Flutter with Firebase: (Refer app: cupped\_lightning)

Okay, now that we know at least a little bit about developing apps using the flutter framework, and are comfortable with working in the DART language, we’ll explore the process of integrating our app with Google’s Firebase framework at the back-end. Currently we’ll be exploring the firebase integration for Android only. Later, we shall see about extending this functionality to iOS.

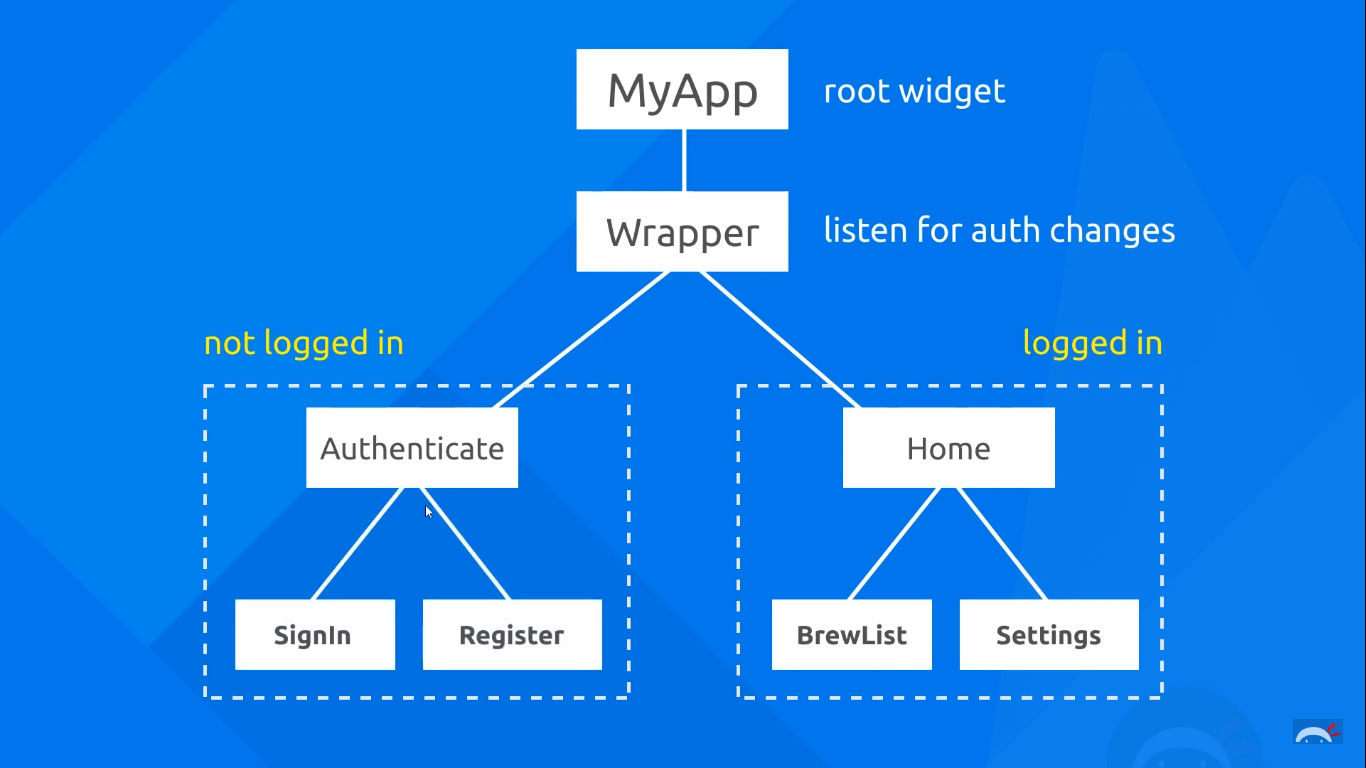
Now, just a quick look at Firebase. It is a service developed by Google which manages various app activities like user authentication, database management, storage, and a host of other activities. It makes the task of actually developing a database manually, and setting up authentication activities extremely simple and allows us to focus on flutter.



We’re going to be starting off with a very simple app during the development of which we shall learn all the different aspects of flutter+firebase. We’ll call this app as ‘Cupped Lightning’ and this app will be used to record the brew preferences of a community. Based on these brew preferences one member of the community will be able to prepare the brew for everyone.

Basic App Structure:

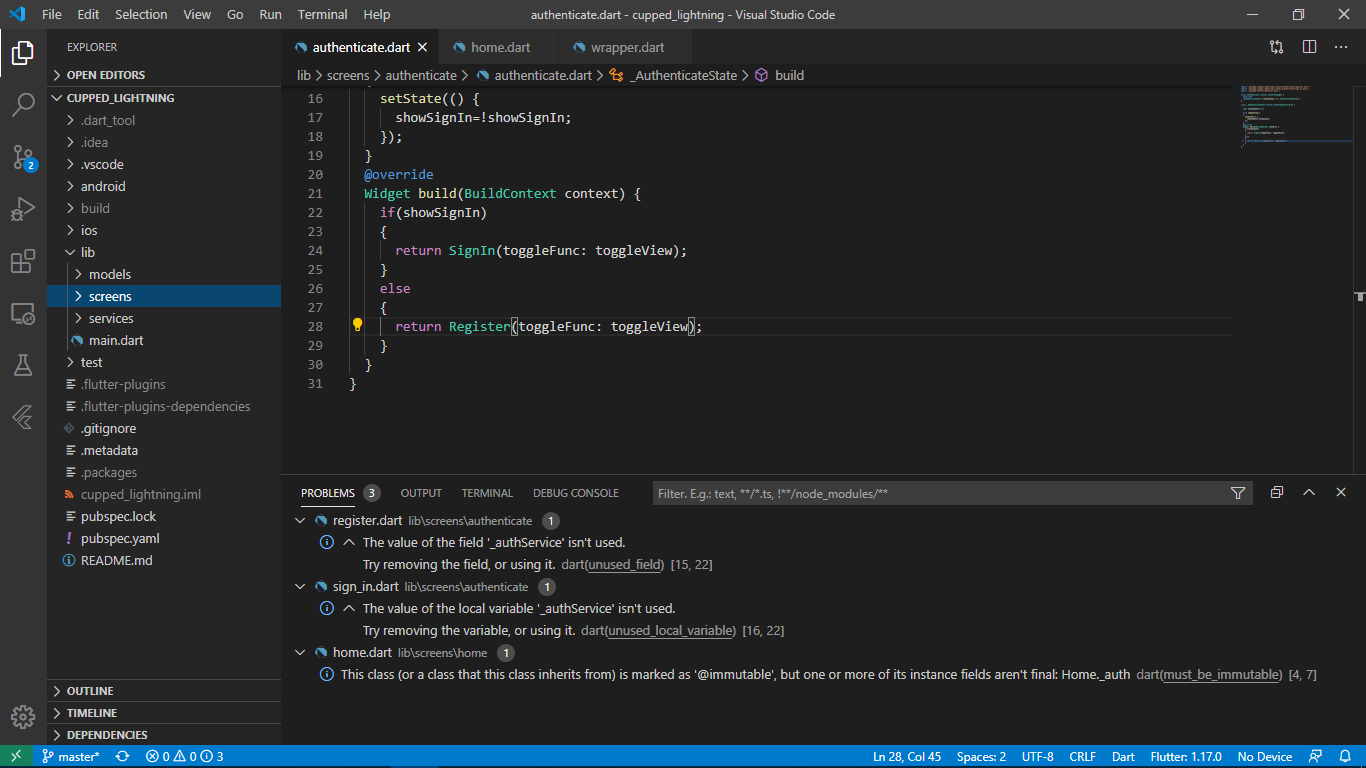
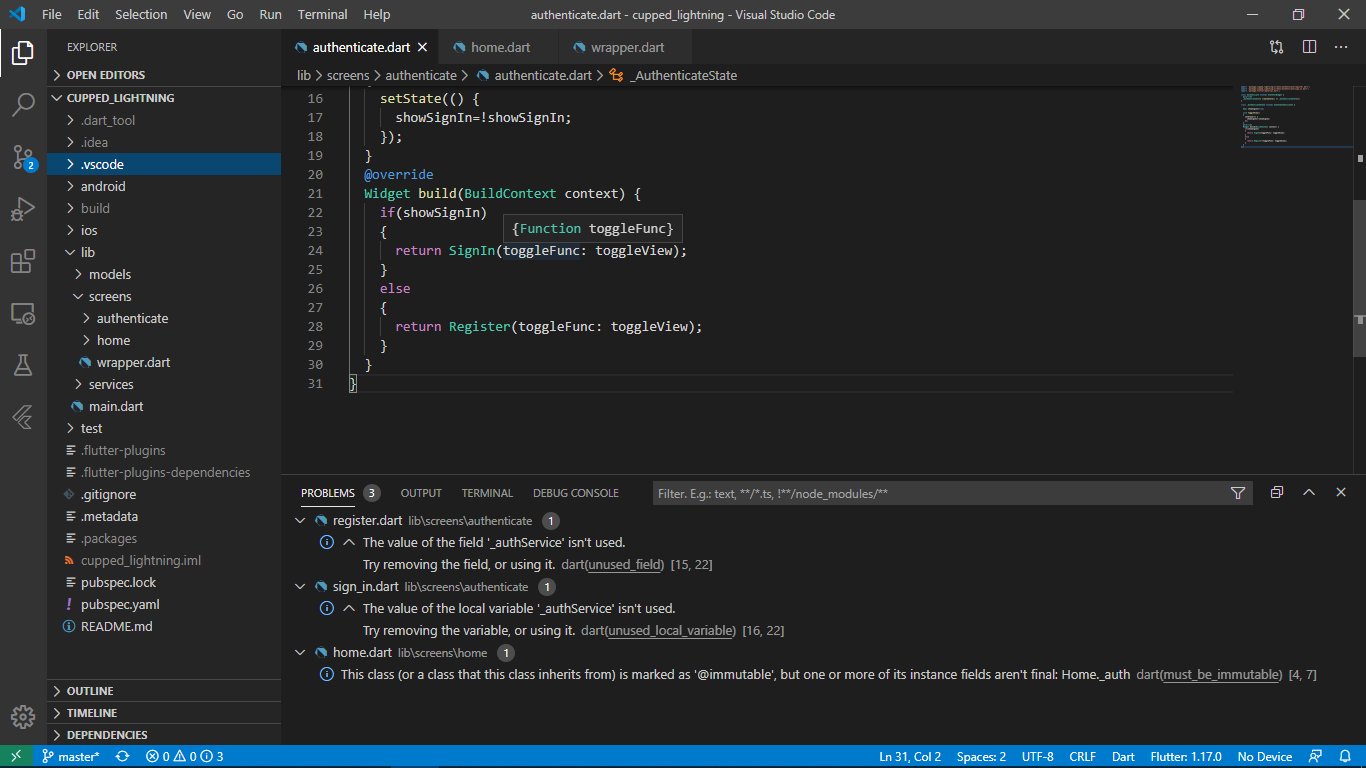
Since we’re going to be using the FirebaseAuth, as well as the Firebase realtime database, we need to think clearly and design the structure of the app beforehand. Let’s look our app tree first of all:

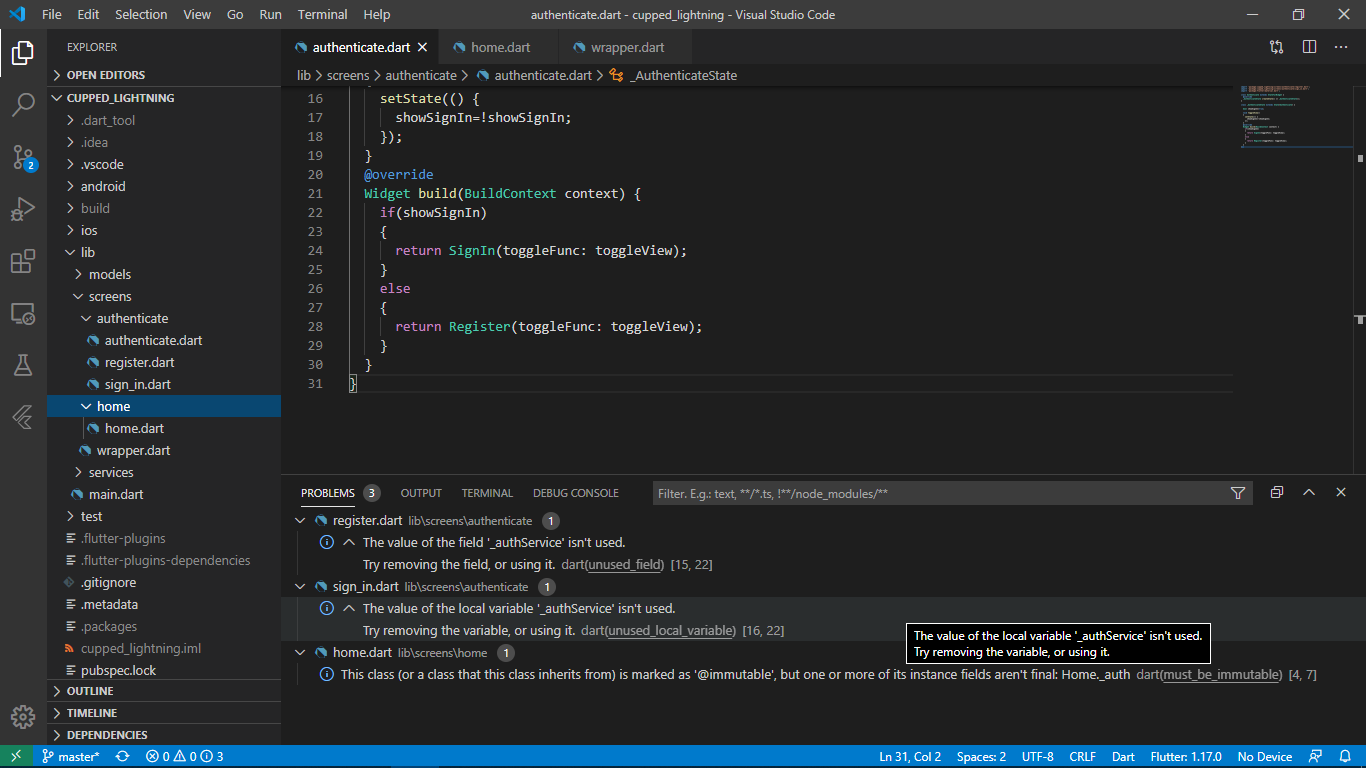


Our app will have a root widget called ‘MyApp’. Nested inside will be a widget called ‘Wrapper’. This widget is like the fulcrum of the app. This widget constantly listens for auth changes in the app. Whenever a user signs in or logs out, it shall display the Home screen or the Authenticate screen respectively. Thus, this widget will always be on the lookout for auth changes.

Inside the authentication screen, we will have a sign in and register screen, and the user must be able to switch between them. Similarly inside the home screen, we shall have the BrewList widget which will display the list of different types of brews that all different users have requested. There will also be a settings panel which enables the current user to change his brew preference.

Now, in our directory structure, we’ll create a few folders and files which will enable us to keep things organised.



So, we have a main folder called ‘Screens’ which has 2 sub folders—‘authenticate’ and ‘home’ and a file called ‘wrapper.dart’. The authenticate folder has all the files which go into making the authentication screen like signIn, or register. Similarly, the home folder has the files which are used for making the home widget.

The authenticate folder:

This has the authenticate.dart file. This file is responsible for showing the signIn or register screens based on user input. Thus, this file switches between these two screens when the user clicks the respective button. This file is nothing but a stateful widget called ‘Authenticate’. It is stateful because we’re going to be using states inside this in the future.

import 'package:flutter/material.dart';

class Authenticate extends StatefulWidget {

  @override

  \_AuthenticateState createState() => \_AuthenticateState();

}

class \_AuthenticateState extends State<Authenticate> {

@override

  Widget build(BuildContext context) {

return SignIn();

}

The home folder:

This has the home.dart file. This file is the widget that represents what the user sees whenever he has logged in. This will have the brew list and the settings button. This will also have the sign out button. The home.dart file is nothing but a stateless widget called ‘Home’.

import 'package:flutter/material.dart';

class Home extends StatelessWidget {

@override

  Widget build(BuildContext context) {

    return Container(Text('Home'));

}

The wrapper.dart file:

As discussed earlier, this file represents a widget which continuously listens for auth changes. And depending on the auth state, it changes between the Authenticate() or the Home() widget that we discussed earlier. This is a stateless widget with the name ‘Wrapper’. This is because we’re not directly be using states in this widget.

import 'package:flutter/material.dart';

import 'package:cupped\_lightning/screens/home/home.dart';

class Wrapper extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

//Return either Home or Authenticate widgets

return Home();

}

The main.dart file:

This file contains the root widget of our app. The root widget does nothing but contain the wrapper widget. So, all our root widget has is the Wrapper() widget as the home property. Here we’ve made slight changes to the basic theme of our app. We’ve changed the main color to ‘pink’ and the accent color to ‘pinkAccent’. If we don’t specify the theme of our app, flutter does it automatically for us. The rest of the code does nothing but call the Wrapper() widget as the home property.

import 'package:cupped\_lightning/models/user.dart';

import 'package:cupped\_lightning/screens/wrapper.dart';

import 'package:cupped\_lightning/services/auth.dart';

import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

void main() {

  runApp(MyApp());

}

class MyApp extends StatelessWidget {

  // This widget is the root of your application.

  @override

  Widget build(BuildContext context) {

    Return MaterialApp(

        home: Wrapper(),

        theme: ThemeData(

    // Define the default brightness and colors.

    brightness: Brightness.dark,

    primaryColor: Colors.pink,

    accentColor: Colors.pinkAccent,

    textSelectionHandleColor: Colors.pinkAccent

    ),

  )

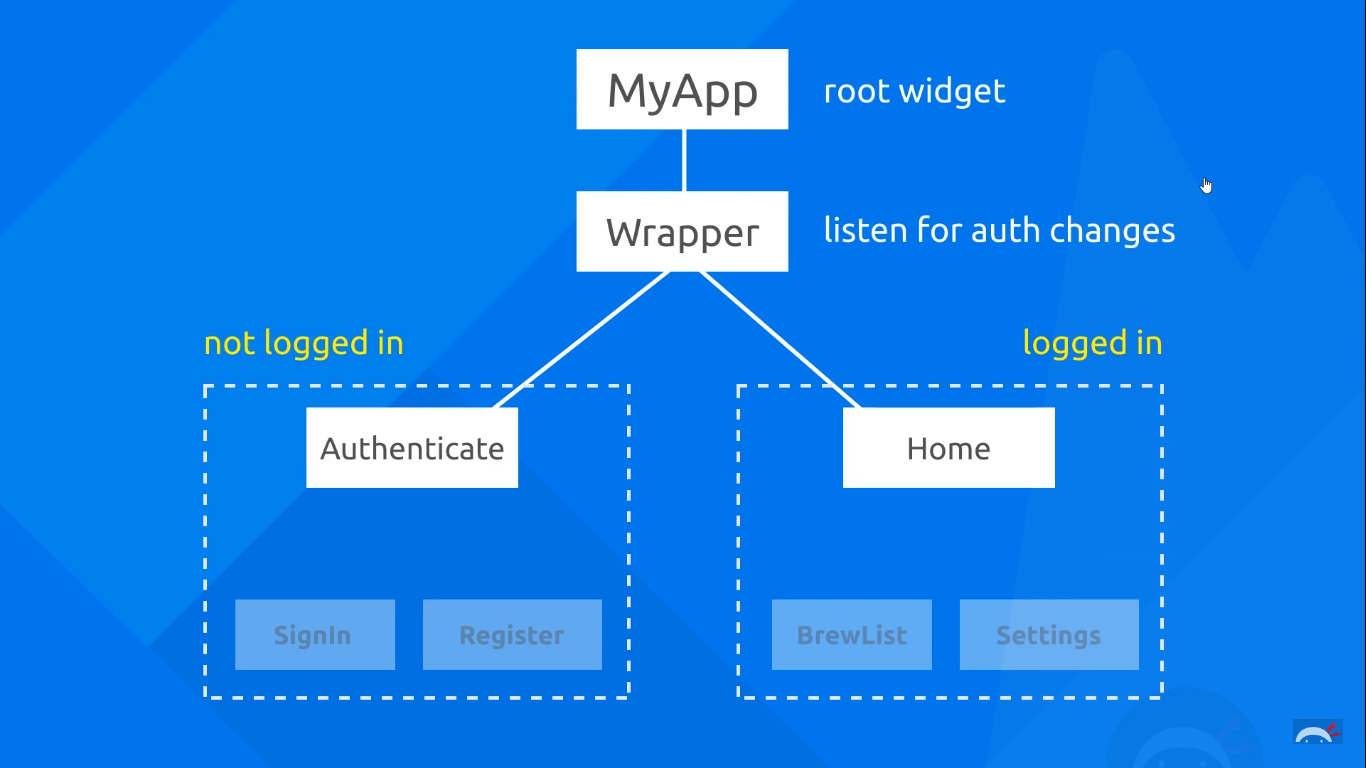
);

  }

}

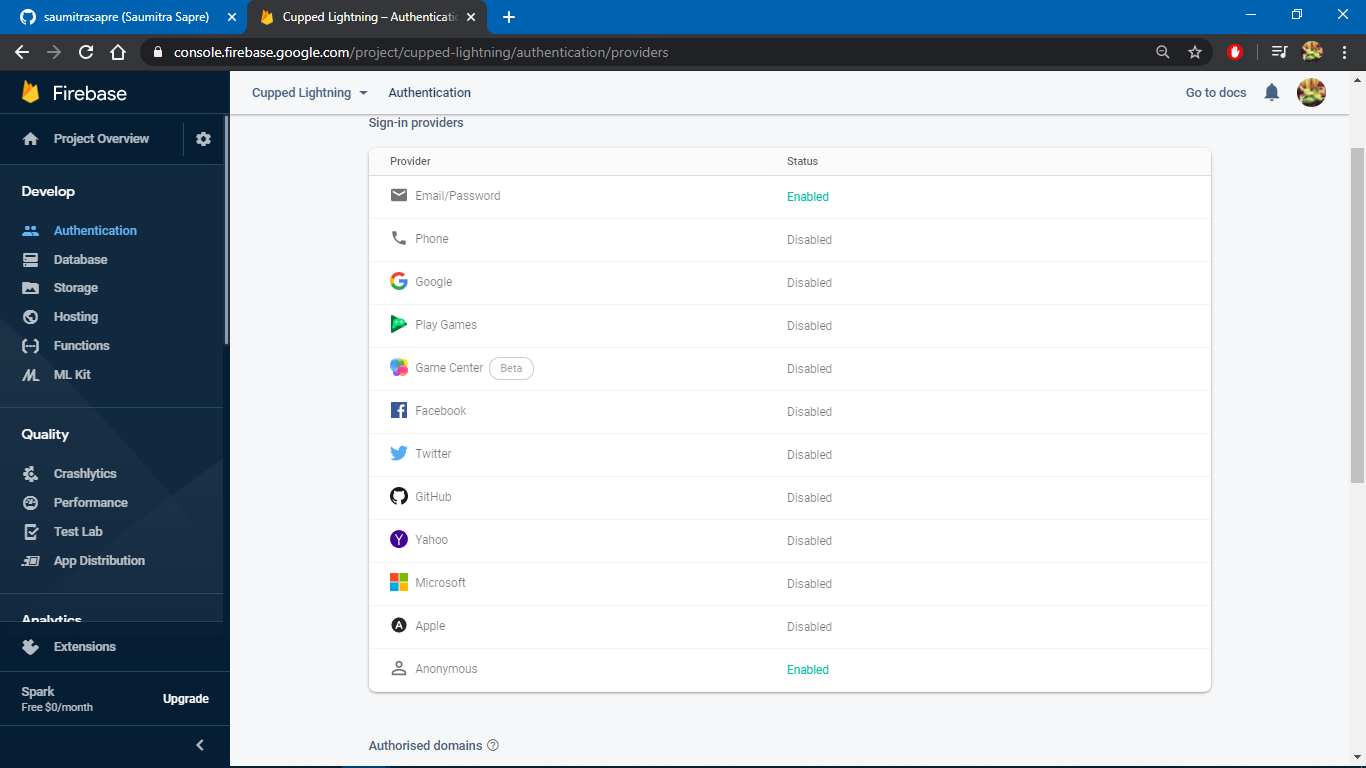
Firebase Auth:

Now we have the basic skeleton of the app developed, and it looks somewhat like this:

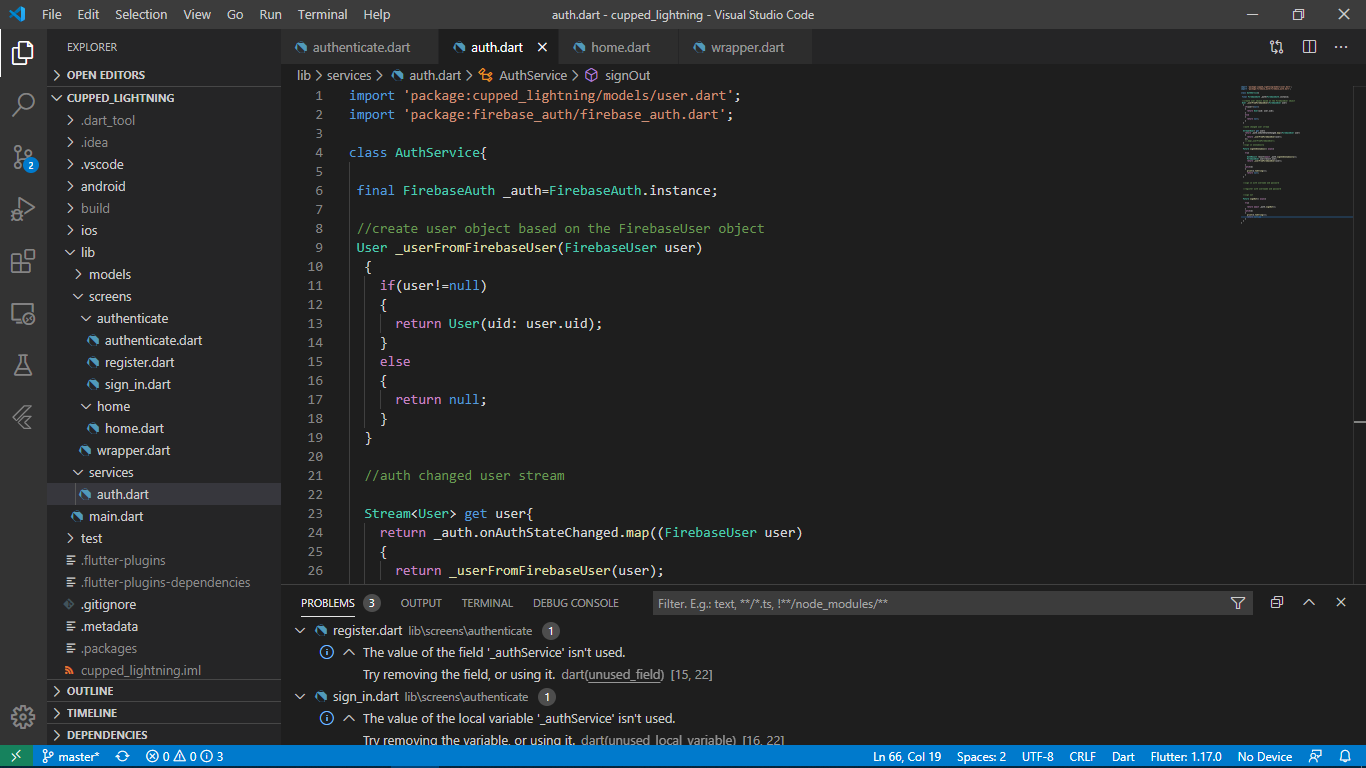


It’s now time to introduce firebaseAuth into the mix. Our aim is to show the Home screen to only those users which have been authenticated. To do this, and to keep track of the authentication status of the user, we use the firebaseAuth which is as service by firebase which manages authentication for us.

Before we code, we should first enable the sign-in methods from our firebase console. We’ll enable Email/Password and the Anonymous sign-in methods. The anonymous sign in method just for testing our sign in functionalities, we’ll disable it later on.



Now that we have enabled these services, we need to access them through our code. In order to bring structure to our app, we shall create a separate dart file which contains the code for signing-in, signing-out, registering, etc. We shall create a new folder called ‘Services’ inside which we have a dart file called ‘auth.dart’.



auth.dart file:

Here we shall first import the firebase\_auth package.

import 'package:firebase\_auth/firebase\_auth.dart';

Here we shall create a class called AuthService. This class holds all the methods which are going to interact with firebaseAuth for us. Thus these methods include signIn(), register(), signOut() and so on.

class AuthService{

//sign in anonymously

//sign in with username and password

//register with username and password

//sign out

}

Before we move on to the functions, we need to create an instance of the FirebaseAuth class. This instance enables us to communicate with firebase at the back-end. Thus, all communication from the app to the firebase server is handled by this object.

class AuthService{

 final FirebaseAuth \_auth=FirebaseAuth.instance;

}

Notice the underscore before the instance name. This implies that the variable is private and can be accessed only in the current class. The above statement therefore creates a private instance of the FirebaseAuth class called ‘\_auth’.

Now, let’s create a function called ‘signInAnonymous’ which enables us to sign in anonymously. This function as you might have guessed is an async function. The function sends in a sign-in request to the firebase server, and the server responds with a FirebaseUser object. The rest of the code should not proceed without the completion of this signIn request. Thus we shall have to use the await keyword. The whole function returns a ‘Future’.

We can imagine the Future object like a ‘box’. Because the function can also result in an error, the caller of the function is not sure about what exactly the function returns. Returning a ‘Future’ from an async method just gives a box to the caller. This box will be opened only when the async task completes—either successfully or with an error. Learn more [here](https://youtu.be/OTS-ap9_aXc).

We use the FirebaseAuth instance that we created and call the method called signInAnonymously(). This method returns an object of the AuthResult class. We’ll call this object as ‘result’. We create an instance of the FirebaseUser class, and extract the user information from the ‘result’ object. The result object contains all the information about the sign in response. One of which is an instance of FirebaseUser. This is nothing but the user information about the user who signed in.

We enclose the code inside a try-catch block in order to handle errors. The catch block will return a ‘null’ value in case the sign in fails.

//sign in anonymously

  Future signInAnonymous() async{

    try{

      AuthResult result=await \_auth.signInAnonymously();

      FirebaseUser user=result.user;

      return user;

    }

    catch(e)

    {

      print(e.toString());

      return null;

    }

  }

Signing In Anonymously:

Now that we have established the anonymous signIn service, we need to create the signIn page on our app. Thus we create a new dart file called signIn.dart. This file is the sign in screen which shows the user textfields to enter the username and password.

Because currently we are using an anonymous sign-in, we will create these textfields later. Right now, we just want to test out the sign-in functionality.

