Firebase (10 pts.)

Lab Objectives:

- Get started with Firebase
- Learn about and use Firebase's Firestore NoSQL database

Lab overview

In this lab, adapted from the Cloud Firestore Android Google Codelab, you will work with a sample app that uses Firebase's Firestore NoSQL database. If you aren't familiar with NoSQL, check out this YouTube video. Otherwise, follow the instructions below. (There are many steps, but most of them are just copy/pasting or clicking buttons.)

Create a Firebase project

A Firebase project is a related collection of Firebase apps and services. Generally, you should create one Firebase project for one Android app (possibly multiple Android apps, if you have multiple apps interacting with the same data/services). The same Firebase project could also be used for a web app, an iOS app, etc.

1. Sign in to the Firebase console at https://console.firebase.google.com/u/0/. This uses a Google account. You will need to either use your personal account or create a temporary Google account (https://accounts.google.com/).

2. In the Firebase console, click on *Get started with a new Firebase project*.

3. Enter a name for your Firebase project. The sample app for this lab is called "Friendly Eats."

4. You can disable Google Analytics for the project. Otherwise, take the defaults for any other options. When your Firebase project is ready, click Continue.

Set up the sample Android Studio project

1. Download the source code from the file attached in canvas:

2. Open the downloaded project in Android Studio. There may be errors - these will be fixed in the following sections.

Connect the Android Studio project to Firebase

This requires configuration on both the Firebase side and the Android Studio side. On the Firebase side, you need to register your app. This needs to be done for every platform using your Firebase project.

1. In the Firebase console, select Project Overview in the left navigation.

2. Click the Android button to register an Android app for your Firebase project.

3. Enter com.google.firebase.example.fireeats as the package name. Note that this must match the package name of your Android app.

4. Click Register App.

5. Follow the instructions to download google-services.json and copy it into the app/directory of the Android Studio project.

6. Click Next. Skip the step about adding the Firebase SDK (that has already been done in the sample app).

Configure the Android app to use production Firebase

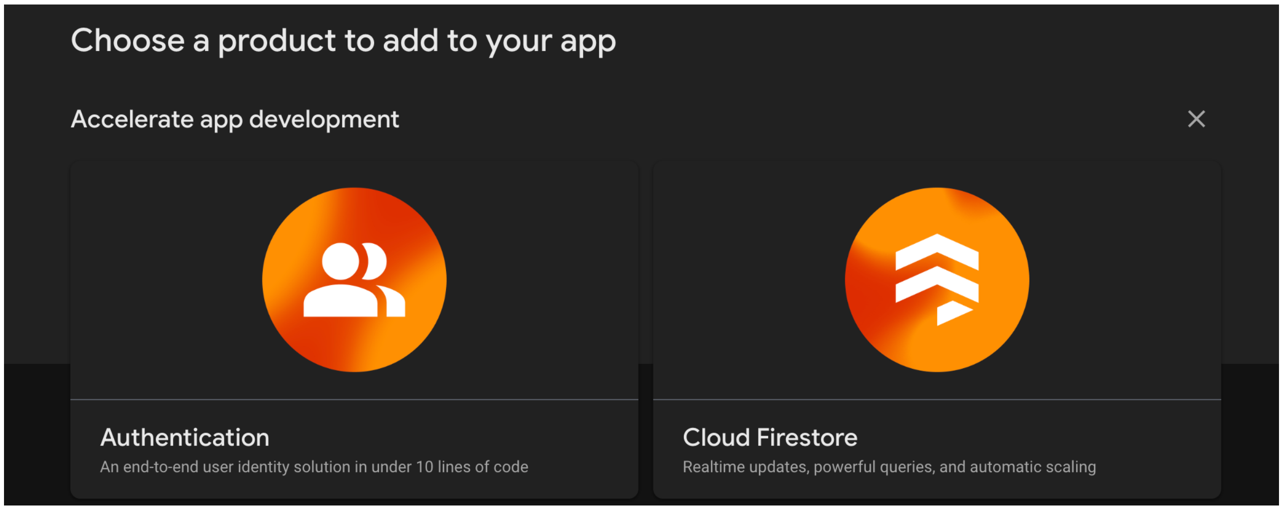
Firebase has a development tool that allows you to emulate a Firebase backend on your local machine. This allows a developer to, for example, test some new code that updates the database, without running it in the production database. Such tools are critical for the development process; however, in this lab, we want to work with the real cloud!

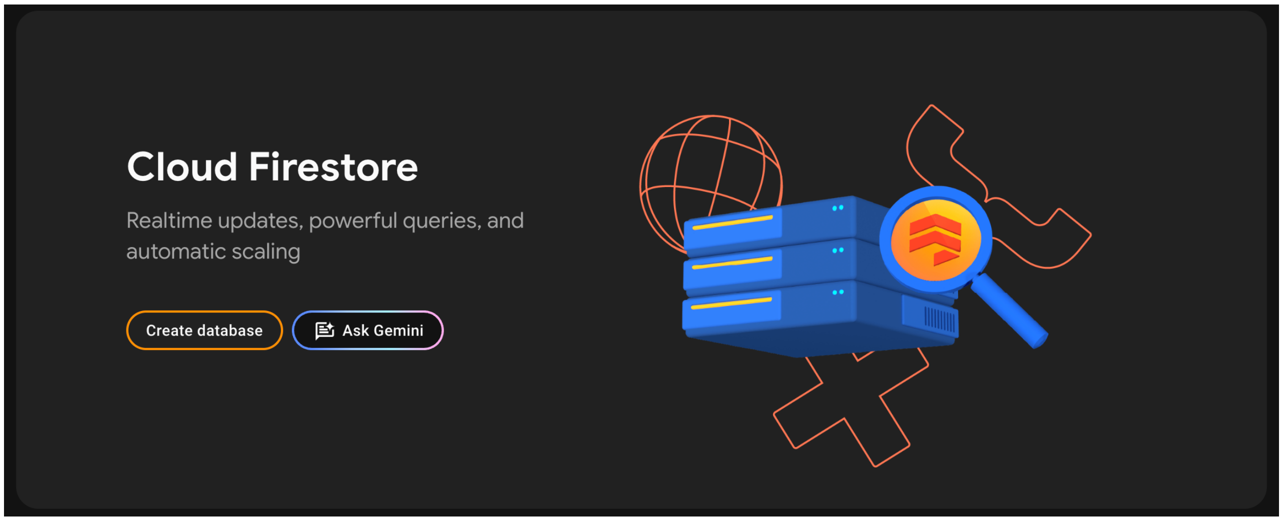
1. In Android Studio, open FirebaseUtil.java. Its located at app/src/main/java/…/util/..

2. Find the line that declares sUseEmulators. Currently this is set to BuildConfig.DEBUG, which means if we are building the app for debugging, we will use the emulator, but if we build the app in release mode, we will use the real cloud.

3. Change this line so that sUseEmulators = false.

4. Later we will use the Firestore database. Let's create it now. In the Firebase console, navigate to Firestore and click Create Database.


Choose a product to add to your app
Accelerate app development
Authentication
An end-to-end user identity solution in under 10 lines of code
Cloud Firestore
Realtime updates, powerful queries, and automatic scaling

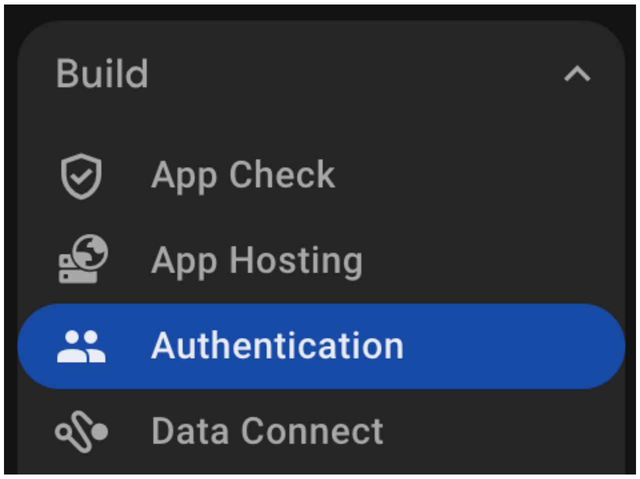


5. Choose nam5 (us-central) for the location
6. Choose to start in **test mode**. We will add security rules later.

Firebase Authentication

Firebase includes a cloud authentication service, and the sample app is set up to use it.

1. In the Firebase console, navigate to Build>Authentication and click Get Started.



2. Under Sign-in providers, enable the email/password sign-in method. Leave "email link" disabled.

3. In MainActivity.java, look at onStart(). If the user is not signed in, startSignIn() is called. This method generates an Intent to start the authentication UI provided by the Firebase Android SDK.

4. Run the app (it should run at this point).

5. Sign in using some email/password combination. If it's slow, try updating your emulator version. Open the SDK manager, go to the SDK tools tab, and update the Android emulator. *Note: don't use your real email password here - you are creating an account for your app, not signing in to your Google account! The Firebase Authentication UI does also support Google sign-in, but it's not configured for this app.*

6. In the Firebase console, navigate to Authentication->Users. You should now see the user you just created.

Write data to Firestore

Firestore is a NoSQL database offered by Firebase, and the recommended database for most new Firebase projects. Data in Firestore is stored in documents, which are organized into collections or subcollections; see here for more details. The Firestore database for your project currently has no data.

1. In MainActivity.java, look for onAddItemsClicked(). This is run when you want to populate the database with dummy data.

2. In onAddItemsClicked(), add code that will add 10 random Restaurant objects to the database. Hints:
    a. Start by getting a reference to a collection named "restaurants" with the following:
    CollectionReference restaurants = mFirestore.collection("restaurants");
    b. You can add an object to this collection using the following:
    restaurants.add(restaurant);
    c. The file RestaurantUtil.java has a static method that will help you generate a random restaurant, if desired.

3. Run the app again, and select "Add Random Items" from the overflow menu.

4. In the Firebase console, navigate to Firestore Database and take a look at the data you just created and pushed to the cloud!

Display data from Firestore

You can retrieve data from Firestore using a Query, which is typically built from a CollectionReference object like that obtained above. You can call get() on a Query or CollectionReference to get data. But Query objects can also be "live," meaning that you can listen to them and receive updates when the database is updated (without explicitly rerunning the query). We will do the latter here.

1. In onCreate() of MainActivity, add the following code right below the call to FirebaseUtil.getFirestore(). This creates the Query.
```
// Get the 50 highest rated restaurants
mQuery = mFirestore.collection("restaurants")
.orderBy("avgRating",Query.Direction.DESCENDING)
.limit(LIMIT);
```

2. We will show the data in a RecyclerView, which is a special view designed for displaying lists of data. RecyclerViews use an Adapter object to adapt the list of data to the slots in the view.

    Open FirestoreAdapter.java. Add the below code to make the class implement EventListener for QuerySnapshots (the format of the data returned by a Query):
    implements EventListener<QuerySnapshot>

If this generates an error, make sure that com.google.firebase.firestore.EventListener is imported, not java.util.EventListener.

3. Now add the onEvent() method to implement this interface:

```
// Add this method
@Override
public void onEvent(QuerySnapshot documentSnapshots,
                                FirebaseFirestoreException e) {

    // Handle errors
    if (e != null) {
        Log.w(TAG, "onEvent:error", e); return;
    }

    // Dispatch the event
    for (DocumentChange change : documentSnapshots.getDocumentChanges()) {
        // Snapshot of the changed document
        DocumentSnapshot snapshot = change.getDocument();

        switch (change.getType()) { case ADDED:
                // TODO: handle document added
                break;
            case MODIFIED:
                // TODO: handle document modified
                break;
            case REMOVED:
                // TODO: handle document removed
                break;
        }
    }

    onDataChanged();
}
```

4. Fix any missing imports using the alt-enter shortcut.

5. Note that onEvent() allows you to distinguish between whether the updated document was just added, modified, or removed. Add three methods to define the behavior in these cases:

```
protected void onDocumentAdded(DocumentChange change) {
    mSnapshots.add(change.getNewIndex(), change.getDocument());
    notifyItemInserted(change.getNewIndex());
}

protected void onDocumentModified(DocumentChange change) {
if (change.getOldIndex() == change.getNewIndex()) {
```

```
            // Item changed but remained in same position
            mSnapshots.set(change.getOldIndex(), change.getDocument());
            notifyItemChanged(change.getOldIndex());
        } else {
            // Item changed and changed position
            mSnapshots.remove(change.getOldIndex());
            mSnapshots.add(change.getNewIndex(), change.getDocument());
            notifyItemMoved(change.getOldIndex(),  change.getNewIndex());
        }
    }

    protected void onDocumentRemoved(DocumentChange change) {
        mSnapshots.remove(change.getOldIndex());
        notifyItemRemoved(change.getOldIndex());
    }
```

6. Now update onEvent() so that the corresponding methods are called for the different cases.

7. Finally, finish the implementation of startListening() to attach the listener:

```
    public void startListening() {
        if (mQuery != null && mRegistration == null) {
            mRegistration = mQuery.addSnapshotListener(this);
        }
    }
```

8. Run the app. You should now see the list of restaurants added in the previous section.

9. In the Firebase console, change one of the values in the database while the app is running. You should see this value get updated in the app automatically.

Implement sorting and filtering

You may have noticed a sorting and filtering menu in the app. The logic for building a Filters helper object from this menu is already implemented; let's implement the sorted and filtered Query.

1. In MainActivity, find onFilter(). The argument to onFilter() is a Filters object. Refer to Filters.java to see the options available.

2. Use the methods of Filters and Query to complete the following code in onFilter():

```
    @Override
    public void onFilter(Filters filters) {
        // TODO: Construct query basic query
        Query query = ??? //hint: CollectionReference extends Query

        // TODO: Filter by category (equality filter)
        //   TODO:  Filter by City (equality filter)

        //   TODO:  Filter by Price (equality filter)

        //   TODO:  Sort by specified order (orderBy with direction)

        //    Limit   items
        query = query.limit(LIMIT);

        // Update the query
        mQuery = query;
        mAdapter.setQuery(query);

        // Set header
        mCurrentSearchView.setText(Html.fromHtml(filters.getSearchDescription(this)));
        mCurrentSortByView.setText(filters.getOrderDescription(this));

        // Save filters
        mViewModel.setFilters(filters);
    }
```

3. Run the app again and try filtering by multiple values. If you get the below error, it's because the complex query requires a compound index to the database. Follow the link in the error message to create the index, and then re-run the app.
   ```
   FAILED_PRECONDITION: The query requires an index.
   ```

Use a transaction to update data

We want to be able to add a Rating in a "ratings" subcollection of restaurants. However, doing so requires not only adding a rating document, but also updating the average rating and number of ratings for the restaurant document. If we make these updates one at a time, there are race conditions that could result in inconsistent data. Instead, we should make all of these updates at once in a Transaction.

1. Open RestaurantDetailActivity.java and find the addRating() method. Implement it with the following code. Particularly note how the restaurant is retrieved from the database and converted into a Restaurant object.

```
    private Task<Void> addRating(final DocumentReference restaurantRef,final Rating rating) {
        // Create reference for new rating, for use inside the transaction
        final DocumentReference ratingRef = restaurantRef.collection("ratings").document();

        // In a transaction, add the new rating and update the aggregate totals
        return mFirestore.runTransaction(new Transaction.Function<Void>() {
            @Override
```

```
                public Void apply(Transaction transaction) throws
                        FirebaseFirestoreException {

                    Restaurant restaurant = transaction.get(restaurantRef)
                            .toObject(Restaurant.class);

                    // Compute new number of ratings
                    int newNumRatings = restaurant.getNumRatings() + 1;

                    // Compute new average rating
                    double oldRatingTotal = restaurant.getAvgRating() * restaurant.getNumRatings();
                    double newAvgRating = (oldRatingTotal + rating.getRating()) / newNumRatings;

                    // Set new restaurant info
                    restaurant.setNumRatings(newNumRatings); restaurant.setAvgRating(newAvgRating);

                    // Commit to Firestore
                    transaction.set(restaurantRef, restaurant); transaction.set(ratingRef, rating);

                    return null;
                }
            });
        }
```

2. Run the app again. You should now be able to navigate to a restaurant's detail page and add a new review. After doing so, your review should show up on the restaurant page, and the number of reviews and average rating for the restaurant should be up to date.

Add security rules

Our final step is to explore the security rules for the database. By default, any user can read/write to any document in the database. This is not good, but Firebase allows you to configure rules to lock down security for relevant services.

1. In the Firebase console, navigate to Firestore -> Rules.

2. Look at the existing rules. Try to understand the format, and who/what the rules are granting access to.

3. Replace the default rules with the following. Note how to check which user is making the request. Also note that you can define functions within the rules for more complex, modular logic.

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    // Determine if the value of the field "key" is the same
    // before and after the request.
```

```
        function isUnchanged(key) {
          return (key in resource.data)
            && (key in request.resource.data)
            && (resource.data[key] == request.resource.data[key]);
        }

        // Restaurants
        match /restaurants/{restaurantId} {
          // Any signed-in user can read
          allow read: if request.auth != null;

          // Any signed-in user can create
          // WARNING: this rule is for demo purposes only!
          allow create: if request.auth != null;

          // Updates are allowed if no fields are added and name is unchanged
          allow update: if request.auth != null
                              && (request.resource.data.keys().hasAny(resource.data.keys())) &&
                              isUnchanged("name");

          // Deletes are not allowed.
          // Note: this is the default, there is no need to explicitly state this.
          allow delete: if false;

          // Ratings
          match /ratings/{ratingId} {
            // Any signed-in user can read
            allow read: if request.auth != null;

            // Any signed-in user can create if their uid matches the document
            allow create: if request.auth != null
                              && request.resource.data.userId == request.auth.uid;

            // Deletes and updates are not allowed (default)
            allow update, delete: if false;
          }
        }
      }
    }
```

4. Click Publish and run the app again. Hopefully it still works!

What to Turn in:

- Demonstrate your working app to a TA, including filtering and creating a new review.