CMU 04-801: Advanced Mobile Application Development
Week 6: Firebase

Firebase

https://firebase.google.com/
Google's Firebase is a backend-as-a-service (BaaS) that allows you to get an app with a server-side real-time database up and running very quickly. Providing this as a service means you don't have to set up a database, write all the code to synchronize the data, and figure out security and authentication. Firebase stores data as JSON in the cloud in a NoSQL database.

With SDKs available for the web, Android, iOS, and a REST API it lets you easily sync data across devices/clients on iOS, Android, and the Web.

Features: https://firebase.google.com/products/

- Database
- Cloud Storage
- Authentication
- Hosting
- Cross platform support – web, iOS, Android

Firebase was founded by Andrew Lee and James Tamplin in 2011, officially launched in April 2012, and purchased by Google two years later.

Get Started
https://firebase.google.com/ Get Started
Get started by logging in with your Google login to create a Firebase account
Get Started:
Create a new Firebase project called Recipes.
A project in Firebase stores data that can be accessed across multiple platforms so you can have an iOS, Android, and web app all sharing the same project data. For completely different apps use different projects.
Firebase has two databases, the Realtime Database(Firebase) and Cloud Firestore. Both are noSQL databases but Firestore is their newer version so we'll use that. (Be careful of the difference between Firebase and Firestore online as they're different and have different structures, methods, etc)
Then you'll be taken to the dashboard where you can manage the Firebase project.

Cloud Firestore

https://firebase.google.com/docs/firestore
Cloud Firestore is a flexible, scalable noSQL database for mobile, web, and server development. It keeps your data in sync across client apps through real-time listeners and offers offline support for mobile and web.

Get Started
https://firebase.google.com/docs/firestore/quickstart
From the console's navigation pane, select **Database**, then click **Create database** for Cloud Firestore.
Start in test mode.

- Good for getting started with the mobile and web client libraries
- Allows anyone to read and overwrite your data.

Select the multi-regional location nam5(us-central)

Data Model
https://firebase.google.com/docs/firestore/data-model
Cloud Firestore is a NoSQL, document-oriented database. Unlike a SQL database, there are no tables or rows. Instead, you store data in documents, which are organized into collections.
Documents

- Document is the unit of storage
- A document is a lightweight record that contains fields, which map to values.
- Each document is identified by a unique id or name
- Each document contains a set of key-value pairs.
- All documents must be stored in collections.
- Documents within the same collection can all contain different fields or store different types of data in those fields.
- However, it's a good idea to use the same fields and data types across multiple documents, so that you can query the documents more easily.
- The names of documents within a collection are unique. You can provide your own keys, such as user IDs, or you can let Cloud Firestore create random IDs for you automatically.

Collection

- A collection is a container for documents
- A collection contains documents and nothing else.
- A collection can't directly contain raw fields with values, and it can't contain other collections.
- You can use multiple collections for different related data (orders vs users)

You do not need to "create" or "delete" collections. After you create the first document in a collection, the collection exists.

Every document in Cloud Firestore is uniquely identified by its location within the database. To refer to a location in your code, you can create a *reference* to it.
A reference is a lightweight object that just points to a location in your database. You can create a reference to a collection or a document.

```
CollectionReference recipesCollectionRef = db.collection("recipes");

DocumentReference recipeDocumentRef = db.collection("recipes").document(ID);
```

Let's create a collection and the first document using the console so we know our collection exists.

In the console go into Database | Data
Create a collection called recipes.
Now create a document using auto id with the fields name, and url.

Note that because we're in test mode our security rules are public and you'll see the following in the rules tab:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
```

```
    allow read, write;
   }
  }
}
```

You'll want to change this when not in test mode or when using authentication.

Read Data

There are two ways to retrieve data stored in Cloud Firestore. Either of these methods can be used with documents, collections of documents, or the results of queries:

- Call a method to get the data.
- Set a listener to receive data-change events.

Methods:

https://firebase.google.com/docs/firestore/query-data/get-data

Get a single document:

```
DocumentReference docRef = db.collection("cities").document("SF");
DocumentSnapShot docSnapShot = docRef.get();
```

Get a single document and convert it to an object of your model class (field names must match). You can then access the instance of the POJO class.

```
DocumentSnapShot docSnapShot = db.collection("cities").document("SF").get();
City city = docSnapShot.toObject(City.class);
```

Get multiple documents from a collection. You then need to loop through the result or access the one you want. You can use the various where() methods to define your query.

```
CollectionReference citiesCollectionRef = db.collection("cities");
QuerySnapshot qSnapShot = citiesCollectionRef.get();
```

Listeners:

https://firebase.google.com/docs/firestore/query-data/listen

When you set a listener, Cloud Firestore sends your listener an initial snapshot of the data, and then another snapshot each time the document changes.

You can listen to a document with the onSnapshot() method. When listening for changes to a document, collection, or query, you can pass options to control the granularity of events that your listener will receive.

```
DocumentReference docRef = db.collection("cities").document("SF");
docRef.addSnapshotListener(new EventListener<DocumentSnapshot>() {
    @Override
    public void onEvent(@Nullable DocumentSnapshot snapshot,
                        @Nullable FirebaseFirestoreException e) {
        ... handle data change
    }
});
```

An initial call using the callback you provide creates a document snapshot immediately with the current contents of the single document. Then, each time the contents change, another call updates the document snapshot.

Local writes in your app will invoke snapshot listeners immediately. This is because of a feature called "latency compensation." When you perform a write, your listeners will be notified with the new data *before* the data is sent to the backend.

Queries:
https://firebase.google.com/docs/firestore/query-data/queries
Cloud Firestore provides powerful query functionality for specifying which documents you want to retrieve from a collection or collection group. These queries can also be used with both the methods and listeners above.
By default, a query retrieves all documents that satisfy the query in ascending order by document ID. You can specify the sort order for your data using orderBy(), and you can limit the number of documents retrieved using limit().

Write Data
When you write a document to a collection in Cloud Firestore each document needs an identifier. You can explicitly specify a document identifier or have Cloud Firestore automatically generate it. You can also create an empty document with an automatically generated identifier, and assign data to it later.

.set() creates or overwrites a single document and requires a document identifier.
.add() adds a document and automatically generates a document identifier for you.
.update() lets you update some fields of a document without overwriting the entire document.

Delete Data
.delete() deletes a specific document
Note that deleting a document does NOT delete any subcollections it might have.

To delete an entire collection or subcollection in Cloud Firestore, retrieve all the documents within the collection or subcollection and delete them. This is not recommended from a mobile client

Offline Support
https://firebase.google.com/docs/firestore/manage-data/enable-offline
Cloud Firestore supports offline data persistence. A copy of the Cloud Firestore data that your app is actively using will be cached so your app can access the data when the device is offline. You can write, read, listen to, and query the cached data. When the device comes back online, Cloud Firestore synchronizes any local changes made by your app to the Cloud Firestore backend.

- For Android and iOS, offline persistence is enabled by default. To disable persistence, set the PersistenceEnabled option to false
- For the web, offline persistence is disabled by default

FirebaseUI
https://github.com/firebase/FirebaseUI-Android
FirebaseUI is an open-source library for Android that allows you to quickly connect common UI elements to Firebase APIs. FirebaseUI has separate modules for using Firebase Realtime Database, Cloud Firestore, Firebase Auth, and Cloud Storage.

FirebaseUI makes it simple to bind data from Cloud Firestore to your app's UI.
https://firebaseopensource.com/projects/firebase/firebaseui-android/firestore/readme.md/
FirebaseUI offers two types of RecyclerView adapters for Cloud Firestore:

- FirestoreRecyclerAdapter — binds a Query to a RecyclerView and responds to all real-time events included items being added, removed, moved, or changed. Best used with small result sets since all results are loaded at once.
    - For the FirestoreRecyclerAdapter to work seamlessly with your model class the data members in your class must match the field names of your documents and the getter and setter method names must follow the standard naming pattern
    - To perform some action every time data changes or when there is an error you override the onDataChanged() and onError() methods of the adapter
- FirestorePagingAdapter — binds a Query to a RecyclerView by loading data in pages. Best used with large, static data sets. Real-time events are not respected by this adapter, so it will not detect new/removed items or changes to items already loaded.

Android Studio Setup
https://firebase.google.com/docs/android/setup
New project called Recipes
Basic Activity template

1. Create a Firebase project
Chose existing Recipes database or create a new one

2. Register your app with Firebase
Click the Android icon or Add app
Enter your app's application id which is the same as the package name which you can get from your Gradle files or Android manifest.

3. Add the Firebase configuration file
Click Download google-services.json to obtain your Firebase Android config file (google-services.json).
Go into the Project view.
Move your config file into the app-level directory of your app.

4.Add Cloud Firestore to your app
In your project-level `build.gradle` file, make sure Google's Maven repository is included in both your `buildscript` and `allprojects` sections.

```
buildscript {
    repositories {
        google()
        jcenter()
    }
```

Add the following dependencies and Google services plugin to your project gradle files:
`build.gradle (project-level)`

```
dependencies {
    classpath 'com.android.tools.build:gradle:3.5.3'
    classpath 'com.google.gms:google-services:4.3.3'
```

4. In your `app/build.gradle` file add the Firebase Gradle plugin

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

Also add the Cloud Firestore Android libraries to your `app/build.gradle` file:

```
implementation 'com.google.firebase:firebase-firestore:21.4.0'
implementation 'com.firebaseui:firebase-ui-firestore:4.3.1'
```

The second dependency is for FirebaseUI (https://github.com/firebase/FirebaseUI-Android) which is a library that helps bind data into our app's UI.
You will need to add other dependencies for other Firebase functionality.

Model class

Go into your project's Java folder and create a new package called model and add a new Java class called Recipe for our model.

```
public class Recipe {
    private String name;
    private String url;

    public Recipe(){
        // Default constructor
    }

    public Recipe(String name, String url) {
        this.name = name;
        this.url = url;
    }

    public String getName(){
        return name;
    }

    public String geturl(){
        return url;
    }
}
```

Repository

Create a package called data and a class called AppRepository that will handle all interaction with Firebase.
If your project doesn't recognize FirebaseFirestore go back to the setup steps and make sure you downloaded the google-services.json file and put it in the app level directory of your app.
Here we're creating data members for the database and the recipes collection and initialize them in the constructor.
We have a method that returns options which we'll need for the recycler view as well as methods that add and delete a document.

```
public class AppRepository {

    // get Cloud Firestore instance
    private FirebaseFirestore db;

    //recipe collection
    private CollectionReference reciperef;
```

```
    public AppRepository(){
        db = FirebaseFirestore.getInstance();
        reciperef = db.collection("recipes");
    }

    //options to set up the adapter
    public FirestoreRecyclerOptions<Recipe> getOptions(){
        Query myquery = reciperef;
        FirestoreRecyclerOptions<Recipe> options = new
FirestoreRecyclerOptions.Builder<Recipe>()
                .setQuery(myquery, Recipe.class)
                .build();
        return options;
    }

    public void insertRecipe(Recipe newRecipe){
        reciperef.add(newRecipe);
    }

    public void deleteItem(String id){
        reciperef.document(id).delete();
    }
}
```

Layout

I used the same layout files for my recyclerview as I have in the past projects so there's nothing new here.

ViewHolder

Before we create our FirebaseRecyclerAdapter we need to define our ViewHolder class.

```
public class RecipeViewHolder extends RecyclerView.ViewHolder{
    private TextView mTextView;

    public RecipeViewHolder(@NonNull View itemView) {
        super(itemView);
        mTextView = itemView.findViewById(R.id.textView);
    }

    public void setRecipeName(String name){
        mTextView.setText(name);
    }
}
```

Adapter

Although we could extend the RecyclerView as we have been doing, the Firestore UI library provides the FirestoreRecyclerAdapter class which binds a Query to a RecyclerView. It also has a snapshot listener built in that monitors changes to the Firestore query so when documents are added, removed, or change these updates are automatically applied to your recyclerview and UI in real time.
The FirestoreRecyclerAdapter requires a FirestoreRecyclerOptions which we set up in our AppRepository class.

```
public class RecipeAdapter extends FirestoreRecyclerAdapter<Recipe,
RecipeViewHolder>  {
    private Context mContext;
    private AppRepository mappRepository;

    public RecipeAdapter(FirestoreRecyclerOptions<Recipe> options, Context
context, AppRepository appRepository){
        super(options);
        this.mContext = context;
        this.mappRepository = appRepository;
    }

    @Override
    protected void onBindViewHolder(@NonNull final RecipeViewHolder
recipeHolder, int i, @NonNull final Recipe recipe) {
        recipeHolder.setRecipeName(recipe.getName());
    }

    @NonNull
    @Override
    public RecipeViewHolder onCreateViewHolder(@NonNull ViewGroup parent,
int viewType) {
        LayoutInflater inflater = LayoutInflater.from(parent.getContext());
        View itemView = inflater.inflate(R.layout.list_item, parent, false);
        RecipeViewHolder recipeHolder = new RecipeViewHolder(itemView);
        return recipeHolder;
    }
}
```

MainActivity

In our MainActivity class we need to set up the repo, recycler view, and adapter. We also need to start the snapshot listener the FirestoreRecyclerAdapter uses to monitor changes to the Firebase query. To begin listening for data we call the startListening() method in the onStart() lifecycle method. To remove the snapshot listener and all data in the adapter call the stopListening() method in the onStop() lifecycle method.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Toolbar toolbar = findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);

    //get the recycler view
    recyclerView = findViewById(R.id.recyclerView);

    //divider line between rows
    recyclerView.addItemDecoration(new DividerItemDecoration(this,
LinearLayoutManager.VERTICAL));

    //set a layout manager on the recycler view
    recyclerView.setLayoutManager(new LinearLayoutManager(this));
```

```
    setuprepo();

    setupadapter();

    FloatingActionButton fab = findViewById(R.id.fab);
    fab.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(final View view) {
        }
    });
}

private void setuprepo() {
    appRepository = new AppRepository();
}

private void setupadapter() {
    FirestoreRecyclerOptions<Recipe> options = appRepository.getOptions();
    recipeAdapter = new RecipeAdapter(options, this, appRepository);
    recyclerView.setAdapter(recipeAdapter);
}

@Override
protected void onStart() {
    super.onStart();
    recipeAdapter.startListening();
}

@Override
protected void onStop() {
    super.onStop();
    recipeAdapter.stopListening();
}
}
```

You should now be able to run your app and see all your Firebase data. If you make any changes through the console your app should automatically update.

Add data

In MainActivity a FAB is already set up in onCreate() so we'll use that to let to user add a recipe
We'll use an AlertDialog again but this time we'll need two EditTexts, for the name and url, so we'll need a linear layout object to put them in.
When we create the AlertDialog it's in an anonymous function so when implementing the listener, outer class RecipeMainActivity has to be specified to refer to the Activity instance and the keyword this in Java applies to the most immediate class being declared.
If the user is entering a lot of data you might prefer a separate activity instead.

```
FloatingActionButton fab = findViewById(R.id.fab);
fab.setOnClickListener(new View.OnClickListener() {
    @Override
```

```
    public void onClick(final View view) {
        //create a vertical linear layout to hold edit texts
        LinearLayout layout = new LinearLayout(MainActivity.this); //try
getApplicationContext()
        layout.setOrientation(LinearLayout.VERTICAL);

        //create edit texts and add to layout
        final EditText nameEditText = new EditText(MainActivity.this);
        nameEditText.setHint("Recipe name");
        layout.addView(nameEditText);
        final EditText urlEditText = new EditText(MainActivity.this);
        urlEditText.setHint("URL");
        layout.addView(urlEditText);

        //create alert dialog
        AlertDialog.Builder dialog = new
AlertDialog.Builder(MainActivity.this);
        dialog.setTitle("Add Recipe");
        dialog.setView(layout);
        dialog.setPositiveButton("Save", new
DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                //get entered data
                String recipeName = nameEditText.getText().toString();
                String recipeURL = urlEditText.getText().toString();
                if (recipeName.trim().length() > 0) {
                    //create new recipe item
                    Recipe newRecipe = new Recipe(recipeName, recipeURL);
                    //add to Firebase
                    appRepository.insertRecipe(newRecipe);
                }
                Snackbar.make(view, "Item added", Snackbar.LENGTH_LONG)
                        .setAction("Action", null).show();
            }
        });
        //sets cancel action
        dialog.setNegativeButton("Cancel", new
DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int whichButton) {
                // cancel
            }
        });
        //present alert dialog
        dialog.show();
    }
});
```

You should now be able to add recipes and see them in your recyclerview as well as in Firebase through the console.

Delete data

To delete items through the app we'll present a context menu on a long press which we'll add to the onBindViewHolder() method in RecipeAdapter.java. After getting the document's id we'll pass it to the deleteItem() method in our AppRepository class to delete the document from the collection.

```
//context menu
recipeHolder.itemView.setOnCreateContextMenuListener(new
View.OnCreateContextMenuListener() {
    @Override
    public void onCreateContextMenu(ContextMenu menu, final View v,
ContextMenu.ContextMenuInfo menuInfo) {
        //set the menu title
        menu.setHeaderTitle("Delete " + recipe.getName());
        //add the choices to the menu
        menu.add(1, 1, 1, "Yes").setOnMenuItemClickListener(new
MenuItem.OnMenuItemClickListener() {
            @Override
            public boolean onMenuItemClick(MenuItem item) {
                //get recipe that was pressed
                int position = recipeHolder.getAdapterPosition();
                //get document id
                String id = getSnapshots().getSnapshot(position).getId();
                //delete from repository
                mappRepository.deleteItem(id);

                Snackbar.make(v, "Item removed", Snackbar.LENGTH_LONG)
                        .setAction("Action", null).show();

                return false;
            }
        });
        menu.add(2, 2, 2, "No");
    }
});
```

Now you should be able to delete items on a long click and see them removed from your Firebase database and your RecyclerView.

Load web page

When the user taps on a recipe we want to load the recipe url in an app that can load web pages like a browser. We're going to use an implicit intent so the user will be prompted to chose an app to load the web page if the device has more than one capable of doing so.
In RecipeAdapter.java add an onClickListener to the onBindViewHolder() method in our adapter.
A DataSnapshot contains data from a Database location. We can then use the toObject() method to return the contents of the document converted to our Recipe class so we can get the url.
Because this class doesn't extend an activity class it can't start a new activity. We use our content, which is an activity, to start the new activity.

```
//onclick listener
recipeHolder.itemView.setOnClickListener(new View.OnClickListener() {
    @Override
```

```
    public void onClick(View v) {
        //get recipe that was pressed
        int position = recipeHolder.getAdapterPosition();
        //get snapshot
        DocumentSnapshot documentSnapshot =
getSnapshots().getSnapshot(position);
        //get recipe url
        String recipeURL = documentSnapshot.toObject(Recipe.class).geturl();
        //create new intent
        Intent intent = new Intent(Intent.ACTION_VIEW);
        //add url to intent
        intent.setData(Uri.parse(recipeURL));
        //start intent
        mContext.startActivity(intent);
    }
});
```

Now when you tap on a recipe its web page should open in a browser.