MBaaS in Android

Mobile Computing - Android

Objectives

- Students will be able to:
  - explain the purpose of MBaaS
  - explain how MBaaS compares to other database technologies
  - write and debug apps that use MBaaS

Introduction

- MBaaS — Mobile Backend as a Service — provides a series of services that many mobile apps need, including:
  - cloud storage using a no-sql database
  - user management
  - social media integration
  - push notifications
  - server-side computing
- In this document, alongside the MBaaS demo and the documentation available at Google Firebase, you will learn much of what you need to do in order to use MBaaS in your projects. You will still need to do some study in order to master MBaaS.

Getting Started

- Obtain a Google account
- Sign in to Firebase
- Click on Get Started and then Create a project.
- Fill in a project name, check the boxes and click on Continue

Getting Started

- You may wish to turn on analytics. If you do so, you will have to carefully consider what data you will/will not share with Google.
- Click on **Continue**
- Configure Analytics Sharing
- Click on **Create Project**

Plans

- Google has two plans – spark and blaze.
  - Spark is free but usage limited.
  - Blaze is free and pay if the usage goes over.

Database Options

- You have two options
  - Firebase RT – Original database, better for simpler data/queries. Limited scaling.
  - Firestore - Supports transactions, Scales better.
- Both are NOSQL
- Choosing your Database
- For demoing, we are going to go with FirebaseRT

Adding Android App

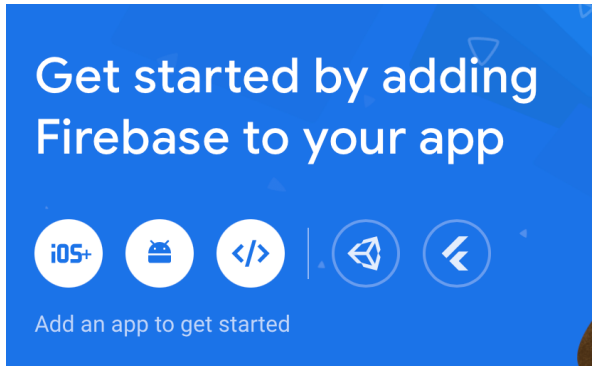
- Now that we have a project (remote database), we need to connect an android studio app to it. There are instructions for adding Firebase to an Android app.
- Requirements
  - Target API 19 (KitKat) or later
  - Uses JetPack (Android X)
  - Gradle v3.2.1
  - Compile SDK 28
- Create a project on Android Studio.

Typical Setup Tasks for MBaaS

- Get a app id and client key for the remote DB
- Install a package/SDK
- Add permissions to the manifest
- Import packages in Java
- Use app id and client key to establish a link to the data base.

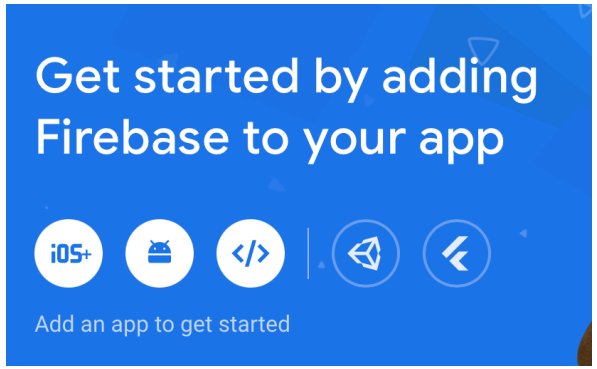
Using the Console

- You can interact with your data base through the console. It is convenient to be able to check the status of your database and repair in development.
- The console is also useful for shared access over multiple platforms.



Using the Console

- Click the Android button and fill out the package name. For more security, use SHA.



Configure Android App

- From the gear next to the overview button, select project settings.

- Download google-services.json
- Put it in the app folder

Add dependencies to App

- In the root level build.gradle.kts (Project) add in google services. The application version may be later than 7.3.0. The services plugin version should be the most recent.

```
plugins {
  id ("com.android.application") version "7.3.0" apply false
  // ...

  // Add the dependency for the Google services Gradle plugin
  id ("com.google.gms.google-services") version "4.4.0" apply false
}
```

Add dependencies to App

- In the module level build.gradle.kts (Module) add in google service.

```
plugins {
  id ("com.android.application")


  // Add the Google services Gradle plugin
  id ("com.google.gms.google-services")
  // ...
}
```

Add dependencies to App

- In the module level build.gradle.kts (Module) add in FireBase SDK services. Goes in dependencies

```
dependencies {
  // ...

  // Import the Firebase BoM
  implementation(platform("com.google.firebase:firebase-bom:32.3.1"))

  // When using the BoM, you don't specify versions in Firebase library dependencies

  // Add the dependency for the Firebase SDK for Google Analytics
  implementation("com.google.firebase:firebase-analytics")

  // TODO: Add the dependencies for any other Firebase products you want to use
  // See https://firebase.google.com/docs/android/setup#available-libraries
  // For example, add the dependencies for Firebase Authentication and Realtime database
  implementation("com.google.firebase:firebase-auth")
  implementation("com.google.firebase:firebase-database")
}
```

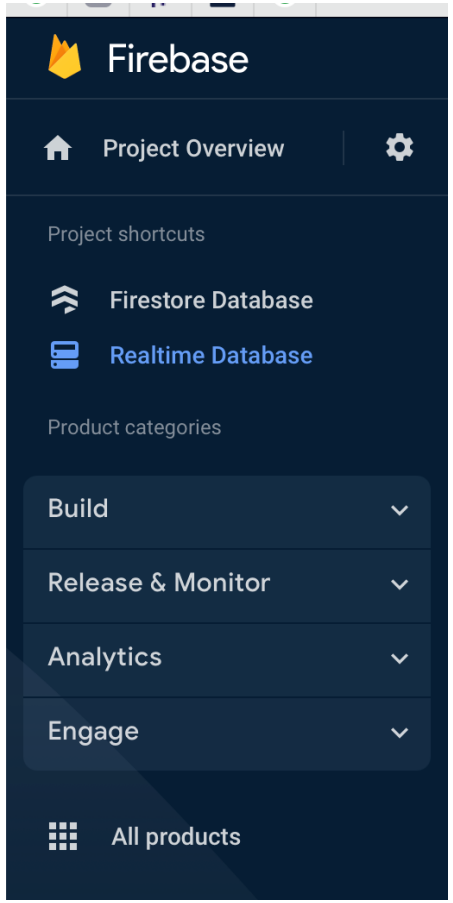
Sync

- After making changes in the build files you will need to sync to your project.

Selecting the DataBase

- From the overview console menu, we can select the kind of data base to use. We will choose RealTime:

- Click the Create DataBase button.



Questions to Answer

- Location: uscentral1 is good
- DB access rules: locked mode (we can update later)
- Enable

Questions to Answer

- Location: uscentral1 is good
- DB access rules: locked mode (we can update later)
- Enable

Create Models

- We will want to have objects that we can use to interact with the database. Create a *models* package under the java folder in your app. This is where we will create model classes.
- The model class will just define attributes and constructors.

- For our app, we will want to store movies each of which has a title (string) , director (string) , and year (integer)

Movie

```
public class Movie {
    public String title;
    public String director;
    public int year;

    public Movie (){
        // Default
    }
    public Movie(String title, String director, int year) {
        this.title = title;
        this.director = director;
        this.year = year;
    }
}
```

Saving data

- We need to get an instance of our database which will be a tree like structure. We will allow the addition of data at any point in the tree.

```
FirebaseDatabase database = FirebaseDatabase.getInstance();

DatabaseReference dbRoot = database.getReference();

// table of Movie objects and a count living under root
moviesEndPoint = dbRoot.child("Movies");
countEndPoint = dbRoot.child("Count");
```

Saving data

- I now have three places I can put my data… root, Movies and Count.
  - setValue(some value) will replace the current contents with the value we passed in.
  - If I did this to root, it would wipe out anything under it.

```
countEndpoint.setValue(7)
```

```
Movie m = new Movie(“Jaws”, “Spielberg”, 1980);
moviesEndpoint.setValue(m)
```

DB rules Example (Server Side)

- This will fail because my database rules say that all of my content is private for read and write. It is safe!
- Ideally, I will have users sign in and be authenticated, but for now we just want to work with the database directly. We will set the access to public. This exposes the database and one of the first things we will want to do is to implement authentication.
- The link shows an example of database rules that are more robust

Rules example

Saving data

- The code we have seen will keep one movie object at a time at the endpoint, but suppose I want a collection.
  - push().setValue(some value) will add in a new object to the collection with a randomly generated unique id.
  - This will be safe if multiple people try to add objects to the collection at the same time.

```
Movie m = new Movie(“Kill Bill”, “Taretino”, 1995);
moviesEndpoint.push().setValue(m)
```

Saving Maps

- We can create a hash map and use that to set objects in a collection. The mapping will be from a String to an Object. Each string will be used as a key.
- Update will add to existing or create objects as needed.

```
models.Movie m3 = new models.Movie("Ghost Busters", "Spielberg", 1980);
models.Movie m4 = new models.Movie("Resevoir Dogs", "Taratino", 1995);
HashMap<String,Movie> all = new HashMap<String, Movie>(){};
all.put("gb",m3);
all.put("rd",m4);
moviesEndPoint.update(all);
```

Saving data

Reference on saving data

Queries

- We need to have the ability to get data out of the database. Since this is not an SQL database that is not an option. We don't even have domain objects either. While we did create a Movie object which we can save, once it is in the database, I can add in new attributes.
- We will use a reference, order and limit the objects and then use a callback function to process them all.
- We will always need to iterate over the items for a list of objects.
- We can also set up call back functions at the same time to listen for changes in the data.

Queries (get all)

```
public void allMovies(View v){
   Query allQ = moviesEndPoint.orderByChild("director");
   allQ.addListenerForSingleValueEvent(new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot listSnapshot) {
            for (DataSnapshot singleSnapshot : listSnapshot.getChildren()) {
                  Log.d("DB", " change " + singleSnapshot);
                  DataSnapshot title = singleSnapshot.child("title");
                  DataSnapshot year = singleSnapshot.child("year");
                  DataSnapshot director = singleSnapshot.child("director");

                  Log.d("DB", "Callback: Movie is  " + title.getValue()
                        + " in year " + child("year").getValue()
                        + " and key " + singleSnapshot.getKey());
               }
            }
```

Queries (get all)
Snapshot

- A snapshot is a local version of the truth.
- In the previous code we had a snapshot that is listening for changes in the Movies list of objects. If there is a change, it can trigger our call back. We then iterate over the listSnapshot and have one snapshot for each object that meets our criteria.
- We use getValue on a snapshot to get the value.
- We can also get references as needed.

Listeners

We have 3 kinds of listeners.

- ChildEvent – Listen for changes on a list of objects. We have callbacks for different kinds of changes. Can be called multiple times per object
- ValueEvent - Listen for changes on a value. Can be for an object. Can be called multiple times per value.
- SingleValueEvent – We have callbacks as before, but they are removed immediately.

Watch on all

```
moviesEndPoint.orderByChild("year").addChildEventListener(
        new ChildEventListener() {
                @Override
                public void onChildAdded(@NonNull DataSnapshot snapshot,
                                         @Nullable String previousChildName) {
                Movie mv = snapshot.getValue(Movie.class);
                 Log.d("DB", "Movie added is  " + mv.title + " in year ”
                        + mv.year + " and key " + snapshot.getKey());
    TextView showMoviesTV = findViewById(R.id.showMoviesTV);
    String current = showMoviesTV.getText().toString();
    current += mv.title + "\n";
    showMoviesTV.setText(current);
  }
```

Queries (get some)

```
moviesEndPoint.orderByChild("year").limitToFirst(2).addChildEven
tListener(
new ChildEventListener() {
    @Override
    public void onChildAdded(@NonNull DataSnapshot snapshot,
                                @Nullable String previousChildName) {
        Movie mv = snapshot.getValue(Movie.class);
        Log.d("DB", "Movie is  " + mv.title + " in year ”
+ mv.year + " and key " + snapshot.getKey());
    }
});
```

Orders

- orderByChild – we give it a key and it orders the object according to the value associated with the key.
- orderByKey – Order by the keys (if using autogenerated keys for objects in a collection, they will be time ordered.)
- orderByValue - we give it an object and it orders the values in the object

- Only one order is allowed.

Limits

- limitToFirst(n) .
- limitToLast(n)
- startAt(value)
- endAt(value)
- equalTo(value)

Queries

- Reference on Retrieving Data

Updating data

- Get a **reference** to the item you want to change and use setValue() to change its value. In the following example, snapshot was one of the results from a query. We get a database reference to that object and use the tree to get to the property (“title”) that we are going to change.

```
snapshot.getRef().child("title").setValue("Fish");
```

Updating data

```
Query getSpielbergs = moviesEndPoint.orderByChild("director").equalTo("Spielberg");
getSpielbergs.addListenerForSingleValueEvent(new ValueEventListener() {
    @Override
    public void onDataChange(DataSnapshot dataSnapshot) {
        Log.d("DB", " changing all spielbergs " + dataSnapshot);
        for (DataSnapshot singleSnapshot : dataSnapshot.getChildren()) {
            DataSnapshot title =  singleSnapshot.child("title");
            DataSnapshot year = singleSnapshot.child("year");
            DataSnapshot director = singleSnapshot.child("director");
            Log.d("DB", " Movie is  " + title.getValue()
                    + " in year "+ year.getValue() + " and key " + singleSnapshot.getKey());

            director.getRef().setValue("Sealberg");
            title.getRef().setValue("Fish");
        }
    }
```

Deleting data

- If you have a reference to an object you can remove it via the removeValue() method.

```
for (DataSnapshot delSnapshot : dataSnapshot.getChildren()) {
    Log.d("DB", "remove snapshot " + delSnapshot);
    delSnapshot.getRef().removeValue();
}
```

Deleting Data

- https://firebase.google.com/docs/database/admin/save-data

Questions