

Name: Allan Rodrigues
Class: TE IT A
Roll no: 59
Pid:191104

St. Francis Institute of Technology, Mumbai-400 103
Department of Information Technology

A.Y. 2021-2022
Class: TE-ITA/B, Semester: VI
Subject: **MAD & PWA LAB**

Experiment – 6: Connecting Flutter UI with firebase database.

1. **Aim:** To connect Flutter UI with firebase database.
2. **Objectives:** After study of this experiment, the student will be able to
 - Create a production ready Flutter App by including files and firebase backend service.
3. **Outcomes:** After study of this experiment, the student will be able to
 - Analyze and Build production ready Flutter App by incorporating backend services and deploying on Android/iOS
4. **Prerequisite:** Dart Programming Language.
5. **Requirements:** Android Studio, Flutter framework, Internet Connection.

6. Pre-Experiment Exercise:

Brief Theory:

Firebase is a platform developed by Google for creating mobile and web applications. Firebase is a set of server-side services and tools. If you use Firebase, you don't need to buy or rent your own server. Firebase is made up of over a dozen tools. Let's glance at these three:

- Cloud Firestore – A database with an API to read and write data
- Cloud Functions – Logic that is kicked off by an API call
- Authentication – Single sign-on to allow users to securely log in to your app using their social accounts or a username/password combination

Steps to connect UI with firebase database:

1. Visit <http://firebase.google.com> to register an app with Firebase.
2. Create a Firebase Project.
3. Create the database by using the option given on your project's dashboard.
4. Create an Android app.
 - a. Going through the next step while creating an android app, the wizard will create a google-services.json file and will tell you where to save it.

5. Add FlutterFire plugins.

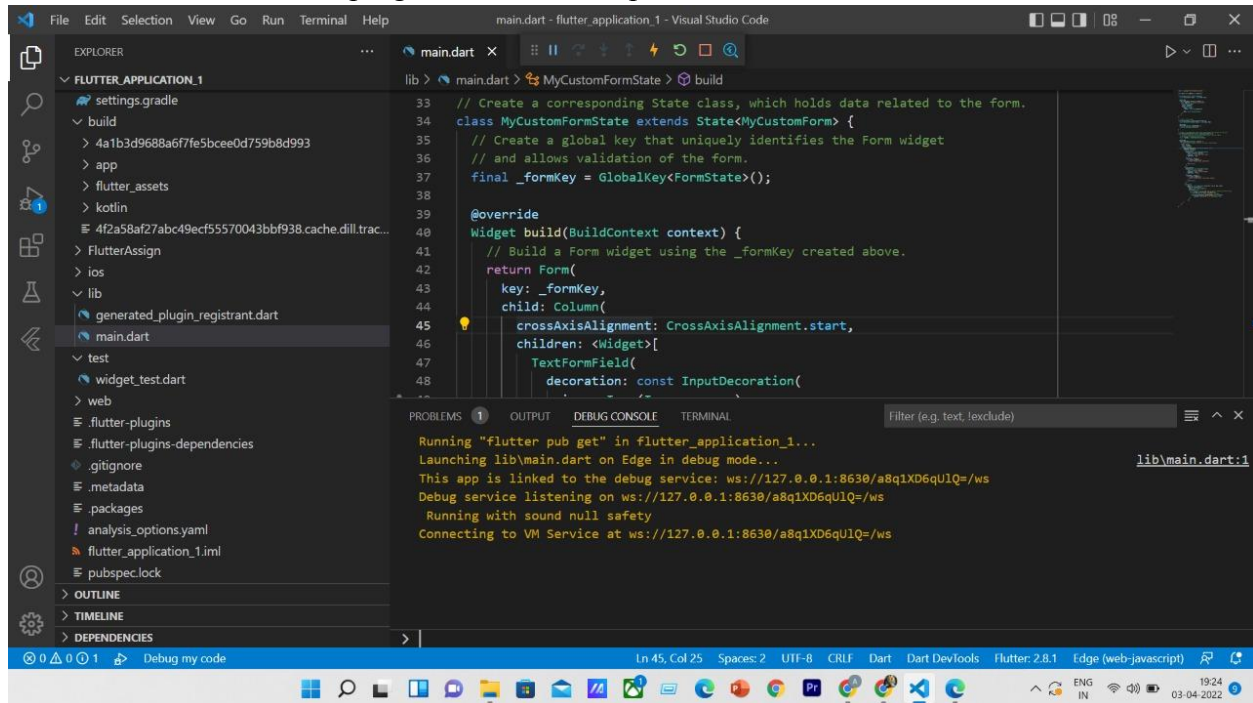
7. Laboratory Exercise

A. Program

1. Connect your UI with the firebase database and show the values entered in UI being reflected in the database.

B. Result/Observation

1. Print out of program code and output.



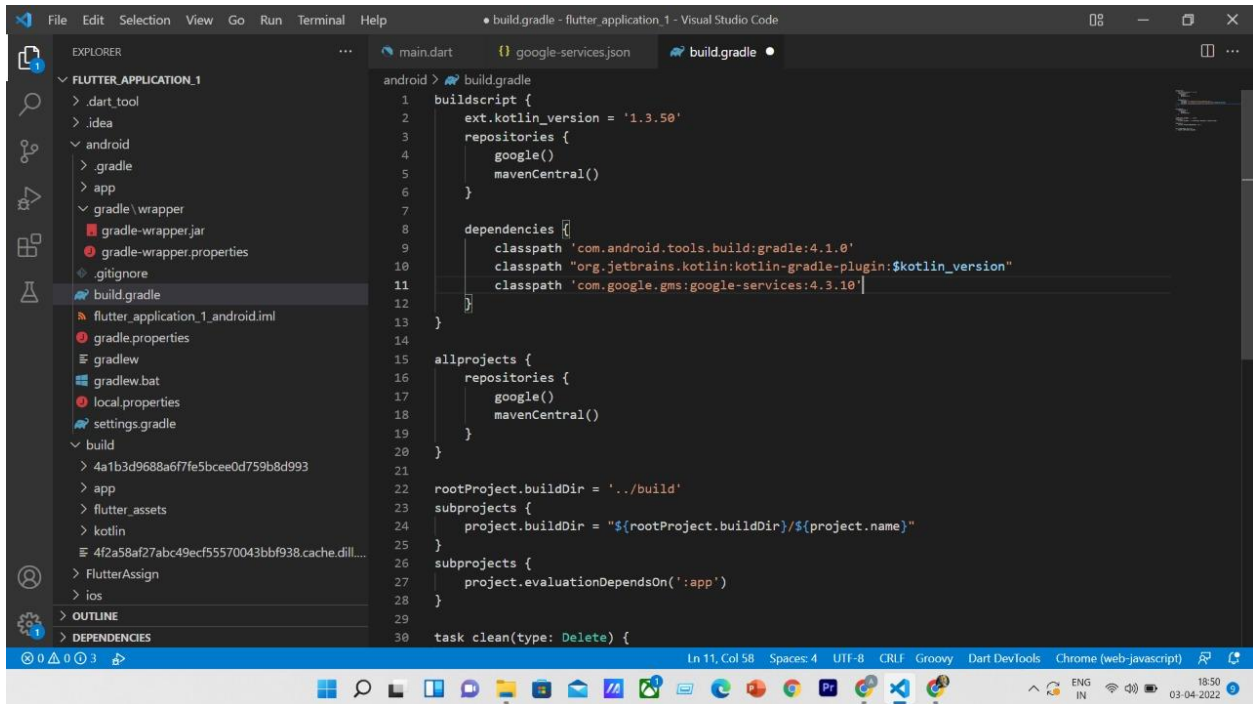
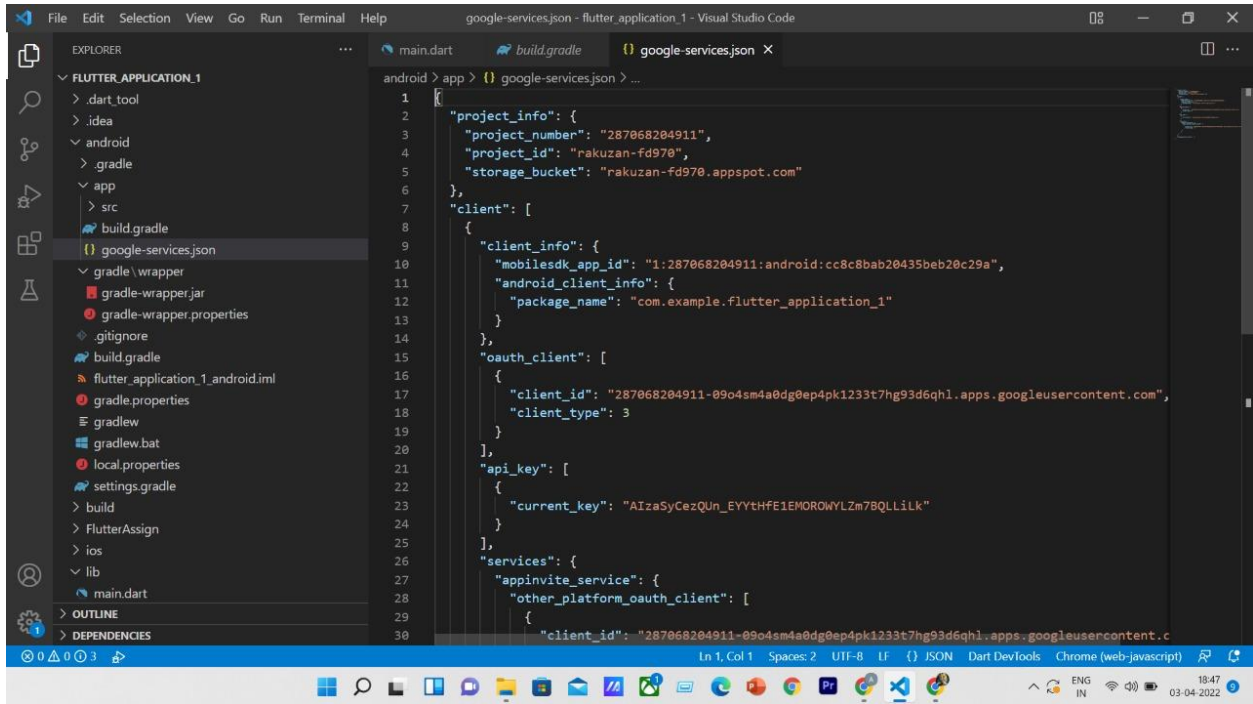
The screenshot displays the Visual Studio Code interface for a Flutter application. The Explorer panel on the left shows the project structure, including files like `settings.gradle`, `build`, `app`, `flutter_assets`, `kotlin`, `FlutterAssign`, `ios`, `lib`, `generated_plugin_registrant.dart`, `main.dart`, `test`, `widget_test.dart`, `web`, `.flutter-plugins`, `.flutter-plugins-dependencies`, `.gitignore`, `.metadata`, `.packages`, `analysis_options.yaml`, `flutter_application_1.iml`, and `pubspec.lock`. The main editor shows the `main.dart` file with the following Dart code:

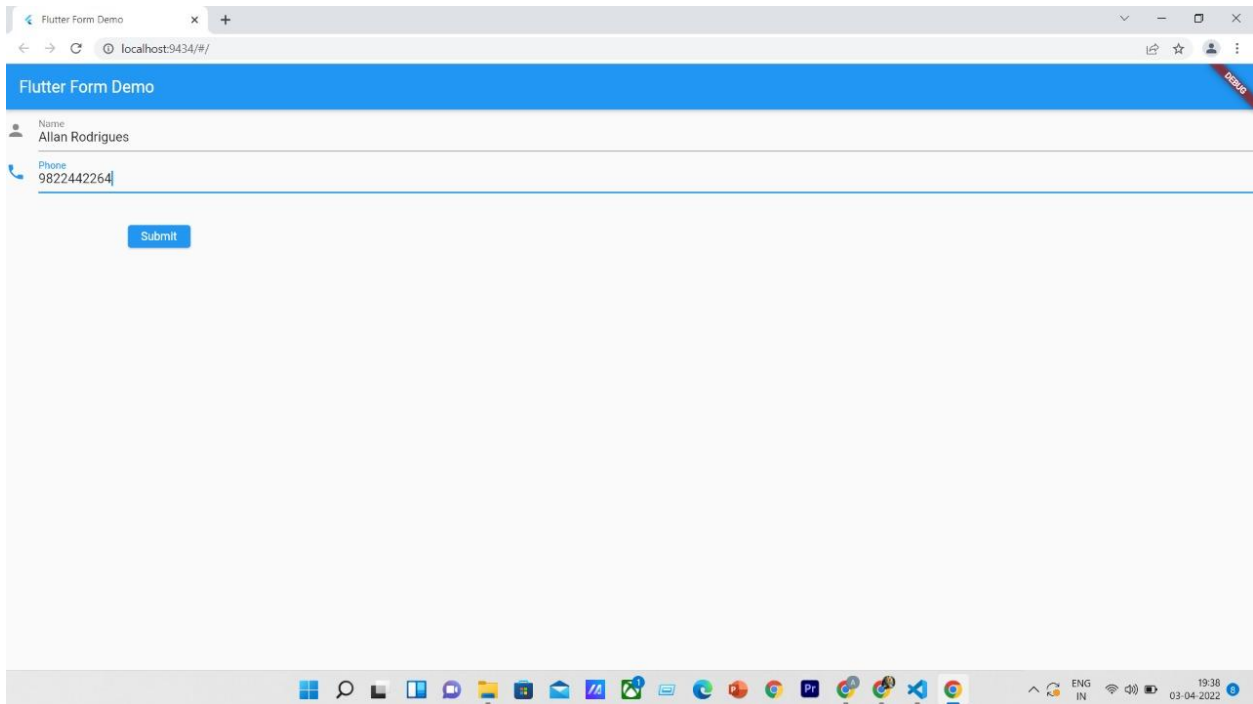
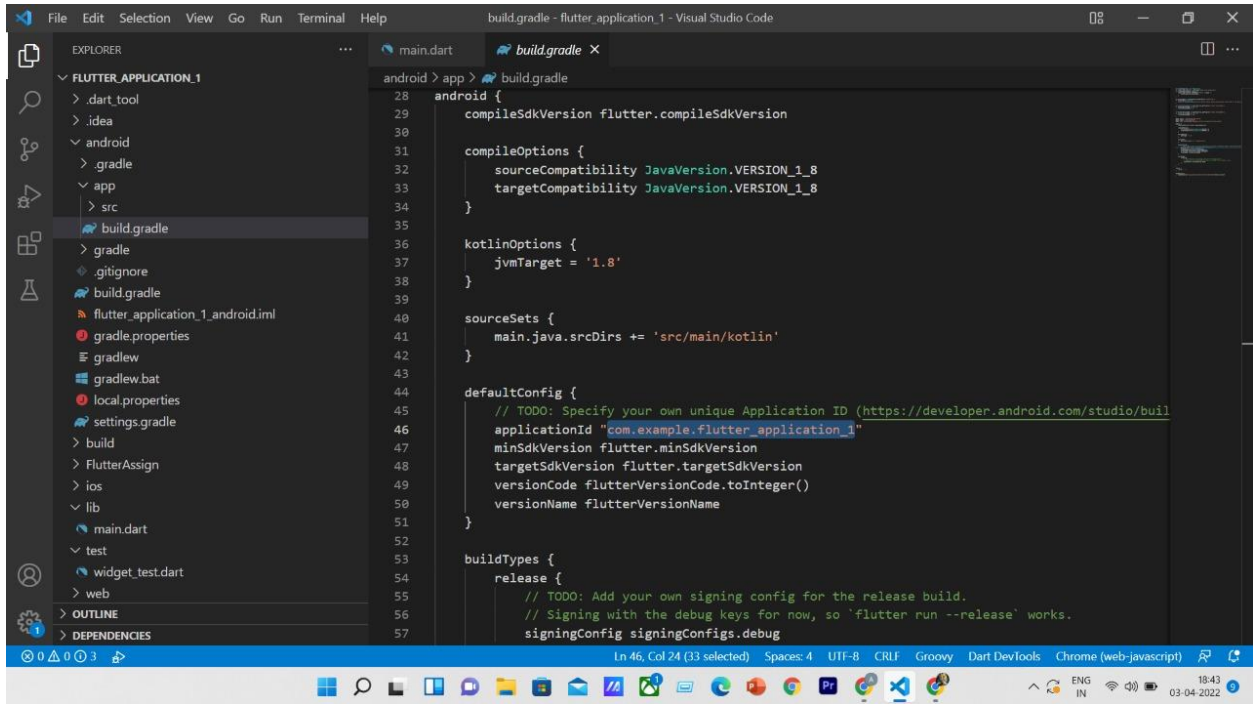
```
33 // Create a corresponding State class, which holds data related to the form.
34 class MyCustomFormState extends State<MyCustomForm> {
35   // Create a global key that uniquely identifies the Form widget
36   // and allows validation of the form.
37   final _formKey = GlobalKey<FormState>();
38
39   @override
40   Widget build(BuildContext context) {
41     // Build a Form widget using the _formKey created above.
42     return Form(
43       key: _formKey,
44       child: Column(
45         crossAxisAlignment: CrossAxisAlignment.start,
46         children: <Widget>[
47           TextFormField(
48             decoration: const InputDecoration(
```

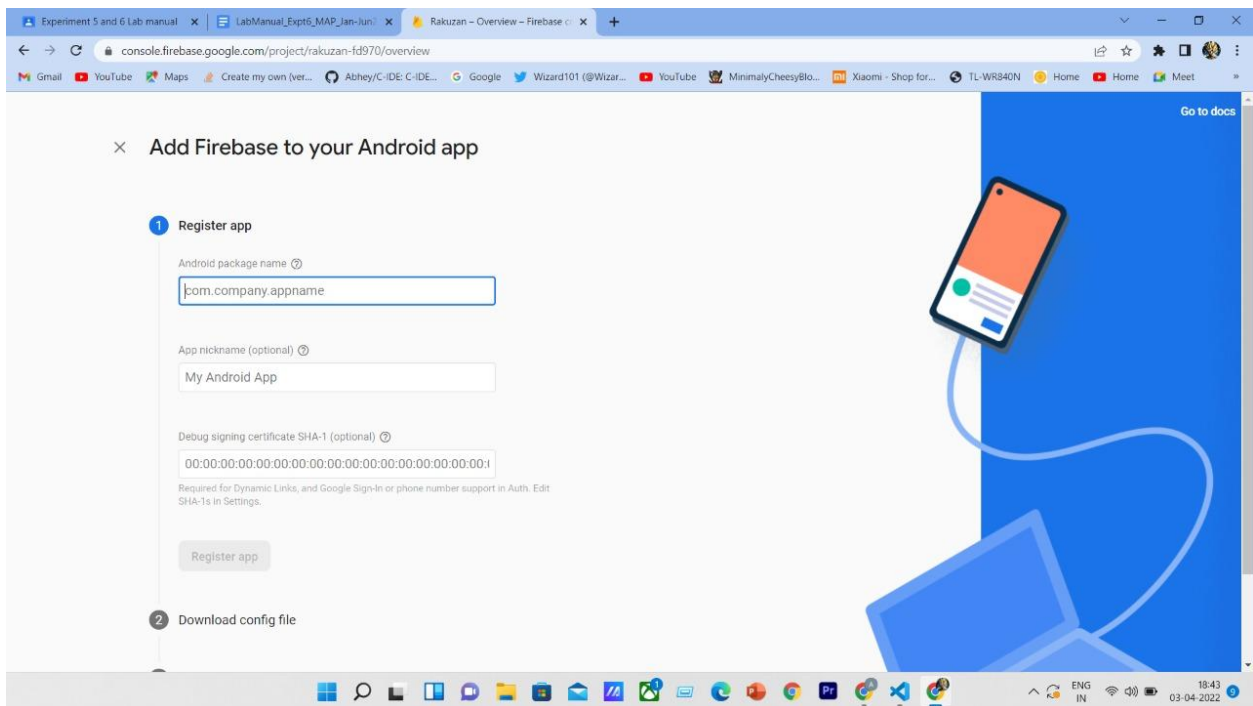
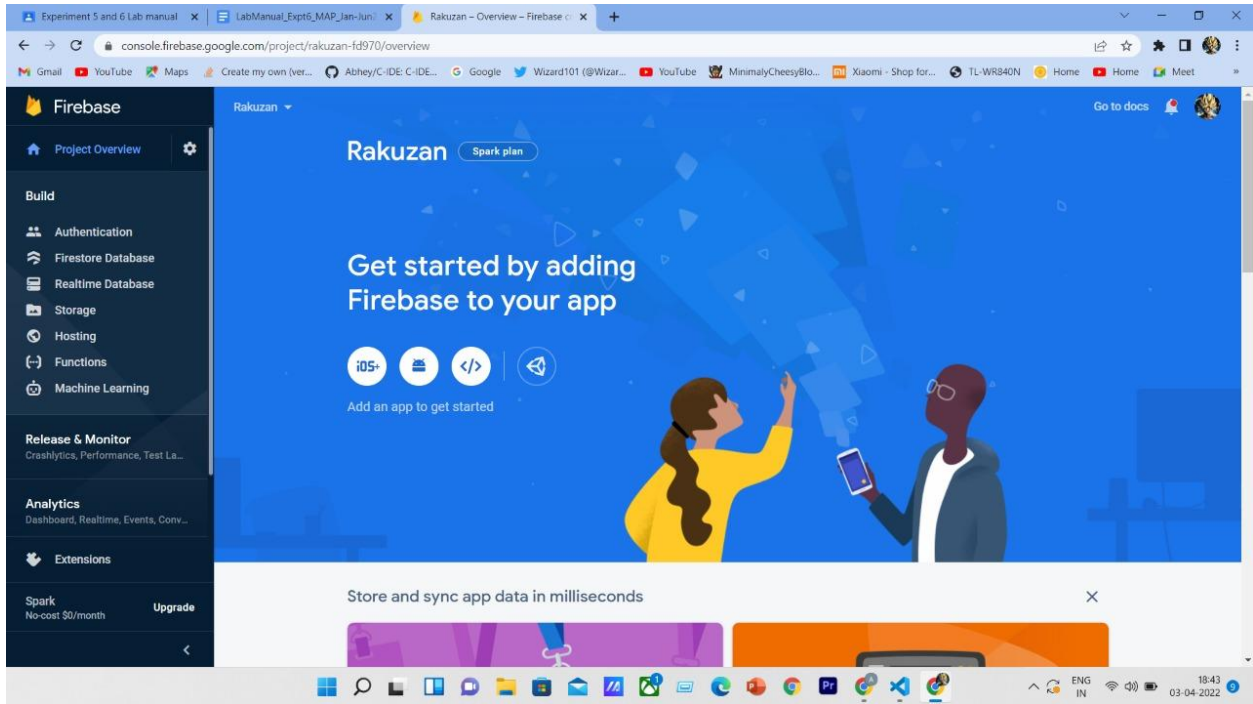
The Debug Console at the bottom shows the output of running the application:

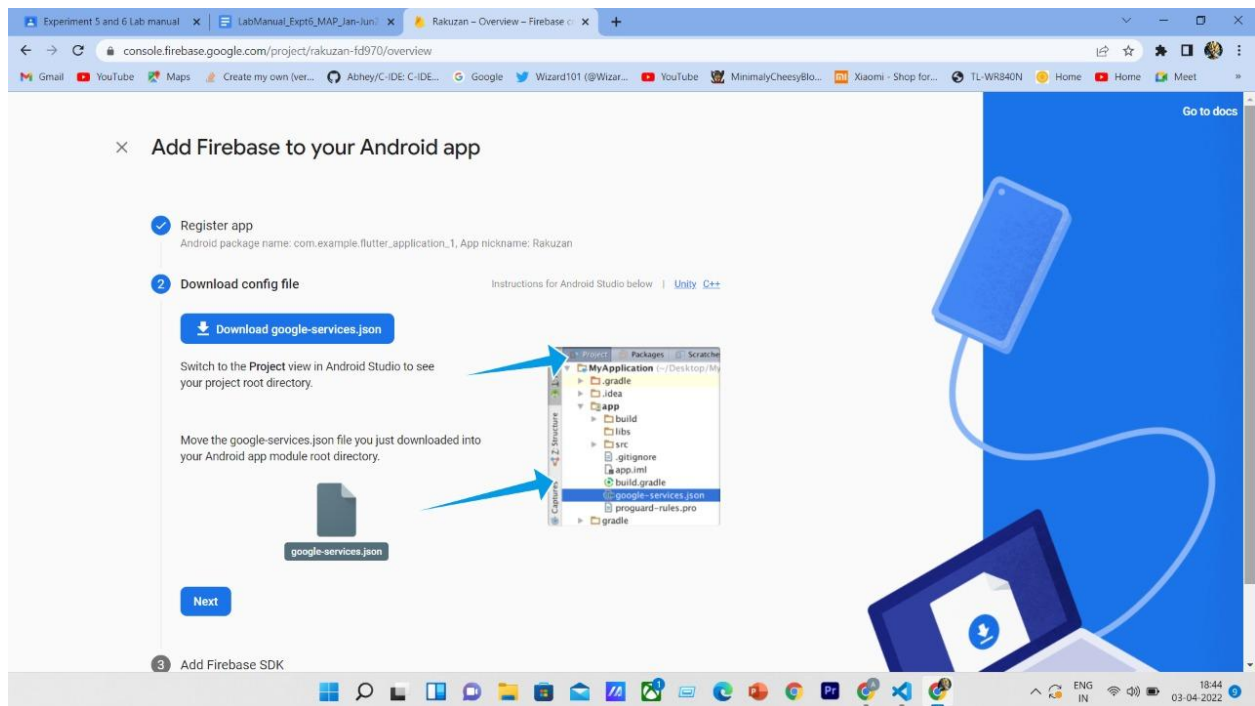
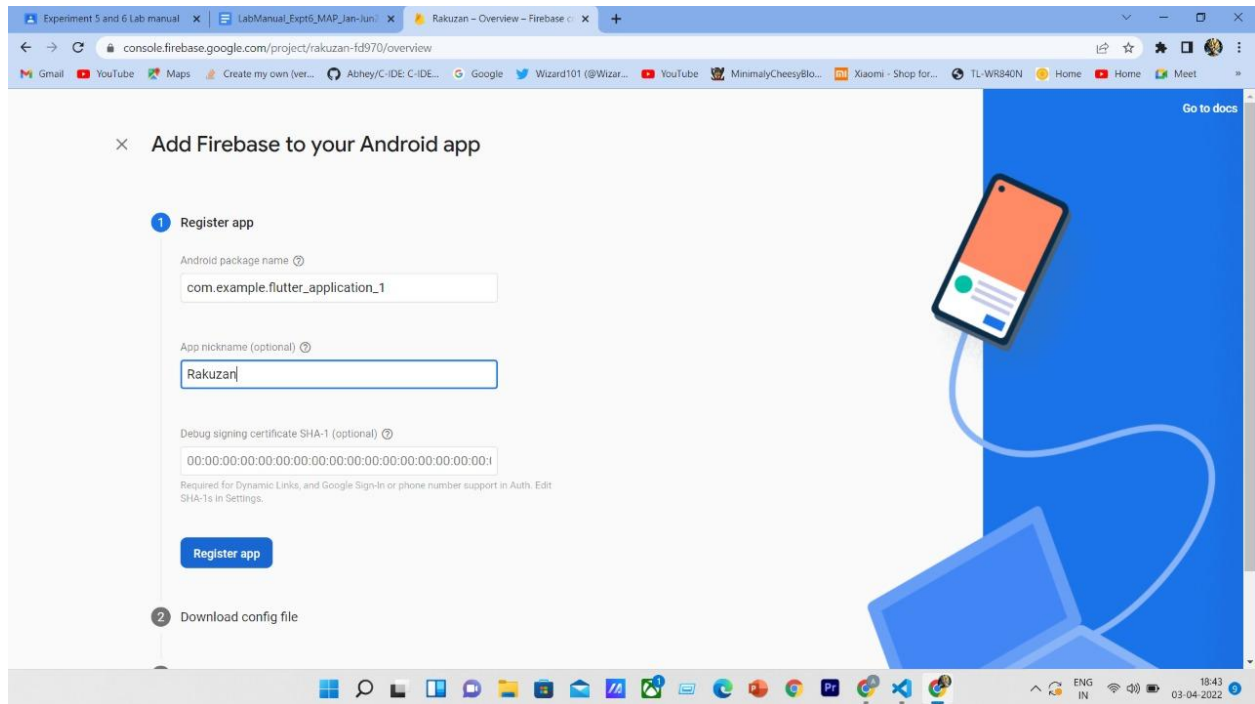
```
Running "flutter pub get" in flutter_application_1...
Launching lib/main.dart on Edge in debug mode...
This app is linked to the debug service: ws://127.0.0.1:8630/a8q1XD6qU1Q=/ws
Debug service listening on ws://127.0.0.1:8630/a8q1XD6qU1Q=/ws
Running with sound null safety
Connecting to VM Service at ws://127.0.0.1:8630/a8q1XD6qU1Q=/ws
```

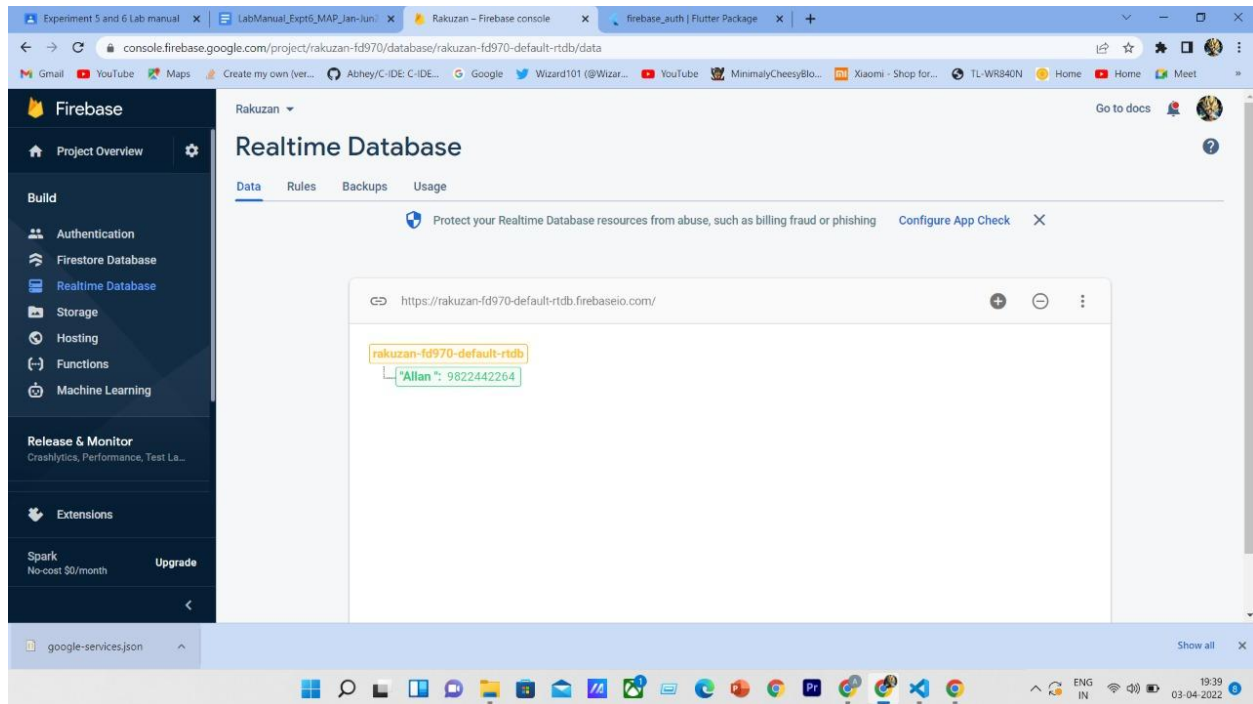
The status bar at the bottom indicates the current file is `Ln 45, Col 25`, using `UTF-8` encoding with `CRLF` line endings, and the Flutter version is `Flutter: 2.8.1`.









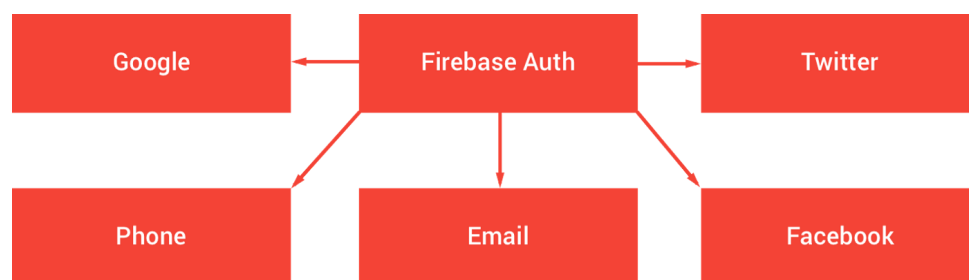


8. Post-Experiments Exercise

A. Questions:

1. What are the other different databases that you can use to connect your UI?
2. Demonstrate the use of firebase Authentication using UI.

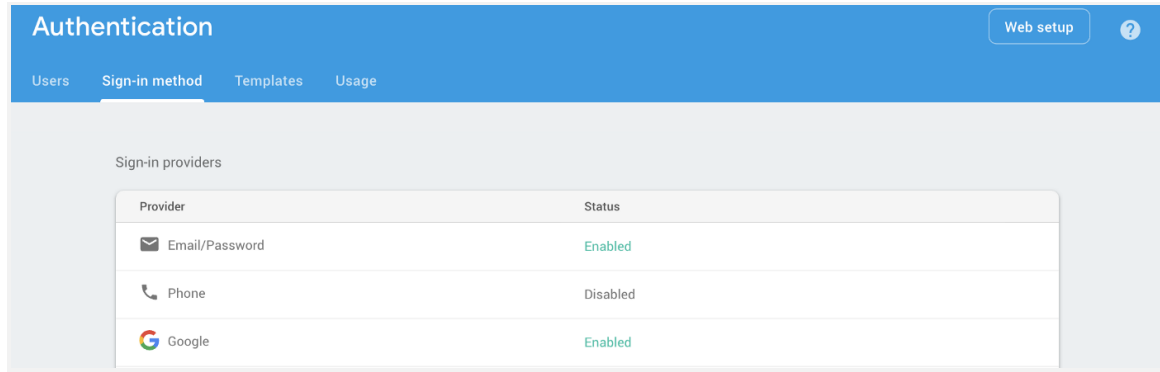
When it comes to implementing Firebase UI, we need to begin by declaring the supported methods of authentication. Firebase will then show a button for each of these methods within the launched authentication screen. The current methods of authentication that Firebase support via its UI library are email, phone, Google, Twitter and Facebook.



Once you have decided the authentication methods which are to be accepted by your application, you can create a list of them ready for use with the Firebase Auth UI builder:

```
val providers = arrayListOf(
    AuthUI.IdpConfig.EmailBuilder().build(),
    AuthUI.IdpConfig.PhoneBuilder().build(),
    AuthUI.IdpConfig.GoogleBuilder().build(),
    AuthUI.IdpConfig.FacebookBuilder().build(),
)
```

```
AuthUI.IdpConfig.TwitterBuilder().build()  
)
```



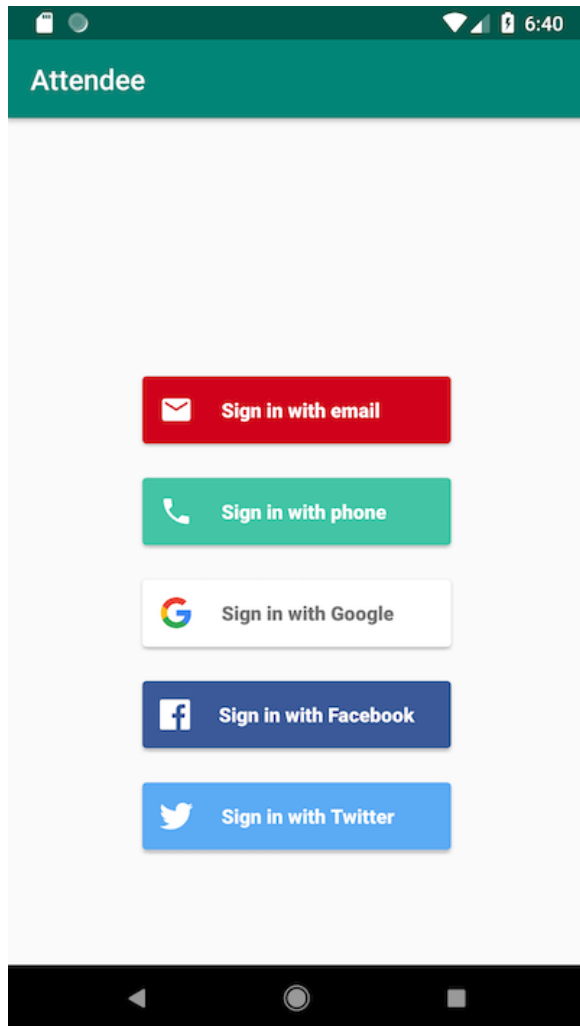
Triggering the Authentication flow

Now that we have our supported authentication methods, we can go ahead and make use of them when launching our authentication screen. We'll begin here by declaring a request code — we need to do this so that once the authentication process has completed and Firebase Authentication returns data to our Activity, we have a way of picking up this data via a unique request code.

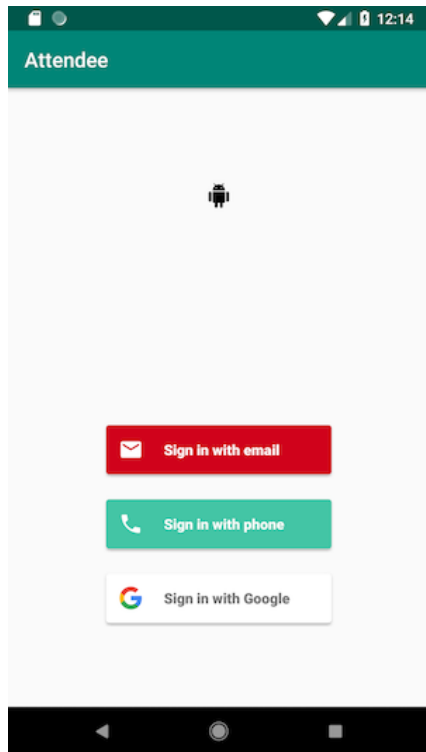
```
private val RC_SIGN_IN = 1822
```

The Firebase Authentication UI library contains an activity which handles the authentication process for us — this means that we need to a) launch this activity and b) listen for the result that is to be returned by it using our unique request code. For this, we'll make use of the `createSignInIntentBuilder()` from our `AuthUI` instance — triggering the intent when our desired button is pressed.

If you launch your activity and hit the authentication button, then you will be shown a screen similar to this, listing the available authentication options:



We can also further customise this screen to display additional content alongside the authentication buttons. For example, we may want to show a logo to add a touch of our brand to the authentication screen. We can do so by making use of the **setLogo()** function on our **AuthUI** instance.

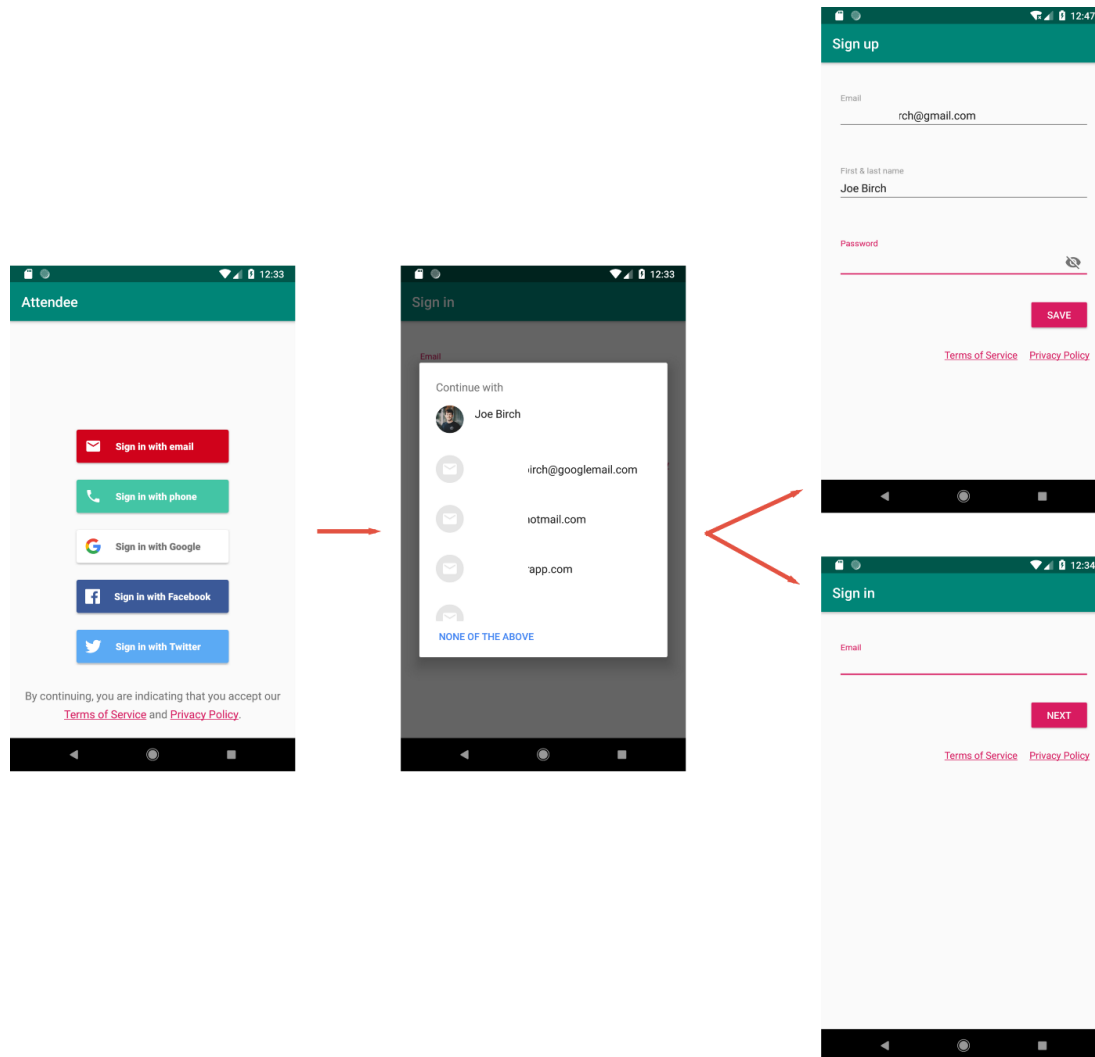


```
private fun setupAuthenticateListener() {  
    button_authenticate.setOnClickListener {  
        startActivityForResult(  
            AuthUI.getInstance()  
                .createSignInIntentBuilder()  
                .setAvailableProviders(providers)  
                .setLogo(R.drawable.ic_logo)  
  
                .build(),  
  
            RC_SIGN_IN  
        )  
    }  
}
```

We can also set a theme to be used for Firebase UI authentication screens. By default your application theme will be used (such as the primary color etc). But if you want further customisation, then you can do so by using the **setTheme()** function on the AuthUI instance.

Email Authentication

Email authentication allows our users to sign up / in to our service via the use of a chosen email address. When it comes to Firebase UI we get presented with a simple flow which allows our users to make use of any Google accounts which may have been connected on the device:



After hitting the **sign in with email** button the email authentication screen is launched. At this point an overlay is displayed which lists the auto-fill accounts on the mobile device — this allows the users to pre-fill **email** and **name** fields automatically from one of these accounts. Selecting **None of the above** will take the user to enter these pieces of information one-by-one.

You'll also notice here that the service and policy links that were originally set are also displayed here. Once the user continues and saves their data they will be returned to the main authentication screen where you will need to handle the authentication response inside of `onActivityResult()`.


B. Conclusion:

1. Write what you have learnt in the experiment.

C. References:

1. Beginning App Development with Flutter: Create Cross-Platform Mobile Apps, By Rap Payne, 2019.
2. Google Flutter Mobile Development Quick Start Guide, Packt Publishing, 2019.

Alfon Rodriguez TE IT M4

 **ST. FRANCIS INSTITUTE OF TECHNOLOGY**
(Engineering College)

DATE : _____ PAGE NO. : _____

Exp 6 - MAP.

Q.8

A. 1) The database landscape for flutter is still very limited. The databases other than Firebase realtime DB are:

1) Hive:

It is lightweight key-value database written in Dart for flutter application, inspired by Bitcask.

2) ObjectBox: It is a highly performant lightweight NoSQL database with an integrated Data Sync.

3) sqflite is a wrapper around SQLite, which is a relational database without direct support for Dart ~~Support~~ Objects.

4) Moor is a reactive persistence library for flutter & Dart, built on top of sqflite.

B. Conclusion.

In this experiment we learnt how to connect flutter UI with Firebase realtime database. After that we can create a production ready flutter App by incorporating backend service Firebase.