

# The AnDev Guide to Firebase

by Paul Trebilcox-Ruiz

 **AnDevCon**  
The Android Developer Conference

# Introduction

Although Firebase has been around for some time, it received a massive overhaul recently and was reintroduced to the development community during Google's I/O conference. Firebase is a service from Google that provides multiple features to developers so that they can easily put together a backend for their apps and web pages, and interact with their users.

This tutorial is the first in a series that will dive into the current state of Firebase and the great features that are available to help you quickly build apps that serve your users. In this first installment, you will be introduced to the setup process for Firebase and how to implement it into your Android application. You will then learn about the Firebase Real-time Database, which is arguably the backbone of Firebase. In order to demonstrate how Firebase works, I will use a fictitious 'bad jokes' app to demonstrate implementation details.

## Setting Up the Firebase Console

Before you can start using Firebase in your applications, you will need to create a Firebase project through Google. You can start by going to [Google's Firebase console](#) site and clicking on the blue **Create New Project** button.

# Welcome to Firebase

Tools from Google for developing great apps, engaging with your users, and earning more through mobile ads. [Learn more](#)

CREATE NEW PROJECT

[or import a Google project](#)

Next you will receive a dialog window asking you to name your project and select a country/region. For this tutorial I have created a project called *AnDevCon-BadJokes*, and I have selected the *United States* for the country/region, which will end up displaying all currency information in US dollars.

## Create a project

×

Project name

AnDevCon-BadJokes

Country/region ?

United States ▼

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](#)

**By proceeding and clicking the button below**, you agree that you are using Firebase services in your app and agree to the applicable [terms](#).

CANCEL

CREATE PROJECT

After you hit the blue **Create Project** button, Firebase will generate your project and take you to the Firebase console screen.

## Add Firebase to your Android App

On the main Firebase console screen, you should see three options for adding Firebase to a project. For this tutorial you will want to click on the green **Add Firebase to your Android App** button.

Welcome to Firebase! Get started here.



Add Firebase to  
your Android app



Add Firebase to  
your iOS app



Add Firebase to  
your web app

After you have clicked on the above button, a new dialog will open that will ask you for the necessary information for getting Firebase into your Android application. You will need to enter the package name for your app. In this tutorial I will use *com.andevcon.andevcon\_badjokes* for the package name.

You also have the option of adding the SHA1 for your signing key, which is required in order to use features such as app invites, authentication and dynamic links. For this tutorial I will use the SHA1 for my **debug.keystore**, which you can get from a terminal with the following command.

```
keytool -list -v -keystore ~/.android/debug.keystore -alias androiddebugkey -storepass android -keypass android
```

On Windows you can run the same command with the path set to the location of your debug.keystore file.

## Add Firebase to your Android app

1

2

3

Enter app details

Copy config file

Add to build.gradle

Package name ?

com.andevcon.andevcon\_badjokes

Debug signing certificate SHA-1 (optional) ?

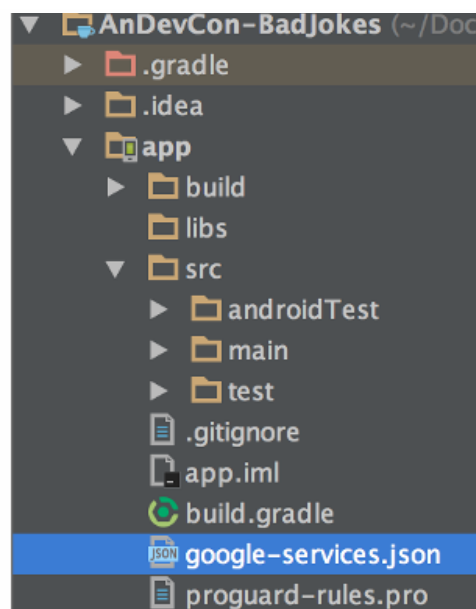
F9:DD:F3:EA:73:3F:18:E4:7D:87:8F:37:E2:C9:80:BB:FA:BF:AD:A5

Required for Dynamic Links, Invites, and Google Sign-In support in Auth. Edit SHA-1s in Settings.

CANCEL

ADD APP

When you click on the blue Add App button, a file named google-services.json will be generated and downloaded to your computer. You will need to place this file into the app/ folder of your project.



Finally, you will need to include the Google Play Services plugin in your project in order to load the JSON file that you just placed into your project. You can do this by opening the project-level `build.gradle` file (the one a directory higher than where you just placed `google-services.json`) and adding a new classpath under the buildscript -> dependencies node.

```
buildscript {  
    repositories {  
        jcenter()  
    }  
    dependencies {  
        classpath 'com.android.tools.build:gradle:2.1.0'  
        classpath 'com.google.gms:google-services:3.0.0'  
    }  
}
```

Once you have added the plugin to the project-level **build.gradle** file, you will need to apply it in your module level **build.gradle** file. At the bottom of the file, add the following line.

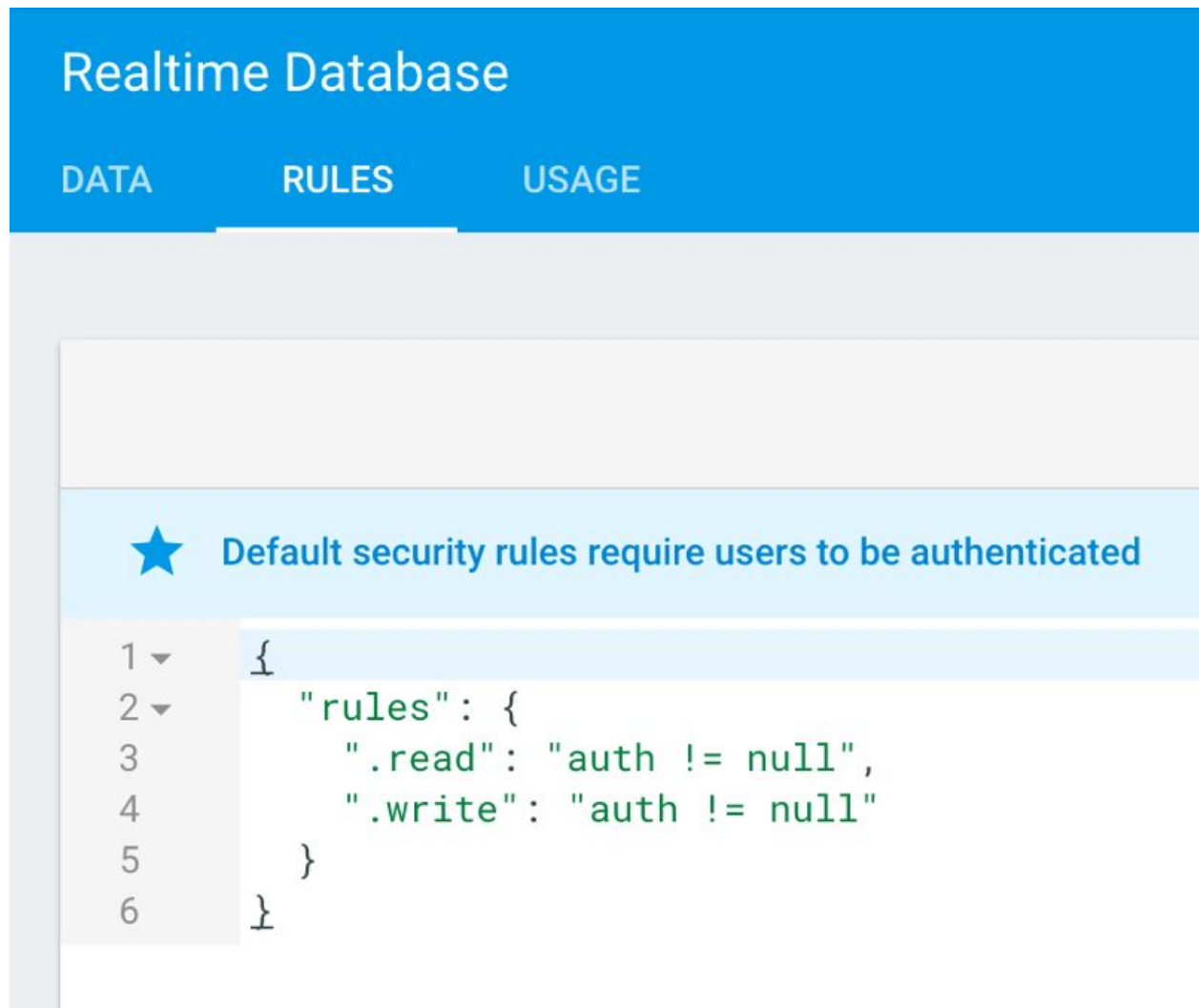
```
apply plugin: 'com.google.gms.google-services'
```

At this point you should be able to start using Firebase in your project.

## Database Access

One of the core feature of Firebase is that it allows developers to create an easy to use NoSQL database that can be accessed from a client app, known as the Firebase Real-time Database. This backend can push updates to all clients when data is changed in real-time, can be easily edited from the Firebase console, and supports storing data in the app when the device has gone offline.

One thing to note is that, by default, users must be authenticated in order to read and write data from your Firebase database. For this tutorial we will make all data both readable and writable, though in your actual projects you should carefully consider your security requirements before making these changes. Under the **database** section of the Firebase console, switch to the **Rules** tab.



In the JSON that comes up, change the `auth != null` items to `auth == null`. This will allow anyone using your app to read and write data in your Firebase backend without having to first authenticate.



## Data Structure and Types

Before you can start using the Firebase Real-time Database, you will need a general understanding of how the data is stored and structured. Luckily, Firebase uses a standard JSON tree rather than records and tables, so the typical types that you would expect in a JSON tree are available.

- String
- Boolean
- Long
- Double
- Map<String, Object>
- List<Object>

Firebase is great for representing simple data, but where things get a little tricky is when you need to use arrays. While working with arrays in Firebase and techniques for structuring your data are beyond the scope of this tutorial, you can get some great information on arrays from this [official Firebase blog post](#).

There is one design consideration that you should keep in mind: although Firebase allows objects to be nested up to 32 layers deep, you should try and keep your JSON tree structure as flat as possible, as any object that is loaded by the client will load all child objects as well. An example list of jokes for the tutorial app could look like this:



To add your own data, go into the **Database** section of the Firebase Console and click on the green plus mark to add a **jokes** container object, and then press on the green plus mark to add additional objects to that container.

## Initializing the Firebase Real-time Database

To use the Firebase Real-time Database in your Android app, you will need to import the **firebase-database** library into your project. You can do this by adding the following line to your module level **build.gradle** file under the **dependencies** node. Note that the version number will change, so you will want to check the [Firebase documentation for the latest version number](#).

```
compile 'com.google.firebase:firebase-database:9.2.1'
```

Once you have the Firebase Real-time Database library imported into your project, you will need to create a Firebase database reference. For this tutorial we will create our reference from a URL pointing to our Firebase **jokes** object.

```
DatabaseReference databaseRef = FirebaseDatabase.getInstance().getReferenceFromUrl(JOKES_URL);
```

Now that you have a reference to your Firebase Real-time Database, you can start reading and writing values from it.

## Reading Data from Firebase in Android

No matter how much data you have in your database, it's not going to do your app any good if you can't read that data into your client application. When retrieving values from the Firebase database, you have two options: the `ValueEventListener` and the `ChildEventListener`.

### ValueEventListener

`ValueEventListener`, when attached to your `DatabaseReference`, will call `onDataChanged(DataSnapshot snapshot)` while initializing and then any time a reference or any of its children has changed. In our bad jokes app, we can update a list of all jokes by pointing to the jokes node and adding a `ValueEventListener`.

```
DatabaseReference databaseRef = FirebaseDatabase.getInstance().getReferenceFromUrl(JOKES_URL);
databaseRef.addValueEventListener(new ValueEventListener() {

    @Override

    public void onDataChange(DataSnapshot dataSnapshot) {

        mAdapter.clear();

        for( DataSnapshot snapshot : dataSnapshot.getChildren() ) {

            mAdapter.add(snapshot.getValue(Joke.class));

        }

        mAdapter.notifyDataSetChanged();

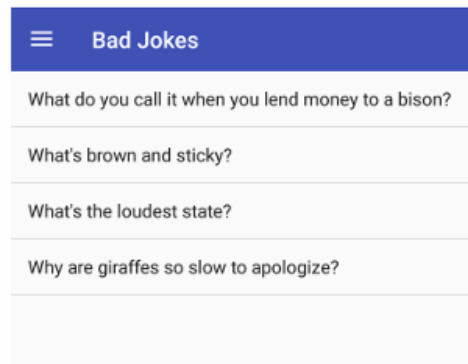
    }

    @Override

    public void onCancelled(DatabaseError databaseError) {
```

```
}  
});
```

When this code is run in the app, our `ListView` will display each question from the Firebase Real-time Database. If any items are added, the list will clear and add each item again.



## ChildEventListener

The other option for reading data is the `ChildEventListener`, which has various methods related to changes in child objects on your `DatabaseReference`. There are four methods related to the children of the reference that exist within this listener: `onChildAdded`, `onChildRemoved`, `onChildChanged`, and `onChildMoved`.

`onChildAdded` is called for each child of the reference when the listener is first added to the `DatabaseReference`. It is also called whenever a new child object is added to the reference.

`onChildChanged` is called whenever a child is changed at all, such as adding or removing a new property from that child in the Firebase JSON tree.

`onChildRemoved`, as you've probably guessed, is called when an item is removed.

Finally, `onChildMoved`, is called whenever a child item is modified in a way that would change the order of the items in the returned JSON tree. This method

only applies when you are using a Query created from a DatabaseReference that sorts received items by using orderByValue() or orderByChild(String child)

While it may seem like there's a lot going on here, implementation is just as straight forward as using the ValueEventListener.

```
DatabaseReference databaseRef = FirebaseDatabase.getInstance().getReferenceFromUrl(JOKES_URL);

//Query is only being used because of onChildMoved.

//Can call addChildEventListener on databaseRef

Query queryOrderByChild = databaseRef.orderByChild("question");

queryOrderByChild.addChildEventListener(new ChildEventListener() {

    @Override

    public void onChildAdded(DataSnapshot dataSnapshot, String s) {

        Log.e(TAG, "onChildAdded");

    }

    @Override

    public void onChildChanged(DataSnapshot dataSnapshot, String s) {

        Log.e(TAG, "onChildChanged");

    }

    @Override

    public void onChildRemoved(DataSnapshot dataSnapshot) {

        Log.e(TAG, "onChildRemoved");

    }

    @Override

    public void onChildMoved(DataSnapshot dataSnapshot, String s) {

        Log.e(TAG, "onChildMoved");

    }

})
```

```

    }

    @Override
    public void onCancelled(DatabaseError databaseError) {

    }

});

```

Now that you've gone over the two types of read listeners, it's time to add some data to the Firebase backend.

## Writing Data to Firebase in Android

While reading from Firebase is a huge help for almost any app, most apps will also need to be able to also write data to the backend. You do have a few options here, and which option you use depends on the needs of your app.

### New and Replaced Items

The first use-case for writing data to Firebase is when you need to add a new value or completely overwrite a value that already exists within the JSON tree. You can do this using the `setValue` method. You will need to provide the exact location and key for the item that you want to add, and if that item already exists, it will be completely overwritten with all of its children.

```

Joke joke = new Joke();

joke.setAnswer("a tuba glue!");
joke.setQuestion("What do you use to fix a broken tuba?");

DatabaseReference databaseRef = FirebaseDatabase.getInstance().getReferenceFromUrl(JOKES_URL);

databaseRef.child("99").setValue(joke);

```

This method will allow you to add a complete object to the JSON tree, or override a single value of an object in Firebase.

## Changing Multiple Items without Overwriting Children

If you want to write to specific nodes in the JSON tree without overwriting any of their child nodes, you can use the `updateChildren` method. You will need to create a `Map<String, Object>` representing the paths to the nodes that you want to update and the nodes that will be changed. You are able to change multiple nodes with one action using this method. Although the bad jokes app does not lend itself to a multi-tiered tree, you can see a very simple example of overwriting multiple items in Firebase using `updateChildren`.

```
Joke joke = new Joke();

joke.setAnswer("a tuba glue!");

joke.setQuestion("What do you use to fix a broken tuba?");

DatabaseReference databaseRef = FirebaseDatabase.getInstance().getReferenceFromUrl(JOKES_URL);

Map<String, Object> childUpdates = new HashMap<>();

childUpdates.put("/0", joke.toMap());

childUpdates.put("/1", joke.toMap());

databaseRef.updateChildren(childUpdates);
```

## Generating Unique Keys in a List

In the above two methods, you will need to know the key for the node that you want to write. If you simply want to add an item to a list without worrying about conflicts, the easiest way to do this is by using the `push()` method. When you use `push`, Firebase will create a new unique key at the specified path, and you can retrieve this key in your app using the `getKey()` method. You can then use this to reference the new item in the database without worrying about overwriting any already existing data or running into conflicts when multiple people are using your app.

```
Joke joke = new Joke();

joke.setAnswer("a tuba glue!");

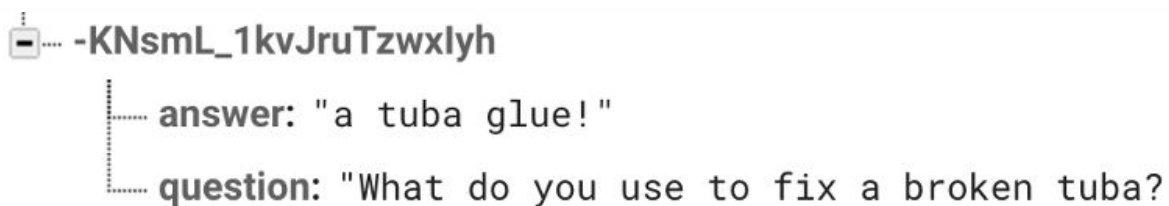
joke.setQuestion("What do you use to fix a broken tuba?");

DatabaseReference databaseRef = FirebaseDatabase.getInstance().getReferenceFromUrl(JOKES_URL);

String key = databaseRef.push().getKey();

databaseRef.child(key).setValue(joke);
```

When you look at the data in the Firebase Console database view, an item similar to the following should appear with a new unique key.



```
-KNsmL_1kvJruTzwxlyh
├── answer: "a tuba glue!"
└── question: "What do you use to fix a broken tuba?"
```

## Avoiding Conflicts When Updating

The last situation that I will cover in this section is when you have an item that may be updated by multiple users simultaneously. Let's say our joke app has the ability to "+1" a joke when a user enjoys it. If multiple users attempt to do this at the same time, we could end up losing some of the data if a conflict arises using `setValue`. In order to solve this, Firebase includes the concept of **transactions**, which allow one piece of data to be updated by multiple users without losing track of any updates. This is done by attempting to submit a value from a user, and if a conflict arises, an error is sent back to the transaction so that the app can attempt again with the current updated information.

To do this, you will need a reference to the item that you want to update, then you will run a new transaction with a `Transaction.Handler`. This Handler will grab the value from the database, update it, and then submit it to Firebase to



either update the value in the backend or receive a rejection so that the app can try again.

```
DatabaseReference databaseRef = FirebaseDatabase.getInstance().getReferenceFromUrl(JOKES_URL +
"/3");

databaseRef.runTransaction(new Transaction.Handler() {

    @Override

    public Transaction.Result doTransaction(MutableData mutableData) {

        Joke joke = mutableData.getValue(Joke.class);

        if( joke == null ) {

            return Transaction.success(mutableData);

        }

        joke.plusOned++;

        mutableData.setValue(joke);

        return Transaction.success(mutableData);

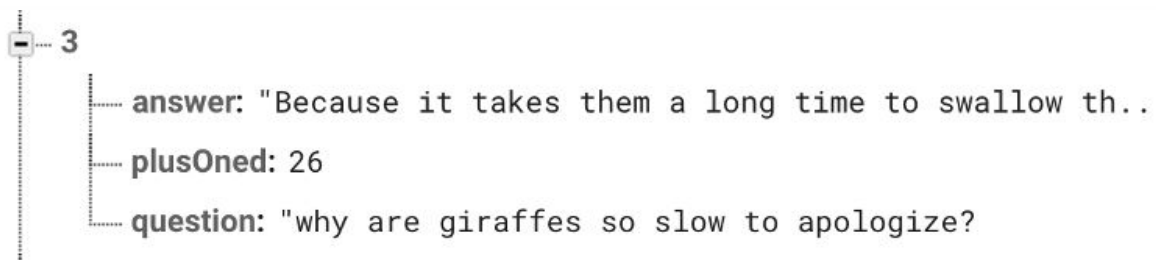
    }

    @Override

    public void onComplete(DatabaseError databaseError, boolean b, DataSnapshot dataSnapshot)
{

    }

});
```



```
3
answer: "Because it takes them a long time to swallow th..
plusOned: 26
question: "why are giraffes so slow to apologize?"
```

## Data Persistence

This has all been fine and dandy, until you realize that your app won't be able to display any data if the user loses their internet connection. You can get around this by setting Firebase to keep data persistent. Once you've done that, your user will be able to close their app and come back without an internet connection, and still be able to use the last bit of data that was pulled down.

```
FirebaseDatabase.getInstance().setPersistenceEnabled(true);
```

While this handles reading while the user is offline, you're probably wondering how you can handle writing events if the user loses connection. The good news here is that Firebase handles this automatically by first writing to a local datastore, and then syncing once a connection has been established.

## **Firebase Database UI Library**

Although I won't go into the implementation details in this tutorial, I do want to address the point that there is an easier way to handle showing items in a UI based off of the Firebase database. Google has provided open sourced libraries called FirebaseUI for Android and iOS. The Android version can be found [here on Github](#). Using the Android FirebaseUI library, you can easily create a `FirebaseListAdapter` or `FirebaseRecyclerViewAdapter` that handles syncing data with your Firebase backend automatically, removing a lot of the hassle associated with displaying a list using changing data.

## **Finishing Up**

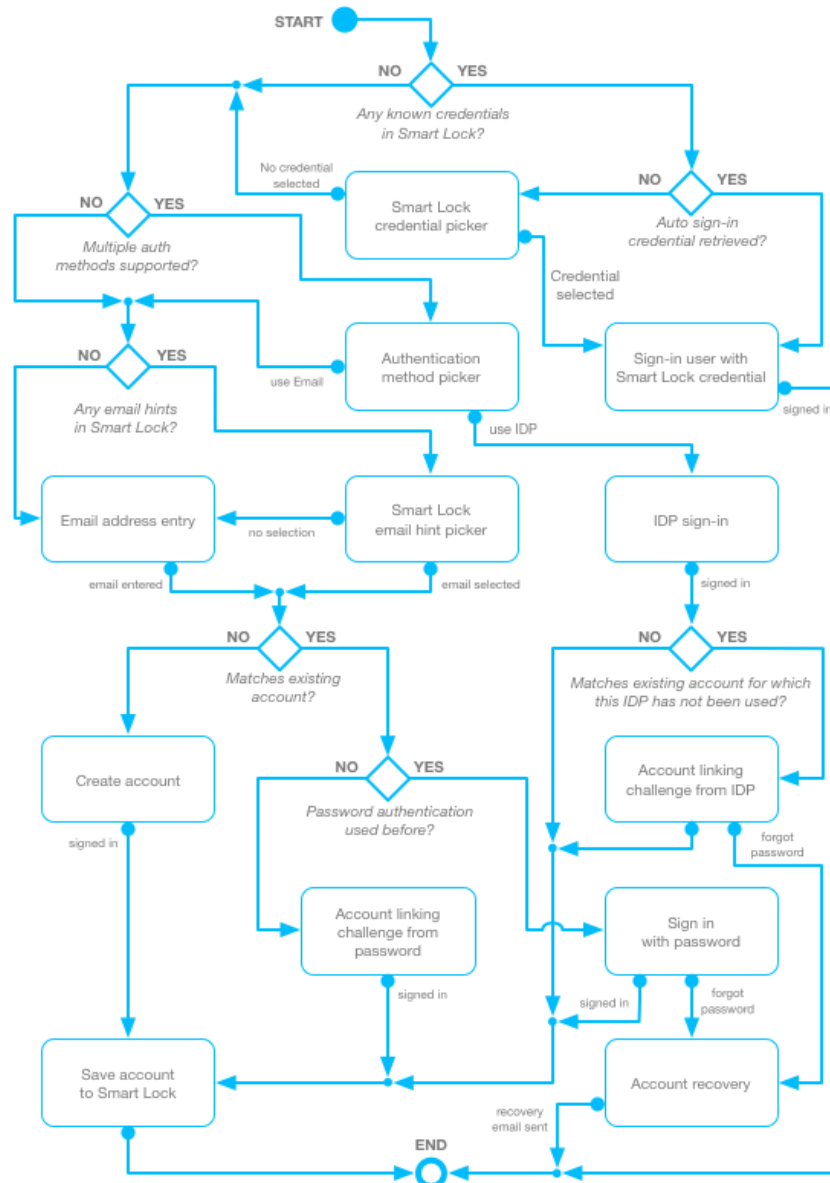
In this tutorial you've covered a lot! You've gone over how to setup Firebase, read items from the Firebase Real-time Database and keep your app in sync with the backend, and you've learned various ways to write data to the backend from your client apps. In the next tutorial in this series you will go over authenticating users so that you can keep track of who can read and write data from your Firebase database.

## Firestore Authentication

As you may have guessed, the user creation and login processes can be complicated and cumbersome. Not only does each provider have its own unique caveats, but keeping track of Android device and authentication state is a beast in and of itself, as can be seen in the following flow chart.

When working with data in Firestore, there will be times when you will want to validate a user to ensure that they should be allowed to access the requested information. Luckily, Firestore provides an authentication framework which will allow you to log users into your app using a username and password, or one of the approved authentication providers, such as Google, Facebook, Twitter or GitHub.

(Continued on next page)



Luckily, the [FirebaseUI library](#) also includes an easy to implement component to handle creating users and authenticating. While you can authenticate with a username and password, this tutorial will focus on using the FirebaseUI library to create and authenticate a user through the Google authentication provider with Firebase. Once you are able to add a user to Firebase, you will learn how to work with the `FirebaseUserObject`.

## Setting Up Authentication in Firebase

Before you begin, you will need to enable authentication in the Firebase console. Under the **Auth** section, select the **Sign-In Method** tab.

# Authentication







USERS

SIGN-IN METHOD

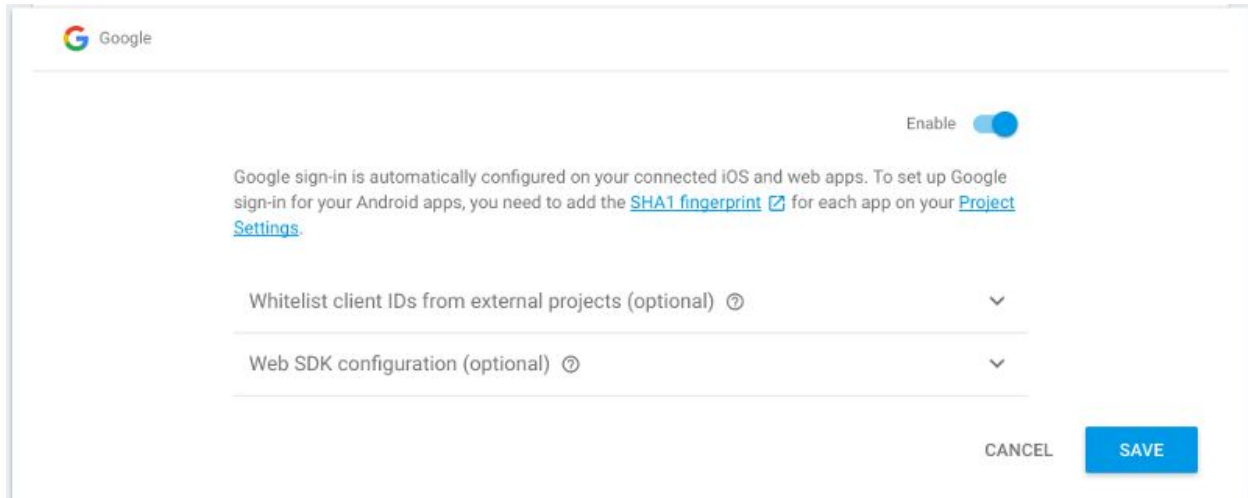
EMAIL TEMPLATES

From here you should see a list of available sign-in providers.

## Sign-in providers

Provider	Status
 Email/Password	Disabled
 Google	Disabled
 Facebook	Disabled
 Twitter	Disabled
 GitHub	Disabled
 Anonymous	Disabled

For this tutorial you will only need to select the Google provider. In the dialog window that opens, click on the toggle in the top right corner to enable the Google sign-in provider. Once this is done, click on the blue **SAVE** button.



Now that you have enabled the Google sign-in provider, it's time to dive into your Android application. One thing to note is that the SHA1 key for your signing key must be registered with Firebase, as covered in the [previous tutorial in this series](#).

## Authenticating with the FirebaseUI Library

Under the **dependencies** node of your app module's **build.gradle** file, import the FirebaseUI auth library. Note that the version number will change as Firebase and the FirebaseUI library continue to be developed. The FirebaseUI auth library also imports the Firebase and Google Play Services libraries that are required for Firebase authentication, so you don't need to worry about those.

```
compile 'com.firebaseui:firebase-ui-auth:0.4.3'
```

Once you have imported the library, you will need to create an Activity to handle authentication. For this tutorial I will simply use the app's MainActivity class. When you want users to sign into your app, you will need to get a reference to the FirebaseAuth object. If the FirebaseAuth `getCurrentUser()` method returns a user, then the user is already signed in. If no user is associated with the FirebaseAuth object, then you can start the FirebaseUI's sign-in Activity with a list of sign-in providers that your app supports.

```
private void handleAuthentication() {  
    FirebaseAuth auth = FirebaseAuth.getInstance();  
    if (auth.getCurrentUser() != null) {  
        //User is signed in already. You're good to go!  
    } else {  
        startActivityForResult(  

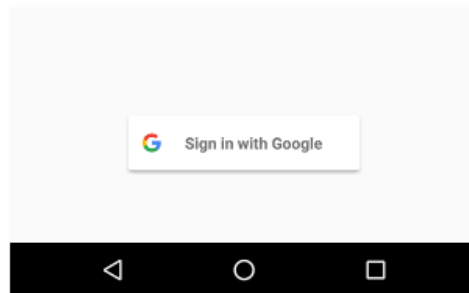
```

```

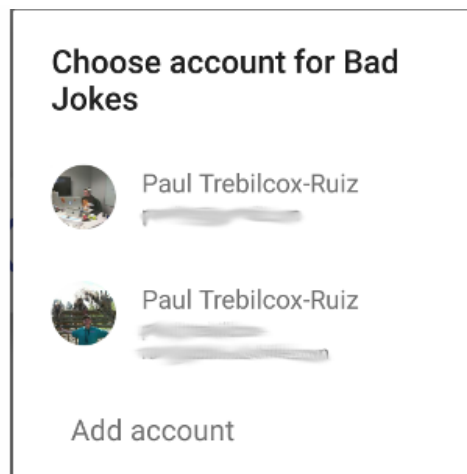
        AuthUI.getInstance()
            .createSignInIntentBuilder()
            .setProviders(AuthUI.GOOGLE_PROVIDER)
            .build(),
        SIGN_IN_REQUEST_CODE);
    }
}

```

The above method will display a list of buttons representing the supported sign-in providers. Although I am not covering it here, there are also options to configure the UI and make the component mesh well with your app's theme.



When the user selects the **Sign in with Google button**, they will either begin to authenticate if they only have one Google account associated with their device, or they will be presented with an account selector dialog.



Once the user has selected an account, the FirebaseUI library will handle working with the rest of the Google sign-in provider to authenticate the user with your Firebase backend. When the process has completed, `onActivityResult` will be called and you can validate if the process was successful or not.

```

protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
}

```



```

if (requestCode == SIGN_IN_REQUEST_CODE) {
    if (resultCode == RESULT_OK) {
        Toast.makeText(this, "User signed in!", Toast.LENGTH_SHORT).show();
    } else {
        // Something didn't work out. Let the user know and wait for them to sign in again
    }
}
}

```

Now that your user has signed in, you should be able to see their account in the **Auth** section of the Firebase console under the **Users** tab.



The screenshot shows the Firebase console's 'Users' tab. At the top, there is a search bar with the placeholder text 'Search by exact email address (email@domain.com) or user UID' and an 'ADD USER' button. Below the search bar is a table with the following columns: 'Email', 'Providers', 'Created', 'Signed In', and 'User UID'. A single user is listed in the table with a Google provider icon, a creation date of 'Jul 31, 2016', a sign-in date of 'Jul 31, 2016', and a User UID of 'QtQ7TkUujgYRJ7JxqU8Uvdv7IFd2'. At the bottom of the table, it says 'Displays up to 500 users' and 'Rows per page: 50'.

Email	Providers	Created	Signed In	User UID
[Redacted]	Google	Jul 31, 2016	Jul 31, 2016	QtQ7TkUujgYRJ7JxqU8Uvdv7IFd2

## Getting User Profile Information

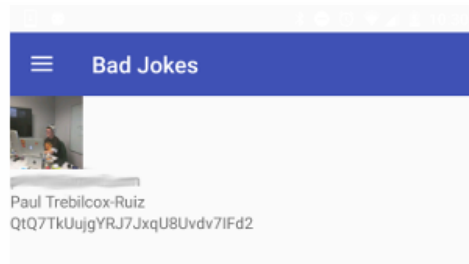
At this point you may be thinking, "OK, my user has authenticated with Firebase, so what?" While it may not seem like much, you can now access the user account to display information for your user and keep track of data based on their unique ID. This means if you have user specific information, such as submitted posts or chat entries, you have a way of adding it to Firebase and ensuring that only certain users have access to it. You can access this information by grabbing the current `FirebaseUser` object from `FirebaseAuth` and calling the appropriate methods for the data you need.

```

private void displayUser() {
    FirebaseAuth auth = FirebaseAuth.getInstance();
    if (auth.getCurrentUser() != null) {
        FirebaseUser user = auth.getCurrentUser();
        Glide.with(this).load(user.getPhotoUrl().toString()).into(mProfileImage);
        mProfileDisplayName.setText(user.getDisplayName());
        mProfileUid.setText(user.getUid());
        mProfileEmail.setText(user.getEmail());
    }
}

```

The above code will simply display the information obtained about the current user.



While this example is very simple, it does show you how to grab your user's information so that you can use it in other parts of the app, such as creating a unique node for that user in your Firebase Real-time Database JSON tree.

## Updating User Information

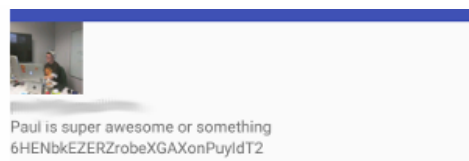
There will be times where users will want to update their information for their user account, such as a change in their name or to update their profile image. This can be done easily enough by creating a `UserProfileChangeRequest` object with the new information that your user would like to use. You can then call `updateProfile` on the `FirebaseUser` object and add an `OnCompleteListener` to act on the results.

```
private void updateUserName() {
    FirebaseUser user = FirebaseAuth.getInstance().getCurrentUser();

    UserProfileChangeRequest updates = new UserProfileChangeRequest.Builder()
        .setDisplayName("Paul is super awesome or something")
        .build();

    user.updateProfile(updates).addOnCompleteListener(this, new OnCompleteListener<Void>() {
        @Override
        public void onComplete(@NonNull Task<Void> task) {
            if( task.isSuccessful() ) {
                displayUser();
            }
        }
    });
}
```

This code will change the display name for the authenticated user and then show it on the screen.



## **Finishing Up**

While this tutorial has just touched the basics of authentication, you should now be able to implement enough to get your own apps up and running with user accounts. The authentication library in Firebase includes many more features that may be useful for your app, such as the ability to change/reset passwords, delete accounts, authentication against multiple providers for one account and anonymous login for users.

# Firebase Analytics, Crash Reporting and Ads

Many of the tools available in Firebase are built around understanding the usage of your applications, and helping you maximize revenue/use by understanding your users.

Continuing with our series on Firebase, this tutorial will focus on implementing analytics, crash reporting and advertisements in your Android apps.

## Analytics

When you implement analytics into your applications, you are adding the ability to understand how your users are interacting with your app, and allowing yourself to make informed decisions about future updates and marketing. Luckily, Firebase provides a great console tool for analyzing data, and the Firebase analytics library makes it easy to report events.

First you will need to include the Firebase core library in your project by importing it in your app's **build.gradle** file.

```
compile 'com.google.firebase:firebase-core:9.4.0'
```

Once you have imported the library, you can obtain a reference to the `FirebaseAnalytics` object and send events to Firebase using the `logEvent` method. This method accepts a `String`, representing the event type, and a `Bundle` of `String` objects for additional parameters/information regarding the analytics event. Here's an example of sending an event when an item in the `NavigationDrawer` is selected.

```
private void updateSelectedSection(int menuItemId) {  
    Bundle bundle = new Bundle();  
    switch( menuItemId ) {  
        case R.id.item_joke_list: {  
            getFragmentManager().beginTransaction().replace(R.id.fragment_container,  
JokeListFragment.getInstance()).commit();  
            bundle.putString(FirebaseAnalytics.Param.ITEM_NAME, "JokeListFragment");  
        }  
    }  
}
```

```

        break;
    }case R.id.item_random_jokes: {
        getFragmentManager().beginTransaction().replace(R.id.fragment_container,
RandomJokeFragment.getInstance()).commit();
        bundle.putString(FirebaseAnalytics.Param.ITEM_NAME, "RandomJokeFragment");
        break;
    }case R.id.item_submit_joke: {
        getFragmentManager().beginTransaction().replace(R.id.fragment_container,
SubmitJokeFragment.getInstance()).commit();
        bundle.putString(FirebaseAnalytics.Param.ITEM_NAME, "SubmitJokeFragment");
        break;
    }default: {
        return;
    }
}

bundle.putString(FirebaseAnalytics.Param.CONTENT_TYPE, "drawer_item");
FirebaseAnalytics.getInstance(this).logEvent(FirebaseAnalytics.Event.SELECT_CONTENT,
bundle);
}

```

As you can see, a `Bundle` is created and `String` values are placed into it with the keys of `FirebaseAnalytics.Param.ITEM_NAME` and `FirebaseAnalytics.Param.CONTENT_TYPE`. The Firebase analytics library provides dozens of pre-defined `String`s under `Param` and `Event`, and you should try and use the ones that match the closest to what you're trying to keep track of in your app. If nothing represents your app's event or parameters well enough, then you can pass in a custom `String` as a key.

When you go into the Firebase analytics console, you will be able to see information that is automatically collected about users, such as country, device type and operating system. There are also events that are automatically recorded, such as when the app is first opened and when it is uninstalled by a user.

Event name ↑	Count	↔	Value	↔	Users	↔	Mark as conversion
app_exception	1	-	-		1	-	<input type="checkbox"/>
app_remove	5	-	-		5	-	<input type="checkbox"/>
first_open	6	-	-		6	-	<input checked="" type="checkbox"/>
notification_foreground	2	-	-		1	-	<input type="checkbox"/>
notification_open	2	-	-		1	-	<input type="checkbox"/>
notification_receive	4	-	-		1	-	<input type="checkbox"/>
select_content	23	-	-		2	-	<input type="checkbox"/>
view_item	25	-	-		3	-	<input type="checkbox"/>

This console will update a few times a day, so analytics information will not be immediately available. For testing purposes, you can record when an event is fired by your app using the **adb** console tool with the following commands.

```
adb shell setprop log.tag.FA VERBOSE
adb shell setprop log.tag.FA-SVC VERBOSE
adb logcat -v time -s FA FA-SVC
```

Now if you look in your terminal or command prompt window, you should see the events as they occur.

```
(31314): Logging event (FE): select_content, Bundle[{item_name=SubmitJokeFragment, _o=app, content_type=drawer_item}]
```

## Crash Reporting

When you release an app into the wild, there's a good chance that crashes will happen. Using the standard Play Store crash reporting, you're only going to know about the crashes that are reported by users. However, when you implement Firebase, crashes

will be automatically reported to the Firebase console, allowing you to correct issues as soon as they creep up, allowing you to prevent poor ratings in the Play Store or uninstalls of your app. You can add Firebase crash reporting to your app by importing the Crash Reporting library into your project by including the following line into the dependencies node of your **build.gradle** file.

```
compile 'com.google.firebase:firebase-crash:9.4.0'
```

Once you have the library imported, fatal crashes will automatically be sent to Firebase with data about the crash and the user's device. One thing to note while testing is that this process may take up to twenty minutes to propagate through to the Firebase console.

While fatal crash reporting is important, not all errors will end up killing your app. Sometimes you'll run into exceptions that are caught by your program, and you'll want to know about those issues as well. In your **catch** statement, you can send reports by using the **FirebaseCrash.report** operation, which will log device information and the caught exception.

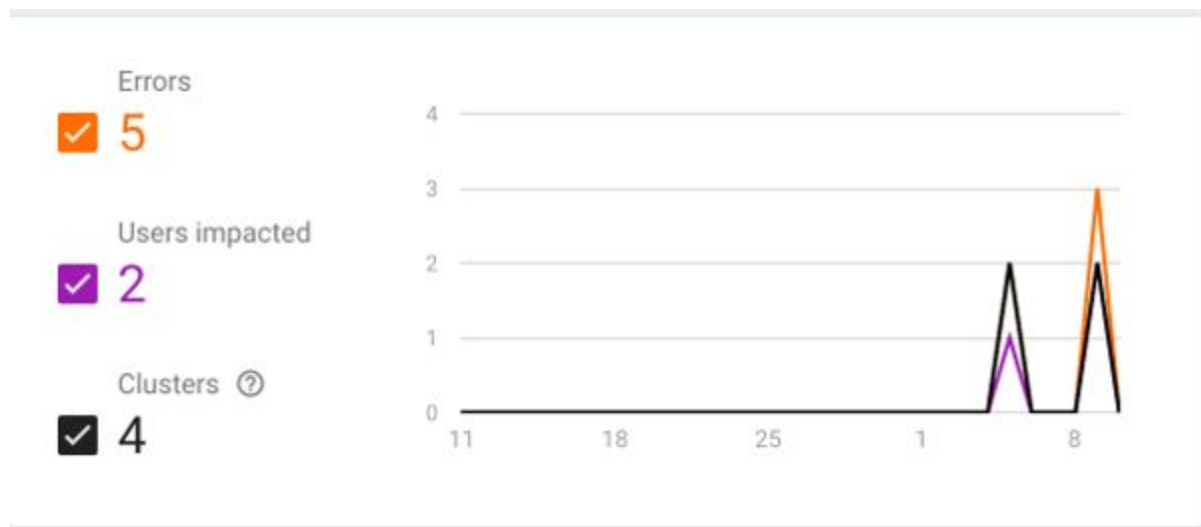
```
try {  
    throw new IOException("Bad URL example exception");  
} catch( IOException e ) {  
    FirebaseCrash.report(e);  
}
```

Another great tool for debugging is the simple log message. While you do not want to leave code in your app that logs messages to the Android console when you release, you can use Firebase to collect log messages that can be associated with crash reports by using the **FirebaseCrash.log** command, like so.

```
FirebaseCrash.log("MainActivity: some important log message");
```

## Crash Reporting Console

Now that you've setup your app to report crashes, it's time to take a look at the Firebase crash reporting dashboard. The first thing you'll see is a graph showing the number of crashes over a period of time, allowing you to easily notice when crash-inducing bugs may have been introduced or caused by a specific event.



Below that you'll receive information on **clusters**, or crashes that have been grouped together due to similarity. This cluster view will give you simple information, such as the number of users impacted, their app version and a summary of the crash stack trace.



2	2	1 (1.0)	java.lang.R... ActivityThread,	android.app.ActivityTh.. android.app.ActivityTh..
			<b>Fatal</b>	

 Upload a ProGuard mapping file to deobfuscate future stack traces for version

 **UPLOAD**

**Exception java.lang.RuntimeException: Unable to start activity ComponentInfo{com.anddevcon.e**

```

android.app.ActivityThread.performLaunchActivity (ActivityThread.java:2416)
android.app.ActivityThread.handleLaunchActivity (ActivityThread.java:2476)
android.app.ActivityThread.handleRelaunchActivity (ActivityThread.java:4077)
android.app.ActivityThread.-wrap15 (ActivityThread.java)
android.app.ActivityThread$H.handleMessage (ActivityThread.java:1350)
android.os.Handler.dispatchMessage (Handler.java:102)
android.os.Looper.loop (Looper.java:148)
android.app.ActivityThread.main (ActivityThread.java:5417)
java.lang.reflect.Method.invoke (Method.java)
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run (ZygoteInit.java:7
com.android.internal.os.ZygoteInit.main (ZygoteInit.java:616)





```

**CLOSE**

**VIEW DETAILS**

When you click on the blue **View Details** button, you'll be taken into a, you've guessed it, details screen. This screen will show you meta-data about app versions and devices experiencing the crash, as well as more specific information about the user's device and the full stack trace for the received error.

## Data

 User data	 Performance	 Device data	 Error data
Country Code: 310	<b>data</b>	Manufacturer: Motorola	Date: Aug 10, 2016, 5:24:03 PM
Carrier Code: 260	VM free: 7.58MB	Model: Nexus 6	App Version: 1
Carrier: T-Mobile	VM total: 38.92MB	Board: Shamu	
Locale: —	VM max: 256MB	Android API: 23	
	Battery level: —	Android OS: 6.0.1	
	Charging state: Charging USB	Brand: Google	
	Connection State: Wifi	RAM: 2.9GB	
		Available on disk: —	
		Orientation: Portrait	
		Proximity to user: —	

As you've seen, Firebase provides an easy to use and implement tool for keeping track of crashes in your application, and a great way to review those crashes and errors so that you can quickly get them fixed.

## Ads

In-app advertisements can provide a key source of revenue for an Android application. In order to help facilitate the improved use of ads, Google has integrated AdMob with Firebase, allowing developers to get analytical insight into ad views and interactions. In this section we will take a look at implementing simple ads into an Android app from the Firebase Ads library.

## Creating and Linking an AdMob App

Before you can start adding Firebase-enabled ads into your app, you'll need to create an AdMob account using the Google account that you use for your Firebase project. Once your AdMob account is created, you need to set up the [AdMob console](#) for your app. You can start by going to the **Monetize** tab at the top of the screen. If you haven't created an AdMob app before, you will be presented with a screen that will walk you through the process. If you have created one before, you can create a new app by pressing on the red **Monetize New App** button. For this example, select the **Add Your App Manually** tab and enter in the display name and platform for your app.

**1 Select an app**

App name ⓘ

Bad Jokes

Platform ⓘ

ANDROID ▾

Next you will need to set up at least one type of ad that you will display. This tutorial will only use a banner ad.

## 2 Select ad format and name ad unit

BANNER

INTERSTITIAL

REWARDED INTERSTITIAL

NATIVE

Ad type ?

☒ Text ?

☒ Image ?

Automatic refresh ?

☐ No refresh

☒ Refresh rate:  seconds (30-120 seconds)


Text ad style ?

STANDARD ▾

Increase your downloads with AdMob.  
Get ranked, get noticed, get users.

→


Ad unit name ?

 Ad type, size, and placement are specified when you integrate the code using the Google

SAVE

CANCEL

Once you have filled out all of the information for your ad and clicked on the blue **Save** button, you will be given the option of linking your new AdMob app to a Firebase project. When you start this process, you will be asked to enter in an app package name that *cannot be changed*, so be sure to enter it in correctly. Next you will be provided with a dialog that asks you to select a Firebase project to link to your AdMob app.

 **Bad Jokes**  
Android

We found this AdMob app in existing project(s) in your Firebase account. It's recommended to link your AdMob app to an existing Firebase project and existing Firebase app. [Learn more](#)

☒ Link to an existing Firebase project and existing Firebase app

ANDEVCON-BADJOKES ▾

>

App  
com.andecon.andecon\_badjokes

☐ Create a new Firebase project

Link your AdMob app(s) to Firebase and we'll share your data from the free Firebase Analytics tool with AdMob to improve app monetization and user engagement. You understand that this linkage may also allow AdMob Data to be shared with Firebase for enhanced reporting purposes.

CONTINUE

CANCEL

If everything works correctly, you should see a screen that says **Successfully linked** when you hit the blue **Continue** button. After that you will be shown your app and ad unit IDs, which you will need in your Android application.

## Adding Banner Ads to an Android App

Before you can place advertisements in your Android app, you will need to import the Firebase Ads library, which includes Google's ads SDK. You can do this by adding the following line into your app module's **build.gradle** file.

```
compile 'com.google.firebase:firebase-ads:9.4.0'
```

Once you have imported the library and synced your project, you can add the ID for your ad and app to your **strings.xml** file. The end result should look like the following, but with digits in place of the series of Xs.

```
<string name="banner_ad_main_activity">ca-app-pub-4219185563279047/XXXXXXXXXX</string>  
<string name="ad_app_pub_id">ca-app-pub-4219185563279047~XXXXXXXXXX</string>
```

With your IDs in place, it's time to add an `AdView` to your app. For this example I will simply add the item into the layout file for `MainActivity`. Since the `AdView` class is added as a standard XML View item, you can align it within your layout like you would with any other item.

```
<com.google.android.gms.ads.AdView
    android:id="@+id/ad_view"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    ads:adSize="BANNER"
    ads:adUnitId="@string/banner_ad_main_activity" />
```

Since `AdView` uses custom attributes, you will also need to include the appropriate namespace in the root element for the layout.

```
<android.support.v4.widget.DrawerLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:ads="http://schemas.android.com/apk/res-auto"
    ...
```

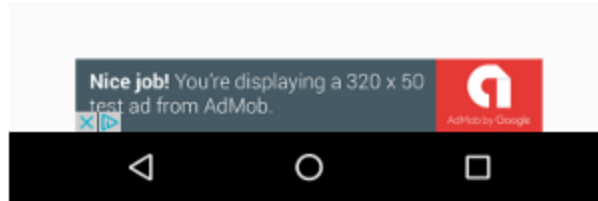
When you're finished with your layout file, you'll need to go into `MainActivity` and initialize the mobile ads SDK by passing in your ads app ID into the `MobileAds.initialize` method.

```
MobileAds.initialize(getApplicationContext(), getString(R.string.ad_app_pub_id));
```

Finally you can display a new ad by creating an `AdRequest` object and passing it into `AdView`'s `loadAd` method. When debugging, you will need to call `addTestDevice` on the builder in order to display a test ad, or the `AdView` will be invisible. You can find your device's test ID in the Android debug log the first time you display an ad in your app.

```
AdRequest adRequest = new AdRequest.Builder()
    .addTestDevice(getString(R.string.test_device_id))
    .build();

mAdView.loadAd(adRequest);
```



Now when ads are shown and interacted with by users, analytics information will be automatically gathered and viewable in the Firebase console.

## Finishing Up

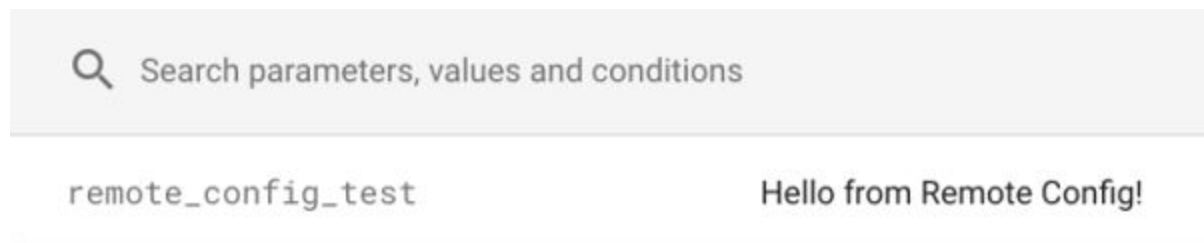
In this tutorial you've learned about how you can use Firebase to gather usage and crash information for your applications, as well as how to implement simple ads into your app with integrated Firebase analytics. In the next articles in this series, you will learn how you can put some of the information from analytics to use by engaging with your users.

# Customizing User Experience and Engagement Through Firebase

While getting users to install your app is no easy task, keeping users engaged can be even more of a challenge. In this tutorial we will explore a few of the tools available through Firebase for Android to not only keep users engaged and interested in your application, but to also bring in new users.

## Remote Config

One of the biggest hassles with distributing a mobile app is making small changes that require pushing an updated APK to the Google Play Store. Luckily, Firebase includes a feature called Remote Config, which acts as a cloud based key/value data store, that can be accessed from your apps. You can add values by selecting **Remote Config** from the side navigation panel in Firebase and clicking on the blue **ADD PARAMETER** button in the top right of the screen.



Given that this key/value store is online, you can change the value at any time to match the needs of your app. This can be useful for triggering special promotions, changing displayed text in an app, or configuring properties for games, such as level difficulty or rewards. Since Remote Config is tied in with Google Analytics, you can also specify groups based on location, device language, OS version, or any other available information so that only select users receive specific values for your keys. Another



handy feature is that a certain percentage of random users can be placed into a group for testing purposes.

**Define a new condition**

Use conditions to provide different parameter values if a condition is met

Name:  Color: ▼

Applies if...

User in random percentile ▼	<= ▼	10.75 %
OS type ▼	Android ▼	
Device region/country ▼	United States ▼	AND

CANCEL CREATE CONDITION

Once you've set up some values for Remote Config, it's time to start using them in an Android app. First you will need to import the required dependency into your **build.gradle** file.

```
compile 'com.google.firebase:firebase-config:9.4.0'
```

Once you have synced your project, it'll be time to dive into your Java code. When testing, you will want to initialize a `FirebaseRemoteConfigSettings` object and enable developer mode, which will allow you to grab data changes immediately rather than having to plan for the standard wait time between allowed pollings. Once the object is initialized, you can associate it with your `FirebaseRemoteConfig` instance.

```
FirebaseRemoteConfigSettings configSettings = new FirebaseRemoteConfigSettings.Builder()
    .setDeveloperModeEnabled(BuildConfig.DEBUG)
    .build();
FirebaseRemoteConfig.getInstance().setConfigSettings(configSettings);
```

The next thing you will want to link to your `FirebaseRemoteConfig` is a set of default values. You can do this by first creating a new XML file that contains map items of keys and values matching your Firebase Remote Config entries.

```
<?xml version="1.0" encoding="utf-8"?>
<defaultsMap>
    <entry>
        <key>remote_config_test</key>
        <value>Some default data here</value>
    </entry>
</defaultsMap>
```

Then apply the new defaults map to your `FirebaseRemoteConfig` instance.

```
FirebaseRemoteConfig.getInstance().setDefaults(R.xml.remote_config_default_values);
```

The final thing you will need to do is call `fetch()` to retrieve the values from Firebase and populate them into your app. When this is done, you can use the new values however you want to customize your user's experience. In addition, you can also include an `OnCompleteListener` to keep track of when the values are up to date and perform an action based on the new data.

```
FirebaseRemoteConfig.getInstance().fetch().addOnCompleteListener(this, new
OnCompleteListener<Void>() {

    @Override

    public void onComplete(@NonNull Task<Void> task) {

        if( task.isSuccessful() ) {

            FirebaseRemoteConfig.getInstance().activateFetched();

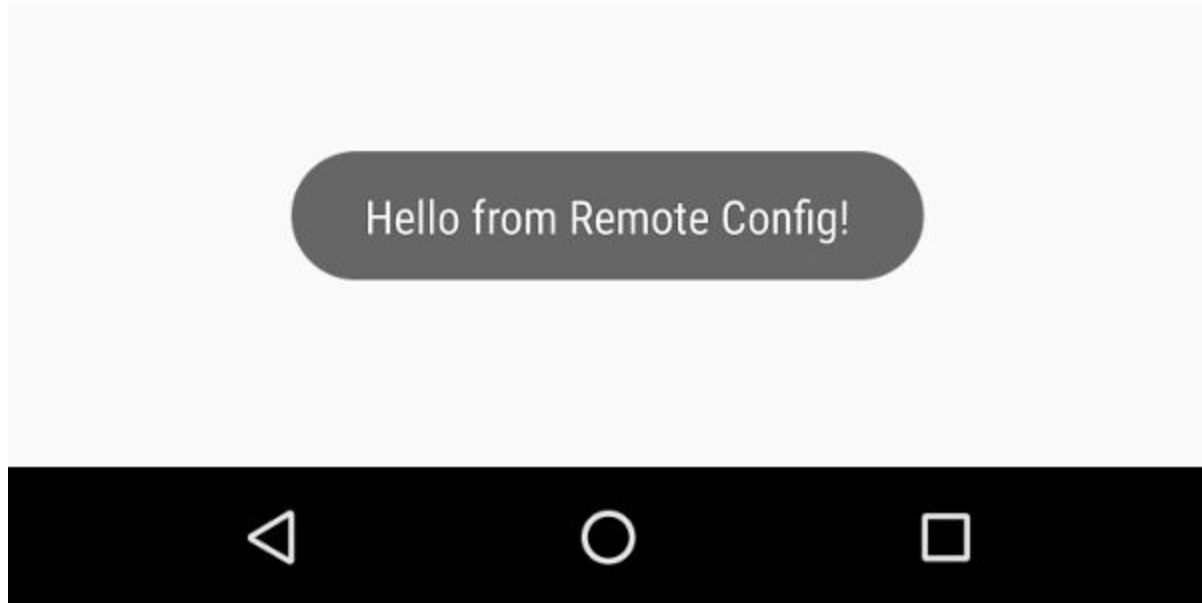
        }

        Toast.makeText(getApplicationContext(),
FirebaseRemoteConfig.getInstance().getString("remote_config_test"),
Toast.LENGTH_SHORT).show();

    }

}
```

```
});
```

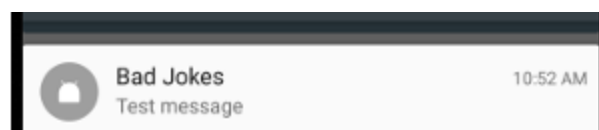


## Cloud Messaging and Notifications

Notifications are one of the best ways to re-engage with your users and get them back into your app, or let them know when new features are available for them to try. Using Firebase, you can easily send notifications to all or segments of your users using the Firebase Notification Console. To get notifications to your users, include the Messaging library in your **build.gradle** file.

```
compile 'com.google.firebase:firebase-messaging:9.4.0'
```

Firebase will handle receiving messages and displaying notifications to your app when it's not open automatically.



It's important to note that this will only display notifications when your app is not opened by your users. If you want to display notifications for your users while they are using

your app, you can create a new class that extends `FirebaseMessagingService` and handle the received message there.

```
public class NotificationService extends FirebaseMessagingService {

    @Override

    public void onMessageReceived(RemoteMessage remoteMessage) {

        super.onMessageReceived(remoteMessage);

        NotificationCompat.Builder builder = new NotificationCompat.Builder(this);

        builder.setContentTitle("Hello from Firebase!");

        builder.setSmallIcon(R.mipmap.ic_launcher);

        builder.setContentText(remoteMessage.getNotification().getBody());

        NotificationManagerCompat.from(this).notify(0, builder.build());

    }

}
```

You will also need to register this new `Service` in **AndroidManifest.xml**.

```
<service

    android:name=".services.NotificationService">

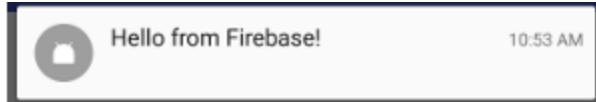
    <intent-filter>

        <action android:name="com.google.firebase.MESSAGING_EVENT"/>

    </intent-filter>

</service>
```

Now when your user receives a message from Firebase while they have your app open, they can receive a notification with whatever information you want to display.



With your app able to display notifications, it's time to look into the Firebase Notification Console. When you go to send a new message, you'll be presented with a form to enter a message data, as well as who you want to receive the notification based on Firebase Analytics groups. Once you have filled out the form, you can click on the blue **Send Message** button and the notification will be on its way.

Message text

Test Message

Message label (optional) ?

Enter message nickname

Delivery date ?

Send Now

Target

☒ User segment ☐ Topic ☐ Single device

Target user if...

App



com.andevcon.andevcon...

AND

## App Invites

When a user finds an app that they really like, it's likely that they'll want to share it with their friends. Normally they would just have to tell their friends the name of the app, or try and find the link on the Play Store to share, though this can be cumbersome if the friend cannot find the app, or they use a non-Android device. Using Firebase, you can include a share option in your app that will make it easy to send an email or SMS message to a friend containing a link for installing your app.

As with everything else in Firebase, you will need to include the proper dependency for App Invites in your **build.gradle** file.

```
compile 'com.google.firebase:firebase-invites:9.4.0'
```

When your project has synced, you will be able to create a new Intent using the `AppInviteInvitation.IntentBuilder` class, which will launch a screen that allows users to select contacts to invite to install the app. This builder provides various options for customizing the app invite screen.

- `setMessage`: This will set the message that users see and can send to contacts over text message or email. It should be noted that this cannot be more than 100 characters long.
- `setCustomImage`: Using this method, you can provide a `URI` to a custom image that will appear in the invite screen and invite email.
- `setCallToActionText`: Sets the text for the install button in emails, which cannot be more than 32 characters long.
- `setDeepLink`: Allows you to provide metadata for your invite, which can be received on install for taking specific actions for your newly invited user.
- `setEmailHtmlContent`: Allows you to override `setMessage`, `setCustomImage` and `setCallToActionText` to create a custom HTML formatted email to send to potential new users.

- `setEmailSubject`: Required if `setEmailHtmlContent` is used. As the name implies, this will set the subject for your custom email.
- `setOtherPlatformsTargetApplication`: One of the more interesting options, this method will allow you to associate the Firebase client app ID for an iOS version of your app, allowing iOS users to install the proper version if it's shared by an Android user.

Once your Intent has been created, you can simply call `startActivityForResult` in order to show the invite screen and get the result of the user's actions.

```
Intent intent = new AppInviteInvitation.IntentBuilder(getString(R.string.invitation_title))

    .setMessage(getString(R.string.invitation_message))

    .setCustomImage(Uri.parse(getString(R.string.invitation_custom_image)))

    .setCallToActionText(getString(R.string.invitation_cta))

    .setDeepLink(Uri.parse("deep link URL here"))

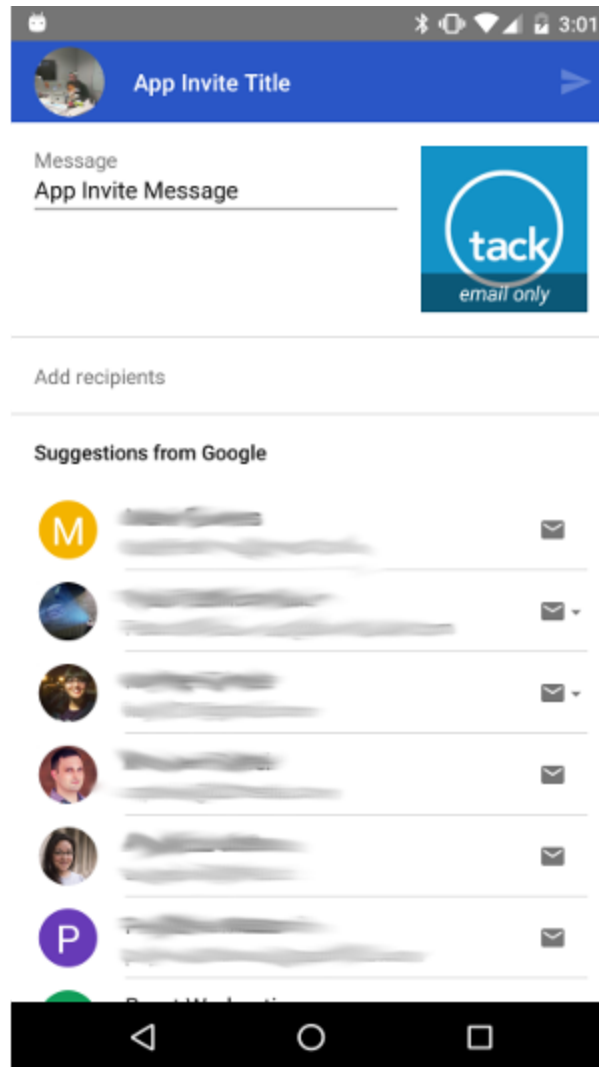
    .setOtherPlatformsTargetApplication(

        AppInviteInvitation.IntentBuilder.PlatformMode.PROJECT_PLATFORM_IOS,

        getString(R.string.ios_app_client_id))

    .build();

startActivityForResult(intent, 42);
```



As mentioned above, your invites can contain some meta-data through the use of the `setDeepLink` method in the App Invite `IntentBuilder` object. When an Android user installs your app from an invite, you can use Google Play Services to detect and retrieve that meta-data, allowing you to provide a custom experience for your new user.

```
boolean autodeeplink = true;

mGoogleApiClient = new GoogleApiClient.Builder(this)
    .addApi(AppInvite.API)
    .enableAutoManage(this, this)
```



```

        .build();

AppInvite.AppInviteApi.getInvitation(mGoogleApiClient, this, autodeeplink)

        .setResultCallback(

            new ResultCallback<AppInviteInvitationResult>() {

                @Override

                public void onResult(AppInviteInvitationResult result) {

                    if (result.getStatus().isSuccess()) {

                        //Get intent information

                        Intent intent = result.getInvitationIntent();

                        String deepLink = AppInviteReferral.getDeepLink(intent);

                        String invitationId = AppInviteReferral.getInvitationId(intent);

                        //if autodeeplink is false, can handle actions manually here

                    }

                }

            }

        );

```

In the above example, you'll notice a boolean named `autodeeplink`, which is set to true. When a value of true is passed into the `AppInvite.AppInviteApi.getInvitation` method, Android will handle the intent and meta-data as defined by your **AndroidManifest.xml** file. If this value is set to false, then you can manually extract the information and handle it however your app requires.

## Conclusion

In this article you've been introduced to some of the tools available in Firebase to customize the user experience of your app, as well as how to use notifications to reengage with users. In the next, and final, article of this series, you will learn about file storage options with Firebase, and be introduced to Firebase's cloud test lab.

# Firebase Storage and Additional Tools

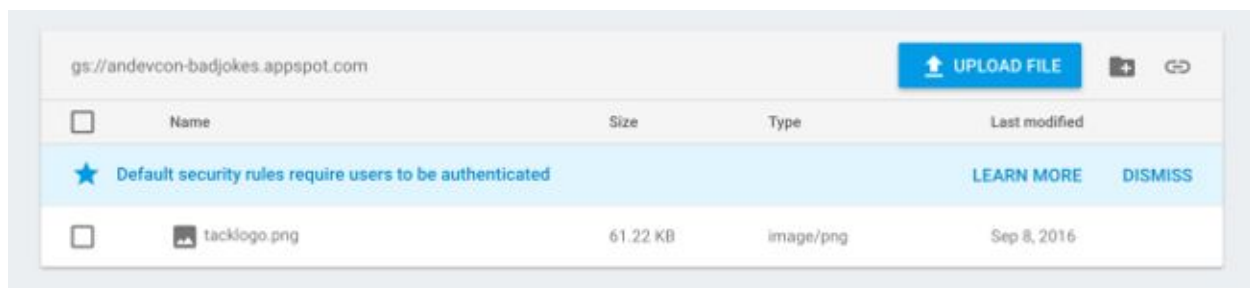
Throughout this series we've covered a lot of the available tools in Firebase for Android developers. In this article we'll finish up by going over Firebase Storage, as well as some of the non code based features of Firebase, such as web hosting and device test lab.

## Firebase Storage

As with the Firebase Real Time Database, you have a set of authentication rules that must be met before an app can access or modify data in Firebase Storage. While the default requires that a user be authenticated, for this example we will simply open the **Rules** tab of the Firebase Storage section in the Firebase console and set the authentication rules to allow any unauthenticated user to access storage.

```
1  service firebase.storage {  
2      match /b/andevcon-badjokes.appspot.com/o {  
3          match /{allPaths=**} {  
4              allow read, write: if request.auth == null;  
5          }  
6      }  
7  }  
8  |
```

While you're in the Firebase console, you can also go into the **Files** tab to manually upload files for use with your apps by clicking on the blue **UPLOAD FILE** button in the top right.



## Downloading a File in Android

Now that you know how to manually upload files and have set up the Firebase authentication rules for accessing files from your app, it's time to dive into some Android code. We'll start by going over how to download files from Firebase Storage. The first thing you will need to do is create a **StorageReference** to your project's URL and the file name.

```
FirebaseStorage storage = FirebaseStorage.getInstance();
StorageReference storageRef =
storage.getReferenceFromUrl("gs://andevcon-badjokes.appspot.com").child("andevcon.jpg");
```

Next you simply need to create a **File** object and use that to call **getFile** on your **StorageReference**. This will create a **FileDownloadTask** that you can assign an **OnSuccessListener** and an **OnFailureListener** to in order to know when a file has finished asynchronously downloading. For this example we will simply download a jpg image and load it into an **ImageView**.

```
try {
    final File localFile = File.createTempFile("images", "jpg");

    storageRef.getFile(localFile).addOnSuccessListener(new
    OnSuccessListener<FileDownloadTask.TaskSnapshot>() {
        @Override
        public void onSuccess(FileDownloadTask.TaskSnapshot taskSnapshot) {
            Log.e("Test", "success!");
            Bitmap bitmap = BitmapFactory.decodeFile(localFile.getAbsolutePath());
            mImageView.setImageBitmap(bitmap);
        }
    }).addOnFailureListener(new OnFailureListener() {
        @Override
        public void onFailure(@NonNull Exception exception) {
            Log.e("Test", "fail :( " + exception.getMessage());
        }
    });
} catch (IOException e ) {}
```

If you run the above code, you should see the image from Firebase appear in your **ImageView**.



## Getting a File's URL

Sometimes you won't want to download the file to the device, but will simply want the URL. Doing this is equally straight forward, as you will create another `StorageReference` to the file that you want the URL for, and then will call `getDownloadUrl` on that reference. You can also use `OnSuccessListener` and `OnFailureListener` for this task.

```
FirestoreStorage storage = FirestoreStorage.getInstance();
StorageReference storageRef =
storage.getReferenceFromUrl("gs://andevcon-badjokes.appspot.com").child("andevcon.jpg");

storageRef.getDownloadUrl().addOnSuccessListener(new OnSuccessListener<Uri>() {
    @Override
    public void onSuccess(Uri uri) {
        Log.e("AnDevCon", "uri: " + uri.toString());
    }
}).addOnFailureListener(new OnFailureListener() {
    @Override
```

```

        public void onFailure(@NonNull Exception exception) {

        }
    });

```

The above code will output the path to the file, including the token used to retrieve it.

```

E/AnDevCon: uri:
https://firebasestorage.googleapis.com/v0/b/andevcon-badjokes.appspot.com/o/andevcon.jpg?alt=media&token=09f59ce8-620b-4b88-8d69-d4576cd55fe6

```

One interesting thing to note is that if you go to the URL, but cut off everything after **andevcon.jpg**, you will see the metadata for the file in JSON format.

```

{
  name: "andevcon.jpg",
  bucket: "andevcon-badjokes.appspot.com",
  generation: "1476483156071000",
  metageneration: "1",
  contentType: "image/jpeg",
  timeCreated: "2016-10-14T22:12:36.053Z",
  updated: "2016-10-14T22:12:36.053Z",
  storageClass: "STANDARD",
  size: "22982",
  md5Hash: "bh0pPRnw3B6mkSURcWBOGQ==",
  contentEncoding: "identity",
  crc32c: "E8JRGw==",
  etag: "CNiko7Co288CEAE=",
  downloadTokens: "09f59ce8-620b-4b88-8d69-d4576cd55fe6"
}

```

## Uploading a Byte Array

When it comes to uploading data to Firebase Storage, you have three options: a **byte array**, an **InputStream**, and a standard **File**. All three will require creating a **StorageReference** exactly like you did before, but this time the file name will be the name of the file that as you want it stored in Firebase.

```

FirebaseStorage storage = FirebaseStorage.getInstance();
StorageReference storageRef =
storage.getReferenceFromUrl("gs://andevcon-badjokes.appspot.com").child("ic_launcher.jpg");

```

For this example, I will create a **byte array** from the image stored in memory via an **ImageView**, though your **byte array** of data can come from any source.

```

mImageView.setDrawingCacheEnabled(true);
mImageView.measure(View.MeasureSpec.makeMeasureSpec(0, View.MeasureSpec.UNSPECIFIED),

```

```

View.MeasureSpec.makeMeasureSpec(0, View.MeasureSpec.UNSPECIFIED));
mImageView.layout(0, 0, mImageView.getMeasuredWidth(), mImageView.getMeasuredHeight());
mImageView.buildDrawingCache();
Bitmap bitmap = Bitmap.createBitmap(mImageView.getDrawingCache());

ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
bitmap.compress(Bitmap.CompressFormat.JPEG, 100, outputStream);
byte[] data = outputStream.toByteArray();

```

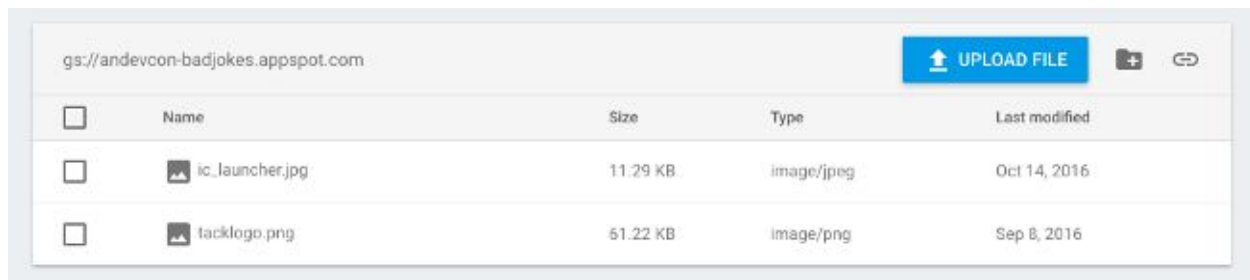
Once you have your **array**, you can call **putBytes** on your **StorageReference** and add your status listeners to the **UploadTask** that is created.

```

UploadTask uploadTask = storageRef.putBytes(data);
uploadTask.addOnFailureListener(new OnFailureListener() {
    @Override
    public void onFailure(@NonNull Exception exception) {
        Log.e("Test", "failed: " + exception.getMessage());
    }
}).addOnSuccessListener(new OnSuccessListener<UploadTask.TaskSnapshot>() {
    @Override
    public void onSuccess(UploadTask.TaskSnapshot taskSnapshot) {
        Log.e("Test", "successfully uploaded file");
    }
});

```

If the above code is successful, you should see your file available in the Firebase console.



## Uploading from an InputStream or File

Uploading content from a **File** object or an **InputStream** is almost identical, though you will use the **putStream** or **putFile** methods, respectively. Uploading an **InputStream** can be done like so.

```

FirebaseStorage storage = FirebaseStorage.getInstance();
StorageReference storageRef =
storage.getReferenceFromUrl("gs://andevcon-badjokes.appspot.com").child("test.txt");

InputStream stream = getResources().openRawResource(R.raw.test);

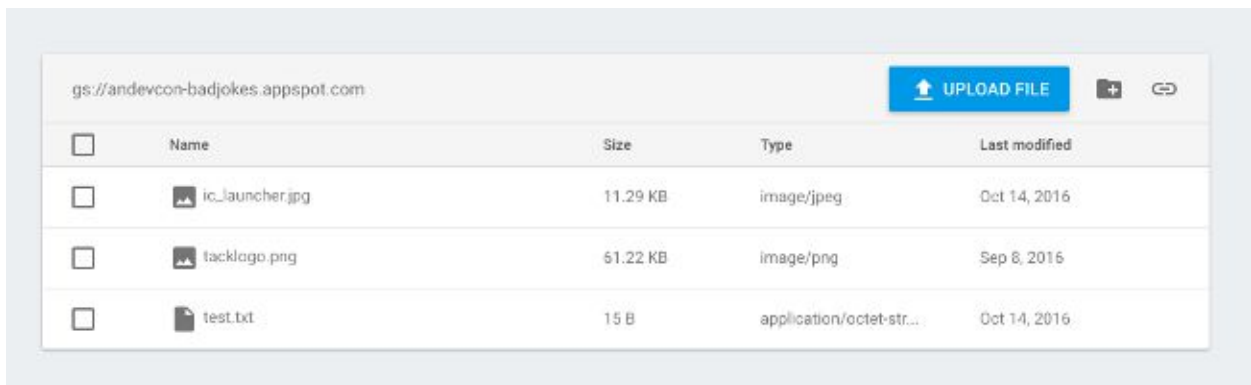
```

```

UploadTask uploadTask = storageRef.putStream(stream);

uploadTask.addOnFailureListener(new OnFailureListener() {
    @Override
    public void onFailure(@NonNull Exception exception) {
        Log.e("Test", "failed: " + exception.getMessage());
    }
}).addOnSuccessListener(new OnSuccessListener<UploadTask.TaskSnapshot>() {
    @Override
    public void onSuccess(UploadTask.TaskSnapshot taskSnapshot) {
        Log.e("Test", "successfully uploaded file");
    }
});

```



And uploading a file can be done with the following code.

```

FirebaseStorage storage = FirebaseStorage.getInstance();
StorageReference storageRef =
storage.getReferenceFromUrl("gs://andevcon-badjokes.appspot.com").child("test2.txt");

File file = null;
try {
    file = File.createTempFile("test2", "txt");
} catch( IOException e ) {

}

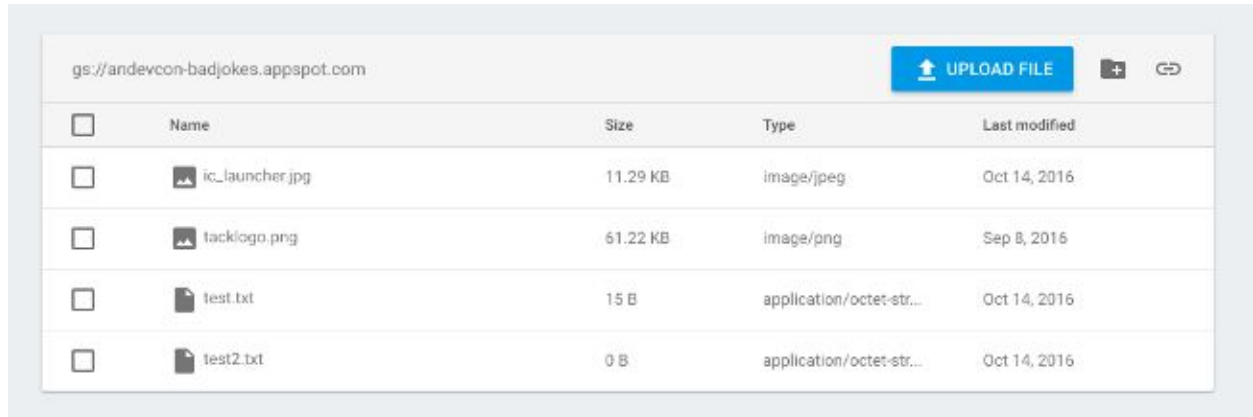
UploadTask uploadTask = storageRef.putFile(Uri.fromFile(file));

uploadTask.addOnFailureListener(new OnFailureListener() {
    @Override
    public void onFailure(@NonNull Exception exception) {
        Log.e("Test", "failed: " + exception.getMessage());
    }
}).addOnSuccessListener(new OnSuccessListener<UploadTask.TaskSnapshot>() {
    @Override
    public void onSuccess(UploadTask.TaskSnapshot taskSnapshot) {
        Log.e("Test", "successfully uploaded file");
    }
});





```



```
}  
});
```



The screenshot shows the Firebase Storage console interface. At the top, the URL is 'gs://andevcon-badjokes.appspot.com'. There is an 'UPLOAD FILE' button and icons for adding and linking. Below is a table with columns: Name, Size, Type, and Last modified. The table contains five entries: 'ic\_launcher.jpg' (11.29 KB, image/jpeg, Oct 14, 2016), 'tacklogo.png' (61.22 KB, image/png, Sep 8, 2016), 'test.txt' (15 B, application/octet-str..., Oct 14, 2016), and 'test2.txt' (0 B, application/octet-str..., Oct 14, 2016). Each entry has a checkbox on the left.

<input type="checkbox"/>	Name	Size	Type	Last modified
<input type="checkbox"/>	 ic_launcher.jpg	11.29 KB	image/jpeg	Oct 14, 2016
<input type="checkbox"/>	 tacklogo.png	61.22 KB	image/png	Sep 8, 2016
<input type="checkbox"/>	 test.txt	15 B	application/octet-str...	Oct 14, 2016
<input type="checkbox"/>	 test2.txt	0 B	application/octet-str...	Oct 14, 2016

As you can see, working with Firebase Storage is incredibly easy, and provides a great tool for storing assets for your app online without a lot of code overhead.

## Web Hosting and Test Lab

While both the web hosting and test lab features of Firebase are incredibly useful, I won't go too heavily into how they work, as they don't require any Android code on your part, however they are both important to know about as they can be great tools for you and your projects.

Firebase Hosting, as the name implies, is a simple to use web hosting service that allows you to upload a webpage that can then be served to users.

Test Lab, which is only available for Android app testing, is a paid service that allows developers to upload APKs to Firebase that are then tested against on either real devices, or more cheaply, virtual devices. Tests then produce logs, crash analysis reports, and other useful information that you can use to improve your apps before deploying to your users.

## Conclusion

Whew! In this article we've wrapped up our series on Firebase for Android, and walked through how to use Firebase Storage in your apps. You've also been, very briefly, introduced to Firebase Hosting and Test Lab, which rounds off the tools available in Firebase for your apps. I hope you've enjoyed learning about this awesome backend tool and find it useful in either your current or future projects.