Advanced Mobile Application Development
Week 15: Android and Firebase

Firebase

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.

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.

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 CollectionReference to a collection or a document.

```
val recipeRef = db.collection("recipes")
```

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:

```
val docRef = db.collection("cities").document("SF").get()
```

Get a single document and convert it to an object of your model class (field names must match) using .toObject<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.

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.

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
https://firebase.google.com/docs/firestore/manage-data/add-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

Access Data Offline
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
Empty Activity template

1. Create a Firebase project
Chose existing Recipes database or create a new one
If you use the same Recipes Firebase project that we used for iOS you'll need to make it public so our app can read and write without authentication.

In the Firebase console chose your database and then go into Authentication and in sign-in method disable all the sign-in providers.
Also back in Firestore Database go to the Rules tab and make sure the documents permissions for both read and write are set to true so our app can access the database.

```
match /{document=**} {
  allow read, write;
}
```

2. Register your app with Firebase
In the Firebase console 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, Android manifest, or top of your MainActivity file.

3. Add the Firebase configuration file
In project settings 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 the `buildscript` section. You'll also need the Google services plugin as a dependency.

```
buildscript {
    repositories {
        google()
    }
  dependencies {
    classpath 'com.google.gms:google-services:4.3.10'
  }
}
```

In your `app/build.gradle` file apply the Google services plugin

```
plugins{
       id 'com.google.gms.google-services'
}
```

Also add these libraries to your `app/build.gradle` file:

```
implementation platform('com.google.firebase:firebase-bom:29.3.1')
implementation 'com.google.firebase:firebase-firestore-ktx'
implementation 'com.firebaseui:firebase-ui-firestore:8.0.1'
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.1"
implementation 'androidx.activity:activity-ktx:1.4.0'
```

The Firebase Android BoM (Bill of Materials) enables you to manage all your Firebase library versions by specifying only one version — the BoM's version. The BoM automatically pulls in the individual library versions mapped to BoM's version. All the individual library versions will be compatible. When you update the BoM's version in your app, all the Firebase libraries that you use in your app will update to the versions mapped to that BoM version.
The second dependency is for the cloud firestore kotlin library.

The third 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.
The last dependency is for the viewmodel libraries.

Add colors, create a theme with a Material Components parent theme, and apply it to your app.

Layout
activity_main.xml
Delete the text view.
In design view drag out a RecyclerView to display the list (can be found in common or containers).
Add start, end, top, and bottom constraints.
Make sure the RecyclerView has layout_width and layout_height set to "match_constraint" (0dp).
Give the RecyclerView the id recyclerView.
Add a Floating Action Button (under buttons) to the bottom right of the view and select the android drawable ic_input_add.
I shortened the id to fab.
Layout width and height should be wrap_content. Margin bottom and end 16dp.
I also added a string value to use for the content description.

```
<string name="fab_add">Add</string>
```

@string/fab_add

List Item Layout
We also need a layout file to determine what each row in our list will look like. Ours will be super simple with just a TextView.
File | New | Layout resource file
File name: list_item
Root: androidx.constraintlayout.widget.ConstraintLayout
Source set: main
Directory name: layout

Drag out a TextView and make sure it has an id (textView).
Make sure the TextView has layout_width and layout_height set to "wrap_content".
Add missing constraints.
I also added some padding so the text is in the vertical center of the row and made the text larger.

```
android:paddingStart="10dp"
android:paddingTop="8dp"
android:paddingBottom="8dp"
android:textAppearance="@style/TextAppearance.MaterialComponents.Headline6"
```

Once you don't need the default text to help with layout, remove it.

Also change the constraint layout's height to "wrap_content". If the height is match_constraint each text view will have the height of a whole screen.

Model class
Go into your project's Java folder and create a new package called model and add a new Kotlin class called Recipe for our model. Our model will match our database document structure with name and url properties.

Firestore also requires an empty constructor method so I added a secondary empty constructor as well.

```
data class Recipe(val name: String, val url:String) {
    constructor(): this("", ""){}
}
```

ViewModel

Now we'll create our view model class.
Select the model package and add a new class called RecipeViewModel.
You'll notice we're not using LiveData and that's because the FirestoreRecyclerAdapter class we're going to use has listeners built in.
We also have methods to add and delete a recipe from the database. The method to delete takes an id because that's what the Firestore delete() method will need.

```
class RecipeViewModel: ViewModel() {
    private val recipedb = RecipeDatabase()
    val options = recipedb.getOptions()

    fun addRecipe(recipe: Recipe){
        recipedb.addRecipe(recipe)
    }

    fun deleteRecipe(id: String){
        recipedb.deleteRecipe(id)
    }
}
```

RecipeDatabase() will give you an error because we haven't created that class yet.

Database

Create a package called util and a class called RecipeDatabase 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 constants for the database and the recipes collection.
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.

```
class RecipeDatabase {
    //Firestore instance
    private val db = FirebaseFirestore.getInstance()

    //recipe collection
    private val recipeRef = db.collection("recipes")

    //FirestoreRecycler options for the adapter
    fun getOptions(): FirestoreRecyclerOptions<Recipe> {
        //define query
        val query = recipeRef
        val options = FirestoreRecyclerOptions.Builder<Recipe>()
```

```
            .setQuery(query, Recipe::class.java)
            .build()
        return options
    }

    fun addRecipe(recipe: Recipe){
        recipeRef.add(recipe)
    }

    fun deleteRecipe(id: String){
        recipeRef.document(id).delete()
    }
}
```

The import for Query should be from firestore.

```
import com.google.firebase.firestore.Query
```

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 RecipeDatabase class.

In the java folder create a new class for our adapter.
File | New | Kotlin Class
Name: RecipeAdapter
Kind: Class

The class will need to subclass FirestoreRecyclerAdapter and have FirestoreRecyclerOptions and our view model in the primary constructor.

```
class RecipeAdapter(options: FirestoreRecyclerOptions<Recipe>, private val
viewModel: RecipeViewModel) : FirestoreRecyclerAdapter<Recipe,
RecipeAdapter.ViewHolder>(options) {}
```

You will give you errors until you implement the required methods.
Select the light bulb and choose implement methods to get rid of the error.

You will still have an error because we need to define the ViewHolder class which will set up the bindings to the view in the layout file.
In our RecipeAdapter class we'll create a ViewHolder class to render the item and a default constructor.

```
class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
    var nameTextView: TextView = view.findViewById(R.id.textView)
}
```

The import for View should be from view.

```
import android.view.View
```

Now we'll implement the required methods.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
RecipeAdapter.ViewHolder {
    val layoutInflater = LayoutInflater.from(parent.context)
    val recipeViewHolder = layoutInflater.inflate(R.layout.list_item,
parent, false)
    return ViewHolder(recipeViewHolder)
}

override fun onBindViewHolder(holder: RecipeAdapter.ViewHolder, position:
Int, model: Recipe) {
    //use model that is passed in and assign the properties defined in the
ViewHolder class
    holder.nameTextView.text = model.name
}
```

MainActivity

In our MainActivity class we need to set up the view model, adapter, and recycler view. 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.

```
class MainActivity : AppCompatActivity() {
    private val viewModel: RecipeViewModel by viewModels()
    private var recipeAdapter: RecipeAdapter? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        //get the recycler view
        val recyclerView: RecyclerView = findViewById(R.id.recyclerView)

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

        //set a layout manager on the recycler view
        recyclerView.layoutManager = LinearLayoutManager(this,
LinearLayoutManager.VERTICAL, false)

        //set up options
        val options = viewModel.options
        //create adapter using the options
        recipeAdapter = RecipeAdapter(options, viewModel)
        //set adapter on the recyclerview
        recyclerView.adapter = recipeAdapter

    }
```

```
    override fun onStart() {
        super.onStart()
        recipeAdapter?.startListening()
    }

    override fun 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

Let's set the FAB up in the MainActivity class to add recipes to the list.

Add these strings to your string resource file since we'll be using them in our alert dialog.

```
<string name="recipeHint">Recipe name</string>
<string name="urlHint">URL</string>
<string name="addRecipe">Add Recipe</string>
<string name="add">Add</string>
<string name="recipeAdded">Recipe Added</string>
<string name="action">Action</string>
<string name="cancel">Cancel</string>
```

In MainActivity set up the fab at the end of onCreate() 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.
If the user is entering a lot of data you might prefer a separate activity instead.

```
        findViewById<FloatingActionButton>(R.id.fab).setOnClickListener {
view ->
            //create a vertical linear layout to hold edit texts
            val layout= LinearLayout(this)
            layout.orientation = LinearLayout.VERTICAL

            //create edit texts and add to layout
            val nameEditText = EditText(this)
            nameEditText.setHint(R.string.recipeHint)
            layout.addView(nameEditText)
            val urlEditText = EditText(this)
            urlEditText.setHint(R.string.urlHint)
            layout.addView(urlEditText)

            // create alert dialog
            val dialog = AlertDialog.Builder(this)
            //set dialog title
            dialog.setTitle(R.string.addRecipe)
            //add layout to dialog
            dialog.setView(layout)
```

```
            //set OK action
            dialog.setPositiveButton(R.string.add) {dialog, which ->
                val recipeName = nameEditText.text.toString()
                val recipeURL= urlEditText.text.toString()
                if (!recipeName.isEmpty()){
                    //create new recipe item
                    val newRecipe = Recipe(recipeName, recipeURL)
                    //add item
                    viewModel.addRecipe(newRecipe)
                    Snackbar.make(view, R.string.recipeAdded,
Snackbar.LENGTH_LONG)
                            .setAction(R.string.action, null).show()
                }
            }
            //sets cancel action
            dialog.setNegativeButton(R.string.cancel) { dialog, which ->
                //cancel
            }
            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

We want an item to be deleted when the user does a long-press on an item. We'll do this though a context menu so they can choose to delete the item or not.

Add these strings to your string resource file since we'll be using them in our alert dialog.

```
<string name="delete">Delete?</string>
<string name="deleteRecipe">Recipe Deleted</string>
<string name="yes">Yes</string>
<string name="no">No</string>
```

In RecipeAdapter.kt we'll update onBindViewHolder() to set up the long click listener and create the context menu.

When the user clicks on the "yes" menu we'll call the delete method in our viewmodel.

```
//context menu
holder.itemView.setOnCreateContextMenuListener(){menu, view, menuInfo ->
    //set the menu title
    menu.setHeaderTitle(R.string.delete)

    //add the choices to the menu
    menu.add(R.string.yes).setOnMenuItemClickListener {
        //get recipe that was clicked
       //snapshots gets the array that the adapter is populated with
       //getSnapshot returns the snapshot at the position
        val id = snapshots.getSnapshot(position).id
        //delete item
        viewModel.deleteRecipe(id)
        Snackbar.make(view, R.string.deleteRecipe, Snackbar.LENGTH_LONG)
            .setAction(R.string.action, null).show()
```

```
            true
        }
        menu.add(R.string.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 in a browser. We're going to use an implicit intent so the user will be prompted to choose an app to load the web page if the device has more than one capable of doing so.
In RecipeAdapter.kt 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 the context of our view, which is the MainActivity, to start the new activity.

```
//onclick listener
holder.itemView.setOnClickListener{view->
    //get recipe that was clicked
    val documentSnapshot = snapshots.getSnapshot(position)
    val recipeURL = documentSnapshot.toObject<Recipe>()?.url

    //create new intent
    var intent = Intent()
    intent.action = Intent.ACTION_VIEW
    //var intent = Intent(Intent.ACTION_VIEW, it.context)
    intent.data = Uri.parse(recipeURL)

    //start activity
    view.context.startActivity(intent)
}
```

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