This is a very raw work in progress. The steps are in place, but I haven't gone through and pulled screen captures to make it pretty yet.

Assumptions:

We're building a project for release, not just poking around. Our intent is to actually have a release APK. If you're planning other build variants and you know what they're going to be, it's a good idea to get all of this setup early.

At the end of this, we'll have a hello world app that can authenticate to firebase when run from debug or release.

Getting things up and on Firebase

Clean new project
MinSDK to support Firebase UI is 16
Go to project view, and build the release and debug directories
Generate Signed APK
Create Keystore
Create Signing Config (right click app module, open module settings F4)
Assign Signing Config to release build.
Modify build.gradle to add applicationIdSuffix to release and debug build types (gist to copy/paste in at this early stage would be great)
Change storeFile to relative path to keystore. Matters when you collaborate.

Checkpoint.
installRelease gradle task that pushes signed apk to your device
Ability to button run debug
Ability to button run release

Use the wizard to connect to firebase at this point. Have the wizard create the project. It takes a minute. Let it do it's thing.

Then click Step 2 and add Firebase Authentication to your app. Let the wizard add the dependencies, but follow these directions in terms of what to paste in your app:

Declare these at the method level:

```
private String TAG ="MainActivity";
private int RC_SIGN_IN = 69;
private String userID = "some_user";

private FirebaseAuth mAuth;
private FirebaseAuth.AuthStateListener mAuthListener;
private Boolean userIsLoggedIn;
```

Override onStart()/onStop() as directed by the wizard, or just paste this in:

```
@Override
public void onStart() {
  super.onStart();
  mAuth.addAuthStateListener(mAuthListener);
}

@Override
public void onStop() {
  super.onStop();
  if (mAuthListener != null) {
     mAuth.removeAuthStateListener(mAuthListener);
  }
}
```

Basing the next steps on the directions found here:
https://github.com/firebase/FirebaseUI-Android/blob/master/auth/README.md

There are far more details in that document. Use them. This is just basic stuff to get you off the ground.

Modify build.gradle:

```
dependencies {
   // ...
   compile 'com.firebaseui:firebase-ui-auth:1.2.0'
}
```

and add the Fabric repository

```
allprojects {
   repositories {
      // ...
      maven { url 'https://maven.fabric.io/public' }
   }
}
```

Enable Authentication via email and Google in Firebase console for this project.
(probably build some screen shots for this)

Drop this method in your activity:

```
private void authenticateUser(){

   /**
    * Authenticates user, and watches for changes.
    *
    * **/

   mAuth = FirebaseAuth.getInstance();
   userIsLoggedIn = false;

   mAuthListener = new FirebaseAuth.AuthStateListener() {
       @Override
       public void onAuthStateChanged(@NonNull FirebaseAuth firebaseAuth) {
           FirebaseUser user = firebaseAuth.getCurrentUser();
           if (user != null) {
               // User is signed in
               userIsLoggedIn = true;
               userID = user.getUid();
               Toast.makeText(MainActivity.this, "Welcome, " + user.getDisplayName()
                       , Toast.LENGTH_SHORT).show();
           } else {
               userIsLoggedIn = false;
               startActivityForResult(
                       AuthUI.getInstance()
                               .createSignInIntentBuilder()
                               .setProviders(Arrays.asList(new
AuthUI.IdpConfig.Builder(AuthUI.EMAIL_PROVIDER).build(),
                                       new AuthUI.IdpConfig.Builder(AuthUI.GOOGLE_PROVIDER).build()))
                               .build(),
                       RC_SIGN_IN);
           }
       }
   };
}
```

Then call the authenticateUser() method from onCreate. If the user is not logged in, the FirebaseUI login activity takes over. If they are logged in, they get a toast. Replace the toast with whatever your program needs to get going.

You can and probably should build response handlers for the different types of login responses.

Drop in a method to let them sign out. Wire up to a button or menu option later:

```
public void signoutUser() {
  AuthUI.getInstance()
        .signOut(this)
        .addOnCompleteListener(new OnCompleteListener<Void>() {
           public void onComplete(@NonNull Task<Void> task) {
              // user is now signed out
              // do something if you need to
              finish();
           }
        });
}
```

Pushing any version of this should now connect to firebase for authentication.

Working with the realtime database.

The objective here is to connect to the database, push an object to the database, and read the object from the database. You will need a firebase connected project, with the installRelease task built, that is able to authenticate with firebase whether run from debug or release. You will also need to define a class for the objects you want to push to your database, and build an instance of that object.

Fire up the Firebase wizard and setup the realtime datbase
Add Realtime database to you app (button) Accept the gradle changes. Let it do its thing.

Since you're already a project that uses authentication, you don't have to do anything with the default rules to get started. Default rules allow authenticated users read and write access. Depending upon how your app works, you'll have to go back and tailor them for your use case. That's beyond scope here. We're just getting to the point where we can read and write our object to the database.

Instantiate the object you intend to write, and populate it with test values.
Drop in a method like this to write your object to the database. I'm writing an instance of a class called Spot. Replace this with your own Class and instance name.

```
private void saveSpotToFirebase(Spot spot){
  final DatabaseReference mDatabase;
  mDatabase = FirebaseDatabase.getInstance().getReference();
  mDatabase.child("spots")
```

```
        .child("users")
        .child(userID)
        .child(spot.getSignature())
        .setValue(spot);
}
```

Where you call this is important. You can’t just fire this off from onCreate(), as you may or may not be logged in yet. For testing, I like putting the call to write to Firebase in the mAuthListener. Once the login state changes and the user is no longer null, it’s save to write.

Could just as easily wire up a button that wouldn’t be available until login was completed.

Validating basic queries

Querying Firebase is reaching across the internet and asking for data. Your program doesn’t hang around waiting for a response when you query firebase. When you build the request, you build a listener that waits for the response from firebase. The response from Firebase hits your listener in the form of a DataSnapshot. This is pretty much just an arraylist of your objects. You’ll iterate through the DataSnapshot, and build instances of your object from what you receive.

There are two kinds of listeners you can build. If you want to query firebase and ask for data once, you’ll .addListenerForSingleValueEvent. If you want firebase to update you every time the data at a specific location changes, you’ll .addValueEventListener

An example of each type of listener that listens for Spots like we wrote earlier.

```
private void listenUp(){
  DatabaseReference mDatabase;
  mDatabase = FirebaseDatabase.getInstance().getReference();
  mDatabase.child("spots").child("users").child(userID)
        .addListenerForSingleValueEvent(new ValueEventListener() {
           @Override
           public void onDataChange(DataSnapshot dataSnapshot) {
              ArrayList<Spot> spots = new ArrayList<>();
              spots.clear();
              for (DataSnapshot child : dataSnapshot.getChildren()) {
                 Spot spot = child.getValue(Spot.class);
                 if (spot != null) {
                    spots.add(spot);
                 }
              }
              for (Spot spot : spots){
                 Log.e(TAG, "singleValueEvent spot.getName(): " + spot.getName());
```

```
                }
            }

            @Override
            public void onCancelled(DatabaseError databaseError) {
                Log.e(TAG, "database error");
            }
        });

    mDatabase.child("jams").child("users").child(userID)
        .addValueEventListener(new ValueEventListener() {
            @Override
            public void onDataChange(DataSnapshot dataSnapshot) {
                ArrayList<Spot> spots = new ArrayList<>();
                spots.clear();
                for (DataSnapshot child : dataSnapshot.getChildren()) {
                    Spot spot = child.getValue(Spot.class);
                    if (spot != null) {
                        spots.add(spot);
                    }
                }
                for (Spot spot : spots){
                    Log.e(TAG, "valueEventListener spot.getName(): " + spot.getName());
                }
            }

            @Override
            public void onCancelled(DatabaseError databaseError) {
                Log.e(TAG, "database error");
            }
        });
}
```

The single event listener in this case is redundant, but it here to demonstrate the structure. You'd use different .child() calls to point to different locations.

Again, when you make the method call is important. You'll have to be logged in, network connected, etc.

Configure ProGuard just as instructed in the Wizard, ignoring the word "optional"

Revisit your Firebase Rules now, or make an access review part of your project plan. Do not skip this step. Do not skip this step. Do not skip this step.

Make sure you revisit the last step. Security is important.

Ensure both your debug and release variants work (and any others you may have).