

Self teaching myself Flutter, part 2: First Steps

In the previous section, I briefly introduced dart and flutter, described some quirks of the installation process for mobile development, and presented the context in which I had to create a mobile app along with what made me choose Flutter specifically.

This part assumes flutter is properly installed, and that VSCode is the editor of choice. In this part, I will recount what learning path I took to understand the basics of Flutter, which I will then present succinctly using the default flutter template as an example, before finally taking a look at Flutter's powerful tooling.

Learning the basics

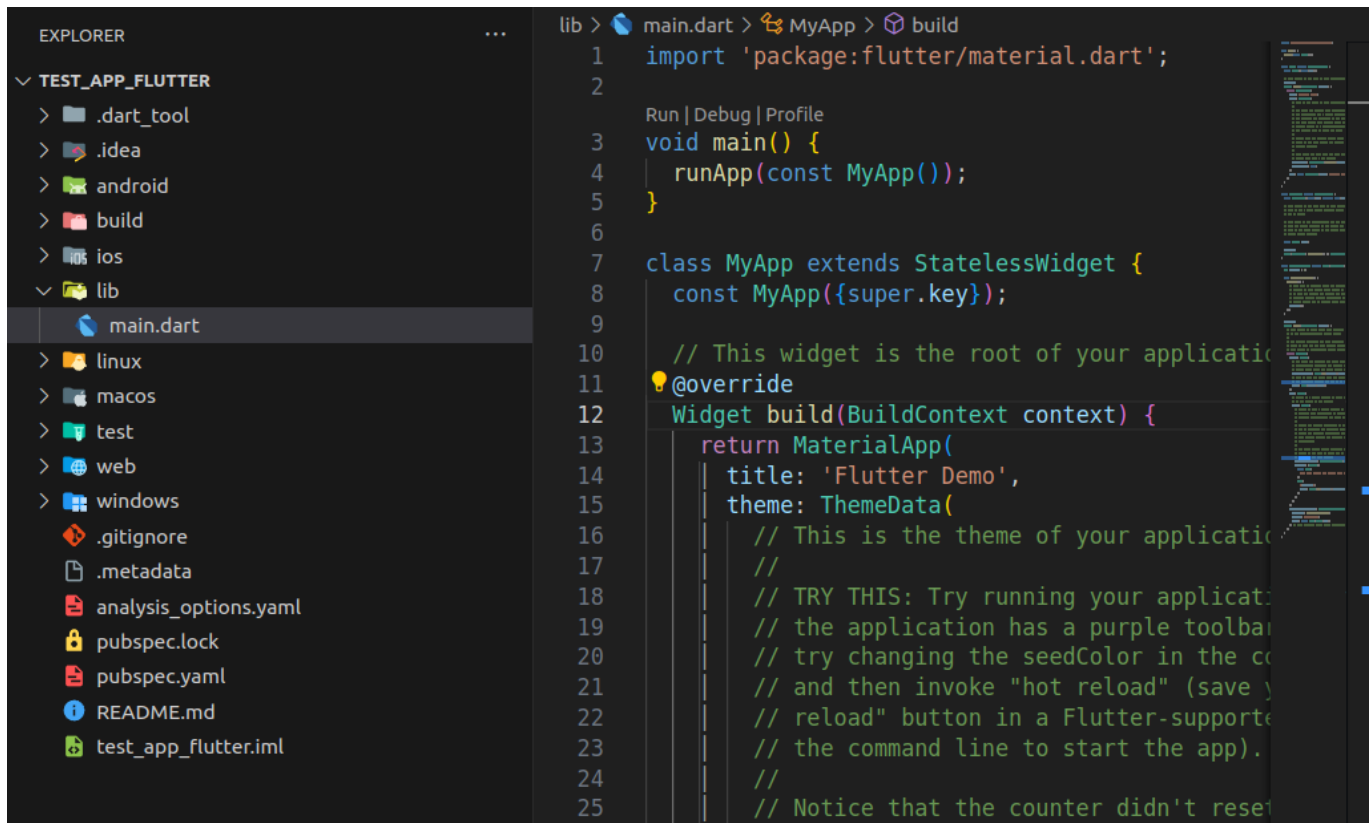
After deciding to use Flutter to create my mobile application, I started with a [video series](#) in order to cover the basics.

One issue I encountered with this series was that, being four years old, some changes had happened to flutter which invalidated some of the elements presented in the tutorials, for example, [some fundamental Widgets had been renamed](#), which caused me, as a total beginner, to get stuck for a few minutes. While not being much, those differences could keep piling up over the years, meaning one should be careful when exploring older learning resources.

In hindsight, I could have instead started with the [official learning resources](#) provided by the Flutter team, but back when I started, I hadn't realized how thorough the official documentation was, I highly recommend using it as your main source of knowledge if you are starting out with flutter.

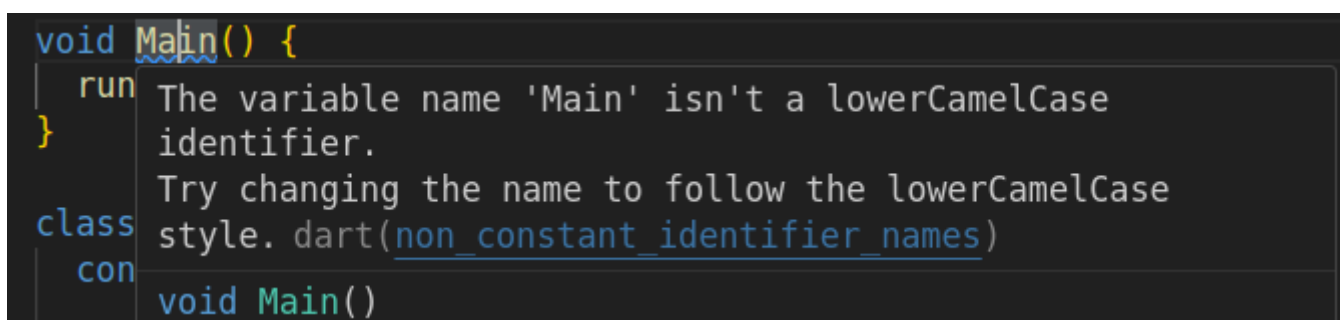
The basics of flutter

When you create your first flutter project using the "Application" template, you start out with a single file inside of your "lib" folder.



Other folders in the project root are mainly for platform-specific code, this aspect may be described in more details in a future article.

One striking aspect of flutter is that it is very Object-oriented. The framework makes heavy use of the class system, and Dart evolves in ways to facilitate oop, such as [the removal of the "new" keyword](#) when instantiating a class. The ambiguity between a class instantiation and a method call is then solved with proper capitalization, which is enforced, by default, with Dart's style guidelines:



Widgets

Widgets are the fundamental building blocks of the framework, they come in two flavors:

- Stateless widgets are static, purely presentational elements, they correspond to "dumb components" in other frontend frameworks. These widgets are eligible for Hot-reload when developing an application;
- Stateful widgets have to keep track of some state (data attached to them) and can dynamically modify it, when updating their code, a Hot-restart needs to be performed.

A stateless widget can contain a stateful widget and inversely.

Let's look at the sample application:

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepPurple),
        useMaterial3: true,
      ),
      home: const MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
```

The main App widget is stateless since it has no data that will change over time (for example, the homepage title will stay the same). It contains the `MyHomePage` widget.

Stateful widgets require a bit more boilerplate to set up.

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

  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

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

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,

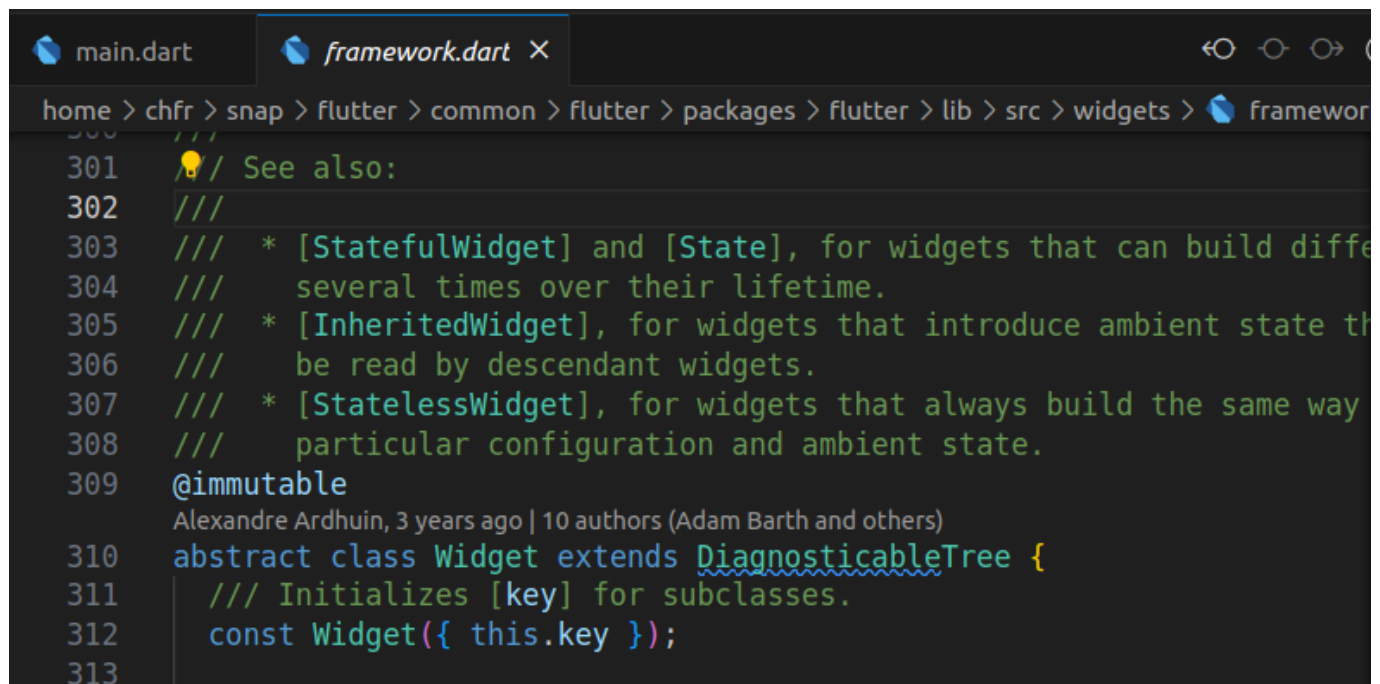
```

```

        children: <Widget>[
          const Text(
            'You have pushed the button this many times:',
          ),
          Text(
            '$_counter',
            style: Theme.of(context).textTheme.headlineMedium,
          ),
        ],
      ),
    ),
    floatingActionButton: FloatingActionButton(
      onPressed: _incrementCounter,
      tooltip: 'Increment',
      child: const Icon(Icons.add),
    ),
  );
}

```

First of all, notice how a stateful widget is made of two classes, one of type `StatefulWidget`, another of type `State<T>`. The reason we cannot manage state directly in the `StatefulWidget` class is that `StatefulWidget` extends `Widget`, which itself is an immutable class.



The screenshot shows an IDE with two tabs: `main.dart` and `framework.dart`. The `framework.dart` tab is active, showing the source code for the `Widget` class. The code is as follows:

```

301  /// See also:
302  ///
303  /// * [StatefulWidget] and [State], for widgets that can build differ
304  ///   several times over their lifetime.
305  /// * [InheritedWidget], for widgets that introduce ambient state th
306  ///   be read by descendant widgets.
307  /// * [StatelessWidget], for widgets that always build the same way
308  ///   particular configuration and ambient state.
309  @immutable
310  abstract class Widget extends DiagnosticableTree {
311    /// Initializes [key] for subclasses.
312    const Widget({ this.key });
313

```

Because it is immutable, all properties are final, meaning the state cannot be changed directly in the `StatefulWidget`.

This is where the `State` class comes into play, a `StatefulWidget` implements its `State` as a separate, mutable class, so that state can be changed. As a result, the main job of the `StatefulWidget` implementation is to define constructor properties (if there are any) and not much else. The reason `State` is a generic class, is so that such constructor properties can still be accessed. Otherwise it would not be possible to show the `title` property that was passed to the widget.

In order to access widget properties from the state, the `widget` variable must be used (like so: `widget.title`), the type of this variable is from the class passed to the generic (in our case: `State<MyHomePage>`). The management of generics is the same as with Typescript.

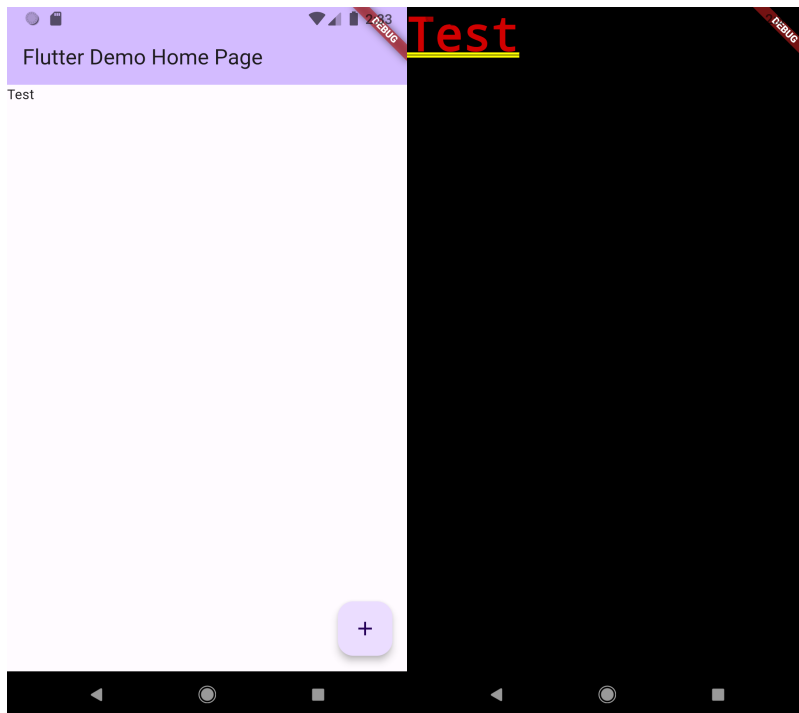
A change in the state must be made using the `setState()` method. When a state property changes, all Widgets of the `build()` method that use the property are updated.

Scaffold

The scaffold is a stateful widget, it is usually the main parent in a page/screen.

```
return Scaffold(  
  body: /*Widgets ici*/,  
  // appBar: AppBar(),  
  // floatingActionButton: FloatingActionButton(),  
);
```

Using a scaffold widget applies default styles for child components. Here is a comparison of what a regular `Text` widget looks like with and without a scaffold.



Style aside, the scaffold also has numerous properties meant to provide basic mobile app functionalities with minimal boilerplate such as:

- A floating action button;
- Sidebars;
- Drawers;
- Title bars;
- App bars;
- etc...

Other widgets

Flutter has many types of widgets that can be learned about from the [widget catalog](#).

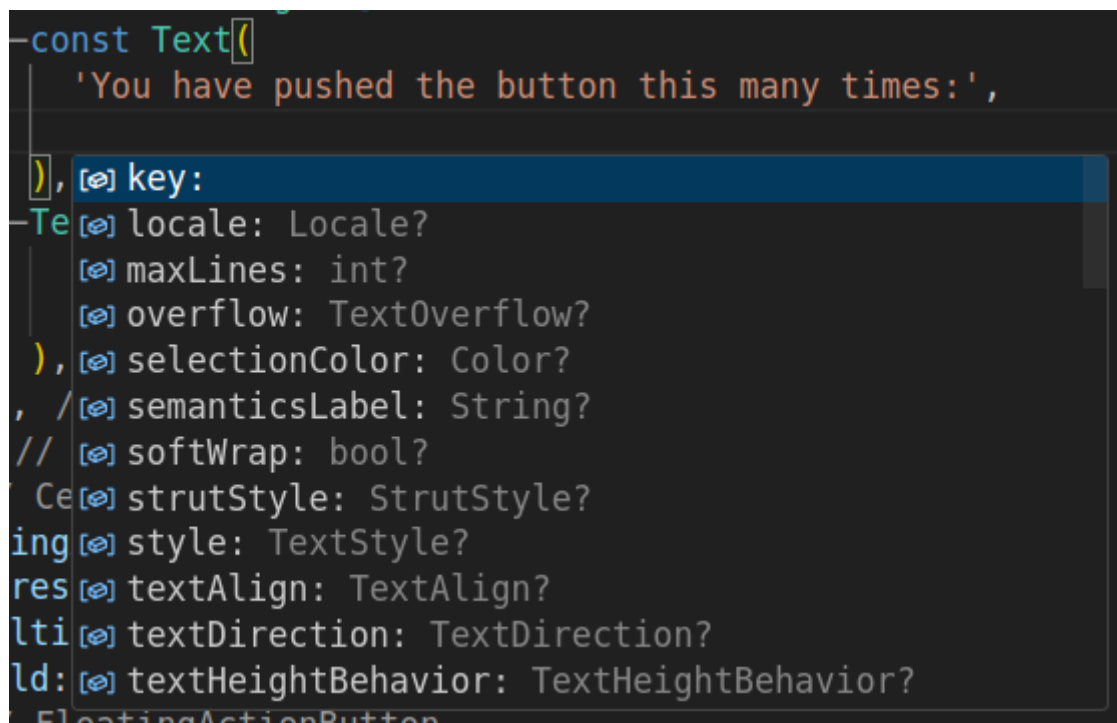
Everything that is done on the UI is done with a widget, these are tasks such as:

- Layout management: `Row`, `Column`, `ListView`, `Grid`, etc...
- Interactivity: `TextButton`, `FilledButton`, `IconButton`, etc...
- And many more (text styling, async, animations, scrolling, interactivity).

The strength of strong typing

Every widget in flutter is described in great details, the same goes for widget properties, and their subsequent properties. This makes the Flutter API highly explorable directly from the editor.

`ctrl + space` is an essential shortcut to learn when developing with flutter, as it forces auto-completion.



```
const Text(  
  'You have pushed the button this many times:',  
),  
Te locale: Locale?  
  maxLines: int?  
  overflow: TextOverflow?  
) , selectionColor: Color?  
, / semanticsLabel: String?  
// softWrap: bool?  
Ce strutStyle: StrutStyle?  
ing style: TextStyle?  
res textAlign: TextAlign?  
lti textDirection: TextDirection?  
ld: textHeightBehavior: TextHeightBehavior?  
FloatingActionButton
```

Using this shortcut is a great way to explore all the properties at your disposal in a class, many of which are self-explanatory from their name alone.

Flutter tooling

Looking at the earlier code-examples, you might realize that Flutter is fairly verbose. For example, adding padding to a text widget, and styling said text widget is done as such :

```
Padding(  
  padding: const EdgeInsets.all(8.0),  
  child: Text(  
    'Text',  
    style: TextStyle(  
      color: Theme.of(context).primaryColor,  
      fontWeight: FontWeight.bold,  
    ),  
  ),  
),
```

Because of the strong type system, it also means most complex configuration properties also use specific classes that must each be instantiated. This can result in a lot of code that is tedious to type manually.

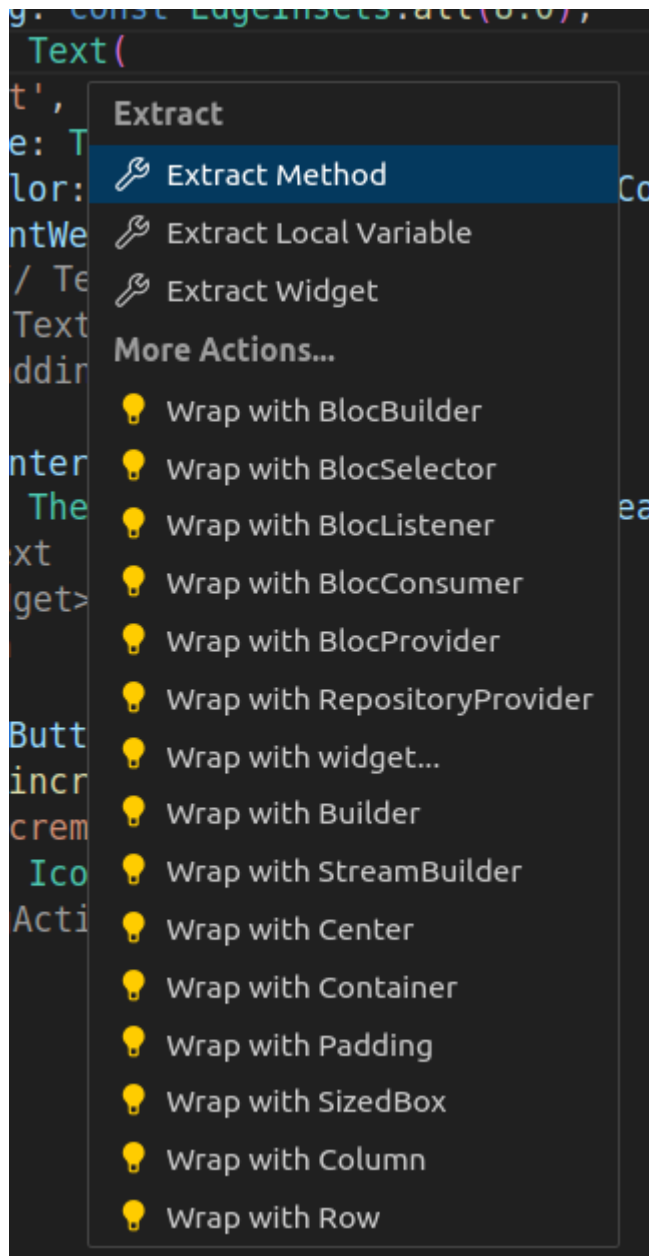
It must also be mentioned that, since Flutter screens are built via a widget tree, the developer will often face heavily nested classes that are cognitively taxing to manage and move around.

Because of this, Flutter comes with many refactoring tools made to improve the developer experience, learning these tools is critical to the development of bigger applications.

Quick fix

The Flutter extension in VSCode hijacks the **Quick Fix** shortcut (**ctrl+.** by default) to provide numerous essential widget helpers to allow functionalities such as:

- Wrapping the widget with another one (padding, centering, column, container, etc...);
- Swapping the widget with a parent/child;
- Turning a stateless widget into a stateful one;
- Extracting a widget tree into its own separate widget;
- etc...

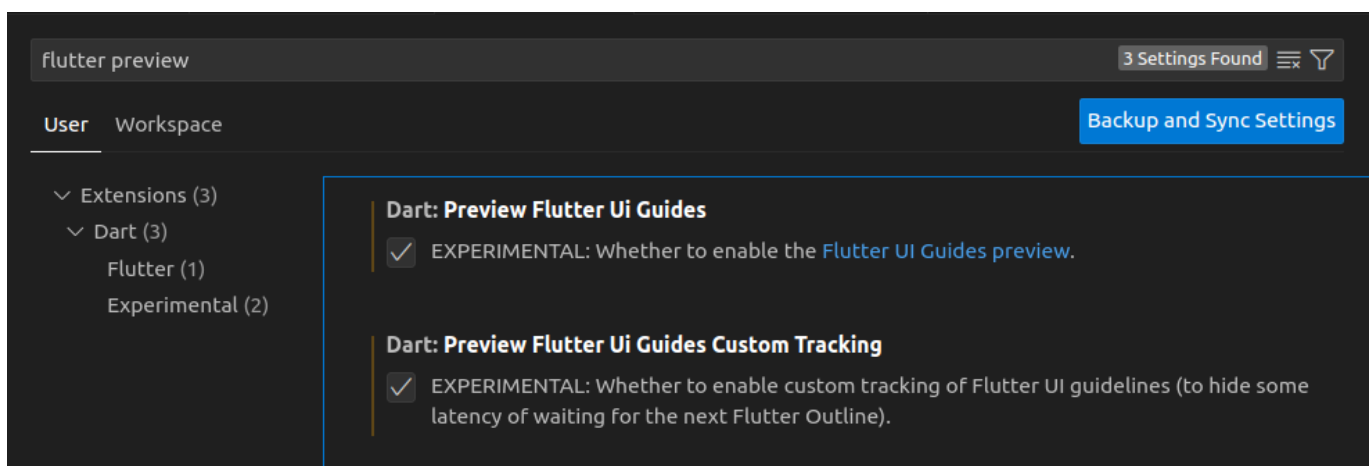


Visualizing the widget tree

A feature that is enabled by default on Android studio, but not on VSCode, is the visualization of the tree structure in the editor.


```
return Scaffold(  
  appBar: AppBar(  
    backgroundColor: Theme.of(context).colorScheme.inversePrimary,  
    title: Text(widget.title),  
  ), // AppBar  
  body: Center(  
    child: Column(  
      mainAxisAlignment: MainAxisAlignment.center,  
      children: <Widget>[  
        const Padding(  
          padding: EdgeInsets.all(8.0),  
          child: Text(  
            'Text',  
          ), // Text  
        ), // Padding  
        Text(  
          '$_counter',  
          style: Theme.of(context).textTheme.headlineMedium,  
        ), // Text  
      ], // <Widget>[]  
    ), // Column  
  ), // Center  
)
```

The lines connecting parents to children on the left side are very convenient to understand the structure of the widget tree at a glance, this can be enabled with the following experimental switches in the VSCode settings:



Or in json format:

```
"dart.previewFlutterUiGuides": true,  
"dart.previewFlutterUiGuidesCustomTracking": true,
```

Leveraging Flutter's tooling is crucial in order to deal with the framework's verbose nature.

Conclusion

In this article, I presented how I initially started learning flutter, and shared some more adequate learning resources. I then used the default application template in order to explain some fundamental concepts of the framework, before finishing by talking about the tooling.

All in this article points to Google having put substantial efforts in making the framework as documented and convenient as possible. This care put in the overall developer experience is understandable, as Flutter needs to justify its existence among every other frontend/mobile framework it has to compete with. If the developer experience is not pleasant, then people will use other frameworks that don't require learning a new language.

What's next

When I was watching the initial tutorial playlist, I stopped at the video n°16, about stateful widgets, the reason I halted my progress was because the recommended videos below that one were videos advising **against** the use of stateful widgets for complex state management. That was when I was made aware of third-party state management libraries for dart and flutter. This aspect turned out to be the hardest element to understand during the making of my mobile application.

In the next article, I will talk about state management in flutter using the BLOC library.