

COMPUTAÇÃO MÓVEL

ANO LETIVO 2023/2024

ANDROID SERVICES

Services

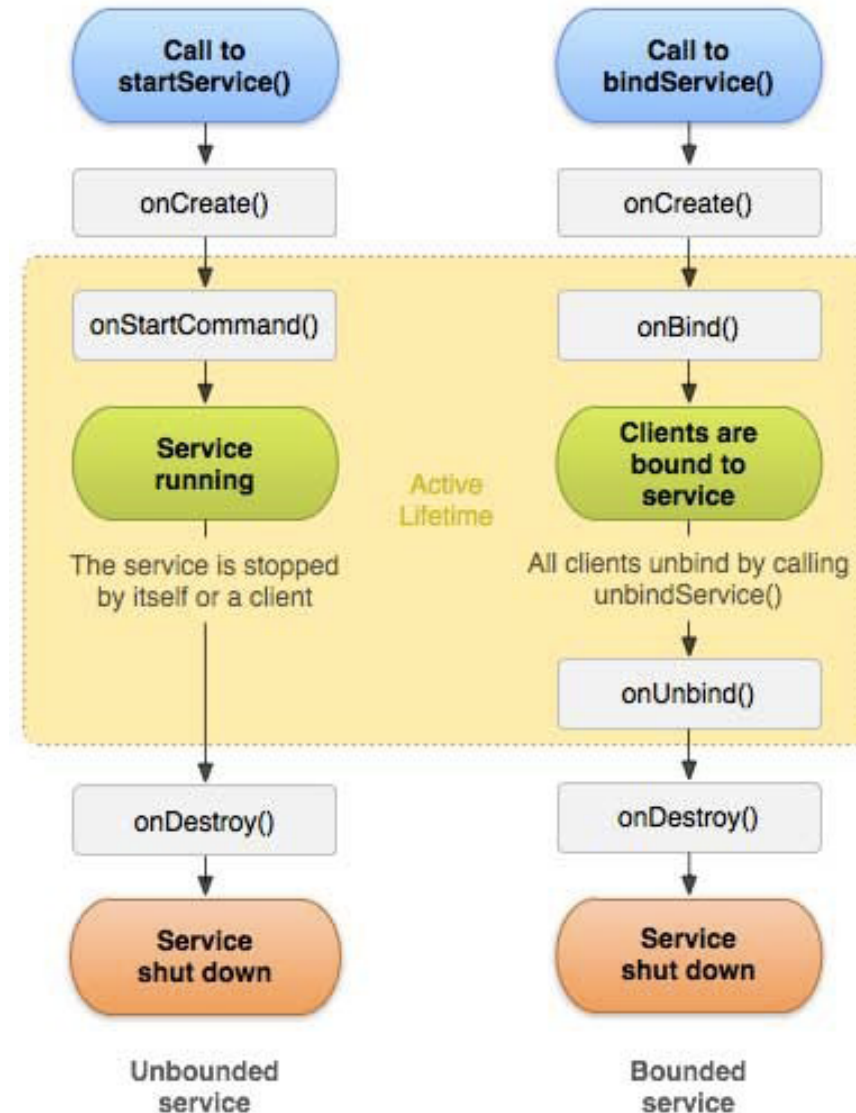
- Service: a background task used by an app.
 - Example: Google Play Music plays the music using a service.
 - Example: Web browser runs a downloader service to retrieve a file.
 - Useful for long-running tasks, and/or providing functionality that can be used by other applications.
- Unlike activities, which have an associated layout, Services work transparently, being designed to operate over a long period, with a priority higher than activities.

Services

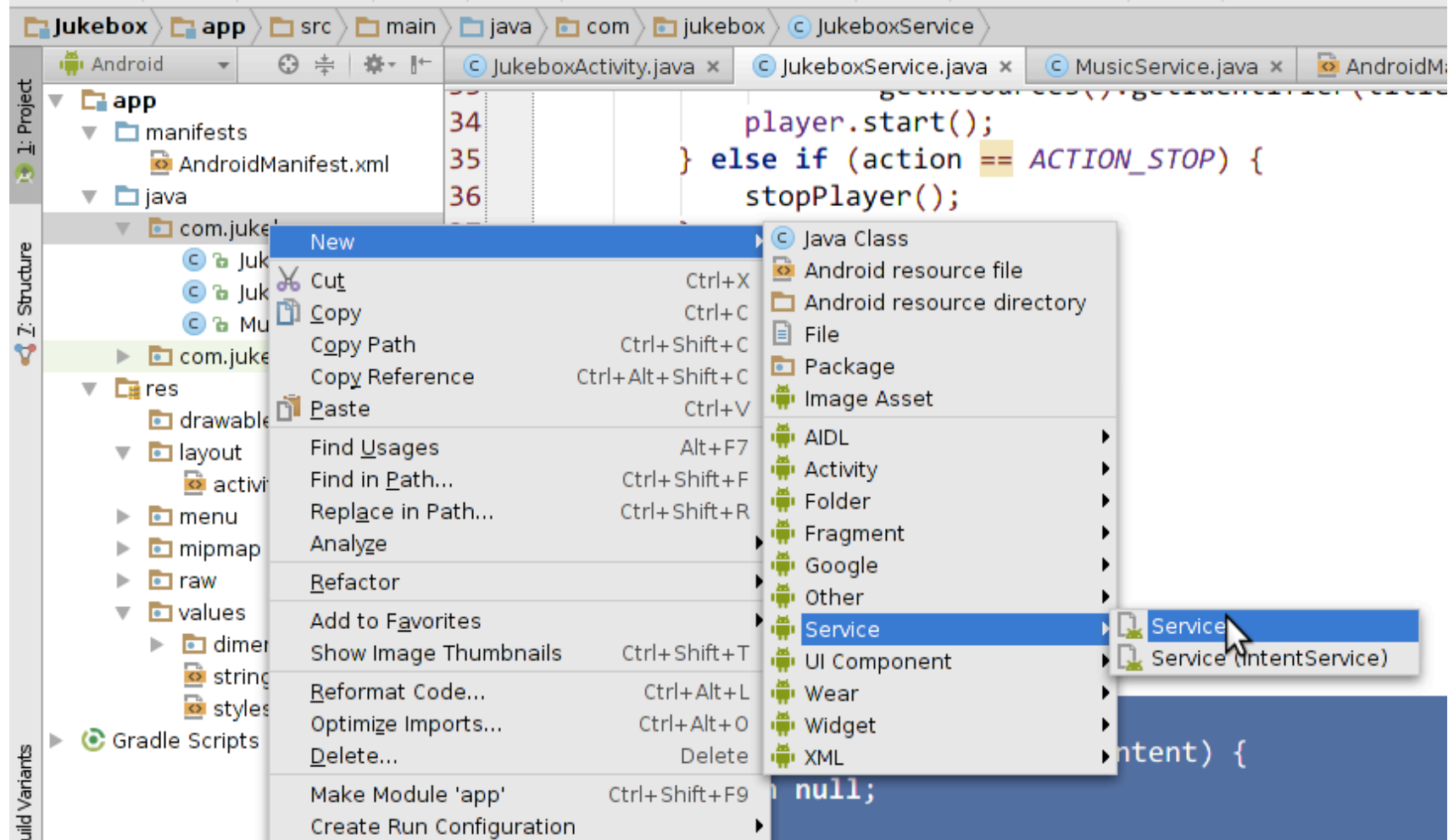
- If an application provides a functionality which is not directly connected to the UI, or it implies a considerable processing overhead, a Service may be considered.
- Android has two kinds of services:
 - **standard services**: For longer jobs; remains running after app closes.
 - **intent services**: For shorter jobs; app launches them via intents. *IntentService* is a base class for Services that handle asynchronous requests (expressed as Intents) on demand. Clients send requests; the service is started as needed, handles each Intent in turn using a worker thread, and stops itself when it runs out of work.
- When/if the service is done doing work, it can broadcast this information to any receivers who are listening.

The service lifecycle

- A service is started by an app's activity using an intent.
- Service operation modes:
 - **start**: The service keeps running until it is manually stopped.
we'll deal mainly with this one
 - **bind**: The service keeps running until no "bound" apps are left.
- Services have similar methods to activities for lifecycle events.
 - onCreate, onDestroy



Adding a Service



Service class template

```
public class ServiceClassName extends Service {

    /* this method handles a single incoming request */
    @Override
    public int onStartCommand(Intent intent, int flags, int id) {
        // unpack any parameters that were passed to us
        String value1 = intent.getStringExtra("key1");
        String value2 = intent.getStringExtra("key2");
        // do the work that the service needs to do ...
        return START_STICKY; // stay running
    }
    @Override
    public IBinder onBind(Intent intent) {
        return null; // disable binding
    }
}
```

AndroidManifest.xml

- To allow your app to use the service, add the following to your app's ***AndroidManifest.xml*** configuration:

(Android Studio does this for you if you use the New Service option)

- the exported attribute signifies whether other apps are also allowed to use the service (true=yes, false=no)
- note that you must write a dot (.) before the class name below!

```
<application ...>
```

```
<service
```

```
    android:name=".ServiceClassName"
```

```
    android:enabled="true"
```

```
    android:exported="false" />
```



Starting a (regular) service

- If someone calls **`Context.startService(intent)`** then the system will retrieve the service (creating it and calling its **`onCreate()`** method if needed) and then call its **`onStartCommand(Intent, int, int)`** method with the arguments supplied by the client.
- The service will at this point continue running until **`Context.stopService()`** or **`stopSelf()`** is called.
- Note that multiple calls to **`Context.startService()`** do not nest (though they do result in multiple corresponding calls to **`onStartCommand()`**), so no matter how many times it is started a service will be stopped once **`Context.stopService()`** or **`stopSelf()`** is called; however, services can use their **`stopSelf(int)`** method to ensure the service is not stopped until started intents have been processed.

Starting a (regular) service

For started services, there are two additional major modes of operation they can decide to run in, depending on the value they return from ***onStartCommand()***:

- **START_STICKY** is used for services that are explicitly started and stopped as needed
- **START_NOT_STICKY** or **START_REDELIVER_INTENT** are used for services that should only remain running while processing any commands sent to them.

Starting a service – binded usage

- Clients can also use ***Context.bindService()*** to obtain a persistent connection to a service. This likewise creates the service if it is not already running (calling ***onCreate()*** while doing so), but does not call ***onStartCommand()***.
- The client will receive the ***IBinder*** object that the service returns from its ***onBind(Intent)*** method, allowing the client to then make calls back to the service. The service will remain running as long as the connection is established (whether or not the client retains a reference on the service's ***IBinder***). Usually the ***IBinder*** returned is for a complex interface that has been written in Android Interface Definition Language (AIDL).

Starting a service – binded usage

- A service can be both started and have connections bound to it. In such a case, the system will keep the service running as long as either it is started or there are one or more connections to it with the ***Context.BIND_AUTO_CREATE*** flag.
- Once neither of these situations hold, the service's ***onDestroy()*** method is called and the service is effectively terminated. All cleanup (stopping threads, unregistering receivers) should be complete upon returning from ***onDestroy()***.

Starting a (regular) service

- In your Activity class:

```
Intent intent = new Intent(this, ServiceClassName.class);  
intent.putExtra("key1", "value1");  
intent.putExtra("key2", "value2");  
startService(intent); // not startActivity!
```

- or if the same code is launched from a fragment:

```
Intent intent = new Intent(getActivity(),  
    ServiceClassName.class);  
...
```

Intent actions

- Often a service has several "actions" or commands it can perform.
 - Example: A music player service can play, stop, pause, ...
 - Example: A chat service can send, receive, ...
- Android implements this with set/getAction methods in Intent.
 - In your Activity class:

```
Intent intent = new Intent(this, ServiceClassName.class);
intent.setAction("some constant string");
intent.putExtra("key1", "value1");
startService(intent);
```
 - In your Service class:

```
String action = intent.getAction();
if (action == "some constant string") { ... } else { ... }
```

Broadcasting a result

- When a service has completed a task, it can notify the app by "sending a broadcast" which the app can listen for:
 - As before, set an action in the intent to distinguish different kinds of results.

```
public class ServiceClassName extends Service {
    @Override
    public int onStartCommand(Intent tent, int flags, int id) {
        // do the work that the service needs to do ...
        ...
        // broadcast that the work is done
        Intent done = new Intent();
        done.setAction("action");
        done.putExtra("key1", value1); ...
        sendBroadcast(done);
        return START_STICKY; // stay running
    }
}
```

Receiving a broadcast

- Your activity can hear broadcasts using a BroadcastReceiver.
 - Extend ***BroadcastReceiver*** with the code to handle the message.
 - Any extra parameters in the message come from the service's intent.

```
public class ActivityClassName extends Activity {  
    ...  
    private class ReceiverClassName extends BroadcastReceiver {  
        @Override  
        public void onReceive(Context context, Intent intent) {  
            // handle the received broadcast message  
            ...  
        }  
    }  
}
```


Listening for broadcasts

- Set up your activity to be notified when certain broadcast actions occur.
 - You must pass an intent filter specifying the action(s) of interest.

```
public class ActivityClassName extends Activity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        ...  
        IntentFilter filter = new IntentFilter();  
        filter.addAction("action");  
        registerReceiver(new ReceiverClassName(), filter);  
    }  
}
```

Services and threading

- By default, a service lives in the same process and thread as the app that created it.
 - This is not ideal for long-running tasks.
 - If the service is busy, the app's UI will freeze up.
- Bear in mind that *IntentServices* run on a separate thread.
- To make the service and app more independent and responsive, the service should handle tasks in threads.

Service with thread

```
public class ServiceClassName extends Service {  
    /* this method handles a single incoming request */  
    @Override  
    public int onStartCommand(Intent intent,  
        int flags, int id) {  
        // unpack any parameters that were passed to us  
        String value1 = intent.getStringExtra("key1");  
        Thread thread = new Thread(new Runnable() {  
            public void run() {  
                // do the work that the service needs to do  
            }  
        });  
        thread.start();  
        return START_STICKY; // stay running  
    }  
}
```

EVENTING AND MESSAGING

Convenient cloud services

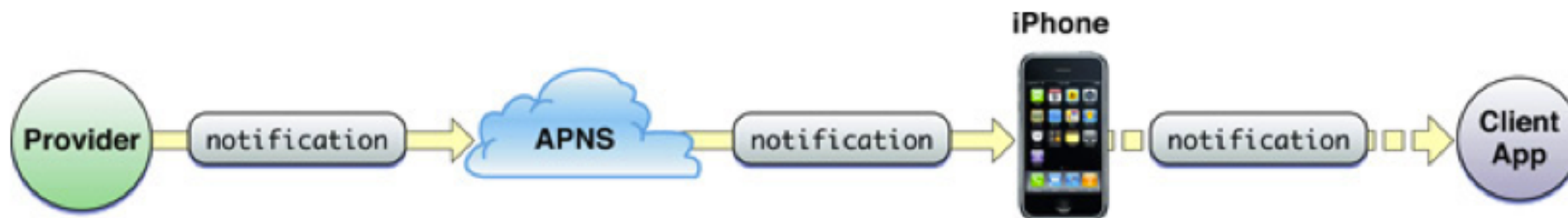
Messaging and eventing – why ?

- Various mechanisms to keep an app in synch with changes in the server (e.g. Twitter, Facebook)
 - Polling: app periodically polls the servers for changes
 - Push: servers push changes to app
- Polling can be inefficient if server data changes infrequently
 - Unnecessary battery drain and network overhead (signaling and data)
- Several apps polling independently without coordination can also be inefficient
 - High battery drain and radio signaling every time the devices moves from IDLE to CONNECTED state

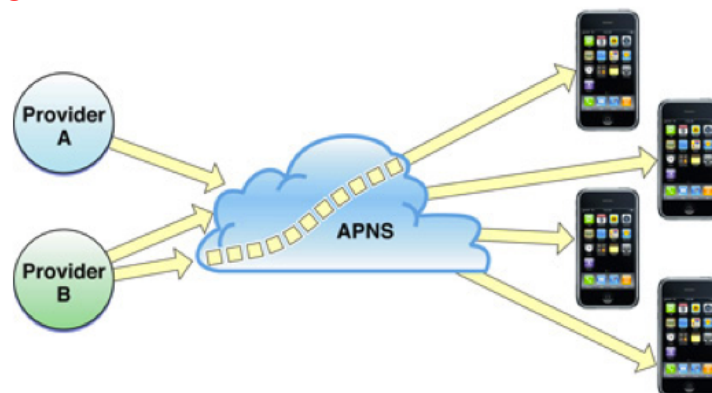
Apple Push Notifications architecture

The Apple way

- iOS device maintains a persistent TCP connection to a Apple Push Notification Server(APNS)



A push notification from a provider to a client application



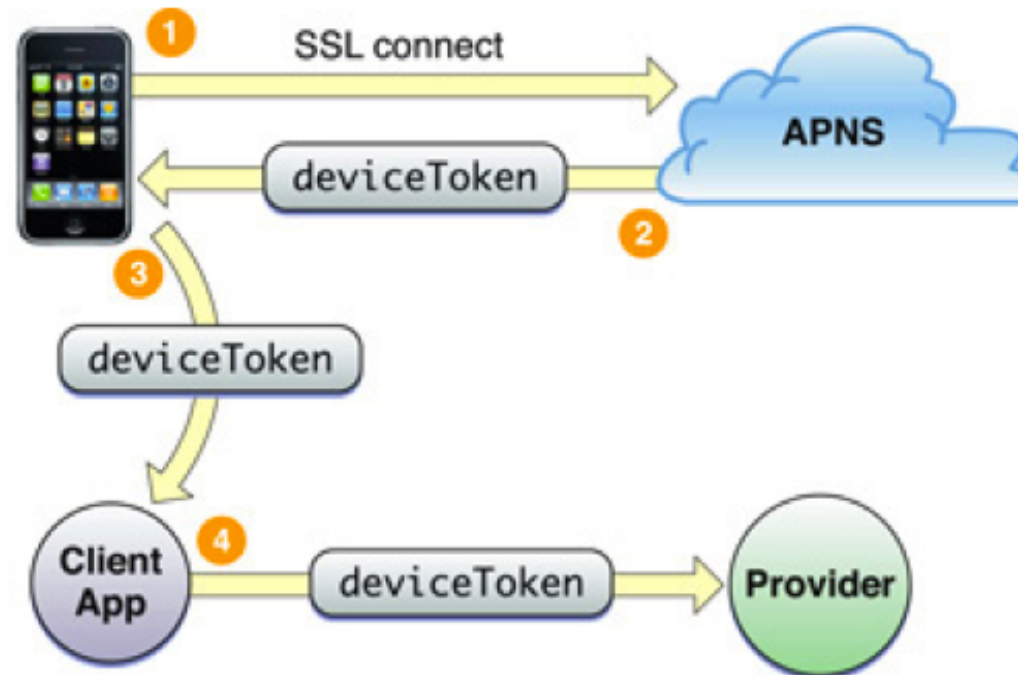
Multi-providers to multiple devices

Push Notification

- Push notification
 - Delivery is best effort and is not guaranteed
 - Max size is 256 bytes
 - Providers compose a JSON dictionary object
 - This dictionary must contain another dictionary identified by the key **apns**
 - Action:
 - An alert message to display to the user
 - A number to badge the application icon with
 - A sound to play

Device Token

- Device token is similar to a phone number
 - Contains information that enables Apple Push Notification servers to locate the device
 - Client app needs to provide the token to its provider
 - Device token should be requested and passed to providers every time your application launches



Google Firebase Cloud Messaging



What is FCM ?

- FCM is the successor to GCM.
- Minimum requirements for FCM clients:
 - Android 2.3+
 - Google Play Store App
(or a suitable Emulator with the Google APIs)
- FCM supports three types of messages:
 - Notification Message
 - Data Message
 - Message with both Notification & Data Payload

How it works ?

- An FCM implementation includes an app server that interacts with FCM via HTTP or XMPP protocol, and a client app.
- You can compose and send messages using the app server or the Notifications console.
- Firebase Notifications is built on Firebase Cloud Messaging and shares the same FCM SDK for client development.



Two types of messages

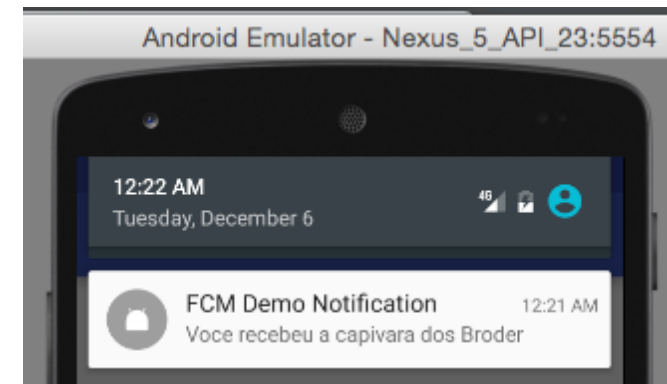
| | Notifications | Data |
|-------------------------------------|---|---|
| Size limit | 2KB | 4KB |
| Use cases | FCM automatically displays the message to end-user devices on behalf of the client app. | Client app is responsible for processing data messages. |
| Can be sent from Firebase console ? | Yes (can be sent from server) | No (requires server) |
| Custom keys ? | No | Yes |

Notifications

Notification messages are handled by firebase SDK itself. Usually contains title, message, icon etc. The notification will be shown automatically when the app is in background.

In order to send notification message, you need to use **notification** key in json data. An example of notification message is given below:

```
{
  "to": "e1w6hEbZn-8:APA91bEUIb2JewYCIiApsMu5JfI5Ak...",
  "notification": {
    "body": "This is a notification",
    "title": "FCM Demo Notification",
    "icon": "appicon"
  }
}
```



Data

Data messages have are handled by the android app. You can add this kind of messages if you want to send some additional data along with the notification. You need to have a server side logic to send the notification using Firebase API. You need to use the **data** key when sending this message.

An example of data message json is given below.

```
{
  "to" : "e1w6hEbZn-8:APA91bEUib2JewYCIiApsMu5JfI5Ak...",
  "data" : {
    "item" : "Electric toothbrush",
    "item_code" : "345333",
    "model" : "Rotoblaster 3K Turbo Mk. II"
  }
}
```

Notifications + data

A message may contain both notification and data payloads. Two handling mechanisms are possible:

- When in the background – the notification is delivered to the device's system tray, and the data payload is delivered in the extras of the intent of the launcher Activity.
- When in the foreground – App receives a message object with both payloads available.

Message handling

To receive messages, use a service that extends ***FirebaseMessagingService***. Your service should override the *onMessageReceived* callback, which is provided for most message types (with the exception of notifications and notifications + data messages received with the app in background).

| App state | Notification | Data | Both |
|------------|---------------------------------|---------------------------------|---|
| Foreground | <i>onMessageReceived</i> | <i>onMessageReceived</i> | <i>onMessageReceived</i> |
| Background | System tray | <i>onMessageReceived</i> | Notification: system tray Data: in extras of the intent. |

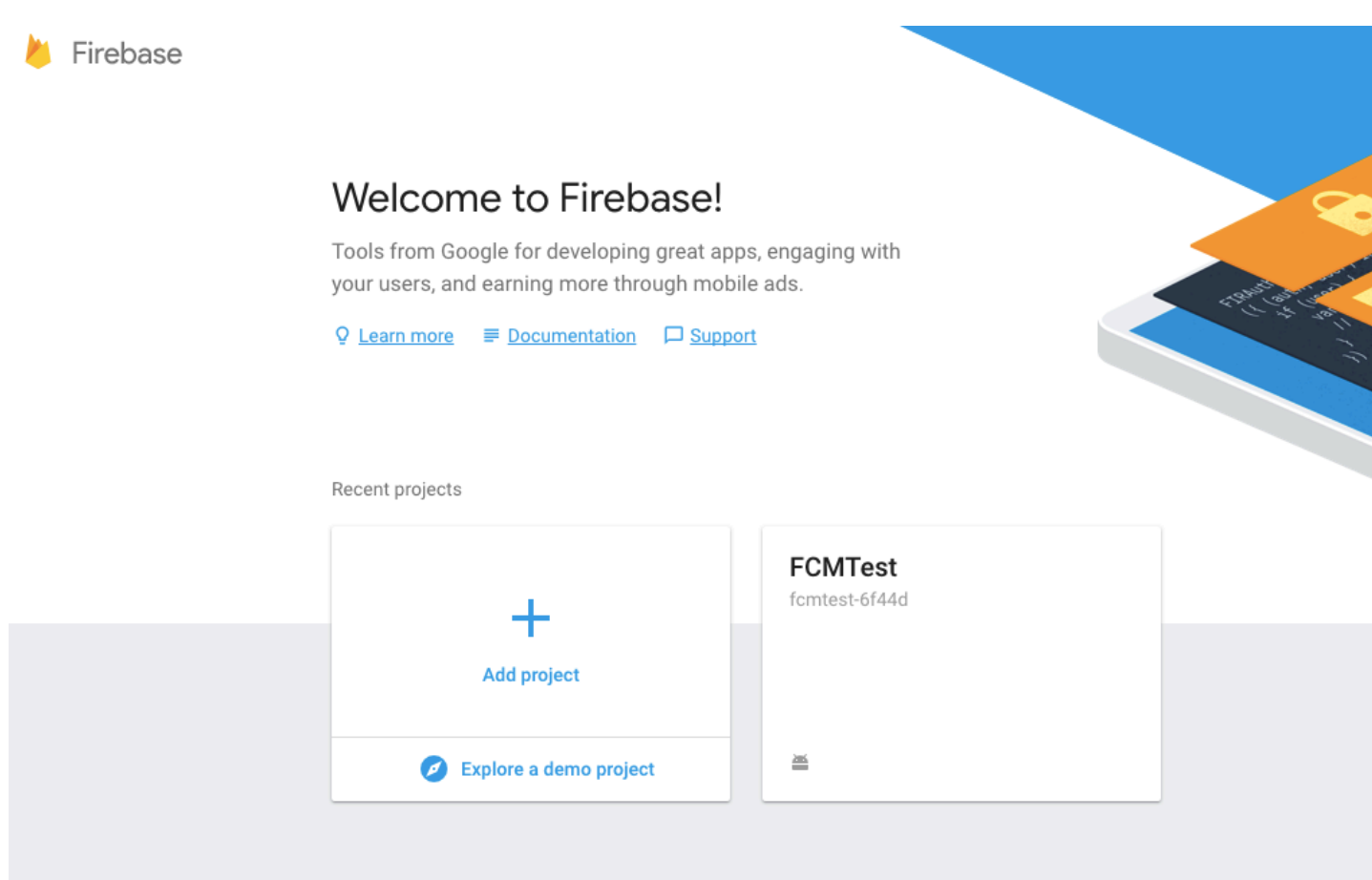
By overriding the method *FirebaseMessagingService.onMessageReceived*, you can perform actions based on the received *RemoteMessage* object and get the message data.

Steps

- **1.** Set the FCM SDK
- **2.** Develop client App
- **3.** Develop an app server (optional for more realistic usage models, not covered here)

Step 1

- Create a Firebase project in the Firebase console (available at <http://console.firebase.google.com>)






Step 1

- Create a Firebase project in the Firebase console (available at <http://console.firebase.google.com>)

Add a project

Project name

My awesome project

 +  + 

Tip: Projects span apps across platforms

Project ID

my-awesome-project-id

Locations

United States (Analytics)

us-central (Cloud Firestore)

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

☒ 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](#)

Cancel

Create project

Step 1

Firebase

CourseCM-2019

[Ir para a documentação](#)

CourseCM-2019 [Plano Spark](#)

Comece adicionando o Firebase ao seu aplicativo

Adicione um app para começar

Desenvolver

- Authentication
- Database
- Storage
- Hosting
- Functions
- ML Kit

Qualidade

Crashlytics, Performance, Test La...

Analytics

- Dashboard
- Events
- Conversions
- Audiences
- Funnels
- User Properties
- Latest Release
- Retention
- Extensions

Spark

Plano gratuito
US\$ 0/mês

[Fazer upgrade](#)

Armazene e sincronize dados de app em milissegundos

Authentication

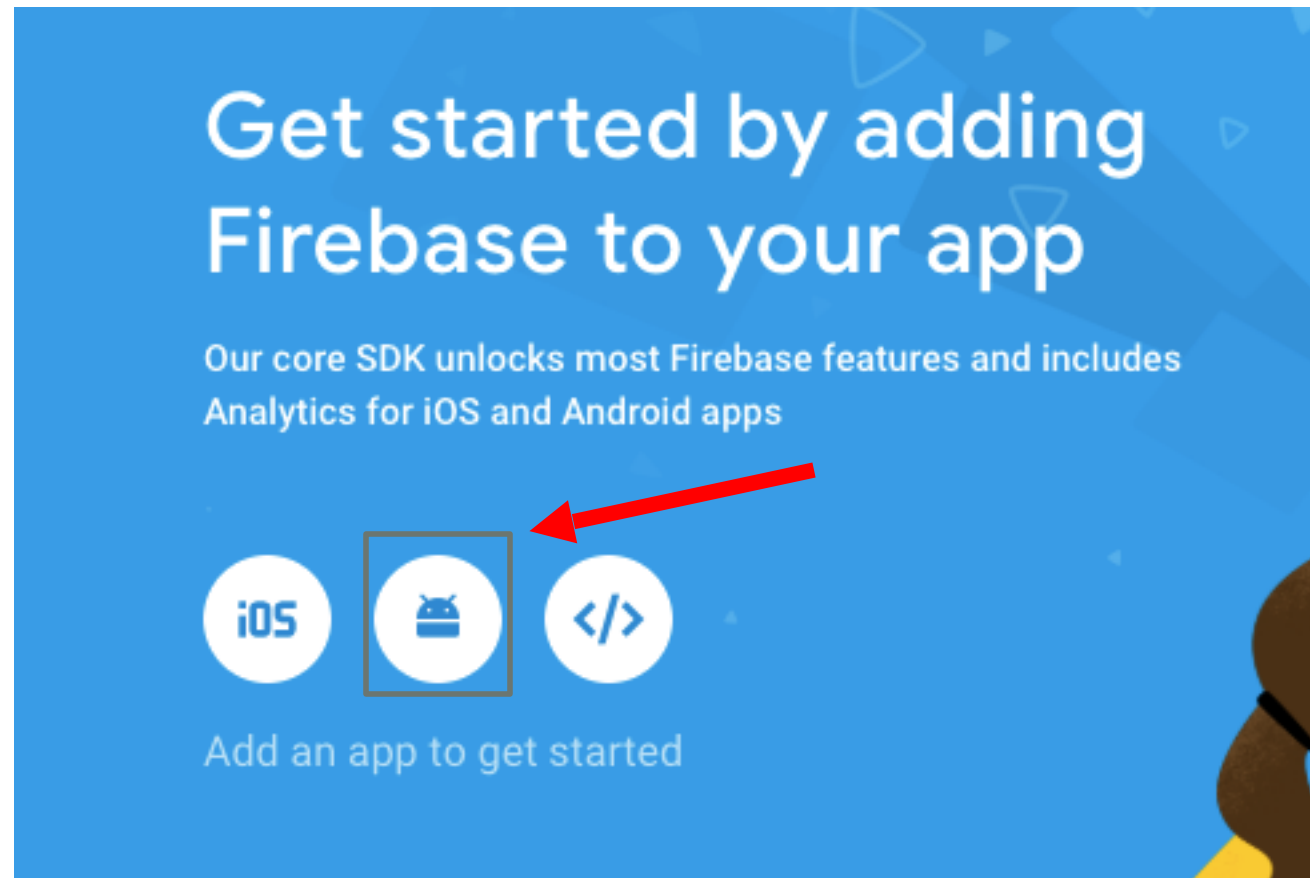
Autenticar e gerenciar usuários

Cloud Firestore

Atualizações em tempo real, consultas eficientes e escalonamento

Step 1

- Choose “Project Overview” and click to add an Android App



Step 1

- Fill correctly with the package name for your app
- Click “Register App”
- Afterwards you’ll be invited to save a “google-services” JSON file.

× Adicionar o Firebase ao seu app para Android

1 Registrar app

Nome do pacote do Android ?



Apelido do app (opcional) ?

Certificado de assinatura de depuração SHA-1 (opcional) ?

Necessário para o Dynamic Links, para o Invites e para o Login do Google ou para receber suporte por telefone no Auth. Edite o SHA-1 nas configurações.

Registrar app


Step 1

Copy the file to the Android app module root directory.

2


Download config file

Instructions for Android Studio below | [Unity](#) [C++](#)

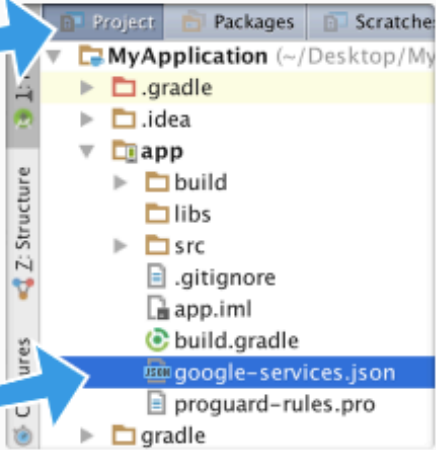
 Download google-services.json

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.


google-services.json

Previous [Next](#)



Step 1

The Google services plugin for Gradle loads the JSON file. Add the SDK to the gradle.

Finally, press **Sync now** in the IDE bar.

Gradle files have changed since last sync

[Sync now](#)

build.gradle no nível do projeto (<project>/build.gradle):

```
buildscript {
    repositories {
        // Check that you have the following line (if not, add it):
        google() // Google's Maven repository
    }
    dependencies {
        ...
        // Add this line
        classpath 'com.google.gms:google-services:4.3.10'
    }
}

allprojects {
    ...
    repositories {
        // Check that you have the following line (if not, add it):
        google() // Google's Maven repository
        ...
    }
}
```

☒ Java ☐ Kotlin

build.gradle no nível do app (<project>/<app-module>/build.gradle):

```
apply plugin: 'com.android.application'
// Add this line
apply plugin: 'com.google.gms.google-services'

dependencies {
    // Import the Firebase BoM
    implementation platform('com.google.firebase:firebase-bom:29.0.1')

    // Add the dependency for the Firebase SDK for Google Analytics
    // When using the BoM, don't specify versions in Firebase dependencies
    implementation 'com.google.firebase:firebase-analytics'

    // Add the dependencies for any other desired Firebase products
    // https://firebase.google.com/docs/android/setup#available-libraries
}
```


Step 2

Add this to your App-level build.gradle (<project>/<app-module>/build.gradle):

```
dependencies {  
    // Import the BoM for the Firebase platform  
    implementation platform('com.google.firebase:firebase-bom:29.0.0')  
  
    // Declare the dependencies for the Firebase Cloud Messaging and Analytics libraries  
    // When using the BoM, you don't specify versions in Firebase library dependencies  
    implementation 'com.google.firebase:firebase-messaging'  
    implementation 'com.google.firebase:firebase-analytics'  
}
```

NOTE: If FCM is critical to the Android app's function, be sure to set *minSdkVersion 19* or higher in the app's *build.gradle*. This ensures that the Android app cannot be installed in an environment in which it could not run properly.

Step 2

Add this to your App manifest:

- A service that extends *FirebaseMessagingService*, for message handling in the foreground:

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

Token Generation

Because the token could be rotated after initial startup, you are strongly recommended to retrieve the latest updated registration token.

```
@Override
public void onNewToken(String token) {
    Log.d(TAG, "Refreshed token: " + token);

    // If you want to send messages to this application instance or
    // manage this apps subscriptions on the server side, send the
    // FCM registration token to your app server.
    sendRegistrationToServer(token);
}
```

Token Retrieval

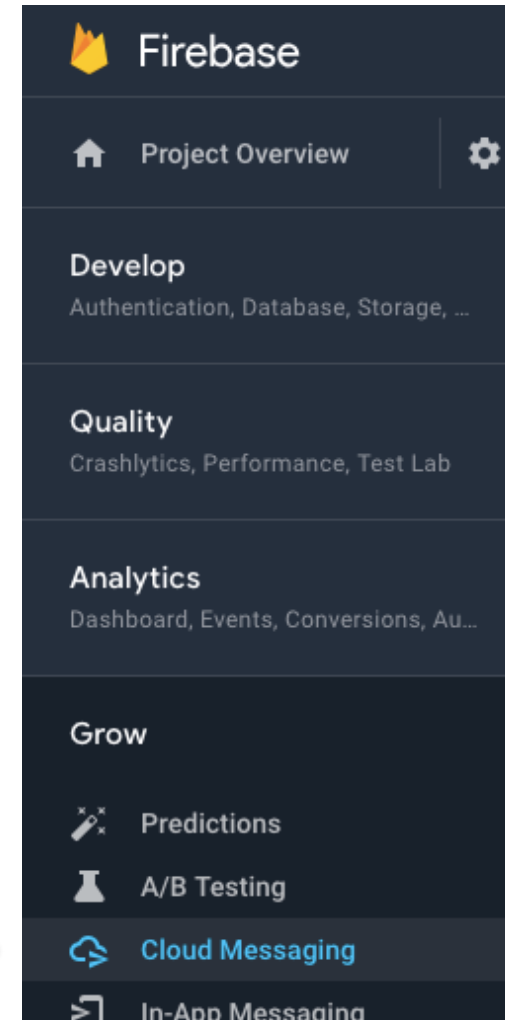
```
FirebaseMessaging.getInstance().getToken()
    .addOnCompleteListener(new OnCompleteListener<String>() {
        @Override
        public void onComplete(@NonNull Task<String> task) {
            if (!task.isSuccessful()) {
                Log.w(TAG, "Fetching FCM registration token failed", task.getException());
                return;
            }

            // Get new FCM registration token
            String token = task.getResult();

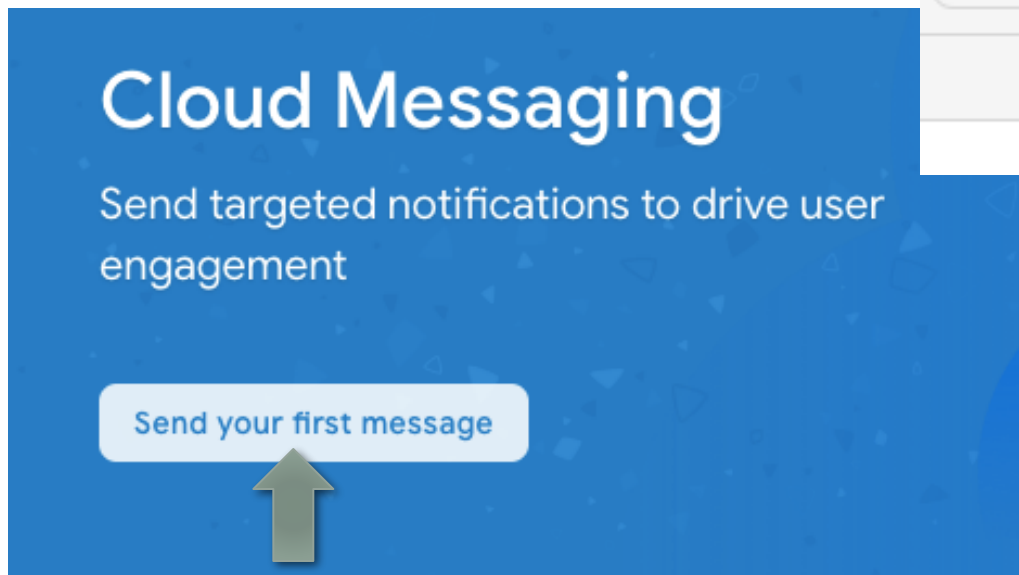
            // Log and toast
            String msg = getString(R.string.msg_token_fmt, token);
            Log.d(TAG, msg);
            Toast.makeText(MainActivity.this, msg, Toast.LENGTH_SHORT).show();
        }
    });
```

Sending messages from the FCM console

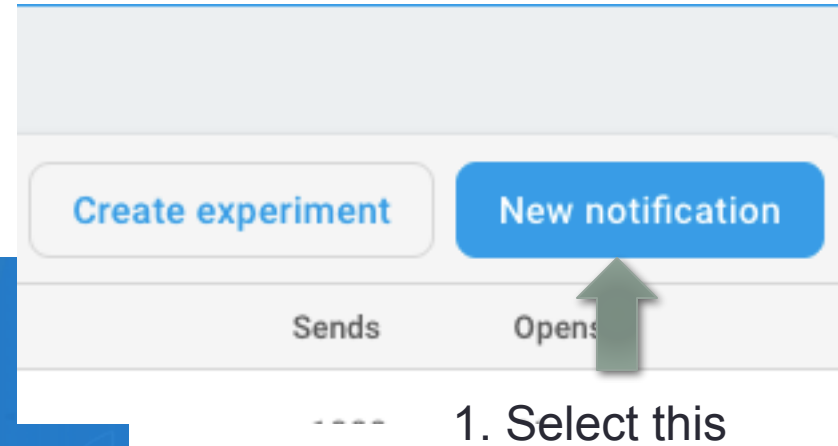
Within the FCM console for your Project, select the **Cloud Messaging** option.



Sending messages from the FCM console



...or this



1. Select this

Notification attributes

1 Notificação

Título da notificação ?

Texto da notificação

Imagem de notificação (opcional) ?



Nome da notificação (opcional) ?

Próxima

Visualização do dispositivo

Por meio dessa visualização, é possível ter uma ideia geral de como sua mensagem será exibida em um dispositivo móvel. Haverá variação na renderização real da mensagem dependendo do dispositivo. Faça o teste com um dispositivo real para resultados reais.

Enviar mensagem de teste

Estado inicial

Visualização expandida



Android



Apple

Target selection

2 Segmentação

Segmento do usuário

Tópico

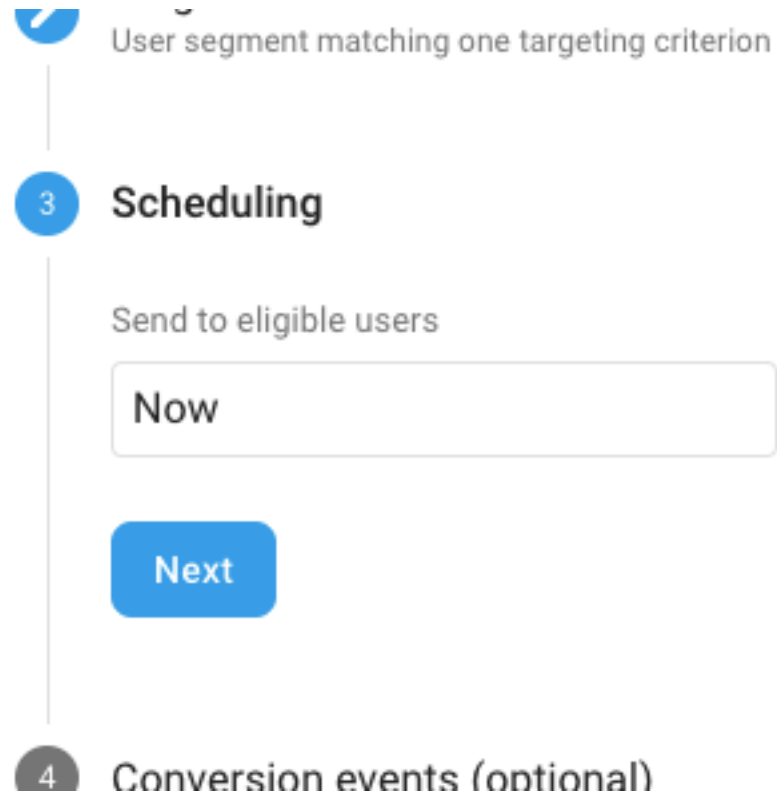
Segmentar usuário se...

| | | | |
|-----------|-----------|----------|---|
| App | 📱 Test ▼ | | |
| Idiomas ▼ | está em ▼ | alemão ▼ | |
| Versão ▼ | >= ▼ | 19 ▼ | e |

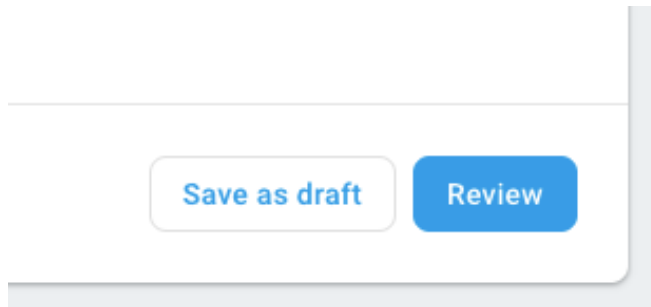
Segmentar outro app

Próxima

Scheduling



Review & publish



All fields optional

Review message

Notification content
Useless message

Target
User segment matching one targeting criterion

Scheduling
Send now

Cancel **Publish**

Report





Cloud Messaging

Notifications

Reports

Create experiment

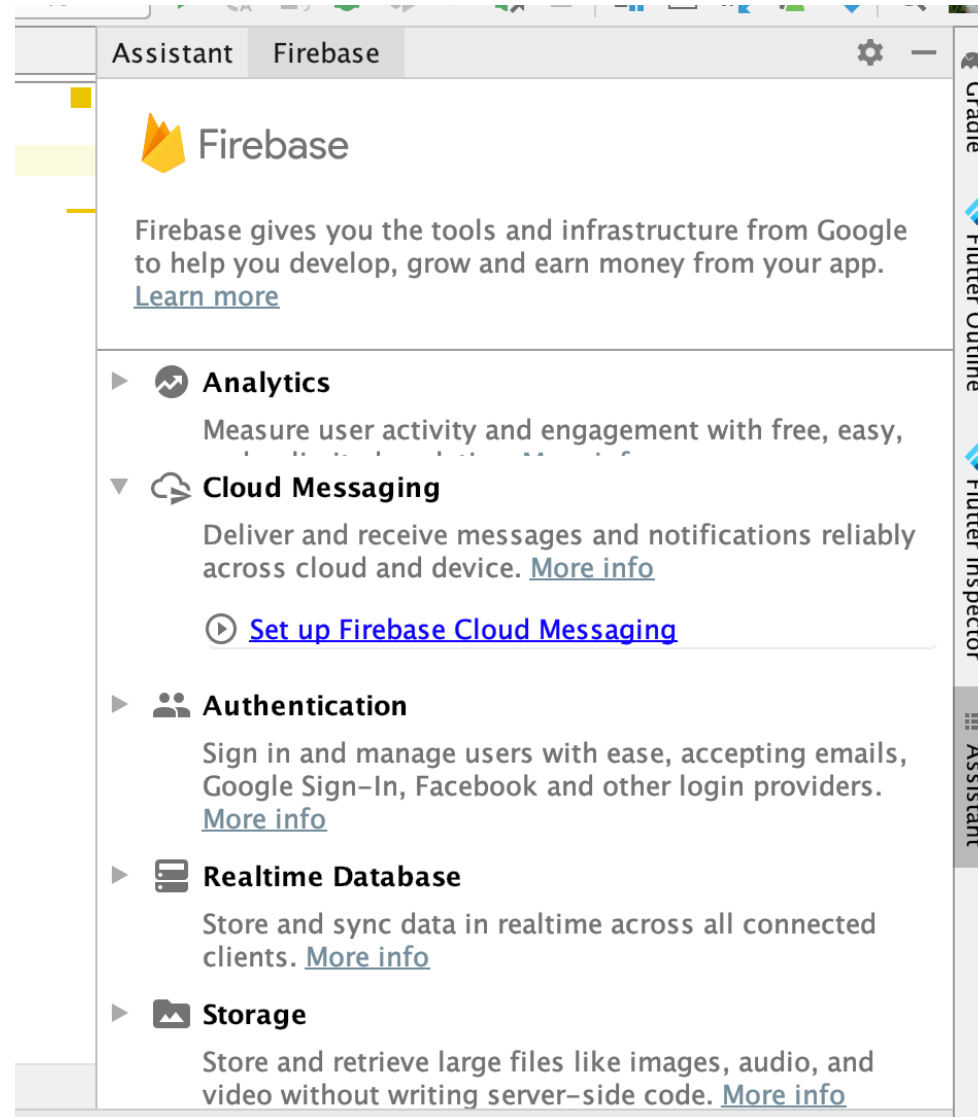
New notification

| Notification | Status ? | Platform | Start / Send | End | Sends | Opens |
|--|-----------------------|---|-------------------------|-----|-------|-------|
|  Useless message | ✓ Completed |  | Dec 2, 2018 11:05 PM | — | <1000 | 0% |
|  Useless message Teste | ✓ Completed |  | Dec 11, 2017 4:52 PM | — | <1000 | 0% |

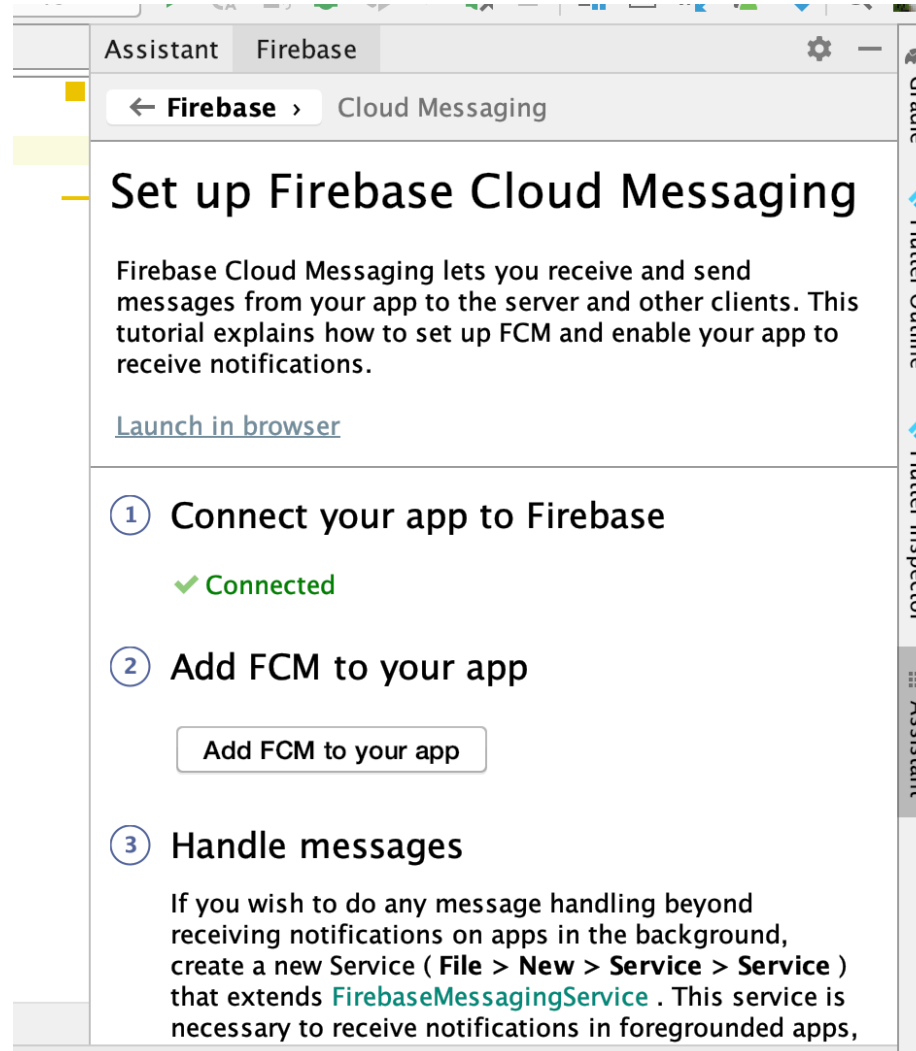
However...

Or... Step 1

- Inside Android Studio
 - Tools > Firebase
 - Select Cloud Messaging



Step 2



The screenshot shows the Flutter Assistant interface with the 'Firebase' tab selected. The breadcrumb navigation shows '← Firebase > Cloud Messaging'. The main content area is titled 'Set up Firebase Cloud Messaging' and includes an introductory paragraph, a 'Launch in browser' link, and a three-step tutorial. Step 1, 'Connect your app to Firebase', shows a green checkmark and 'Connected' status. Step 2, 'Add FCM to your app', features a button labeled 'Add FCM to your app'. Step 3, 'Handle messages', provides instructions on creating a new Service that extends `FirebaseMessagingService`.

Assistant **Firebase** ⚙️

← **Firebase** > Cloud Messaging

Set up Firebase Cloud Messaging

Firebase Cloud Messaging lets you receive and send messages from your app to the server and other clients. This tutorial explains how to set up FCM and enable your app to receive notifications.

[Launch in browser](#)


- 1 Connect your app to Firebase
✓ Connected
- 2 Add FCM to your app
Add FCM to your app
- 3 Handle messages
If you wish to do any message handling beyond receiving notifications on apps in the background, create a new Service (**File > New > Service > Service**) that extends `FirebaseMessagingService` . This service is necessary to receive notifications in foregrounded apps,

Gradle
Flutter Outline
Flutter Inspector
Assistant

Step 2

(you need to link a Google Account to Android Studio)

Connect to Firebase

 **Firebase**

☒ Create new Firebase project [What's this?](#)

Signed in as **jferrovias@gmail.com** [Sign out](#)

☐ Choose an existing Firebase or Google project

1 Android app(s) connected

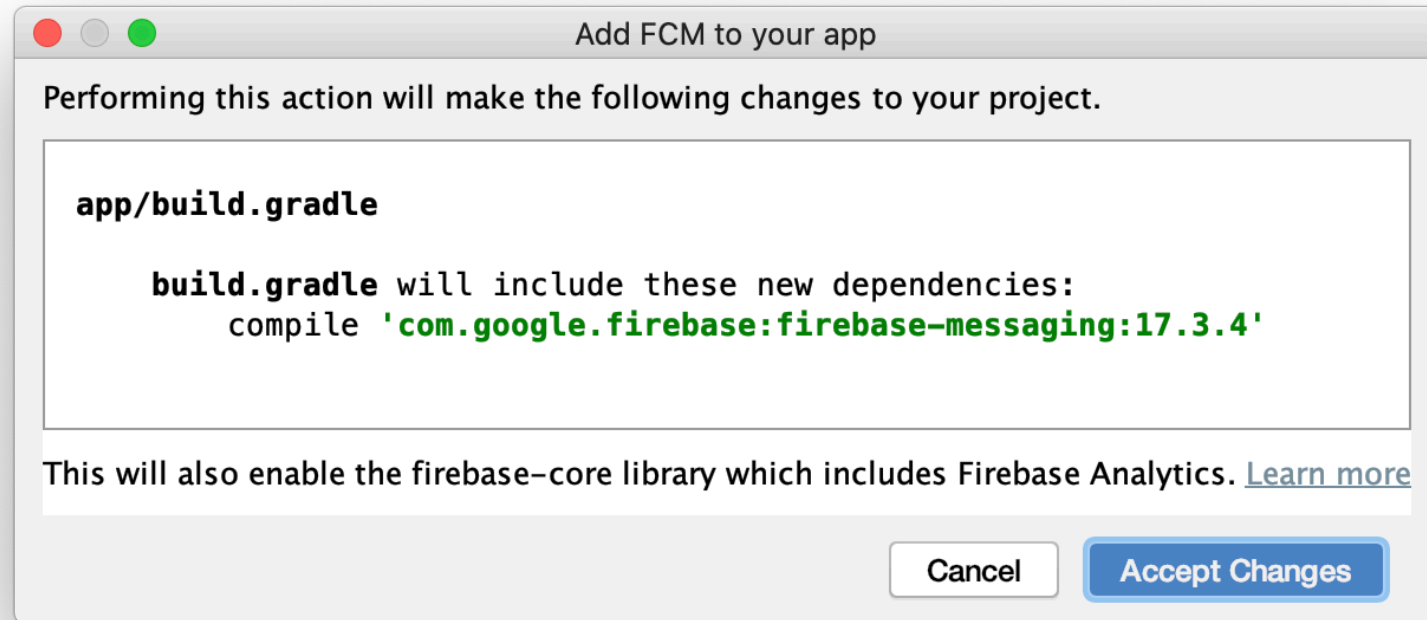
Country/region [What's this?](#)

By default, your Firebase Analytics data will enhance other Firebase features and Google products. You can control how your Firebase Analytics data is shared in your settings at anytime. [Learn more](#)

Cancel

Connect to Firebase

Step 3



Step 3

The screenshot shows the Android Studio IDE with the 'Assistant' tab selected, displaying the 'Firebase > Cloud Messaging' section. The main content area shows the '3 Handle messages' step. The text explains that to handle messages beyond background notifications, a new Service must be created that extends `FirebaseMessagingService`. It then provides a code snippet for the `onMessageReceived` method. Below the code, it instructs the user to declare the service in the application's manifest. The right sidebar contains the 'Gradle', 'Flutter Outline', 'Flutter Inspector', 'Assistant', and 'Device File Explorer' panels. The bottom status bar shows the time as 22:11, the file encoding as UTF-8, and 4 spaces for indentation.

Assistant **Firebase**

← **Firebase** > Cloud Messaging

3 **Handle messages**

If you wish to do any message handling beyond receiving notifications on apps in the background, create a new Service (**File > New > Service > Service**) that extends `FirebaseMessagingService` . This service is necessary to receive notifications in foregrounded apps, to receive data payload, to send upstream messages, and so on.

In this service create an `onMessageReceived` method to handle incoming messages.

```
@Override
public void onMessageReceived(RemoteMessage remote
// ...

// TODO(developer): Handle FCM messages here.
// Not getting messages here? See why this may
Log.d(TAG, "From: " + remoteMessage.getFrom())

// Check if message contains a data payload.
if (remoteMessage.getData().size() > 0) {
    Log.d(TAG, "Message data payload: " + remc

    if (/* Check if data needs to be processed
        // For long-running tasks (10 seconds
        scheduleJob());
    } else {
        // Handle message within 10 seconds
        handleNow();
    }

}

// Check if message contains a notification pe
if (remoteMessage.getNotification() != null) {
    Log.d(TAG, "Message Notification Body: " +

}

// Also if you intend on generating your own r
// message, here is where that should be initi
}
```

Declare the following in your application's manifest:

```
<service android:name=".java.MyFirebaseMessagingSe
<intent-filter>
<action android:name="com.google.firebase.
```

2 Event Log

22:11 LF UTF-8 4 spaces

Step 4

1 Notificação

Título da notificação ?

Test1

Texto da notificação

Test2

Imagem de notificação (opcional) ?

Exemplo: <https://yourapp.com/image.png>

Nome da notificação (opcional) ?

Informa nome opcional

Visualização do dispositivo

Por meio dessa visualização, é possível ter uma ideia geral de como sua mensagem será exibida em um dispositivo móvel. Haverá variação na renderização real da mensagem dependendo do dispositivo. Faça o teste com um dispositivo real para resultados reais.

Enviar mensagem de teste

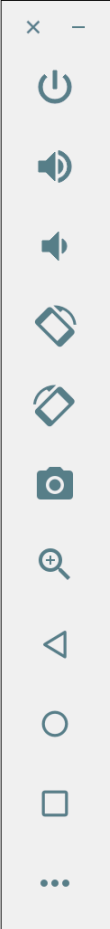
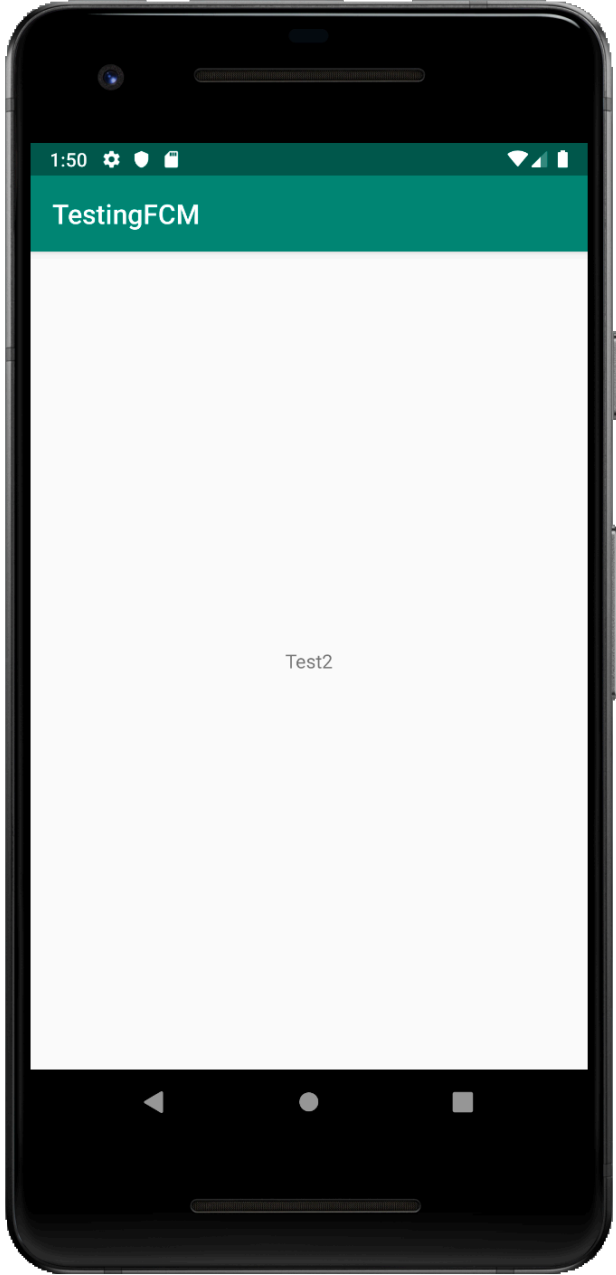
Estado inicial Visualização expandida



Android



iOS



Topic Messaging

Based on the publish/subscribe model, FCM topic messaging allows you to send a message to multiple devices that have opted in to a particular topic. FCM handles routing and delivering the message reliably to the right devices.

Some things to keep in mind about topics:

- Topic messaging supports unlimited topics and subscriptions for each app.
- Topic messaging is best suited for content such as news, weather, or other publicly available information.
- Topic messages are optimized for throughput rather than latency. For fast, secure delivery to single devices or small groups of devices, target messages to tokens, not topics.
- If you need to send messages to multiple devices per user, consider Device Group messaging for those use cases.

Topic subscription and unsubscription

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 with that name is created in FCM and any client can 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("news")`* with the topic name.

FCM – Server considerations

Write server-side app (you may use Tomcat, AppEngine...):

- Fires off properly formatted requests to the FCM server
- Handles requests and resends them as needed, using exponential back-off
- Stores the API key and client registration IDs. The API key is included in the header of POST requests that send messages
- (for XMPP) Generates message IDs to uniquely identify each message it sends. Message IDs should be unique per sender ID
- Must be able to unregister registration ID

You have two ways to interact with the FCM infrastructure: either using the Admin SDK (<https://firebase.google.com/docs/cloud-messaging/admin/> - compatible with Python, Java, C#, etc) or the raw protocols (FCM HTTP v1 API, legacy HTTP or XMPP).

More details at:

<https://firebase.google.com/docs/cloud-messaging/server>

Cloud Firestore

Firebase offers two cloud-based, client-accessible database solutions that support realtime data syncing:

- **Cloud Firestore** is Firebase's newest database for mobile app development. It builds on the successes of the Realtime Database with a new, more intuitive data model. Cloud Firestore also features richer, faster queries and scales further than the Realtime Database.
- **Realtime Database** is Firebase's original database. It's an efficient, low-latency solution for mobile apps that require synced states across clients in realtime.

Cloud Firestore

alovelace

```
first : "Ada"  
last  : "Lovelace"  
born  : 1815
```

alovelace

```
name :  
  first : "Ada"  
  last  : "Lovelace"  
born  : 1815
```

Documents live in collections, which are simply containers for documents. For example, you could have a users collection to contain your various users, each represented by a document:

users

alovelace

```
first : "Ada"  
last  : "Lovelace"  
born  : 1815
```

aturing

```
first : "Alan"  
last  : "Turing"  
born  : 1912
```

Cloud Firestore

Every document in Cloud Firestore is uniquely identified by its location within the database. The previous example showed a document `alovelace` within the collection `users`. To refer to this location in your code, you can create a *reference* to it.

```
DocumentReference alovelaceDocumentRef =  
db.collection("users").document("alovelace");
```

You can also create references to collections:

```
CollectionReference usersCollectionRef = db.collection("users");
```

For convenience, you can also create references by specifying the path to a document or collection as a string, with path components separated by a forward slash (/). For example, to create a reference to the **alovelace** document:

```
DocumentReference alovelaceDocumentRef = db.document("users/alovelace");
```


Store Data

There are several ways to write data to Cloud Firestore:

- Set the data of a document within a collection, explicitly specifying a document identifier.
- Add a new document to a collection. In this case, Cloud Firestore automatically generates the document identifier.
- Create an empty document with an automatically generated identifier, and assign data to it later.

Initialize an instance of Cloud Firestore:

```
// Access a Cloud Firestore instance from your Activity
FirebaseFirestore db = FirebaseFirestore.getInstance();
```

Store Data

```
Map<String, Object> city = new HashMap<>();
city.put("name", "Los Angeles");
city.put("state", "CA");
city.put("country", "USA");

db.collection("cities").document("LA")
    .set(city)
    .addOnSuccessListener(new OnSuccessListener<Void>() {
        @Override
        public void onSuccess(Void aVoid) {
            Log.d(TAG, "DocumentSnapshot successfully written!");
        }
    })
    .addOnFailureListener(new OnFailureListener() {
        @Override
        public void onFailure(@NonNull Exception e) {
            Log.w(TAG, "Error writing document", e);
        }
    });
```

If the document does not exist, it will be created. If the document does exist, its contents will be overwritten with the newly provided data, unless you specify that the data should be merged into the existing document.

Store Data (Data types)

Cloud Firestore lets you write a variety of data types inside a document, including strings, booleans, numbers, dates, null, and nested arrays and objects. Cloud Firestore always stores numbers as doubles, regardless of what type of number you use in your code.

```
Map<String, Object> docData = new HashMap<>();
docData.put("stringExample", "Hello world!");
docData.put("booleanExample", true);
docData.put("numberExample", 3.14159265);
docData.put("dateExample", new Timestamp(new Date()));
docData.put("listExample", Arrays.asList(1, 2, 3));
docData.put("nullExample", null);
```

```
Map<String, Object> nestedData = new HashMap<>();
nestedData.put("a", 5);
nestedData.put("b", true);
```

```
docData.put("objectExample", nestedData);
```

Store Data (Custom objects)

Using Map or Dictionary objects to represent your documents is often not very convenient, so Cloud Firestore supports writing documents with custom classes. Cloud Firestore converts the objects to supported data types.

Each custom class must have a public constructor that takes no arguments. In addition, the class must include a public getter for each property.

```
public class City {  
    private String name;  
    private String country;  
  
    public City() {}  
  
    public City(String name, String country) {  
        // ...  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String getCountry() {  
        return country;  
    }  
}
```

```
City city = new City("Los Angeles", "USA");  
db.collection("cities").document("LA").set(city);
```

Get Data

The following example shows how to retrieve the contents of a single document using `get()`:

```
DocumentReference docRef = db.collection("cities").document("SF");
docRef.get().addOnCompleteListener(new OnCompleteListener<DocumentSnapshot>() {
    @Override
    public void onComplete(@NonNull Task<DocumentSnapshot> task) {
        if (task.isSuccessful()) {
            DocumentSnapshot document = task.getResult();
            if (document.exists()) {
                Log.d(TAG, "DocumentSnapshot data: " + document.getData());
            } else {
                Log.d(TAG, "No such document");
            }
        } else {
            Log.d(TAG, "get failed with ", task.getException());
        }
    }
});
```

Get Data

The previous example retrieved the contents of the document as a map, but in some languages it's often more convenient to use a custom object type. In the previous example you defined a City class that you used to define each city. You can turn your document back into a City object:

```
DocumentReference docRef = db.collection("cities").document("LA");
docRef.get().addOnSuccessListener(new
    OnSuccessListener<DocumentSnapshot>() {
        @Override
        public void onSuccess(DocumentSnapshot documentSnapshot) {
            City city = documentSnapshot.toObject(City.class);
        }
    });
```

Get Data

In addition, you can retrieve all documents in a collection:

```
db.collection("cities")
    .get()
    .addOnCompleteListener(new OnCompleteListener<QuerySnapshot>() {
        @Override
        public void onComplete(@NonNull Task<QuerySnapshot> task) {
            if (task.isSuccessful()) {
                for (QueryDocumentSnapshot document : task.getResult()) {
                    Log.d(TAG, document.getId() + " => " +
document.getData());
                }
            } else {
                Log.d(TAG, "Error getting documents: ", task.getException());
            }
        }
    });
```

Get Data

You can also retrieve multiple documents with one request by querying documents in a collection. For example, you can use `where()` to query for all of the documents that meet a certain condition, then use `get()` to retrieve the results:

```
db.collection("cities")
    .whereEqualTo("capital", true)
    .get()
    .addOnCompleteListener(new OnCompleteListener<QuerySnapshot>() {
        @Override
        public void onComplete(@NonNull Task<QuerySnapshot> task) {
            if (task.isSuccessful()) {
                for (QueryDocumentSnapshot document : task.getResult()) {
                    Log.d(TAG, document.getId() + " => " + document.getData());
                }
            } else {
                Log.d(TAG, "Error getting documents: ", task.getException());
            }
        }
    });
```


And that's all for today.