CMSC 23: Mobile Computing

Week 8: User Authentication and Automated Tests

Objectives

At the end of this session the students should be able to:

- Implement username and password based authentication using Firebase Authentication
- Get familiar with the concept of automated testing in dart and flutter
- Create automated tests for flutter apps

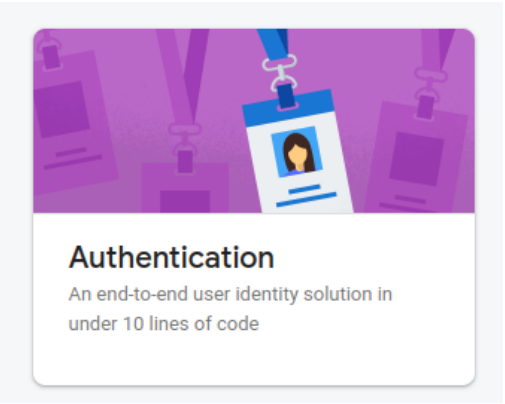
Introduction

The app we are developing might need a way to authenticate and know its users, this allows our app to securely save information in the cloud and ensure privacy and security. You can also personalize the UI/UX when using your app when the app knows its users' identities. A common way of implementing user authentication is the username and password, if done right, it is an effective way of protecting whatever resource or information your application has access to.

There are plenty of options to choose from when implementing user authentication, we have the option to implement our own way of storing, retrieving, and checking credentials or we can use reputable authentication services that can guarantee a level of security that is acceptable for established and well-known applications.

Firebase Authentication

Authentication is one of the services Firebase offers. Other than the basic email/password authentication, we can use sms, email login links or third party authentication providers (Facebook, Google, Microsoft, etc.) but for this lesson, we'll implement the common email and password authentication.



We will be using a modified version of the *firebase* branch of the week 7 discussion code https://github.com/ajhao/week-8---auth make sure to download or clone the **starter** branch

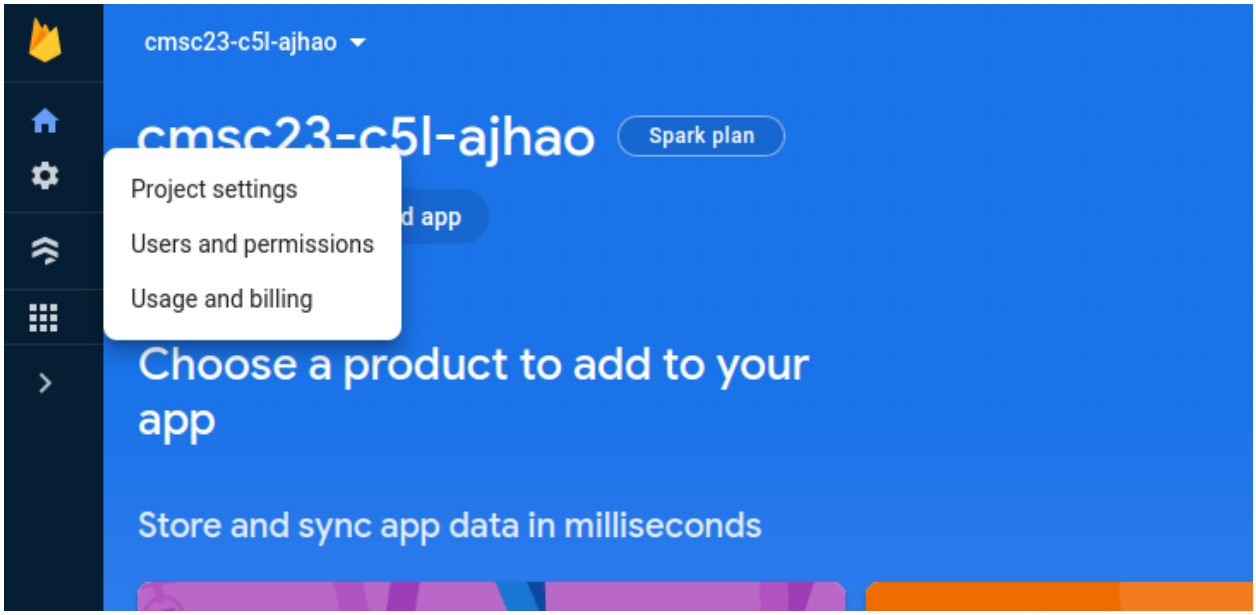
We've already installed the CLI and created a Firebase project in the previous session so we'll be using the same project and activate the Firebase Authentication tool. (If starting from scratch or a new flutter project, then we'll have to refer to parts **I** and **II** of installing and setting up Firebase in the week 7 handout.)

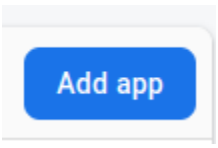
For an existing firebase project, we can access the configuration keys by click on the gear icon on the panel on the left and then click **Project Settings**

I. Set-up Firebase (again)

A. Make sure that the account logged in the machine is yours by running the command **firebase logout** and then **firebase login** on the command line.

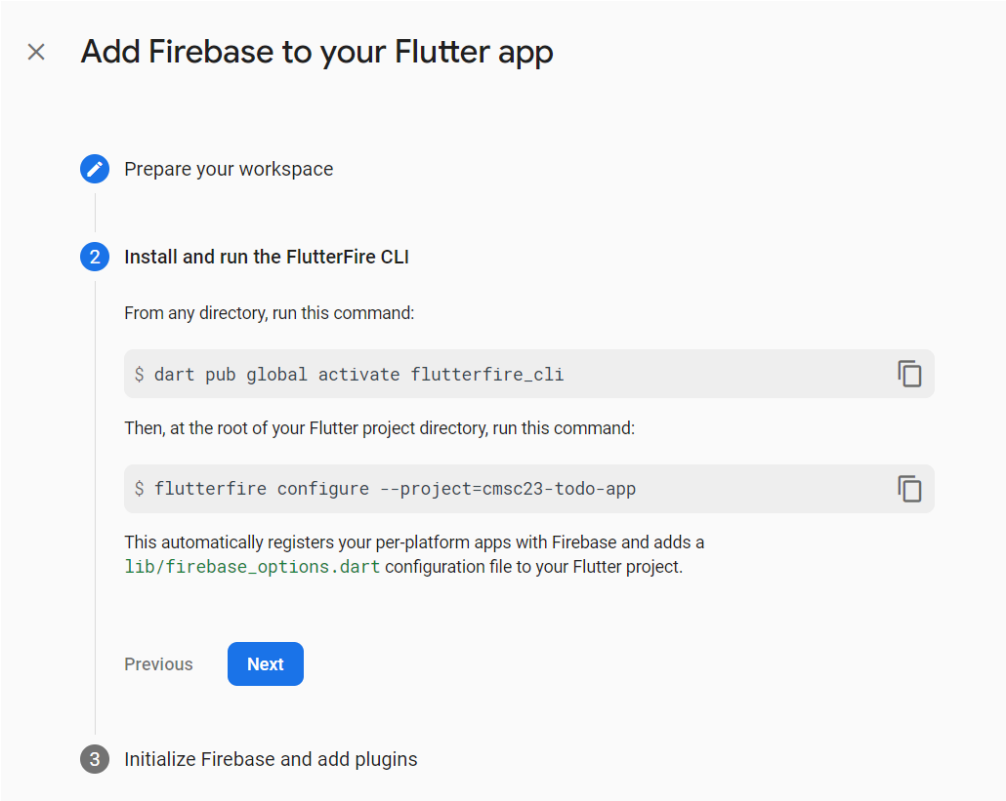
B. In the Firebase console. After selecting your project, click the gear icon on the left panel and then Project settings.



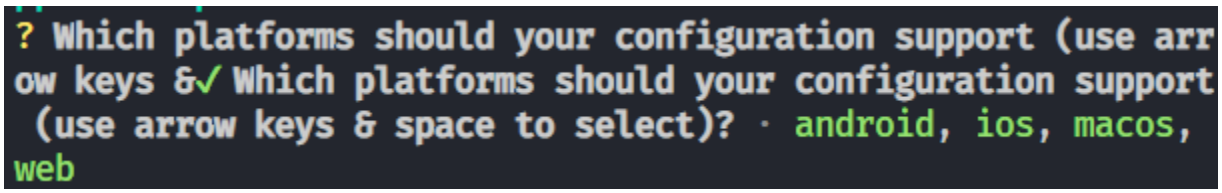
C. Scroll to the bottom and click on the 

button

We've prepared our workspace in previous sessions in **Step 1**. We can proceed with **Step 2** which will add the proper credentials to your app



We can just use the terminal in VS Code to execute these commands. When you are given a prompt asking you to select the platforms to add in your project, just hit enter to enable them all:



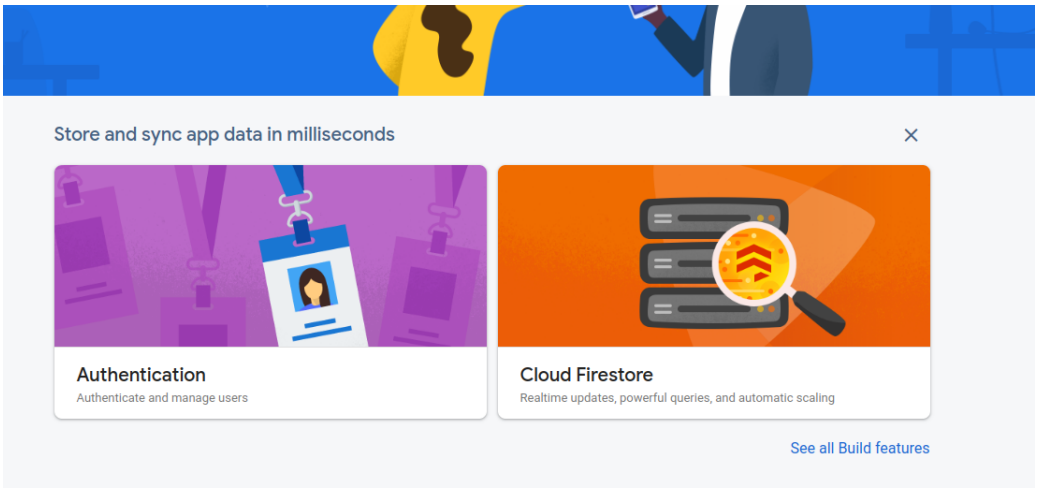
If you see the error below, just add the particular path in your environment variables.

```
Warning: Pub installs executables into C:\Users\Beili\AppData\Local\Pub\
Cache\bin, which is not on your path.You can fix that by adding that dir
ectory to your system's "Path" environment variable.
A web search for "configure windows path" will show you how.
```

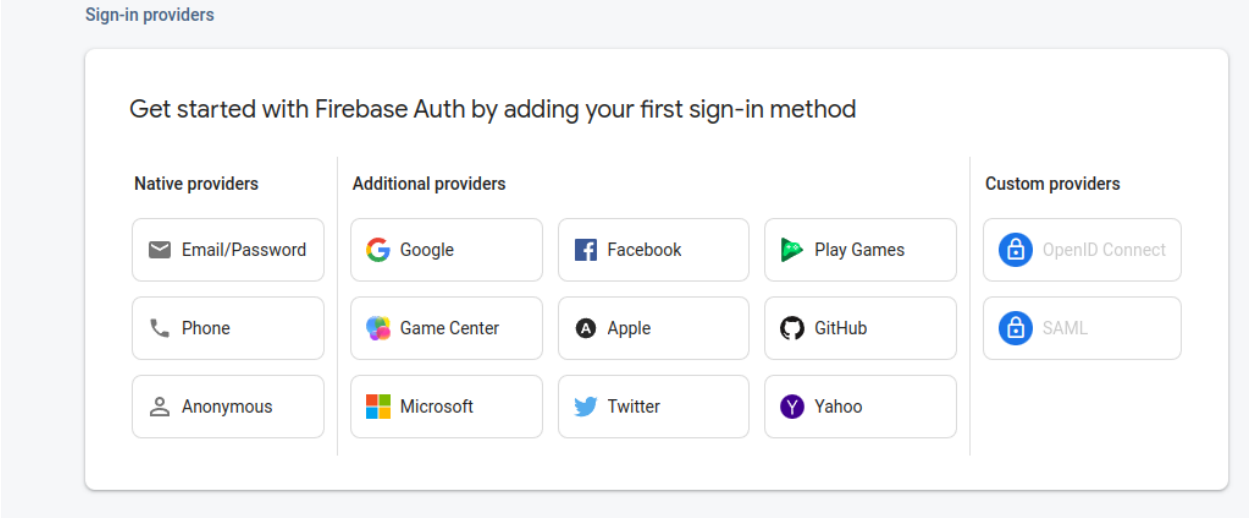
Once the configuration is done, you will notice that another file is added inside lib: **firebase_options.dart**.

II. Set-up Firebase Authentication

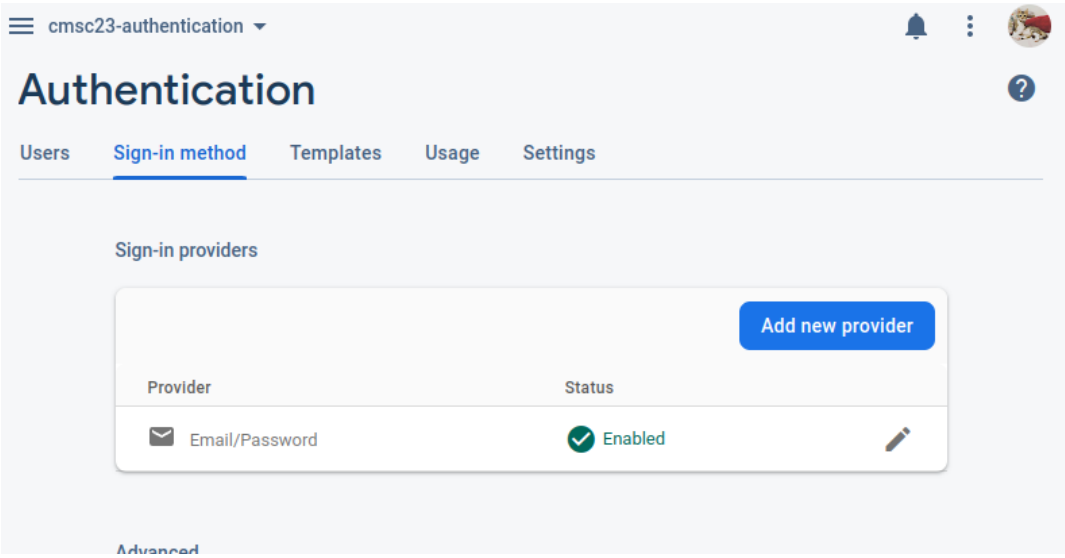
In the Firebase Dashboard, click the Authentication card and then click the “Get Started” button



There are different authentication options and methods available as well as third party authentication providers that you can integrate into our application but for now, we’ll only use the Email/Password sign-in method



After enabling email/password auth. We can check the activated sign-in methods for our project by clicking on the **Sign-in method** tab



III. Create Firebase Authentication Service

In the starter code, cloud_firestore and firebase_core are already added in the list of dependencies in pubspec.yaml so we don't have to add it again. What we need to add is the **firebase_auth** package so we need to enter this command in the terminal

```
flutter pub add firebase_auth
```

A. Firebase Auth API

Using the same design pattern (API <-> Provider<-> Widgets) that we used in the previous session, let's create a separate **firebase_auth_api.dart** in the API folder.

Import the firebase_auth package, create the FirebaseAuthAPI class and create an instance of FirebaseAuth named **auth**

```
import 'package:firebase_auth/firebase_auth.dart';

class FirebaseAuthAPI {
  static final FirebaseAuth auth = FirebaseAuth.instance;

  Stream<User?> getUser() {
    return auth.authStateChanges();
  }


}
```

B. Create the signIn, signOut, and signUp method

```
...
  void signIn(String email, String password) async {
    UserCredential credential;
    try {
      final credential = await auth.signInWithEmailAndPassword(
          email: email, password: password);
    } on FirebaseAuthException catch (e) {
      if (e.code == 'user-not-found') {
        //possible to return something more useful
        //than just print an error message to improve UI/UX
        print('No user found for that email.');
      } else if (e.code == 'wrong-password') {
        print('Wrong password provided for that user.');
      }
    }
  }

  void signUp(String email, String password) async {
    UserCredential credential;
    try {
      credential = await auth.createUserWithEmailAndPassword(
        email: email,
        password: password,
      );
```

```
      if (credential.user != null) {
        saveUserToFirestore(credential.user?.uid, email);
      }
    } on FirebaseAuthException catch (e) {
        //possible to return something more useful
        //than just print an error message to improve UI/UX
      if (e.code == 'weak-password') {
        print('The password provided is too weak.');
      } else if (e.code == 'email-already-in-use') {
        print('The account already exists for that email.');
      }
    } catch (e) {
      print(e);
    }
  }

  void signOut() async {
    auth.signOut();
  }
...
```

C. Save custom user information to Firestore

FirebaseAuth can store basic user information (displayname, profile picture url, etc.) but if we want to store more or custom information about the user, we can save it to Firestore. If **`createUserWithEmailAndPassword`** inside the signUp method above is successful, it will return an instance of the **UserCredential** class which contains information about the new user and then **saveUserToFirestore** method will be called. We then create a new document with the **UserCredential.user.uid** as the document id in Firestore so we can refer to the document by uid when we want to retrieve it.

FirebaseAuthApi class, instantiate Firestore

```
import 'package:cloud_firestore/cloud_firestore.dart';
...

static final FirebaseFirestore db = FirebaseFirestore.instance;
...
  void saveUserToFirestore(String? uid, String email) async {
    try {
      await db.collection("users").doc(uid).set({"email": email});
    } on FirebaseException catch (e) {
      print(e.message);
    }
  }
```

We now have the API calls that we need for signing up,in, and out.

D. Auth Provider

We'll create a provider (**auth_provider.dart**) inside the provider folder, these are the methods that will be called when the user interacts with the UI elements for authentication. The provider will also give the authentication status and information of the user.

```
import 'package:flutter/material.dart';
import 'package:firebase_auth/firebase_auth.dart';
import
'package:week7_networking_discussion/api/firebase_auth_api.dart';

class AuthProvider with ChangeNotifier {
  late FirebaseAuthAPI authService;
  User? userObj;

  AuthProvider() {
    authService = FirebaseAuthAPI();
    authService.getUser().listen((User? newUser) {
      userObj = newUser;
      print('AuthProvider - FirebaseAuth - onAuthStateChanged -
$newUser');
      notifyListeners();
    }, onError: (e) {
      // provide a more useful error
      print('AuthProvider - FirebaseAuth - onAuthStateChanged - $e');
    });
  }

  User? get user => userObj;

  bool get isAuthenticated {
    return user != null;
  }

  void signIn(String email, String password) {
    authService.signIn(email, password);
  }

  void signOut() {
    authService.signOut();
  }

  void signUp(String email, String password) {
    authService.signUp(email, password);
  }
}
```

E. Login and Sign Up Screens

The starter code already has two new widgets for the login and signup screens. All we have to do is call the correct provider methods and pass the required parameters when the user presses the buttons.

login.dart

```
…
      context
          .read<AuthProvider>()
          .signIn(emailController.text, passwordController.text);

…
```

signup.dart

```
…
      context
          .read<AuthProvider>()
          .signUp(emailController.text, passwordController.text);
      Navigator.pop(context);
…
```

F. Which screen to display based on the authentication status

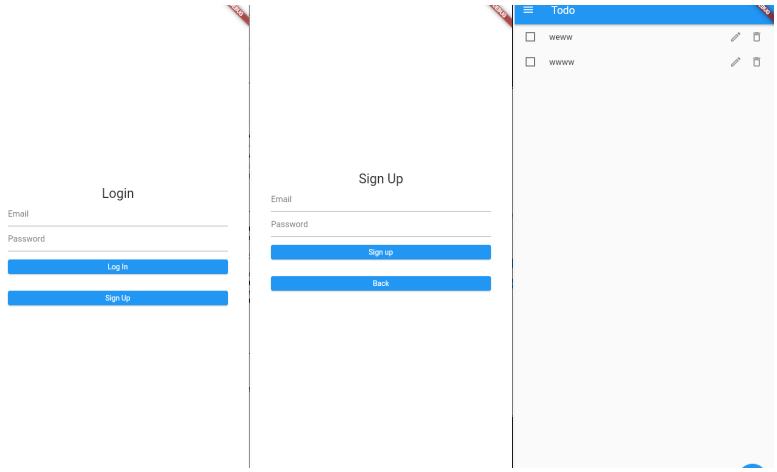
We now have the methods and the UI elements that we need to change the authentication status. The last step that we need to do is to choose which widget is displayed based on the authentication status. In **main.dart**, let's create another widget inside main.dart called **AuthWrapper**, the purpose of this widget is to check the current authentication status by using the provider then display the correct widget when the user is logged in or out.

main.dart

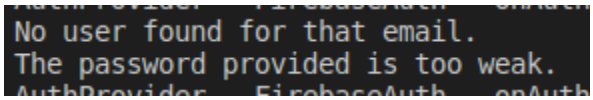
```
…
class AuthWrapper extends StatelessWidget {
  const AuthWrapper({super.key});

  @override
  Widget build(BuildContext context) {
    if (context.watch<AuthProvider>().isAuthenticated) {
      return const TodoPage();
    } else {
      return const LoginPage();
    }
  }
}
```

When the authentication status is updated by signing up, in or out, the provider will be updated and **AuthWrapper** will respond accordingly and display the correct widget.

We should be able to build this flutter app now without errors. The login, and signup screen should be working.

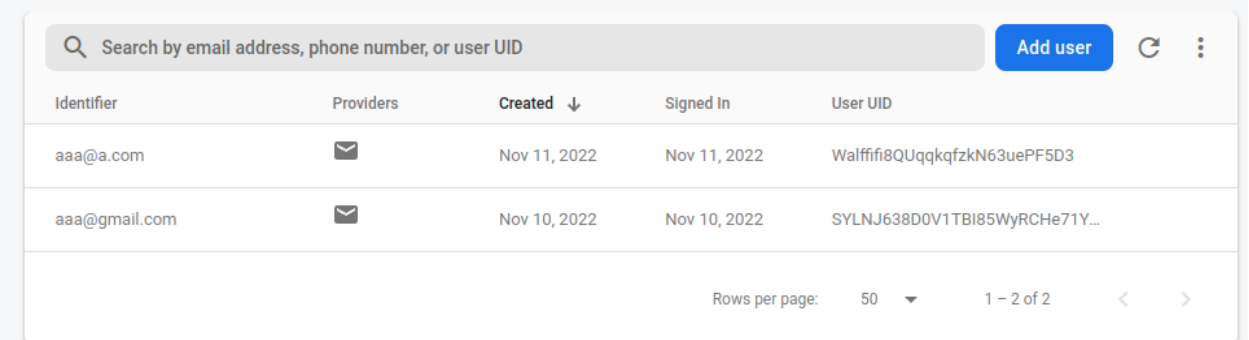


The usual sign up or login errors are only printed in the terminal since we did not add UI validation and proper return statements on the API calls for unsuccessful attempts.

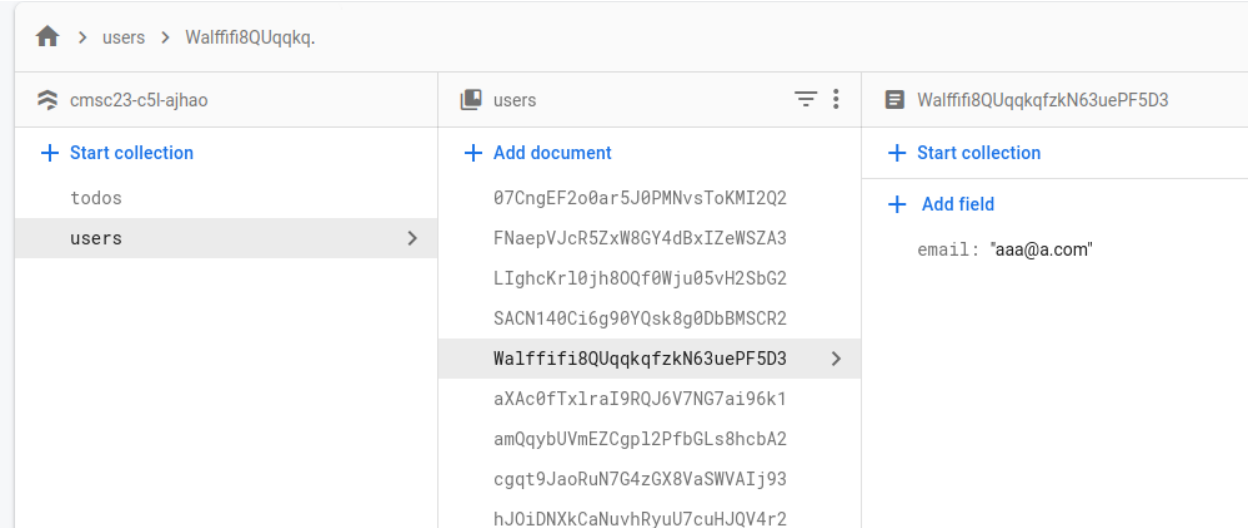


For successful sign ups, new accounts will show up in the Firebase Authentication Dashboard . Take note that, for now, our app will accept any email string as long as it's the correct format. We'll have to implement email verification (send verification email to address) by using the firebase_auth package but nonetheless, we can use the created email to sign in.

Reference: https://firebase.google.com/docs/auth/flutter/start



Finally, in the Firebase Firestore section of the dashboard, a document in the “users” collection with the User UID generated by Google Auth as the document ID will be created which can be retrieved for whatever purpose we’d need it for.



Note: A working branch is also available in the repository but you still need to add the correct configuration and keys to the app by doing **Part I** and **II** of the handout to generate the **firebase_options.dart** file

Introduction to Automated Testing in Flutter

The applications that we are making are getting more complex. The list of things to test will get longer as we add more features and elements to our app. It may lead to us skipping tests since manually testing features that may seem unrelated to the changes that we’ve made can get repetitive. Automated testing can be periodically used to make sure that all is as we expected although it can be bothersome at first since we need to program the use cases that we’ve identified. It is important that we are familiar with creating and running automated tests to make sure that our application behaves how we expect it to in a per unit and widget level and a system as a whole.

Levels of testing

- A **unit test** tests a single function, method, or class.
- A **widget test** (in other UI frameworks referred to as component test) tests a single widget.
- An **integration test** tests a complete app or a large part of an app.

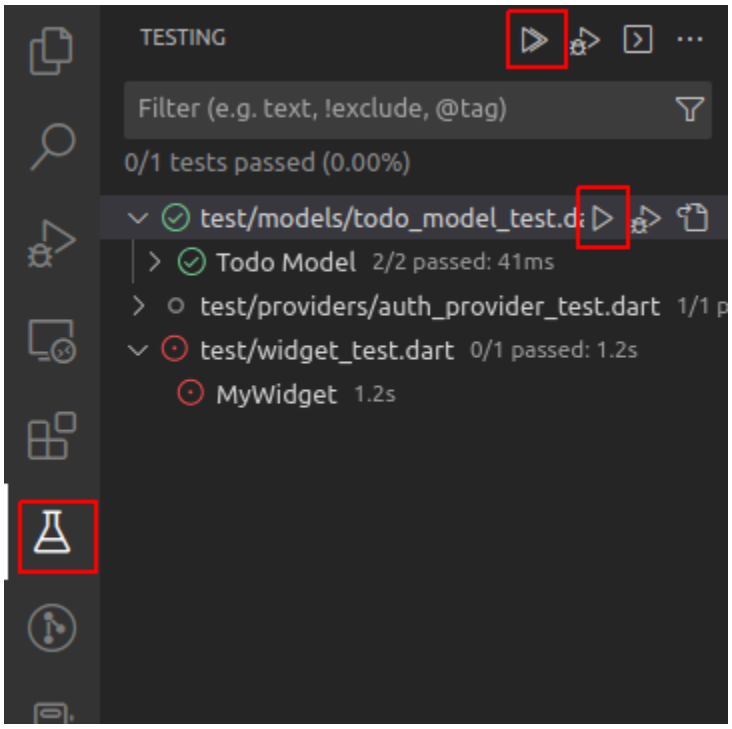
Unit Tests

Let's do a unit test for the existing Todo Model. The Todo model currently doesn't do much but the methods should be enough to demonstrate what unit tests are. Test files are organized the same way as the files in the **lib** directory so inside the **test** folder, create a **models** folder and then a **todo_model_test.dart** file.
*the file must be _test -> to be recognized as a unit test in VSCode.

```
import 'package:flutter_test/flutter_test.dart';
import 'package:week7_networking_discussion/models/todo_model.dart';

void main() {
  group("Todo Model", () {
    test('Test Todo Model constructor', () {
      final modelInstance =
          Todo(userId: 1, completed: false, title: "Test Todo");
      expect(modelInstance.userId, 1);
      expect(modelInstance.completed, false);
      expect(modelInstance.title, "Test Todo");
    });

    test('Test Todo Model toJson method', () {
      final modelInstance =
          Todo(userId: 1, completed: false, title: "Test Todo");

      // do something
      final converted = modelInstance.toJson(modelInstance);

      //test the actual vs the expected
      expect(
          converted, {"userId": 1, "title": "Test Todo", "completed":
false});
    });
  });
}
```

You should be familiar with how each test file looks since it's just a Dart file with some imports and a **main()** program. Unit tests are done using the **test()** method, the method requires a string which will be used as a label or the description of the test for the first parameter and a method that contains what the test will actually do. Although not required, tests can be put into groups for better organization but it's not required.

To run the test in VS Code click the flask icon for Tests on the panel on the left and the test file tree will appear.



To run all the test files you've made, click on the play/run icon at the top. To run individual tests, click the play icon on the right of the filename. The tree shows individual test files with the icon indicating whether the test passed or failed in the previous run.

Widget Tests

We can check what widgets are displayed, what buttons, texts, forms etc. are displayed by using widget tests. But first, we have to modify our code since we can't automate the test of our app while it's actually using Firebase Authentication and Firestore, we have to install a mock Firebase and Firestore for our automated tests.

Let's just run the couple of commands on the terminal to add the mock packages

```
flutter pub add fake_cloud_firestore
```

```
flutter pub add firebase_auth_mocks
```

We'll also comment out this part in **main.dart** so the app won't initialize the real Firebase package

```
  WidgetsFlutterBinding.ensureInitialized();
  // await Firebase.initializeApp(
  //   options: DefaultFirebaseOptions.currentPlatform,
  // );
```

In both the API files, let's import and instantiate the mock packages instead of the actual ones.

firebase_auth_api.dart

```
import 'package:fake_cloud_firestore/fake_cloud_firestore.dart';
import 'package:firebase_auth_mocks/firebase_auth_mocks.dart';

…

 // static final FirebaseAuth auth = FirebaseAuth.instance;
  // static final FirebaseFirestore db = FirebaseFirestore.instance;
  final db = FakeFirebaseFirestore();

  final auth = MockFirebaseAuth(
    mockUser: MockUser(
    isAnonymous: false,
    uid: 'someuid',
    email: 'charlie@paddyspub.com',
    displayName: 'Charlie',
  ));
```

firebase_todo_api.dart

```
import 'package:fake_cloud_firestore/fake_cloud_firestore.dart';
…
// static final FirebaseFirestore db = FirebaseFirestore.instance;
final db = FakeFirebaseFirestore();
```

You can now run the app again, this time it connects to the mock Firebase Authentication and Firestore instances.

Now that we're not using the actual Firebase Authentication and Firestore packages, we can now run automated tests as much as we want on the login widget by creating a **login_widget_test.dart** file in the **test** folder

```
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
import 'package:week7_networking_discussion/main.dart' as app;

void main() {
  // Define a test
  testWidgets('Test Login Widget', (tester) async {
    // Create the widget by telling the tester to build it along with the
provider the widget requires

    app.main();
//find the widgets by the text or by their keys
    final screenDisplay = find.text('Login');
    final userNameField = find.byKey(const Key("emailField"));
    final passwordField = find.byKey(const Key("pwField"));
    final loginButton = find.byKey(const Key("loginButton"));
    final signUpButton = find.byKey(const Key("signUpButton"));

// Use the `findsOneWidget` matcher provided by flutter_test to
// verify that the Text widgets and Button widgets appear exactly once in
the widget tree.
    expect(screenDisplay, findsOneWidget);
    expect(userNameField, findsOneWidget);
    expect(passwordField, findsOneWidget);
    expect(loginButton, findsOneWidget);
    expect(signUpButton, findsOneWidget);

    await tester.tap(signUpButton);
    final signupDisplay = find.text("Sign Up");
    expect(signupDisplay, findsOneWidget);
  });
}
```

We instantiated **MyApp** by calling **app.main().** The next statements find specific widgets (textfield, button, etc.) by their keys. The **expect** method is also called which checks what the **find** method returns against the statement **findsOneWidget** which asserts that there is exactly one of that widget in the widget tree.

User actions can also be simulated by using the method **tester.tap(widget)**. In the emulator, manually tapping the **sign up** button brings up the sign up screen. The last three statements in the code simulates that action and checks whether the correct screen and widget would be displayed.

Run the test by clicking the same button that we used when we ran the unit tests and see what happens. You can also try misspelling the widget keys and the “Sign Up” string that we’re trying to find and see what happens.

Integration Tests

Integration tests or end to end tests are automated tests that run the application as a whole in the device or emulator. We’ll use and expand the code for **login_widget_test.dart** to create an integration test.

First, we have to enable the integration_test package included in the Flutter SDK by adding the following lines to **pubspec.yaml** (.yaml files are sensitive to indentation, don’t add/remove spaces before each line)

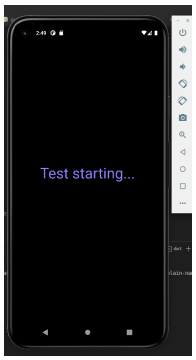
```
dev_dependencies:
  integration_test:
    sdk: flutter
  flutter_test:
    sdk: flutter
```

Create a new directory (**integration_test**) in the app root directory and an **app_test.dart** file inside it. Copy the contents of the **login_widget_test.dart** and add the following lines.

```
import 'package:flutter_test/flutter_test.dart';
import 'package:integration_test/integration_test.dart';
import 'package:week7_networking_discussion/main.dart' as app;
import 'package:flutter/material.dart';

void main() {
  IntegrationTestWidgetsFlutterBinding.ensureInitialized();
  group('end-to-end test', () {
    testWidgets('Test Signup button', (tester) async {
      app.main();
      await tester.pumpAndSettle();

      final screenDisplay = find.text('Login');
      final userNameField = find.byKey(const Key("emailField"));
      final passwordField = find.byKey(const Key("pwField"));
      final loginButton = find.byKey(const Key("loginButton"));
      final signUpButton = find.byKey(const Key("signUpButton"));

// Use the `findsOneWidget` matcher provided by flutter_test to
// verify that the Text widgets and Button widgets appear exactly
once in the widget tree.
      expect(screenDisplay, findsOneWidget);
      expect(userNameField, findsOneWidget);
```

```
      expect(passwordField, findsOneWidget);
      expect(loginButton, findsOneWidget);
      expect(signUpButton, findsOneWidget);

      await tester.tap(signUpButton);
      await tester.pumpAndSettle();

      final signupDisplay = find.text("Sign Up");
      expect(signupDisplay, findsOneWidget);
    });
  });
}
```

Finally, make sure you are connected to the emulator and type **flutter test integration_test/app_test.dart** in the terminal to run the test. The SDK will build an .apk for testing that will be installed in the device/emulator just like when we normally build it. If you have the sound on, you can actually hear the taps and clicks that we simulated just like when we're actually tapping on our device or clicking using the mouse.



References:

- https://docs.flutter.dev/testing
- https://firebase.flutter.dev/docs/testing/testing/
- Flutter for Beginners: An introductory guide to building cross-platform mobile applications with Flutter and Dart 2