flutter team has produced A LOT of content about animations in flutter that will guide you way better than I could cover in a small chapter in this guide.

Read more at Introduction to animations

There are also codelabs available that will allow you to grasp the concepts completely.

The rive animations package

If you don't want to write code for handling animations, there's a great free service called Rive. It is a complete animations editor that will allow you to create really complex animations without any code. In order to use them in your flutter app, you'll need to export them and use the Rive package to load them into the application.

If you want to read more about Rive here's the Rive Flutter runtime, part 1 tutorial.

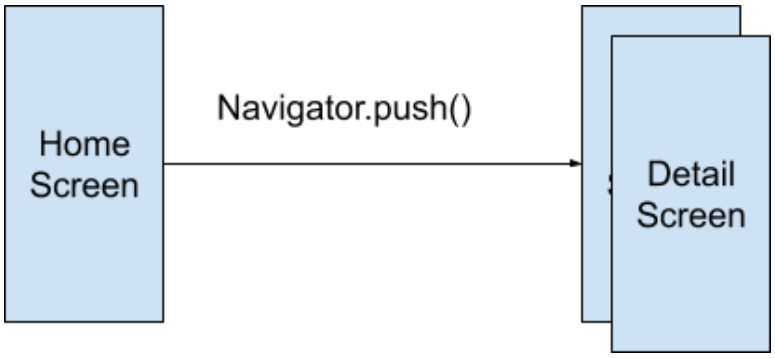
Chapter 5: Navigation and Routing

If you're not building a single page application, which you unlikely are, you'll eventually need to navigate to new pages or routes. Let's see how we do this in flutter.

The basics of navigating between two screens in Flutter are actually pretty straightforward, you can read more at Navigate to a new screen and back documentation.

Simplifying it a bit you do the following:

- You get a navigator instance by passing the context
- You call the `push()` method on it and pass a route to be added to the stack. The route is commonly a `MaterialPageRoute`
- When you want to get back to the previous screen you get the navigator and call `pop()`



You can also define some named routes to handle navigation more easily. I'll leave you the docs about it here: Navigate with named routes.

Navigation 2.0

Whether you like the Imperative API of Navigator 1.0, it has some really important limitations, mostly related to Flutter eventually coming to the web really, really soon.

What are the limitations?

Some of the limitations involve:

- Moving arbitrarily in the navigation stack is really hard. because `push()` and `pop()` only apply to the top most route.
- Opening a specific route, for example, from a push notification is hard, mainly because of the reason above.
- Restoring the app state after a relaunch is also really hard, still, because the navigation stack can't be changed arbitrarily.

The declarative, Navigation 2.0 API has been introduced in Flutter 1.22 to specifically fix this issues that are so important for web, where you're supposed to move from one route to another using urls.

You can read more about this in the Learning Flutter's new navigation and routing system article.

Are there any drawbacks to Navigation 2.0?

There isn't any technical limitation, but it comes with a really steep learning curve.

You can watch a conference called Understanding Navigator 2.0 in Flutter where Dominik Roszkowski goes a bit more in depth and shows how to use it.

Third party navigation libraries

Since the new Navigation API is so different and hard to grasp at first, the community has already started working on third party libraries that simplify it.

One of which is Flow Builder by Felix Angelov, the creator of the BLoC state management library.

My take on Navigation 2.0

Moving to a declarative and non-stack-based API was mandatory, web support with push and popping routes would have been disastrous. On the other hand the Navigator 2.0 API is really complex to pick up, especially for someone that's just starting with Flutter.

Luckily enough the awesome Flutter community has released some pretty great packages that simplify the API a lot.

Chapter 6: State management

State management is by far the hottest topic around flutter development. There's a lot of noise and a huge amount of different solutions.

It is a critical point indeed, state management is really important. I'm going to keep the redundant stuff to a minimum and write about what really matters.

In chapter 3, I mentioned the following formula while talking about the flutter declarative UI:

$$UI = f(state)$$

UI is therefore a function of the state, but what is state? The flutter docs define it as

> The data you need in order to rebuild the UI at any moment of time.

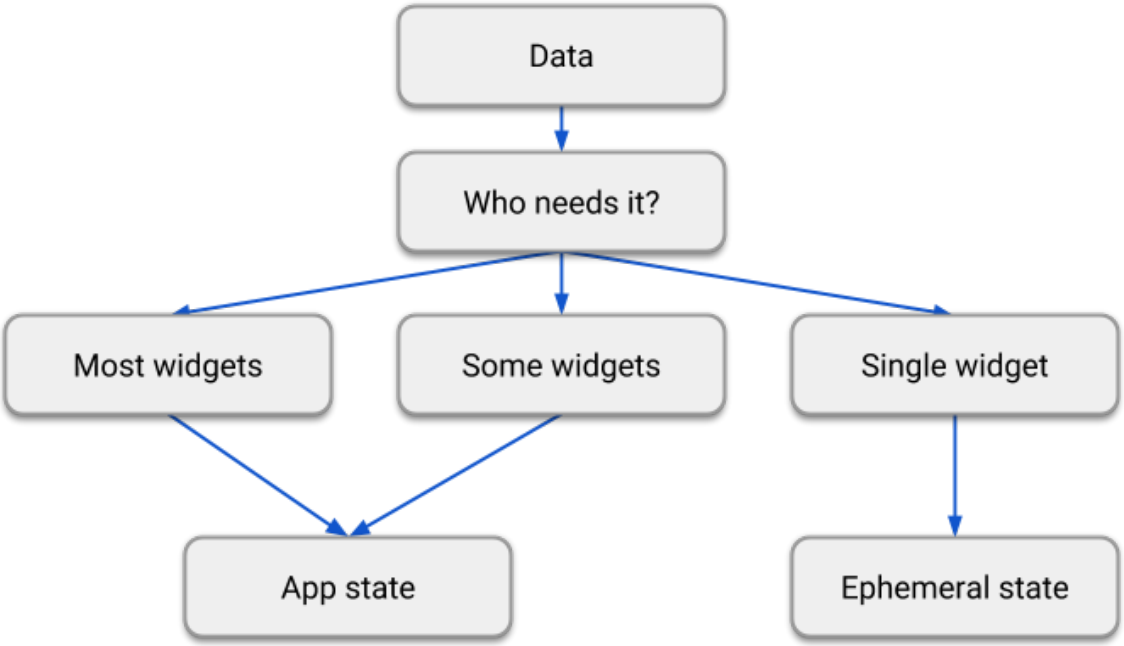
Ephemeral State and App State

The flutter documentation makes a really important distinction and it is really important you get this right. What is the difference between ephemeral state and app state?

Ephemeral State - I'd call it local state, it's state that is only used in a specific widget, for example whether a switch it's toggled on or off.

App state - This is the classic state, every state your app needs, an example could be the authentication status or the user properties.

This scheme explains it exceptionally:



I strongly suggest you use libraries for the app state and stick to setState for handling small local (ephemeral) state.

Yeah, alright, but which library should I use for app state?

The answer is, it depends. It depends on which one you like and on what you need for the project. There's no such thing as a best state management solution.

I've tried a few, I tried BLoC, MobX, GetIt, Provider and Riverpod.

Riverpod is the one I stuck with because it's really simple and feature rich. If you don't know where to start, I suggest you start from Riverpod.

Aside from that, read the docs and see which one would work best for you.

My take on Architecture and State Management

I've created a YouTube video explaining the production architecture and state management techniques I use daily: Flutter architecture - A practical approach.

Chapter 7: Networking and Backend

Chance is, if you're building an app, you'll need to interact with a backend in some way or another. In order to do that, you'll need to write some networking code.

By the way, if you're looking for APIs to base your app on, there's the Public APIs GitHub repo that features thousands of free APIs.

The http package

The HTTP package is integrated into flutter and it's perfect for handling simple REST API requests, it is asynchronous (of course), based on dart's Future and really simple to use.

Third party networking packages

For apps that need more complex handling of the requests, you might not make it with only the http package.

The Dio package is a great choice instead, and comes with some great features such as:

- Interceptors support
- Global configuration
- Request cancellation
- File downloading
- Timeout

The JSON format

Most REST APIs nowadays return the data you requested in JSON format. This data can be handled via dart's `Map` the problem is that key-value pair in the JSON aren't type-safe.

In this case the best practice is to take the JSON data and deserialize it into a strongly typed dart Object, a model class basically.

The model will have a `fromJson()` method that will return the deserialized object, securing the types and avoiding bugs.

You can either write these model classes by yourself, the following example should make it clearer

https://gist.github.com/dancamdev/d939aec8445a2169ed9548babc6ee58b

Or you can rely on code generation and in particular, I'd suggest the freezed package that offers a really convenient way to create these models with some extra features like deep equality and much more.

Chapter 8: Firebase

Firebase is a Backend-as-a-Service (BaaS) app development platform that provides hosted backend services such as a realtime database, cloud storage, authentication, crash reporting, machine learning, remote configuration, and hosting for your static files.

Its integration with Flutter is really, really great and it's perfect to use with small to medium size apps. Moodify, for example, uses Firebase for its backend.

The complete docs, that are really thorough and cover all the different services that Firebase offers can be found in the FlutterFire Overview, from there you can go through the initial installation and setup on the currently supported platforms.

Why is it so convenient?

Because it is really easy to integrate in Flutter and requires really little work and pretty much zero backend knowledge to use it.

It is free, until you reach a quite generous amount of operations per month, so far for Moodify, I haven't paid a penny.

Finally, it offers a big variety of services such as Authentication, Analytics, Database, Crash reporting, and Notifications. It really is quite a complete set of backend features.

Chapter 9: Keeping app data locally

We talked in the previous chapters about backend, database and sending data to the cloud. Along with keeping data on the cloud, you most probably will need to keep some data stored locally too.

> **Note:** the data stored locally will persist if the app gets killed, but it will get deleted if the app is uninstalled from the device and won't be synced across different devices.

An example would be storing settings related flags, e.g. the user selected the light or dark theme. We want to restore the theme if the app gets killed and restarted but we don't necessarily need it to persist across app uninstalls and installs and devices.

In flutter you have many different options and packages to keep local data, I'll focus on the one I use and prefer.

Storing data in key-value pairs

The go-to package for storing unencrypted key-value pairs in flutter is the Shared Preferences package, be careful though, you shouldn't store any sensitive data here, emails and addresses for example shouldn't be stored here, for that there's the...

Storing data in secure key-value pairs

I normally store sensitive data in an encrypted way in my backend database, but at times, you might want to store it on device, make sure you do it securely though.

The flutter secure storage package is just what you need in this case, it uses the Keychain on iOS and the KeyStore on Android so it's really secure. Also, the API is really similar to the shared_preferences one.

Wrap up

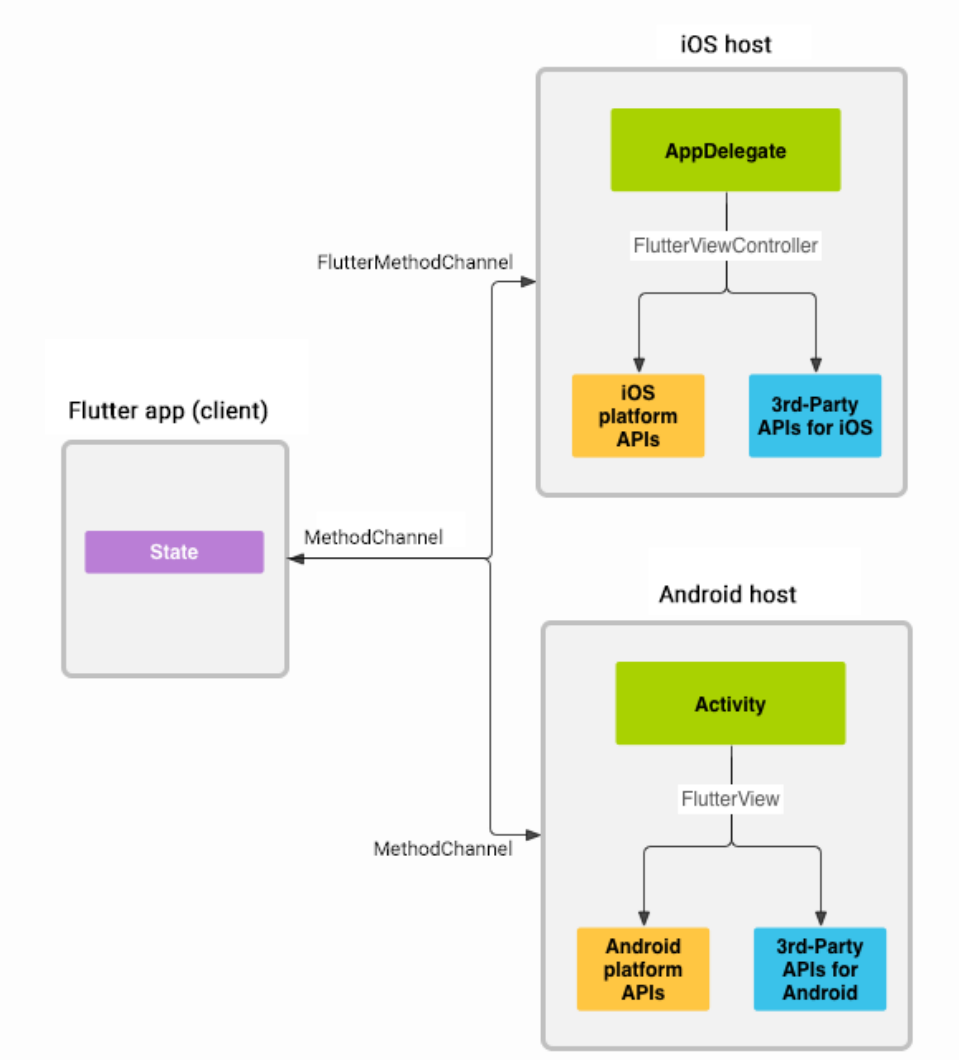
There's an excellent talk made by the flutter team itself for you to learn more, its called Keeping it local: Managing a Flutter app's data

Chapter 10: Integrating native code into your Flutter app

Most of the platform specific functionality is nowadays available as some already built package, but at times, you may want to integrate specific platform code for whatever reason. Well, flutter has you covered in this regard too.

A first-hand example, in Moodify, my app, would be sharing the generated images in a social network, I don't want the OS sharing options to pop up, I just want to open Instagram. There were no libraries available for that, so I wrote my own Kotlin and Swift code to achieve it.

Platform-specific code is called in Flutter by using `MethodChannels` and `EventChannels`, they're basically messengers that move data back and forth from the Flutter engine to the native one.



The difference between `MethodChannels` and `EventChannels` is that method channels would run some native code and return the result, while `EventChannels` will listen to something happening on the native side and return every time a new event is fired by using `Streams`.

Where can I read more?

You can read way more about this in the Writing platform-specific code docs, there's also an example on how to retrieve battery percentage.

There is also this really detailed guide called Flutter Platform Channels which is great and goes in depth about it all, it also covers testing.

My advices on flutter platform channels

In order for you to write platform-specific code, you'll need to become familiar with Kotlin (for Android) and Swift (for iOS). This is a good thing in general as it is important to have a sense of how the platform below the framework works.

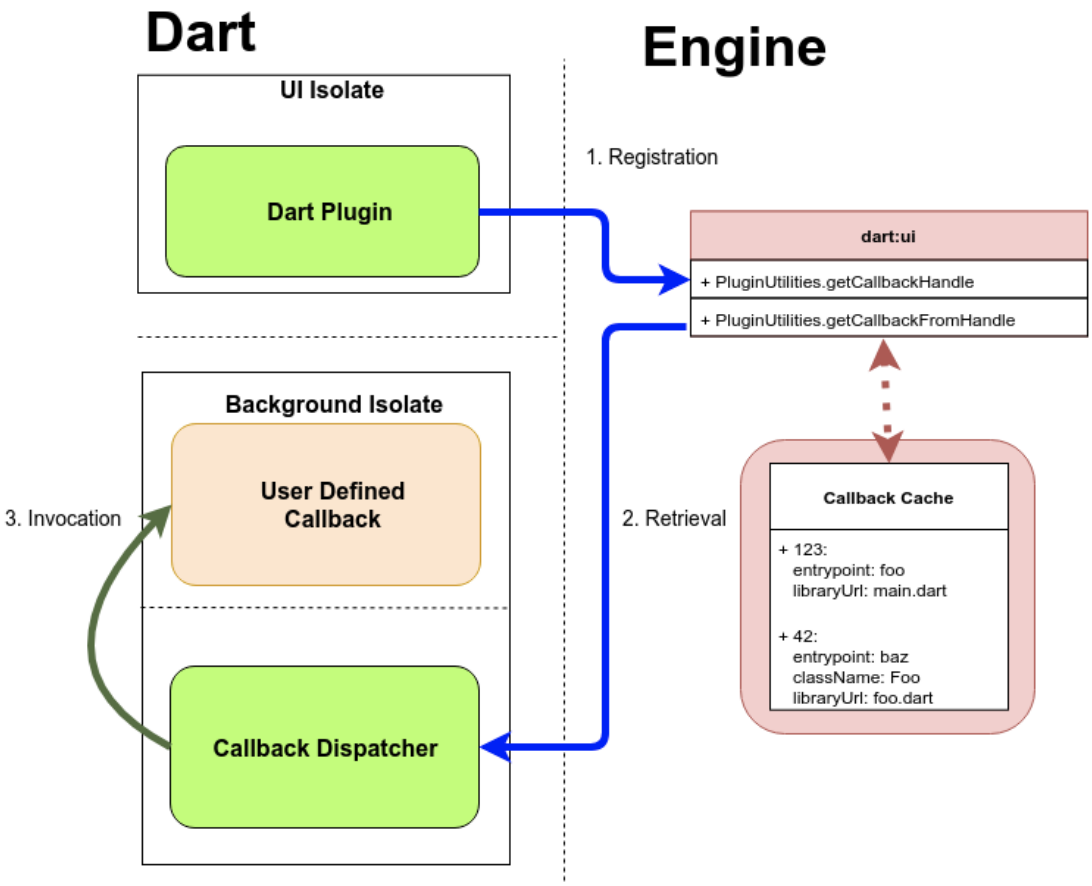
You don't need to be an expert in these languages, but knowing your way around it can help you not only to implement your own platform-specific code but also to contribute to the huge amount of open source libraries out there which always need some tweaking.

Chapter 11: Background processes

More complex apps often need to deal with some kind of background tasks that's going on while the app isn't displayed on the user's screen. Perhaps you'd want to implement a process that watches the time, in a timer app, or notifications to be displayed to the user.

Flutter can do that using Dart's Isolates, which is dart's way of handling multithreading, with some differences.

In order to setup the Isolate you'll use callbacks and a callback dispatcher. The following scheme shows how this works:



This image is taken from an excellent article written by Ben Konyi, which I highly suggest you read if you need to work with background tasks in flutter. It covers it all in depth and features complete examples. You can find it at Executing Dart in the Background with Flutter Plugins and Geofencing.

Chapter 12: Testing in Flutter

Yes, we've reached the chapter where we'll talk about testing.

Testing is way too neglected part of writing software, and I say that while being among those that don't enjoy writing tests and often don't do it. But I'm trying to improve and get better at it myself.

Testing is really, really important if you're serious about building robust and production ready apps, tests allow us to verify automatically that our code is bug free and behaves as we expect it to.

Manual tests, the ones where you launch and use the app basically, are really time consuming and prone to a lot of errors. Bugs can therefore easily occur in production.

Automated tests are there to solve this problem, they're quick to run and should ideally test our whole codebase, acting as a safety measure that allows us developers to find out if something broke while refactoring some code or adding new features.

In flutter we have three types of tests:

- **Unit tests** - tests a single function, method, or class
- **WIdget tests** - tests a single widget
- **Integration tests** - tests a complete app or a large part of it

The Testing Flutter apps documentation will give you a nice introduction.

While Robert Brunhage released a fantastic video on YouTube about this specific topic, called Flutter Testing For Beginners - The Ultimate Guide, which covers all the before mentioned tests types and explains how to use Mockito, which is a great library for mocking objects.

Chapter 13: Wrap up and next steps

Thank you for reading this guide and sticking to it so far. I really hope it was and will be helpful. The idea behind this isn't only to give you a road map, but also a reference to which you can easily come back to whenever you need to refresh some flutter topics.

I am here to help you.

I've created these guides to help you become a better version of yourself, both on the technical and business side of things!

The steps I'll be following from here are to build products specifically tailored to your needs to help you overcome the issues that everyone has faced at first, ourselves included.

I'm very interested in your opinion about this guide, whether you liked it a lot, you'd like to have more content covered, or you didn't like it at all, let me know either by writing an Instagram DM at @dancamdev or an email at support@dancamdev.com.

Next steps

At the time of writing I already offer a Mentorship program, it is available on my Mentorship Website, and it consists on one on one mentorship sessions where we can talk about anything really, wether you want to clarify some doubts on this guide, you want some help fixing a bug who doesn't want to disappear on its own, or talk about your learning or business strategy, I really encourage you to get a mentorship session.

You can get one on the Mentorship website, or if the plans there don't fit your needs, you can email me at mentorship@dancamdev.com

And remember...

Everything you do now, is for your Future!