MODUL PEMROGRAMAN MOBILE

CIM 430

PERTEMUAN 9
CLOUD FIRESTORED

Dosen Pengampu :
8126 - JEFRY SUNUPURWA ASRI, S.KOM, M.KOM
7174 - SAWALI WAHYU, S.KOM, M.KOM

UNIVERSITAS ESA UNGGUL

FAKULTAS ILMU KOMPUTER

TAHUN 2021

Firebase

Building apps using **Firebase** is one of the fastest-growing technology trends in the world. Using Firebase, developers can build apps at a rapid pace without managing the infrastructure, including authentication, storing and syncing data, securely hosting web assets, and cloud storage. Firebase has a base plan that is free, allowing 1 GB the storage and 100 simultaneous connections. If you wish to upgrade, you can check out the plans here: https://firebase.google.com/

Connecting with Firebase

Let's first take a look at how to connect with Firebase. We will first need to ensure whether the connections to the Firebase are made properly; to do so, follow these steps:

1. Create a new Flutter project in your IDE or editor
2. Open the file pubspec.yamlfile
3. Add the following dependency: **a**

```
dependencies: flutter:

    sdk: flutter

  cloud_firestore: ^0.9.5+2                //Add this line
```

4. Next, to make your connection, in your IDE or using command line run the following:

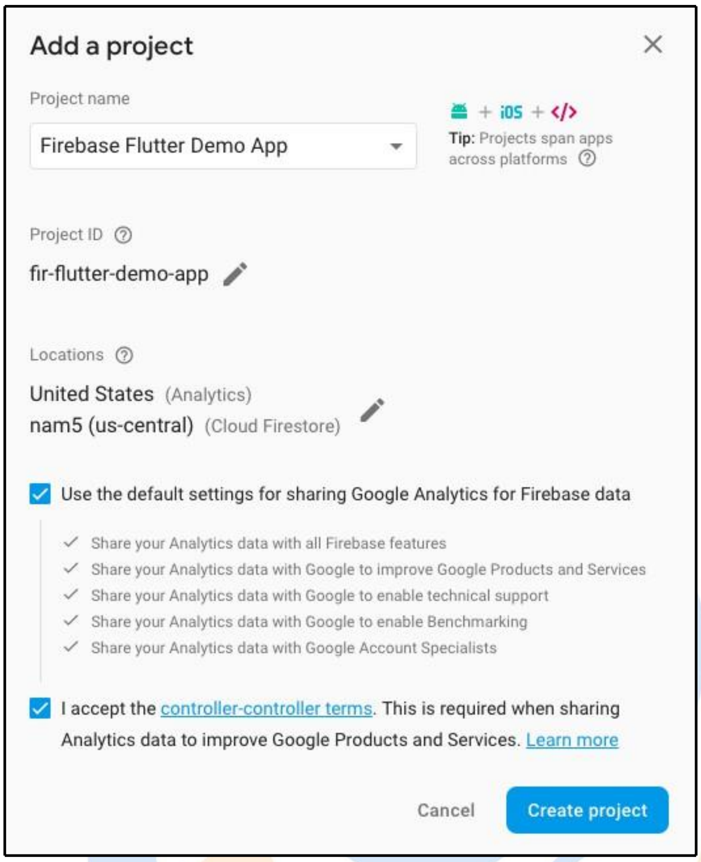
```
flutter packages get
```

Creating a Firebase project

Once the connection has been made, the next step will be to create a Firebase project. So, let's get started. Follow the given steps to create your project:

1. Open the website https://firebase.google.comand log in or sign up. You can use your Google credentials to log in here.
2. Next, click on Add Project.
3. Once you click this option, you will see the following screen:

Universitas Esa Unggul


Add a project

Project name: Firebase Flutter Demo App

Tip: Projects span apps across platforms

Project ID: fir-flutter-demo-app

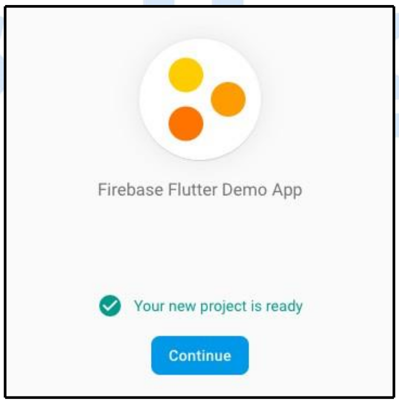
Locations:
United States (Analytics)
nam5 (us-central) (Cloud Firestore)

✓ Use the default settings for sharing Google Analytics for Firebase data

- ✓ Share your Analytics data with all Firebase features
- ✓ Share your Analytics data with Google to improve Google Products and Services
- ✓ Share your Analytics data with Google to enable technical support
- ✓ Share your Analytics data with Google to enable Benchmarking
- ✓ Share your Analytics data with Google Account Specialists

✓ I accept the controller-controller terms. This is required when sharing Analytics data to improve Google Products and Services. Learn more

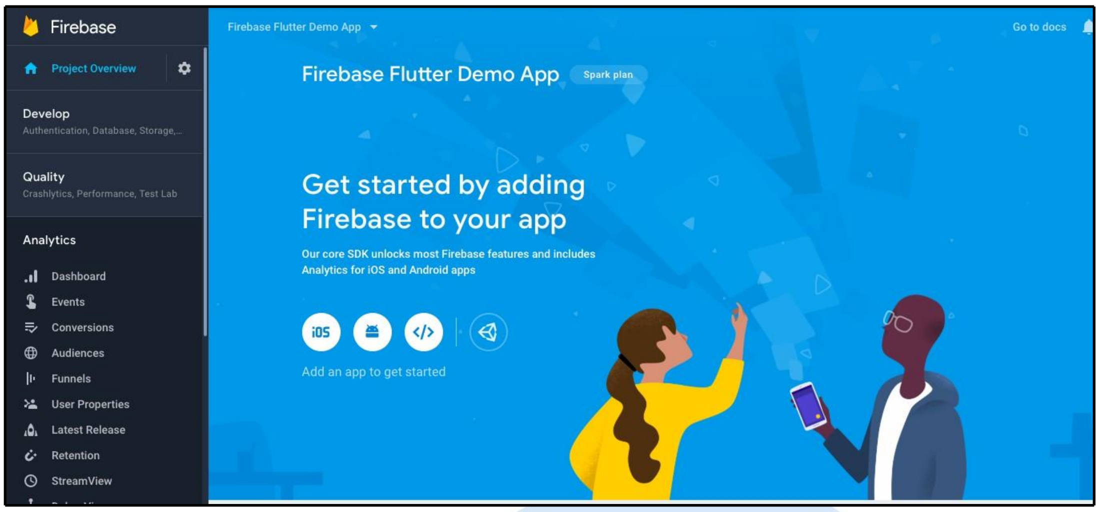
4. Add a **Project name** (for example: FirebaseFlutterDemoApp in our case). The **Project ID** gets auto-generated, or you could type a unique project ID of your own. They are globally unique identifiers.
5. Select the country in **Locations** and then proceed to accept the terms and conditions before you click **Create project.**
6. Click the **Create project** option, and wait a few seconds before the Firebase console shows the message displayed in the following screenshot:


Firebase Flutter Demo App

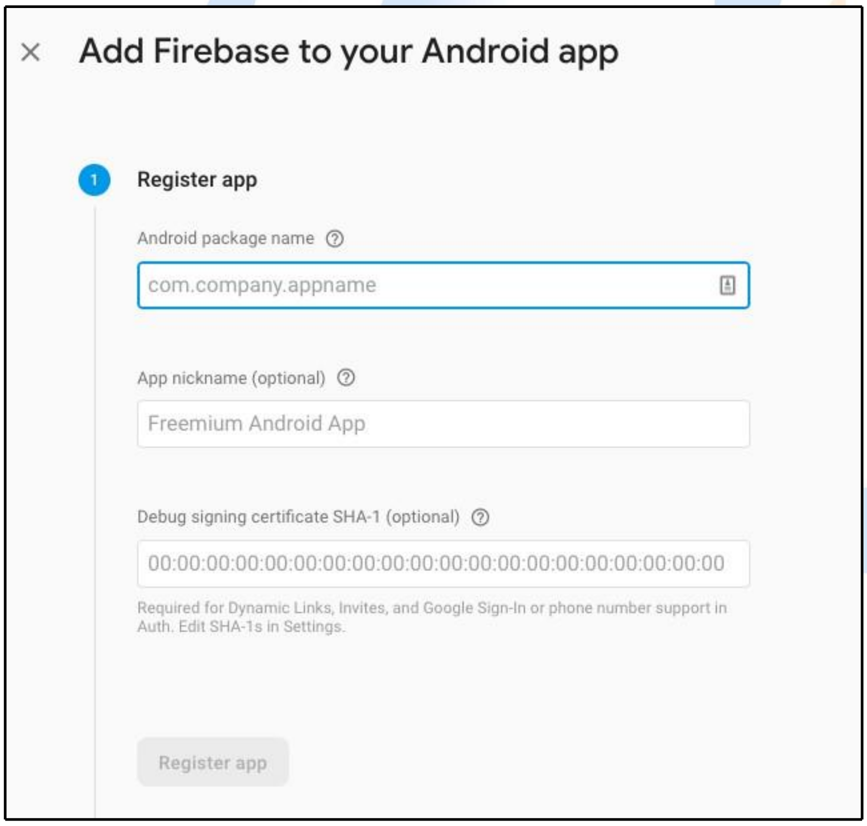
Your new project is ready

7. If your screen shows the **Your new project is ready** text, as shown in the preceding screenshot, you can click the **Continue** button.
8. Once that is done, you will be shown the project settings dashboard of the app as follows:

Universitas Esa Unggul


Firebase
Project Overview
Develop
Authentication, Database, Storage,...
Quality
Crashlytics, Performance, Test Lab
Analytics
Dashboard
Events
Conversions
Audiences
Funnels
User Properties
Latest Release
Retention
StreamView
Firebase Flutter Demo App
Go to docs
Firebase Flutter Demo App
Spark plan
Get started by adding Firebase to your app
Our core SDK unlocks most Firebase features and includes Analytics for iOS and Android apps
iOS
Add an app to get started

9. Choose platform-specific Firebase configurations, based on which app platform you will be building an app for, and click on the respective icon. In our case, since we are building an Android app, we will click on the Android icon to proceed.


Add Firebase to your Android app
1
Register app
Android package name
com.company.appname
App nickname (optional)
Freemium Android App
Debug signing certificate SHA-1 (optional)
00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00
Required for Dynamic Links, Invites, and Google Sign-In or phone number support in Auth. Edit SHA-1s in Settings.
Register app

This step is needed to register your app's platform-specific ID with Firebase. This will generate configuration files that we will add to our project folders. Note that in the top- level directory of your Flutter app, iOS and Android are two of the subdirectories that hold the respective platform-specific configuration files:
In the top-level directory of your Flutter app, you can see subdirectories; called Android and iOS. Here,you'll find platform-specific configuration files for iOS and Android.

The most important field here is the **Android package name.** This is generally the applicationIdin your app-level build.gradlefile. Another way to find the package name is to follow these steps:

1. In the Flutter app directory, check the android/app/src/main/AndroidManifest.xmlfile.
2. Under the Manifest tag, find the string value of the package, which will be the value of the package name.
3. In the Firebase dialog, paste the copied package name from step 2 into the **Android package name** field.

If you are developing the Flutter app for both iOS and Android, you need to register both the iOS and Android versions within the same Firebase project. But if you are developing it just for one platform, you can just click one of them.

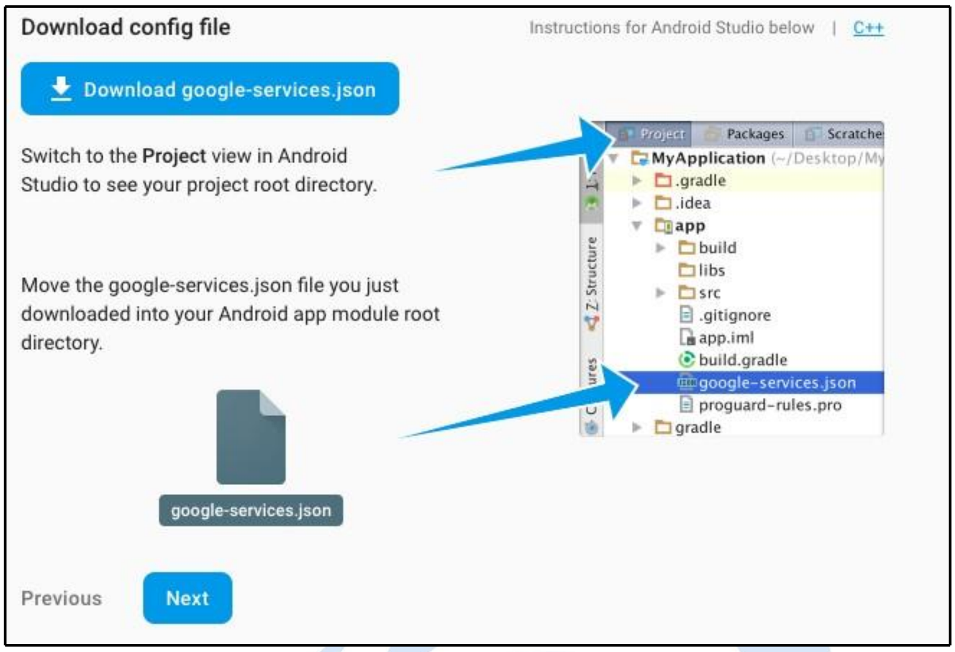
Next, you can add the **App nickname**, which is an optional field. There is another optional field **Debug signing certificate SHA-1,** which has to be used in the same cases if the app uses features such as Google Sign in for Authentication, Dynamic Links, and Invites. In this case, you have to find the debug certificate fingerprint value that you can grab and paste in the field. Refer to the link here, https://developers.google.com/android/guides/ client-auth, for understanding how to build client auth. Since, in this example, we are not using any of these features, we will leave it blank. Click on **Register app**.

Downloading and setting up the config file

The next part will be to download and set up the config file. Follow the given steps to to download and set up the config file:

1. After clicking **Register app**, the console in this step will generate the google-services.jsonfile. Download this file to your computer.
2. Once the file has been downloaded, go to your Flutter app directory, and move the google-services.json file that you downloaded previously into the android/app directory.
3. After the file has been moved, in the Firebase console, click **Next** as shown in the following screenshot:

Universitas Esa Unggul


Download config file
Instructions for Android Studio below | C++
Download google-services.json

Switch to the **Project** view in Android Studio to see your project root directory.

Move the google-services.json file you just downloaded into your Android app module root directory.

Adding Firebase SDK

Now that we have downloaded and set up the config file, the penultimate step is to add Firebase SDK to your project. The Google services plugin for Gradle ensures that the JSON file you downloaded is read. In order to enable Google APIs or Firebase services in your application, you have to add a google-servicesdependency. Two minor modifications are needed to build.gradlefiles to use the plugin. Take a look at the following:

1. Project-level build.gradle(<project>/build.gradle):

```
buildscript {
dependencies

{
// Add this line
classpath 'com.google.gms:google-services:4.2.0'
}
}
```

2. App-level build.gradle(<project>/<app-module>/build.gradle):

```
dependencies {
    // Add this line
    implementation 'com.google.firebase:firebase-core:16.0.1' }
...
// Add to the bottom of the file
apply plugin: 'com.google.gms.google-services'
```

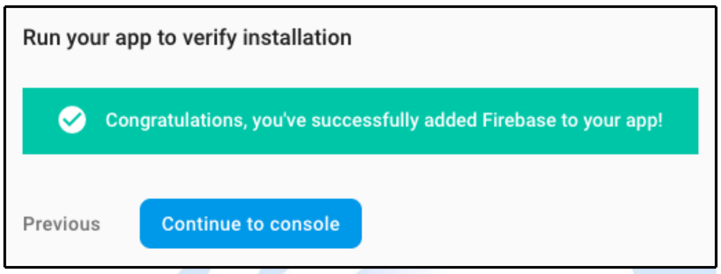
Universitas Esa Unggul

3. Click the **Sync Now** option to complete this process.

Verifying the configuration

Once the previous steps are complete, we have to verify whether your Flutter app is connected to Firebase. To ensure this, follow these steps:

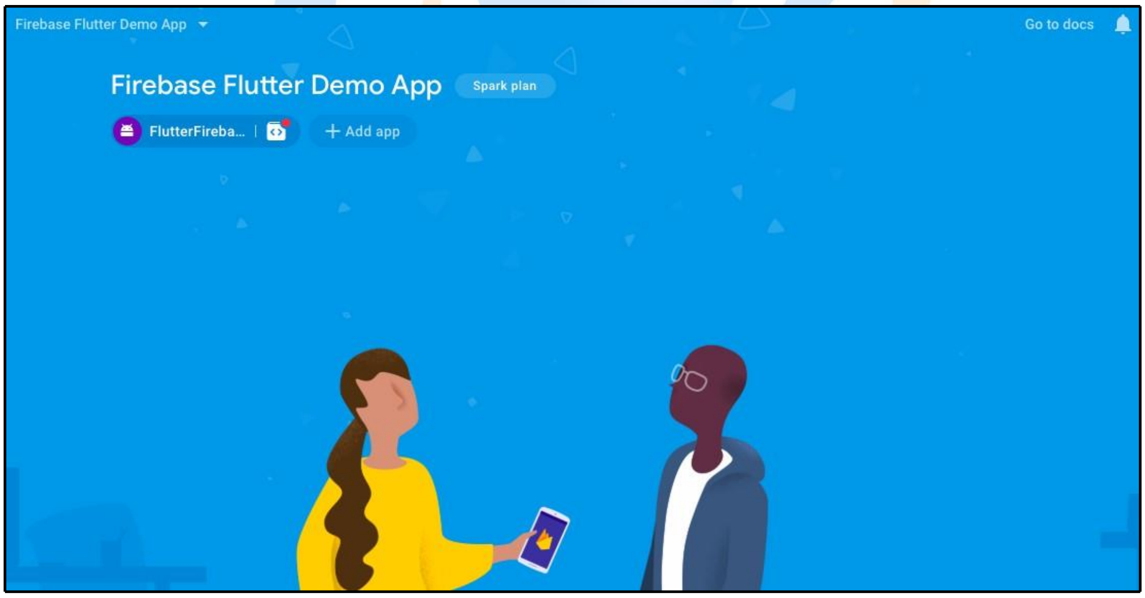
1. Build the project and run it on the device connected to your computer.
2. Once the app gets run on the phone, the Firebase console automatically detects the configurations and displays a success message as follows:



Run your app to verify installation

Congratulations, you've successfully added Firebase to your app!

3. After you click **Continue to console**, you will be taken to the console showing the project name and the other settings:



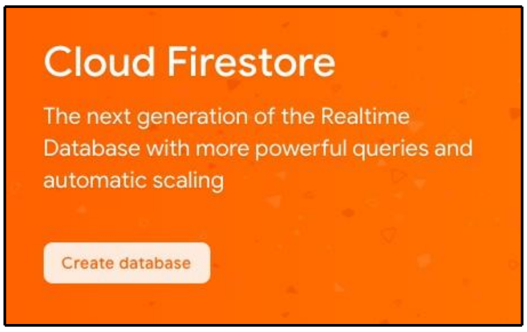
In the next section, we will see how to connect to the Cloud Database.

Creating a Cloud Firestore Database

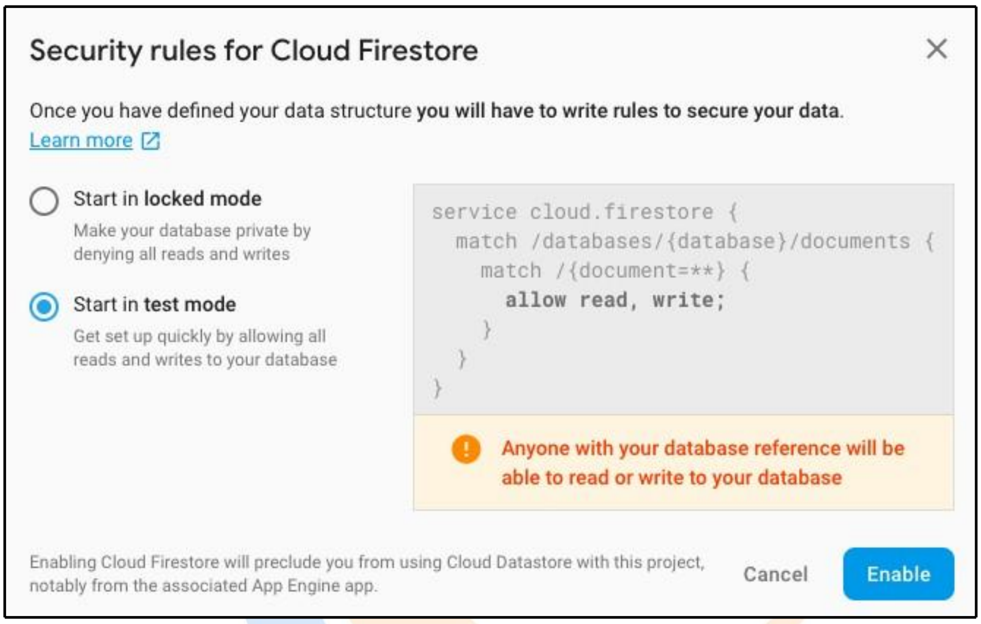
Once the Firebase-Flutter set up is complete, you are all set to build the app. We will now set up a Cloud Firestore database and initialize some values. Follow these steps:

1. Under the **Develop** option, click on **Database.**

2. In the panel shown, click **Create database:**


Cloud Firestore
The next generation of the Realtime Database with more powerful queries and automatic scaling
Create database

3. After clicking, you will see a pop-up panel: **Security rules for Cloud Firestore**. Select **Start in test mode** and enable it:



Security rules for Cloud Firestore

Once you have defined your data structure **you will have to write rules to secure your data.**
Learn more

- [ ] Start in **locked mode**
  Make your database private by denying all reads and writes
- [✓] Start in **test mode**
  Get set up quickly by allowing all reads and writes to your database

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write;
    }
  }
}
```

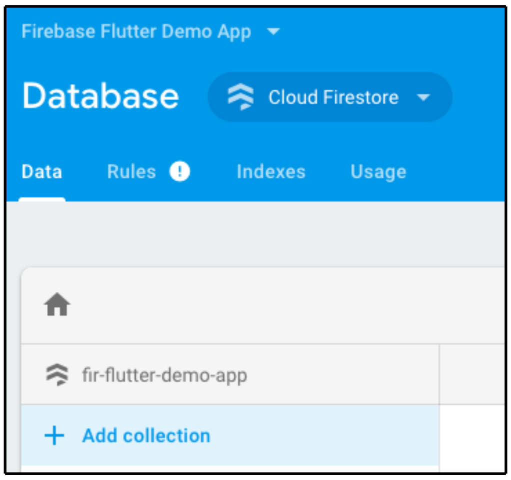
Anyone with your database reference will be able to read or write to your database

Enabling Cloud Firestore will preclude you from using Cloud Datastore with this project, notably from the associated App Engine app.

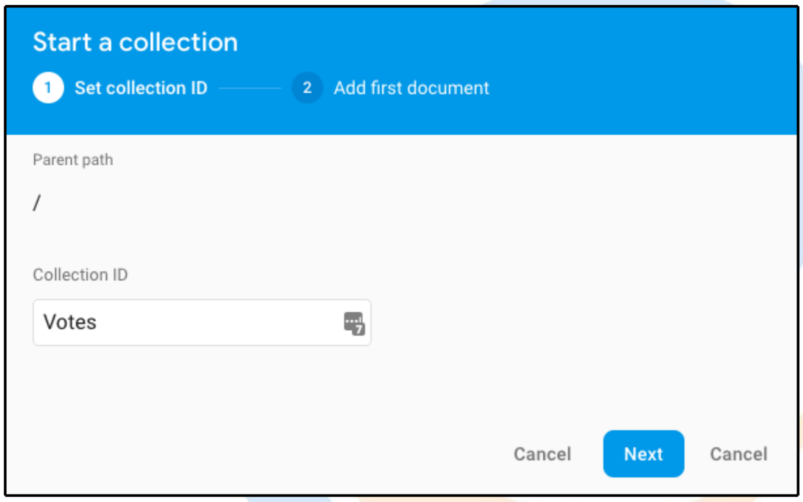
4. We select test mode because we want anyone with the database reference to be able to read or write to the database. When you build the production version of the app, ensure you set up security rules. You can read about these rules here: https://firebase.google.com/docs/reference/rules/rules. After clicking **Enable**, the Cloud Firestore will be provisioned with security rules and will be ready for use.

5. From the following panel, click **Add collection**:

Universitas Esa Unggul



6. We presume that we would have just one collection in Firestore, called **Votes**. A **collection** is a set of documents that comprise the data:



Parent path

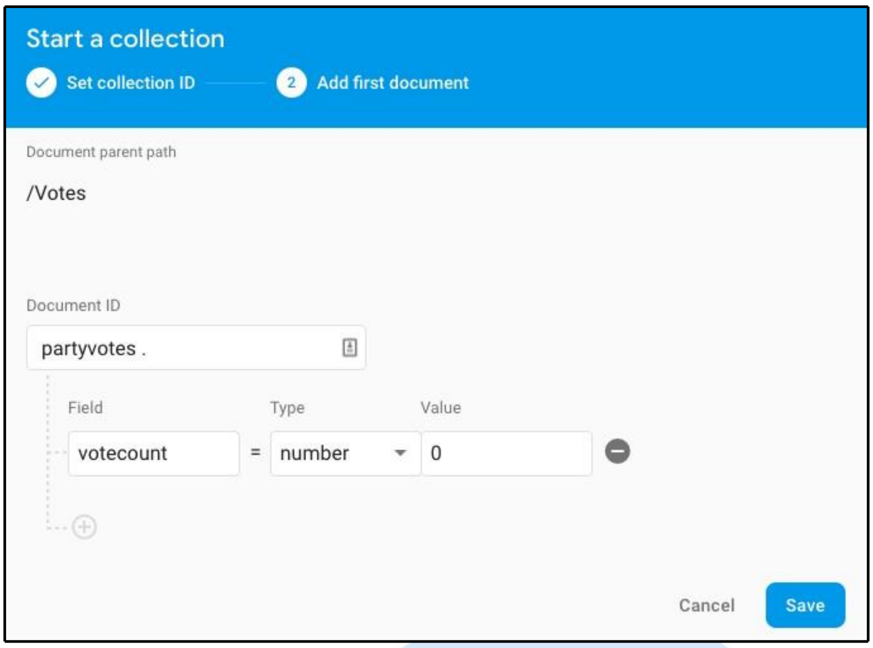
/

Collection ID

Votes

7. Click **Next**.
8. A collection must contain at least one document, which is Cloud Firestore's unit of storage. You can either use an auto-generated ID or have a custom ID. In our case, we use **partyvotes**.
9. For the existing **Field**, enter the value of the name (in our case, it's VoteCount), select the data **Type**, then enter the **Value** of **partyvotes**. Since its an Integer, we select the number and set its initial value to be **0**:

Universitas Esa Unggul


Start a collection

Set collection ID — 2 Add first document

Document parent path

/Votes

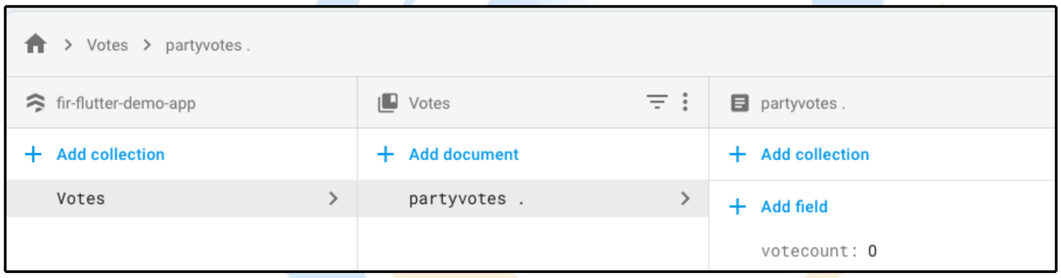
Document ID

partyvotes .

Field | Type | Value
votecount = number 0

Cancel Save

10. Click **Save**.
11. After adding several documents to your collection, your database should look something like this:


Votes > partyvotes .

fir-flutter-demo-app | Votes | partyvotes .

Add collection | Add document | Add collection

Votes | partyvotes . | Add field

votecount: 0

Firestore is a NoSQL database, which means that we would not be working with rows and columns. Now we will build the layout of the app. Using the Firestore details, we will construct the list layout, which will generate the list items runtime based on the values in the Firestore and read/update the fresh values when tapped on the list item into the Firestore database. The following is the main.dartfile:

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

import 'package:flutter/material.dart';


void main() => runApp(MyApp());


// Creating Object to temporary make the list items. We will replace it when we connect it to Firestore


final party = [
  {"partyname": "BJP", "rating": 1},
  {"partyname": "Congress", "rating": 3},
  {"partyname": "AAP", "rating": 5},
```

Universitas Esa Unggul

```
  {"partyname": "Janata Dal Party", "rating": 9},

  {"partyname": "NOTA", "rating": 11},

];

class MyApp extends StatelessWidget { @override

  Widget build(BuildContext context) { return
      MaterialApp(

          title: 'State Party Elections - Worker Profile', home: MyHomePage(),

      );

  }

}

class MyHomePage extends StatefulWidget { @override

  _MyHomePageState createState() { return
      _MyHomePageState();

  }

}

class _MyHomePageState extends State<MyHomePage> { @override

  Widget build(BuildContext context) { return Scaffold(

          appBar: AppBar(title: Text('Party Votes')), body:
          _buildBody(context),

      );

  }

  Widget _buildBody(BuildContext context) {

      // We will add the code here in the next section return _buildList(context,
      party);

  }

  Widget _buildList(BuildContext context, List<Map> snapshot) { return ListView(

          padding: const EdgeInsets.only(top: 22.0),

          children: snapshot.map((data) => _buildListItem(context, data)).toList(),

      );

  }

  Widget _buildListItem(BuildContext context, Map data) { final result =
      Record.fromMap(data);
```

Universitas Esa Unggul

```
// Adding the padding to ensure enough space is given return Padding(

          key: ValueKey(result.name),

          padding: const EdgeInsets.symmetric(horizontal: 15.0, vertical: 7.0), child: Container(

            decoration: BoxDecoration(

              border: Border.all(color: Colors.red), borderRadius:
              BorderRadius.circular(6.0),

            ),
// Showing the list item, with name towards the left and the votes to the right


            child: ListTile(

              title: Text(record.partyname),

              trailing: Text(record.partyvotes.toString()), onTap: () => print(record),

            ),

          ),

        );

  }

}


class Record {

  final String partyname; final int
  partyvotes;

  final DocumentReference reference;


  Record.fromMap(Map<String, dynamic> map, {this.reference})

          : assert(map['partyname'] != null),
            assert(map['partyvotes'] != null), name =
            map['partyname'],

            votes = map['partyvotes'];

  Record.fromSnapshot(DocumentSnapshot snapshot)

          : this.fromMap(snapshot.data, reference: snapshot.reference);


  @override

  String toString() => "Record<$partyname:$partyvotes>";

}
```

We have the collection ready on Firestore Cloud. In the previous example, we have used the party object. It's time we now use the Firestore cloud data from our collection to be shown. We can do that by calling Cloud Firestore using a Firestore.instancereference. For example, if

Universitas Esa Unggul

you wish to call a specific collection from your Firestore Cloud database, you can use the following command to return a stream of snapshots:

```
Firestore.instance.collection('collection_name').snapshots()
```

Streams are of two types: single subscription or broadcast. Streams are responsible for providing an asynchronous sequence of data. User-generated events and data read from files are the two data sequences. Now, using StreamBuilderwidget, we will inject the stream of data into the user interface we have created. One of the classic use cases of the StreamBuilderis that, whenever there is a change in the Firestore values, the list gets updated automatically.

Look for the _buildBodymethod in the previous code and replace the content with this code:

```
Widget _buildBody(BuildContext context) { return
  StreamBuilder<QuerySnapshot>(

      stream: Firestore.instance.collection('party').snapshots(), builder: (context, snapshot) {

          if (!snapshot.hasData) return LinearProgressIndicator(); return _buildList(context,
          snapshot.data.documents);

      },

  );

}
```

Adding the preceding snippet will produce some errors. The _buildListItemmethod still thinks it's getting a map. Hence, we will need to make a couple of changes.

Firstly, make the method to accept DocumentSnapshotinstead of a list of a map:

```
Widget _buildList(BuildContext context, List<DocumentSnapshot> snapshot)

{ ....

}
```

Universitas Esa Unggul

Secondly, use the constructor Record.fromSnapshot()to build the record. The method's updated code as follows:

```
Widget _buildListItem(BuildContext context, DocumentSnapshot data) { final result =
  Record.fromSnapshot(data);
```

Next, use the onTap: () method to ensure whenever a list item is clicked, the votes are updated into the Firestore database. Whenever you click **List Item**, Cloud Firestore notifies all listeners with the updated snapshot. The app is actively engaged using StreamBuilder, which acts to update with the new data. For a single user, it is fine, but when you have multiple users, there is a chance of **Race Condition** may occur.

The complete code for main.dartis as follows:

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

```
void main() => runApp(MyApp());

class MyApp extends StatelessWidget { @override

  Widget build(BuildContext context) { return
      MaterialApp(

          title: 'State Party Elections - Worker Profile', home: MyHomePage(),

      );

  }

}

class MyHomePage extends StatefulWidget { @override

  _MyHomePageState createState() { return
      _MyHomePageState();

  }

}

class _MyHomePageState extends State<MyHomePage> { @override

  Widget build(BuildContext context) { return Scaffold(

          appBar: AppBar(title: Text('Party Votes')), body:
          _buildBody(context),

      );

  }

  Widget _buildBody(BuildContext context) { return
      StreamBuilder<QuerySnapshot>(

          stream: Firestore.instance.collection('party').snapshots(), builder: (context, snapshot) {

              if (!snapshot.hasData) return LinearProgressIndicator();


              return _buildList(context, snapshot.data.documents);

          },

      );

  }

  Widget _buildList(BuildContext context, List<DocumentSnapshot> snapshot) { return ListView(

          padding: const EdgeInsets.only(top: 22.0),

          children: snapshot.map((data) => _buildListItem(context, data)).toList(),

      );
```

Universitas Esa Unggul

```
    }

    Widget _buildListItem(BuildContext context, DocumentSnapshot data) { final result =
        Record.fromSnapshot(data);

        return Padding(
            key: ValueKey(result.name),
            padding: const EdgeInsets.symmetric(horizontal: 15.0, vertical: 7.0),
            child: Container( decoration:
                BoxDecoration(
                    border: Border.all(color: Colors.red), borderRadius:
                    BorderRadius.circular(6.0),
                ),
                child: ListTile(
                    title: Text(record.name),
                    trailing: Text(record.votes.toString()),
                    onTap: () => Firestore.instance.runTransaction((transaction) async {
                        final freshFBsnapshot = await
                            transaction.get(record.reference);
                          final updated = Record.fromSnapshot(freshFBsnapshot);

                              await transaction
                                      .update(record.reference, {'partyvotes': updated.votes + 1});
                            }),
                ),
            ),
        );
    }
  }
    Record.fromSnapshot(DocumentSnapshot snapshot)
            : this.fromMap(snapshot.data, reference: snapshot.reference);

    @override
    String toString() => "Record<$partyname:$partyvotes>";
  }
```

We have the collection ready on Firestore Cloud. In the previous example, we have used the party object. It's time we now use the Firestore cloud data from our collection to be shown.

Universitas Esa Unggul

We can do that by calling Cloud Firestore using a Firestore.instancereference. For example, if you wish to call a specific collection from your Firestore Cloud database, you can use the following command to return a stream of snapshots:

```
Firestore.instance.collection('collection_name').snapshots()
```

Streams are of two types: single subscription or broadcast. Streams are responsible for providing an asynchronous sequence of data. User-generated events and data read from files are the two data sequences. Now, using StreamBuilderwidget, we will inject the stream of data into the user interface we have created. One of the classic use cases of the StreamBuilderis that, whenever there is a change in the Firestore values, the list gets updated automatically.

Look for the _buildBodymethod in the previous code and replace the content with this code:

```
Widget _buildBody(BuildContext context) { return
  StreamBuilder<QuerySnapshot>(

      stream: Firestore.instance.collection('party').snapshots(), builder: (context, snapshot) {

          if (!snapshot.hasData) return LinearProgressIndicator(); return _buildList(context,
          snapshot.data.documents);

      },

  );

}
```

Adding the preceding snippet will produce some errors. The _buildListItemmethod still thinks it's getting a map. Hence, we will need to make a couple of changes.

Firstly, make the method to accept DocumentSnapshotinstead of a list of a map:

```
Widget _buildList(BuildContext context, List<DocumentSnapshot> snapshot)

{ ....

}
```

Secondly, use the constructor Record.fromSnapshot()to build the record. The method's updated code as follows:

```
Widget _buildListItem(BuildContext context, DocumentSnapshot data) { final result =
  Record.fromSnapshot(data);
```

Next, use the onTap: () method to ensure whenever a list item is clicked, the votes are updated into the Firestore database. Whenever you click **List Item**, Cloud Firestore notifies all listeners with the updated snapshot. The app is actively engaged using StreamBuilder, which acts to update with the new data. For a single user, it is fine, but when you have multiple users, there is a chance of **Race Condition** may occur.

The complete code for main.dartis as follows:

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

```
void main() => runApp(MyApp());


class MyApp extends StatelessWidget { @override

  Widget build(BuildContext context) { return
      MaterialApp(

          title: 'State Party Elections - Worker Profile', home: MyHomePage(),

      );

  }

}


class MyHomePage extends StatefulWidget { @override

  _MyHomePageState createState() { return
      _MyHomePageState();

  }

}


class _MyHomePageState extends State<MyHomePage> { @override

  Widget build(BuildContext context) { return Scaffold(

          appBar: AppBar(title: Text('Party Votes')), body:
          _buildBody(context),

      );

  }


  Widget _buildBody(BuildContext context) { return
      StreamBuilder<QuerySnapshot>(

          stream: Firestore.instance.collection('party').snapshots(), builder: (context, snapshot) {

              if (!snapshot.hasData) return LinearProgressIndicator();


              return _buildList(context, snapshot.data.documents);

          },

      );

  }


  Widget _buildList(BuildContext context, List<DocumentSnapshot> snapshot) { return ListView(

          padding: const EdgeInsets.only(top: 22.0),

          children: snapshot.map((data) => _buildListItem(context, data)).toList(),

      );
```

Universitas Esa Unggul

```
  }

  Widget _buildListItem(BuildContext context, DocumentSnapshot data) { final result =
    Record.fromSnapshot(data);

    return Padding(
        key: ValueKey(result.name),
        padding: const EdgeInsets.symmetric(horizontal: 15.0, vertical: 7.0),
        child: Container( decoration:
            BoxDecoration(
                border: Border.all(color: Colors.red), borderRadius:
                BorderRadius.circular(6.0),
            ),
            child: ListTile(
                title: Text(record.name),
                trailing: Text(record.votes.toString()),
                onTap: () => Firestore.instance.runTransaction((transaction) async {
                    final freshFBsnapshot = await
                        transaction.get(record.reference);
                      final updated = Record.fromSnapshot(freshFBsnapshot);

                          await transaction
                                  .update(record.reference, {'partyvotes': updated.votes + 1});
                        }),
            ),
        ),
    );
  }
}
class Record {
  final String partyname; final int
  partyvotes;
  final DocumentReference reference;

  Record.fromMap(Map<String, dynamic> map, {this.reference})
        : assert(map['partyname'] != null),
            assert(map['partyvotes'] != null), name =
            map['partyname'],
            votes = map['partyvotes'];
```

Universitas Esa Unggul

```
  Record.fromSnapshot(DocumentSnapshot snapshot)

                            : this.fromMap(snapshot.data, reference: snapshot.reference);


  @override

  String toString() => "Record<$partyname:$partyvotes>";

}
```

Once you run the code, try clicking on List Items and you will see the updates values mapped on the Firestore Cloud database. You could also try updating the List Item names (in our case, the Party names) in the Firestore Cloud database, and you will see the List Item option updating.

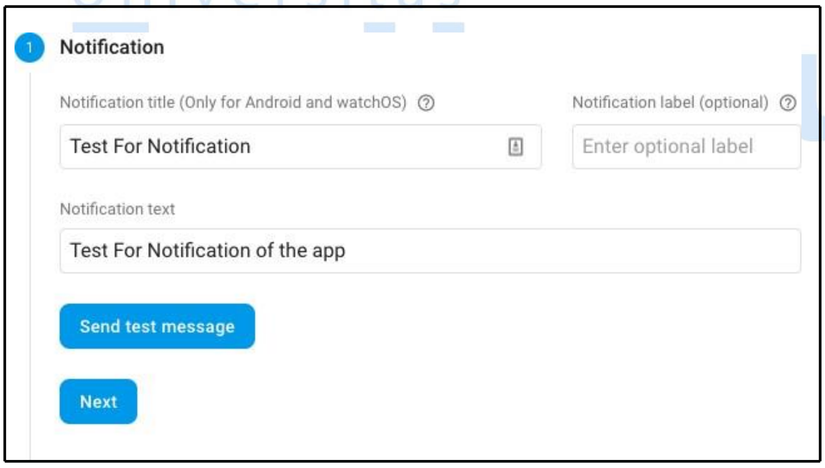
Firebase Cloud Messaging

Firebase Cloud Messaging (**FCM**) is an effective way to drive engagement within the app using the app notification. Using FCM, you can send two kinds of messages to the client device:

1. Notification messages that are directly handled by FCM SDK
2. Data messages

Both these messages have a maximum payload of 4KB. When sending messages from the Firebase console, there is a 1,024 character limit. Firebase has Cloud Messaging as well as In-App messaging, but in this section, we will discuss only Firebase Cloud messaging.

In the Firebase console, click on **Grow | Cloud Messaging** in the left panel. Follow this by clicking **Send your first message.** as shown in the following screenshot:


1
Notification
Notification title (Only for Android and watchOS)
Notification label (optional)
Send test message
Next

Notification title: Test For Notification

Notification label (optional): Enter optional label

Notification text: Test For Notification of the app

To test the message on your device, FCM tokens are needed. Use the following Android code to generate these tokens:

```
FirebaseInstanceId.getInstance().getInstanceId()

        .addOnCompleteListener(new OnCompleteListener<InstanceIdResult> () {

            @Override
```

```
                public void onComplete(@NonNull Task<InstanceIdResult> actionable) {

                    if (!actionable.isSuccessful()) { Log.w(TAG, "getInstanceId
                        failed", actionable.getException()); return;

                    }
                    // Get new Instance ID token

                    String tokenID = actionable.getResult().getToken();

                    // Log and toast

                    String message = getString(R.string.msg_token_fmt, tokenID);

                    Log.d(TAG, message); Toast.makeText(MainActivity.this,
                    message,

                    Toast.LENGTH_SHORT).show();

                }

            });
```

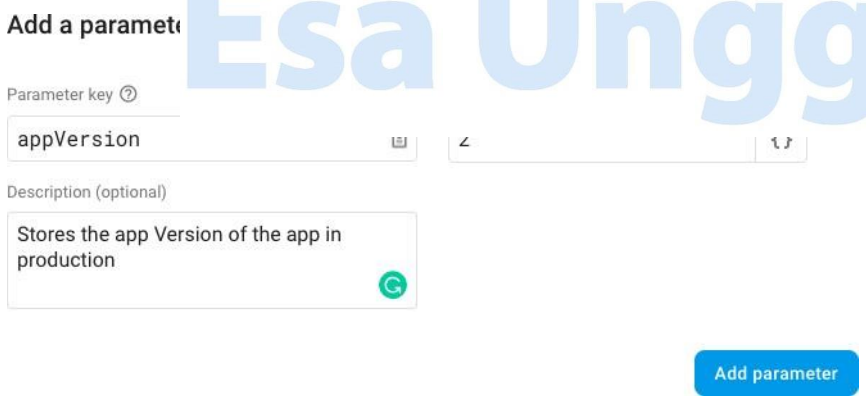
FCM also allows configuring messages specific to targets such as GeoLocations, versions of the app, Languages, and User Audiences. This case is ideal when you wish to send notifications to a specific set of users.

Firebase Remote Config

Using Firebase's Remote Config API, you can make changes to the app without the user actually downloading an app update. One example of this is when you push a new update to the app in production, you can show the pop-up message to the user when they launch the app about the update.

To set up Firebase Remote Config, head over to the **Grow** tab in the Firebase console and click on **Remote Config** as shown in the following screenshot:

Universitas Esa Unggul



appVersion

2

Stores the app Version of the app in production

Add the **Parameter key** and the **Default value**. There is an optional field for adding the description. Click **Add Parameter** to proceed. Avoid storing any confidential information in the remote config. Firebase also allows the setting of conditions for the parameter. For example, if you want to show a specific welcome message to a user in India and a different message to a user in the USA, remote configuration can come in handy.