

# Firestore API

**Firestore** is a cloud services provider and backend as a service. The company makes a number of products for software developers building mobile or web applications. Firestore was founded in 2011 and was acquired by Google in October 2014. Firestore offers an API for android, iOS and web applications. Firestore's primary product is a realtime database which provides an [API](#) that allows developers to store and sync data across multiple clients. But it offers also user management and cloud messaging that will be explained in details here.

## Getting started - Using Firestore in you app

1. Create a Firestore project in the [Firestore console](#). If you already have an existing Google project associated with your mobile app, click **Import Google Project**. Otherwise, click **Create New Project**.
2. Enter your app's package name. It's important to enter the package name your app is using; this can only be set when you add an app to your Firestore project.
3. Download the google-services.json file generated for your project, and copy it into your project's module folder, typically app/.
4. Add this line to the project level gradle file

```
buildscript {  
    // ...  
    dependencies {  
        // ...  
        classpath 'com.google.gms:google-services:3.0.0'  
    }  
}
```

5. Add this four lines to the app level gradle file:

```
dependencies {  
  
    com.google.firebase:firebase-database:9.4.0 //For realtime database  
  
    com.google.firebase:firebase-auth:9.4.0 //For user authentication  
  
    com.google.firebase:firebase-messaging:9.4.0 //For cloud messaging and  
    notifications
```

```
}
```

```
// ADD THIS AT THE BOTTOM OF THE FILE  
apply plugin: 'com.google.gms.google-services'
```

6. Sync your project and update your google repository and google play services if needed(when syncing problem occurs).

That's it, your project is all set and ready to go with Firebase services. In the next chapters we will see which services we can integrate with our app.

## Cloud messaging and notifications

By now you already have the functionality of sending background push notification to all of your users, Just enter the notification pane in your Firebase console, choose new message, enter the text, the sender name and the time if you want to schedule, choose the app and send it. All of your users will receive a push notification if they are outside of the app.

To Gain more functionality like foreground message receiving with payload and manage unique users groups with user token you need to extend two services:

1. A service that extends `FirebaseMessagingService`. This is required if you want to receive notifications in foregrounded apps, to receive data payload, to send upstream messages, and so on. Create your service and extend the `FirebaseMessagingService`, there override the `onMessageReceived(RemoteMessage remoteMessage)` function, this function will be called when your app receive a message while it is in the foreground, there you can get the sender name through the from field the payload through the `getData()` function and the notification text through the `remoteMessage.getNotification().getBody()`. You also need to Add the following lines to the manifest file

```
<service  
    android:name=".MyFirebaseMessagingService">  
    <intent-filter>  
        <action android:name="com.google.firebase.MESSAGING_EVENT"/>  
    </intent-filter>  
</service>
```

2. On initial startup of your app, the Firebase messaging service generates a registration token for the client app instance. If you want to target single devices or create device groups, you'll need to access this token by extending **FirebaseInstanceIdService**. Because the token could be rotated after initial startup, you are strongly recommended to retrieve the latest updated registration token, because it changes every time the user install and uninstall your app or clear app the app data or deletes the instance ID. In your service override the **onTokenRefreshcallback**, it fires whenever a new token is generated, so calling `getToken` in its context ensures that you are accessing a current, available registration token. So just override the function in your service and call the `FirebaseInstanceId.getInstance().getToken()`; this will return you the unique user token and you can send it to your database and store it. If you want to send a message to a specific user just use this token in the user field of the new message window. Don't forget to register your service in the manifest file

```
<service
    android:name=".MyFirebaseInstanceIdService">
    <intent-filter>
        <action android:name="com.google.firebase.INSTANCE_ID_EVENT"/>
    </intent-filter>
</service>
```

## Auth and users Management

Firebase Auth is a service that can authenticate users using only client-side code. It supports social login providers Facebook, GitHub, Twitter and Google. Additionally, it includes a user management system whereby developers can enable user authentication with email and password login stored with Firebase. In this example we will show the steps for creating a user database stored at the firebase with email and password. To do that just follow these steps:

1. In the firebase console go to auth tab and enable the email sign in.
2. Get the FirebaseAuth instance and attach him a listener for authentication changes:

```
FirebaseAuth mAuth = FirebaseAuth.getInstance();
FirebaseAuth.AuthStateListener mAuthListener;
```

3. In your **onStart** attach the listener to the Firebase instance by using `mAuth.addAuthStateListener(mAuthListener);` and in the **onstop** remove the reference by using `mAuth.removeAuthStateListener(mAuthListener);`

4. In your **onCreate** initiate the listener by creating an instance of `FirebaseAuth.AuthStateListener` and override the **onAuthStateChanged**
5. This function receive an instance of `FirebaseAuth` and by using its **getCurrentUser** function we can get the currently registered user as an instance of **FirebaseUser** class, if no one is registered then null is returned. From here you can get the user name and all the user's other details. The complete code for section 4-5 can be found under section 7.
6. Create the sign up function where you get all the user details by calling the **createUserWithEmailAndPassword**(name, pass) of the `FirebaseAuth` instance. You can also add a completion listener to this task by calling **addOnCompleteListener** (chain function calls) in the listener override the **onComplete** function which receives a **Task** object and you can call its **isSuccessful()** function to know whether the login was successfull. Note that the firebase has length limitation on the password and basic email structre validation if you want tmore then that you need to chekc the input before creating the user. Here is the code:

```
mAuth.createUserWithEmailAndPassword(name, pass)
    .addOnCompleteListener(MainActivity.this,new OnCompleteListener <AuthResult>() {
        @Override
        public void onComplete(@NonNull Task<AuthResult> task) {
            Log.d("aa", "createUserWithEmail:onComplete:" + task.isSuccessful());
        }
    })
```

7. If the user creation went well your listener from step 3 will be called with not null user and then you can add the other details of the user like the full name by updating the new user instance. The complete listener code:

```
mAuthListener = new FirebaseAuth.AuthStateListener() {
    @Override
    public void onAuthStateChanged(@NonNull FirebaseAuth firebaseAuth) {
        FirebaseUser user = firebaseAuth.getCurrentUser();
        if (user != null) {
            userTv.setText(user.getDisplayName() + " is logged in!");
            user.updateProfile(new UserProfileChangeRequest.Builder()
                .setDisplayName(registrationFullName).build()).addOnCompleteListener
                (new OnCompleteListener<Void>() {
                    @Override
                    public void onComplete(@NonNull Task<Void> task) {
                        if (task.isSuccessful()) {
                            //profile update sccuessfull
                        }
                    }
                })
        }
    }
});
```

8. For sign in user `signInWithEmailAndPassword(name, pass)`. You can also add on completion listener like before upon signing in the listener will be called passing the user reference that has just been signed in and further data base task can be done there

```
mAuth.signInWithEmailAndPassword(name, pass)
    .addOnCompleteListener(MainActivity.this, new OnCompleteListener<AuthResult>() {
        @Override
        public void onComplete(@NonNull Task<AuthResult> task) {
            if (task.isSuccessful())
                Log.d(TAG, mAuth.getCurrentUser().getDisplayName() + " has logged in");
            else
                Log.d(TAG, "Sign in failed");
        }
    });
```

## Live database

Firebase provides a realtime database and backend as a service. The service provides application developers an API that allows application data to be synchronized across clients and stored on Firebase's cloud. The company provides client libraries that enable integration with Android, iOS, JavaScript, Java, Objective-C and Node.js applications. Using this live data base is very simple. By default, read and write access to your database is restricted so only authenticated users can read or write data.

**To save data** into the database get the data base reference by using

```
DatabaseReference mDatabase = FirebaseDatabase.getInstance().getReference();
```

From here you can save any String, long, double, boolean, Map<String, Object>, List<Object> or any other custom Java object (all in compliance with json files), as long as it defines a default constructor that takes no arguments and has public getters for the properties to be assigned. If you use a Java object, the contents of your object are automatically mapped to child locations in a nested fashion.

For example let's say we have the following User java class:

```
public class User {

    public String username;
    public String email;

    public User() {
        // Default constructor required for calls to DataSnapshot.getValue(User.class)
    }
}
```

```

public User(String username, String email) {
    this.username = username;
    this.email = email;
}
}

```

And the following instance: `User user = new User(name, email);`

We can save it to the database using the `setValue()` method as follows:

```
mDatabase.child("users").child(userId).setValue(user);
```

This line creates a root json object called “users” underneath it it creates an entry for each user defines by the `userId` (this can be achieved using the

```
FirebaseAuth.getInstance().getCurrentUser().getUid())
```

and then we save the user we just created.

### **To retrieve data from the database:**

Because file system and database access can take time the information retrieving mechanism is asynchronous meaning we add a listener to certain event changes we would like to know about and when the data is ready to be read the system will call the `onDataChanges` passing a snapshot of the data we are looking for.

To add a value event listener, use the `addValueEventListener()` or `addListenerForSingleValueEvent()` method (This is useful for data that only needs to be loaded once and isn't expected to change frequently or require active listening, like initial reading of database upon signing in). If we continue to look at the previous user example we can retrieve the data saved before like this:

```

mDatabase.child("users").child(userId).addListenerForSingleValueEvent(
    new ValueEventListener() {
        @Override
        public void onDataChange(DataSnapshot dataSnapshot) {
            // Get user value
            User user = dataSnapshot.getValue(User.class);

            If we want to read a list of values we can do like this:
            for (DataSnapshot userSnapshot : dataSnapshot.getChildren()) {
                User user = userSnapshot.getValue(User.class);
                //...
            }
            // ...
        }
    }
}

```

# Topic messaging

Firebase topic messaging allows you to send a message to multiple devices that have participated in a particular topic. You compose topic messages as needed, and Firebase handles routing and delivering the message reliably to the right devices. For example, users of a local weather forecasting app could opt in to a "severe weather alerts" topic and receive notifications of storms threatening specified areas.

Client apps can subscribe to any existing topic, or they can create a new topic. When a client app subscribes to a new topic name (one that does not already exist for your Firebase project), a new topic of that name is created in Firebase and any client can subsequently subscribe to it.

To subscribe to a topic, the client app calls Firebase Cloud Messaging **subscribeToTopic()** with the FCM topic name:

```
FirebaseMessaging.getInstance().subscribeToTopic("news");
```

To unsubscribe, the client app calls Firebase Cloud Messaging **unsubscribeFromTopic()** with the topic name.

## To send a message topics outside of the console

From the server side, sending messages to a Firebase Cloud Messaging topic is very similar to sending messages to an individual device or to a user group. The app server sets the **to** key with a value like `/topics/<yourTopic>`. To send to combinations of multiple topics, the app server sets the **condition** key to a boolean condition that specifies the target topics. For example, to send messages to devices that subscribed to TopicA and either TopicB or TopicC: `'TopicA' in topics && ('TopicB' in topics || 'TopicC' in topics)`

## The HTTP POST request:

Url: <https://fcm.googleapis.com/fcm/send>

Content-Type:application/json

Authorization:key=<Your cloud messaging key as found in the could messaging in your project preference>

```
{
  "to": "/topics/foo-bar",
  "data": {
    "message": "This is a Firebase Cloud Messaging Topic Message!",
  }
}
```

To topics "dogs" or "cats", instead of the "to" key use "condition": "'dogs' in topics || 'cats' in topics"

Reference: [The Firebase getting started guide in the official firebase site](#)