# Using the Firebase Custom Authentication System to Authenticate with a Phone Number

By **Ali M. Bdeir**

Documentation **V3.0**

**29/March/2017**

**To report an issue or for help, email me at ali@bdeir.org**

## Revision History

| Date of Change | Description of change | Changed/Noticed by |
|---|---|---|
| March 5, 2017 | Initial Documentation | Ali Bdeir |
| March 18, 2017 | Fixed a bug in a url. Changed *“yourappname.herokuapp.com/phoneNumber/{number}”* to what it should be: “*yourappname.herokuapp.com/**token**/{number}”* | Dana Aliahmad |
| March 29, 2017 | Noticed that you need to call thread.start(); to actually start the thread. Fixed. | Dana Aliahmad |
| March 29, 2017 | Noted that Intelli J doesn’t have Gradle included by default, (*might differ if you downloaded some different file for it*) and that I forgot to add the OKHttp dependencies. | Sohaib El Jundi |

# Let’s get started

**What you’ll need:**

- IntelliJ IDE (Or any other IDE like Eclipse used with Gradle. I'll be using IntelliJ.) Download the **Community Version** which is free.
- The Gradle plugin installed in IntelliJ.
- The Heroku CLI downloaded. (OS X or Windows)
- Android Studio
- Git

---

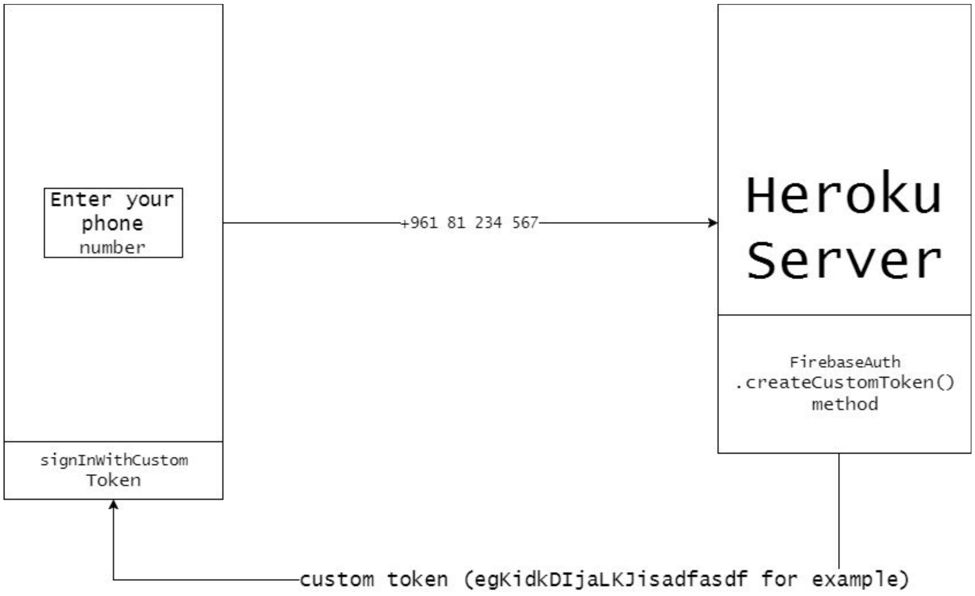
**What we'll achieve:**

1. We will create a Heroku Server
2. We will handle requests to retrieve a custom token from the server to use with Firebase to authenticate the user with a custom parameter.

See the next page to get started.

**Firebase Custom Authentication:**

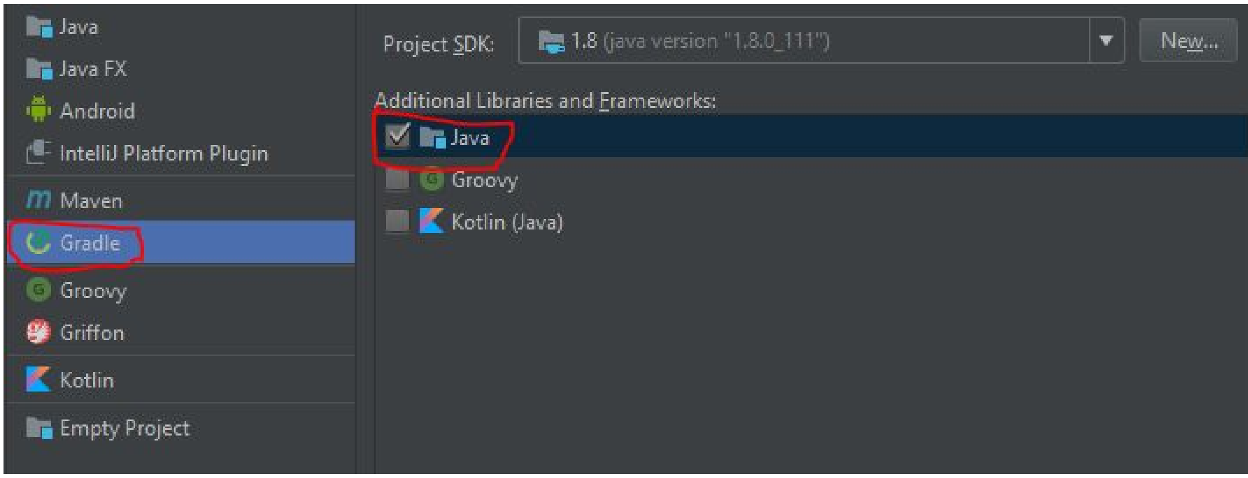
We'll be creating a Java web application using Heroku to generate a custom token that lets our user sign in. Here's the logic:


Enter your phone number
signInWithCustom Token
+961 81 234 567
Heroku Server
FirebaseAuth .createCustomToken() method
custom token (egKidkDIjaLKJisadfasdf for example)

We need to create a Java web app to use with Android. To do that:

1. Open IntelliJ (Not Android Studio), and create a new Gradle project:

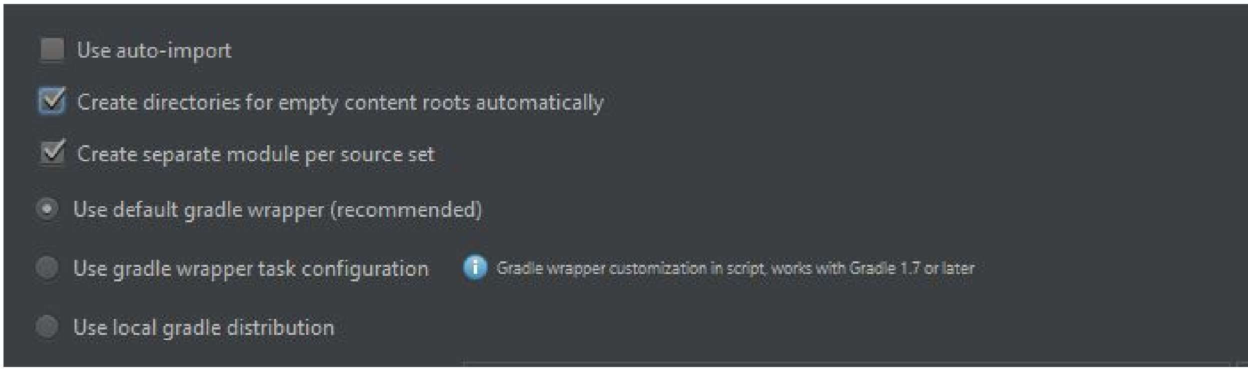
    1. If you have a project open, go to File>Close.
    2. Click "Create New Project"
    3. On the left, select "Gradle", then on the right select "Java".

    Project SDK: 1.8 (java version "1.8.0_111")

    4. Select your Java SDK if it isn't set in the drop down menu.

5. Click next.
6. You can leave your groupID empty. Your ArtifactID is your application name. I named it “CustomAuthTut”. Leave your version as is.
7. On the next screen, check everything I checked:



□ Use auto-import

☑ Create directories for empty content roots automatically

☑ Create separate module per source set

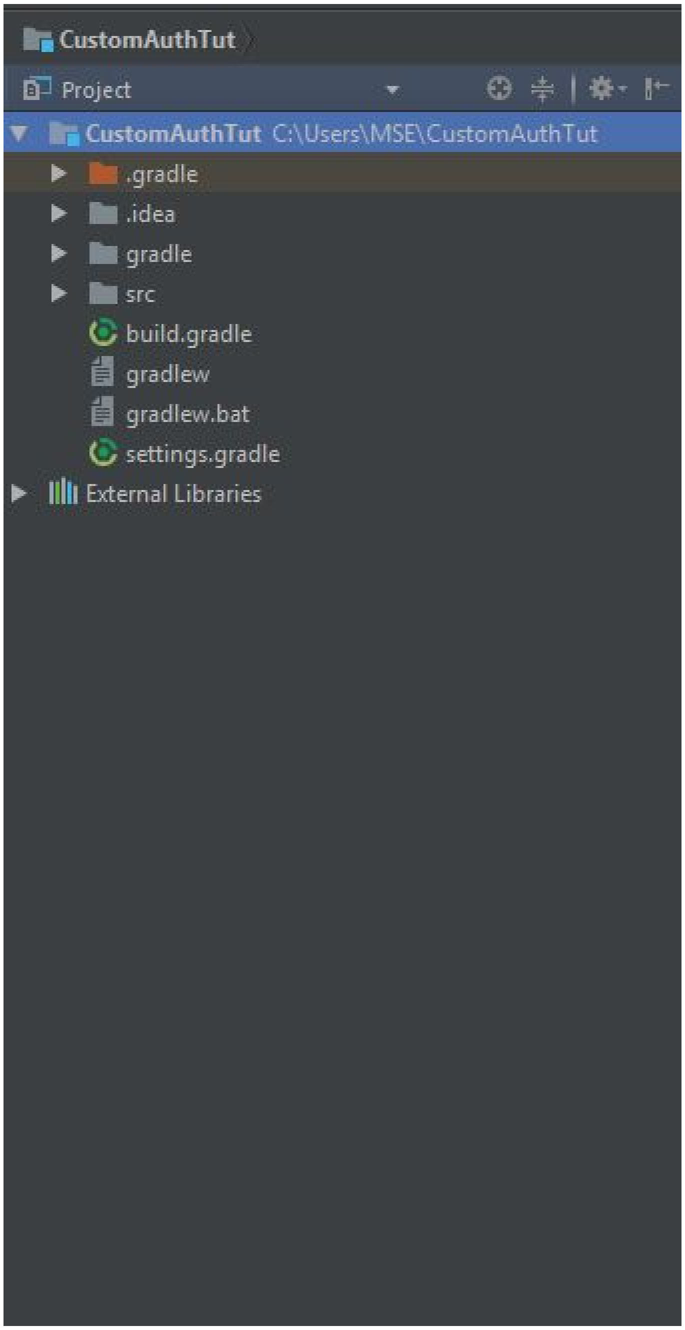
● Use default gradle wrapper (recommended)

○ Use gradle wrapper task configuration — Gradle wrapper customization in script, works with Gradle 1.7 or later

○ Use local gradle distribution

8. Click next, then Finish.
9. Wait until Gradle finishes it’s first build. It could take long depending on the speed of your PC. The first build is always the longest.
10. You’re done creating a new project.

On your left, you’ll have a project explorer.



The place where your Java classes will be placed in is `src/main/java`.

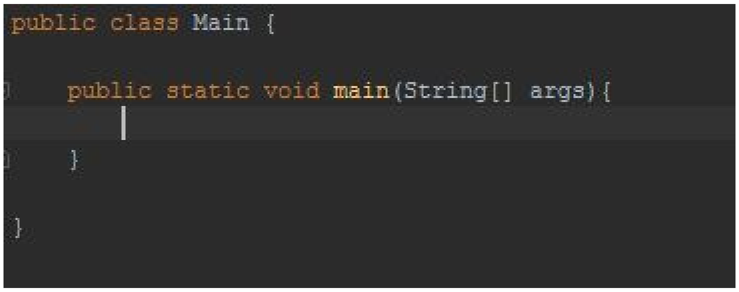
Create a new Java class in that directory by right-clicking on the java directory and selecting New>Java Class.
Name that class `Main`.

**Start programming.**

The `Main` class you just created requires a

```
public static void main(String[] args)
```

Method.



```
public class Main {

    public static void main(String[] args){

    }

}
```

Before you continue, I'd like to clarify something. The build.gradle file is a file for gradle which has some configurations. If you previously used maven, it's basically the `pom.xml`.

That's great. Now we need to set the main method in our `build.gradle:`

```
jar {
    manifest {
        attributes(
                'Class-Path': configurations.compile.collect { it.getName() }.join(' '),
                'Main-Class': 'Main'
        )
    }
}
```

Add this to the root of the code at the **very end of the file**. This tells our project what's the class it should run when starting the project.

### Setting up Heroku

The next step is to set up Heroku and run the project you just created over there.

In IntelliJ, go to View>Tool Windows>**Terminal.**

At the bottom of your screen, you should have a terminal now open. We need to initialize our git repository to use it with Heroku. Do:

```
$ git init
```

You should get a confirmation message.

The next thing you need to do, is create a new Heroku project from the terminal. Use:

```
$ heroku create yourappname
```

You don't have to type an app name, you can just leave it empty and it will generate one for you. Only lowercase letters are allowed. I named mine "`customauthtut`".

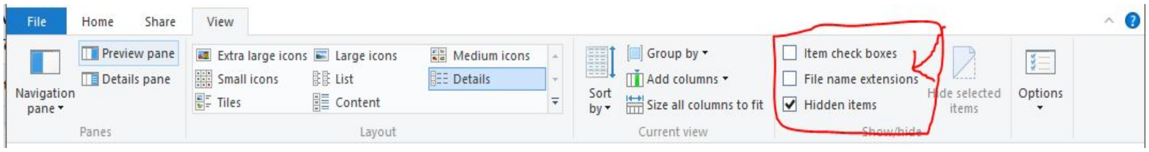
For now, let's just add a simple method to our `main` method:

```
public static void main(String[] args){
   System.out.println("Hello, Heroku!");
}
```

And now, **let's deploy the app to Heroku**:

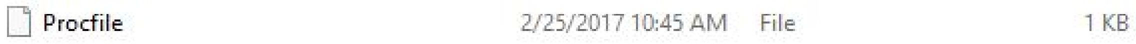
1) In order for Heroku to know what to run, we need a **procfile** file. To create a procfile:
    a) Open your file explorer, go to the root directory of your project.
    b) You'll need to create a file **without an extension. For Windows:**
        i) Right-click on an empty space in the file explorer you opened, go to New>Text Document. Name it ***Procfile*** (case sensitive).

ii) Still in File explorer, go to View (on the top of the window) and check the “File Name Extensions” checkbox.



☐ Item check boxes
☐ File name extensions
☑ Hidden items

iii) Now, go back to the Procfile.txt, right click it, then click “Rename”. Leave the name as is, but remove the .txt. It should look like this:



Procfile 2/25/2017 10:45 AM File 1 KB

c) Now that the procfile is created, open the file you just created in Notepad, Notepad++ , or any text editing software.

d) Go back to IntelliJ for now, and add this to the root of your build.gradle file right above `dependencies {...}` :

```
task stage(dependsOn: ['build', 'clean'])
build.mustRunAfter clean
```

e) Now, go back to the terminal in IntelliJ, and type in:

```
$ gradlew stage
```

f) After the build is done, you should now have a new `build/libs/yourAppName-1.0-SNAPSHOT.jar`

g) Copy the name of the jar file that was just generated

h) Go back to the text editor where you were editing the empty Procfile file. Put this in:

```
web: java -jar build/libs/{The File Name You Just
Copied}.jar
```

Don’t forget to replace `{The File Name You Just Copied}` with the file name you copied earlier.

i) Your Procfile is created.

2) Now that we have a procfile, do these commands in the terminal:

```
$ gradlew stage
$ git add .
$ git commit -m “Added Procfile”
$ heroku local web
```

3) After the last command (which is `heroku local web`, which runs the application locally) the terminal should give you this message:

```
4:37:52 PM web.1 |  Hello, Heroku!
[DONE] Killing all processes with signal  null
4:37:52 PM web.1 Exited Successfully
```

4) Our application is running locally, but it's only **printing** `Hello, Heroku!`, it's not really returning it. We need it to return any message for now, and later a custom token

5) To do that, we can use the Spark Framework. To add Spark to your project, we need to do some changes.

   a) To add Spark to our project, we'll need to add it as a **library.** However, the command to generate our jar file, `gradlew stage`, doesn't include any libraries in the jar. Therefore, we cannot use that command.

      i) What we need to do is use shadowJar. shadowJar creates a jar that INCLUDES libraries.

      ii) What you need to do is go to your build.gradle, and right below the `sourceCompatibility` variable, add this:

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.github.jengelman.gradle.plugins:shadow:1.2.4'
    }
}
```

And at the **very end** of the build.gradle, add this:

```
apply plugin: 'com.github.johnrengelman.shadow'
```

      iii) Now, go to View>Tool Windows>Gradle. Click the refresh button 🔄.

      iv) Now go to your terminal. Type in

```
gradlew shadowJar
```

v) If that build succeeds, that means that shadowJar has been successfully installed. You should see a new jar file in build/libs/ called `yourAppName-1.0-SNAPSHOT-`**`all`**`.jar.` Notice how there's that "`all`" there?

b) Now that we have shadowJar installed, we can install Spark. To do that, go to your build.gradle, and go to dependencies *(**NOT the one in buildscript{...}** but the one in the root of the file)* and add this line:

```
compile "com.sparkjava:spark-core:2.5.5"
```

c) Now, go to View>Tool Windows>Gradle. Click the refresh button ⟳ again.

d) Spark is installed.

e) Now, we need to change the procfile and change the default build method used by Heroku. Go to your procfile again, and just add "-all" to your jar's file name.

**Before:**

```
web: java -jar build/libs/yourAppName-1.0-SNAPSHOT.jar
```

**After:**

```
web: java -jar build/libs/yourAppName-1.0-SNAPSHOT-all.jar
```

f) Now, go to the Terminal and type in this:

```
$ heroku config:set GRADLE_TASK="shadowJar"
```

g) You're done setting it up.

6) Go to your Main class, and add this import:

```
import static spark.Spark.*;
```

7) Then in the `main` method, add this:

```
public static void main(String[] args){
        get("/token", (req,res) -> "Hello World");
}
```

This is a lambda expression. A lambda helps by simplifying this :

```
get("/token", new Route() {
   @Override
   public Object handle(Request request, Response response) throws Exception {
       return "Hello World!";
   }
});
```

Into this:

```
get("/token", (req,res) -> "Hello World");
```

8) If that code line is giving you an error saying it needs a higher language level:

   Go to File>Project Structure and change the dropdown in “Project language level” to 8 or higher.

   Then click “OK”, and go to your build.gradle and change `sourceCompatibility` to 1.8. Now, go to View>Tool Windows>Gradle. Click the refresh button again.

9) Great, it’s working. Your `main` method should look like this:

```
public static void main(String[] args){
   get("/token", (req,res) -> "Hello World");
}
```

10) That’s not enough. Spark needs you to give it a port. So, we create a new method that returns an int:

```
private static int getHerokuAssignedPort() {
         ProcessBuilder processBuilder = new ProcessBuilder();
         if (processBuilder.environment().get("PORT") != null) {
        return Integer.parseInt(processBuilder.environment().get("PORT"));
       }
       return 4567; //return default port if heroku-port isn't set (i.e. on localhost)
}
```

   Now above the `get(...)` we did, add this:

```
Spark.port(getHerokuAssignedPort());
```

11) Now, go to your Terminal and type the following commands:

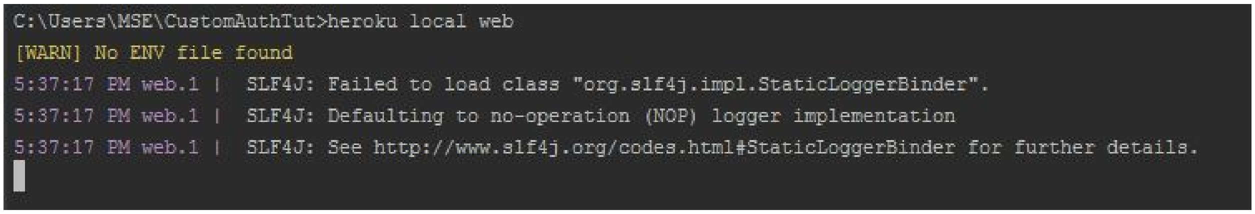
```
$ gradlew shadowJar
$ git add .
$ git commit -m "Added spark"
$ heroku local web
```

12) After 5 seconds or so, you should get something like this:

```
[WARN] No ENV file found
5:37:17 PM web.1 |  SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
5:37:17 PM web.1 |  SLF4J: Defaulting to no-operation (NOP) logger implementation
5:37:17 PM web.1 |  SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for
further details.
```

But don't worry about those warnings. You might not get them anyways.

13) You should see that your terminal looks like this



```
C:\Users\MSE\CustomAuthTut>heroku local web
[WARN] No ENV file found
5:37:17 PM web.1 |  SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
5:37:17 PM web.1 |  SLF4J: Defaulting to no-operation (NOP) logger implementation
5:37:17 PM web.1 |  SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

You can press CTRL+C to stop the local server from running.

14) If you go to your web browser, and type in

`localhost:5000/token`

You should get a "Hello World" message.

15) Next thing we do is put it up on the web.

```
$ git push heroku master
```

And now to open the site, you can do a `$ heroku open` to easily open it. The url would be yourappname.herokuapp.com

16) You'll now get a **404 error**. That's because this is our listener:

```
get("/token", (req,res) -> "Hello World");
```

You can see that we set the path to **/token.** Just go to /token from your url. Instead of https://yourappname.herokuapp.com, put in https://yourappname.herokuapp.com/token.

17) You'll see "Hello World". That's great.

Now that we have our server running, we need to actually generate a Firebase custom token.

To do so, we need the Admin SDK, which is the SDK for doing so.

Go to your build.gradle, and under dependencies (not under buildscript{dependencies}) add this:

```
compile 'com.google.firebase:firebase-admin:4.1.2'
```

Now, go to View>Tool Windows>Gradle. Click the refresh button ↻ again.

What will happen is the following:

- Your Android Application calls on [https://yourappname.herokuapp.com/token](https://yourappname.herokuapp.com/token) and that will return a token.
- However, you want to pass the entered **phone number** as a parameter to your server, so it could generate a custom token using that phone number.

**What's the code?**

At this point, you'll have this code in your Main class::

```
public static void main(String[] args){
   Spark.port(getHerokuAssignedPort());
   get("/token", (req,res) -> "Hello World");
}
```

We don't want to return "Hello World", but the custom token. Let's do this step by step:

Go to console.firebase.google.com and select a project or create a new one.

1. Navigate to the **Service Accounts** tab in your project's settings page.
2. Select your Firebase project. If you don't already have one, click the **Create New Project** button.
3. Click the **Generate New Private Key** button at the bottom of the **Firebase Admin SDK** section of the **Service Accounts** tab. A JSON will be downloaded.
4. Move that JSON file to the root of the IntelliJ project. Rename it to “service-key”

Now, in your `main` method, add this at the beginning:

```
FileInputStream serviceAccount = new
FileInputStream("service_key.json");
FirebaseOptions options = new FirebaseOptions.Builder()

.setCredential(FirebaseCredentials.fromCertificate(serviceAccount))
        .setDatabaseUrl("https://salawat-af7a4.firebaseio.com/")
        .build();
FirebaseApp.initializeApp(options);
```

That code initializes Firebase.

Now, we want to pass the phone number entered by the user as a parameter to be the account’s **USER ID**.

In spark, the `get` method can handle parameters. To do so, you can do this:

```
get("/token/:phoneNumber", (req,res) -> req.params("phoneNumber");
```

This will return the parameter he entered. For example, if a user calls on “https://yourappname.herokuapp.com/token/+96112345678 , it will return +96112345678 .

In our case, we want to get the phone number as a parameter, set it as a user id for the custom token, then return the custom token:

```
get("/token/:phoneNumber", (request, response) -> {
   String uid = request.params("phoneNumber"); //get the passed
parameter
   Task<String> authTask =
FirebaseAuth.getInstance().createCustomToken(uid); //createCustomToken
//takes a UID as a parameter
   try {
       Tasks.await(authTask);
   } catch (ExecutionException | InterruptedException e) {
       return (e.getMessage());
   }
   return authTask.getResult(); //Return the custom token
});
```

Your `main` method will look like this:

```
public static void main(String[] args) throws FileNotFoundException,
UnknownError {
   FileInputStream serviceAccount = new
FileInputStream("serviceAccountKey.json");
   FirebaseOptions options = new FirebaseOptions.Builder()

.setCredential(FirebaseCredentials.fromCertificate(serviceAccount))
           .setDatabaseUrl("https://salawat-af7a4.firebaseio.com/")
           .build();
   FirebaseApp.initializeApp(options);
   Spark.port(getHerokuAssignedPort());
   get("/token/:phoneNumber", (request, response) -> {
       String uid = request.params("phoneNumber");
       Task<String> authTask =
FirebaseAuth.getInstance().createCustomToken(uid);
       try {
           Tasks.await(authTask);
       } catch (ExecutionException | InterruptedException e) {
           return (e.getMessage());
       }
       return authTask.getResult();
   });
}
```

Now, do:

```
$ git add .
$ git commit -m "Returns custom token"
$ git push heroku master
```

Now if you run https://yourappname.herokuapp.com/token/+96181703873

It should return something like this:

```
eyJhbGciOiJSUzI1NiJ9.eyJhdWQiOiJodHRwczovL2lkZW50aXR5dG9vbGtpdC5nb29nbGVhcGlzLmNvbS9n
b29nbGUuaWRlbnRpdHkuaWRlbnRpdHl0b29sa2l0LnYxLklkZW50aXR5VG9vbGtpdCIsImV4cCI6MTQ4ODczO
TQ4MiwiaWF0IjoxNDg4NzM1ODgyLCJpc3MiOiJmaXJlYmFzZS1hZG1pbnNkay1rY2Fid0BzYWxhd2F0LWFmN2
E0LmlhbS5nc2VydmljZWFjY291bnQuY29tIiwic3ViIjoiZmlyZWJhc2UtYWRtaW5zZGsta2NhYndAc2FsYXd
hdC1hZjdhNC5pYW0uZ3NlcnZpY2VhY2NvdW50LmNvbSIsInVpZCI6IiA5NjE4MTcwMzg3MyJ9.juJHRnnKbYW
k4_4f5BeTKfNVfQkfGo7LnHyOwHFSVM-9DAkeRd8C3J5XlBTyplIQ1Eo1hBccUbhUWDzTbiRxWT6bmP1u3mfZ
n9kDl_49unX-1RiPI3cfPcOEcf7769zAzskgIhHamrY0l8JKmwlqLi0IGC6piCNE7_UMM_4MRbI5PW9CZ6FT9
D96qT3HT2b1C6xGV2Zs4VYglmz7Dk-nekHOCSEIFuU24oSYi10JtOepv04odfYVNg1-yqwCyaqiCKovoW1Bsv
ER9jQTJ4GChbdh27tdeQkqAHD0mRQig_CCTK68UDZ4fyXLMO3espkUdEnyiKKKnskF9ntaOy_Wcg
```

That’s the custom token!!!

# Android Studio!

We can go back to Android Studio now! We need to call on our server and get the custom token.

But to do that, we need OkHTTP. This is a library that allows us to do HTTP requests.

Add this to your dependencies (**NOT under buildScript)** in your build.gradle.

```
compile 'com.squareup.okhttp3:okhttp:3.6.0'
```

When the user clicks submit, let’s say we call on signUpOrIn(String phoneNumber). Remember that you can never use a network call on the main thread. You have to create a new thread and do it over there.

```
public void signUpOrIn(String phoneNumber) {
 Thread thread = new Thread(new Runnable() {
  @Override
  public void run() {
   String url = "https://yourappname.herokuapp.com/token/" + phoneNumber; //TODO Replace
yourappname with your app name.
   try {
    String customToken = callOnNetwork(url);
    FirebaseAuth auth = FirebaseAuth.getInstance();
    auth.signInWithCustomToken(customToken).addOnCompleteListener(new OnCompleteListener <
AuthResult > () {
     @Override
     public void onComplete(Task < AuthResult > task) {
      //Handle success or failure here
     }
    });
   } catch (IOException e) {
    e.printStackTrace();
   }
  }
 });
thread.start();
}


OkHttpClient client = new OkHttpClient();
String callOnNetwork(String url) throws IOException {
  Request request = new Request.Builder()
      .url(url)
      .build();

  Response response = client.newCall(request).execute();
  return response.body().string();
}
```

You’re done! The callOnNetwork method calls on the url you passed as a parameter to that method and returns the result. Good Luck!

Part 2: sending SMS