

CMPS 312

Firebase Cloud Services



Dr. Abdelkarim Erradi
CSE@QU

Outline

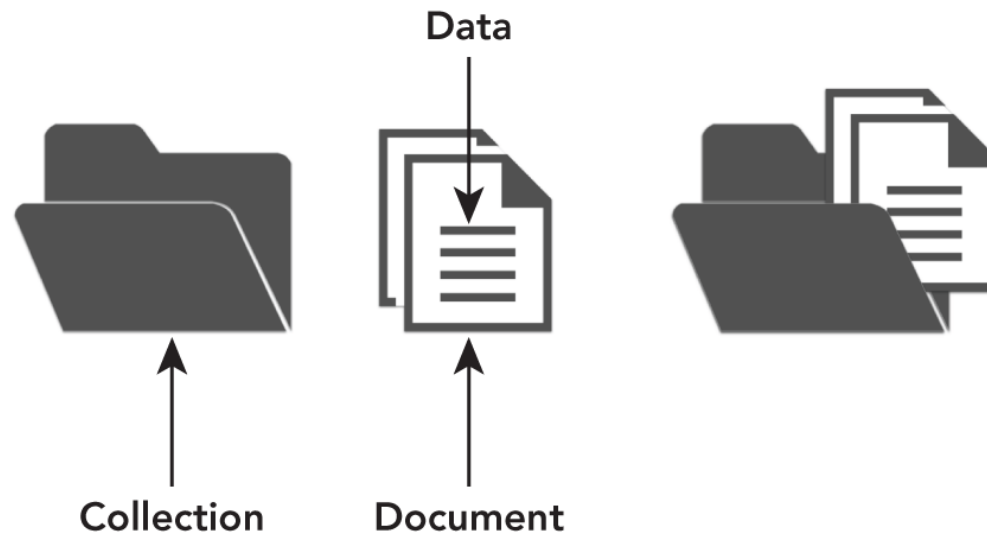
1. Firestore Data Model
2. Firestore CRUD Operations
3. Firebase Cloud Storage
4. Firebase Authentication

Firebase Cloud Services

- Firebase is a cloud platform offering many services that work together as a backend server infrastructure for mobile/web apps
- We will focus on introducing:
 - Cloud Firestore: store/query documents in collections
 - Cloud Storage: store and server files
 - Firebase Authentication: secure user's authentication using various identity provides (e.g., email/password, Google Auth)



Firestore Data Model

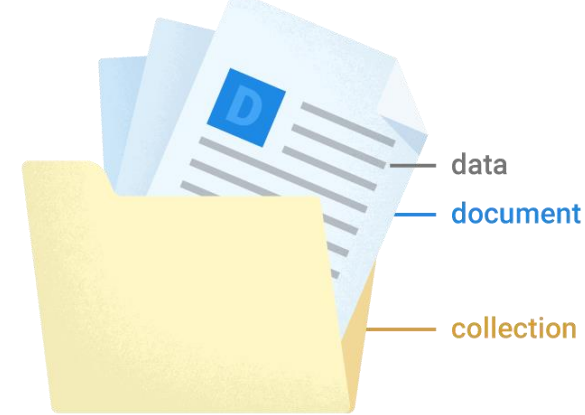




Firestore Database

- Cloud-hosted **scalable** database to manage app data
 - No need to set up or maintain backend servers
- Provides real-time updates and offline support
- Uses a **document-oriented** data model
 - You have a collections, which contain documents, which can contain sub-collections to build hierarchical data structures
- NoSQL (does not use SQL as a query language)
- Access controlled with **security rules**
- Includes a [free tier](#) (1 GiB data, 50K reads/day and 20K writes/day) then pay as you use

Data Model



- Firestore is **Document Oriented Database**
 - **Uses a document data model**: Stores data similar to JSON documents (instead of rows and columns as done in a relational database)
 - **Arrange documents in collections** (documents can vary in structure)
 - **API to query and manage documents**
- Better alternative data management solution for Mobile/Web applications compared to using a Relational Database

Document

```
{  
  "isbn" : "123",  
  "title": "Mr Bean and the Forty Thieves",  
  "category": "Fun",  
  "pages": 250  
  "authors": ["Mr Bean", "Juha Dahak"],  
  "publisher": {  
    "name": "MrBeanCo",  
    "country": "UK"  
  }  
}
```



property : value

property : array

property : map

- **Document = JSON-like object**
- **Document = set of key-value pairs**
- **Document = basic unit of data** in Firestore
 - You can only fetch a document not part of it
- Analogous to **row** in a relational database
- Size limit to **1 MB** per document
- A document can optionally point to subcollections
- A Document **cannot** point to another document

Data Types

- Cloud Firestore supports a variety of data types for values:
 - boolean, number, string,
 - geo point, binary blob, and timestamp
 - arrays, nested objects (called maps) to structure data within a document

Document

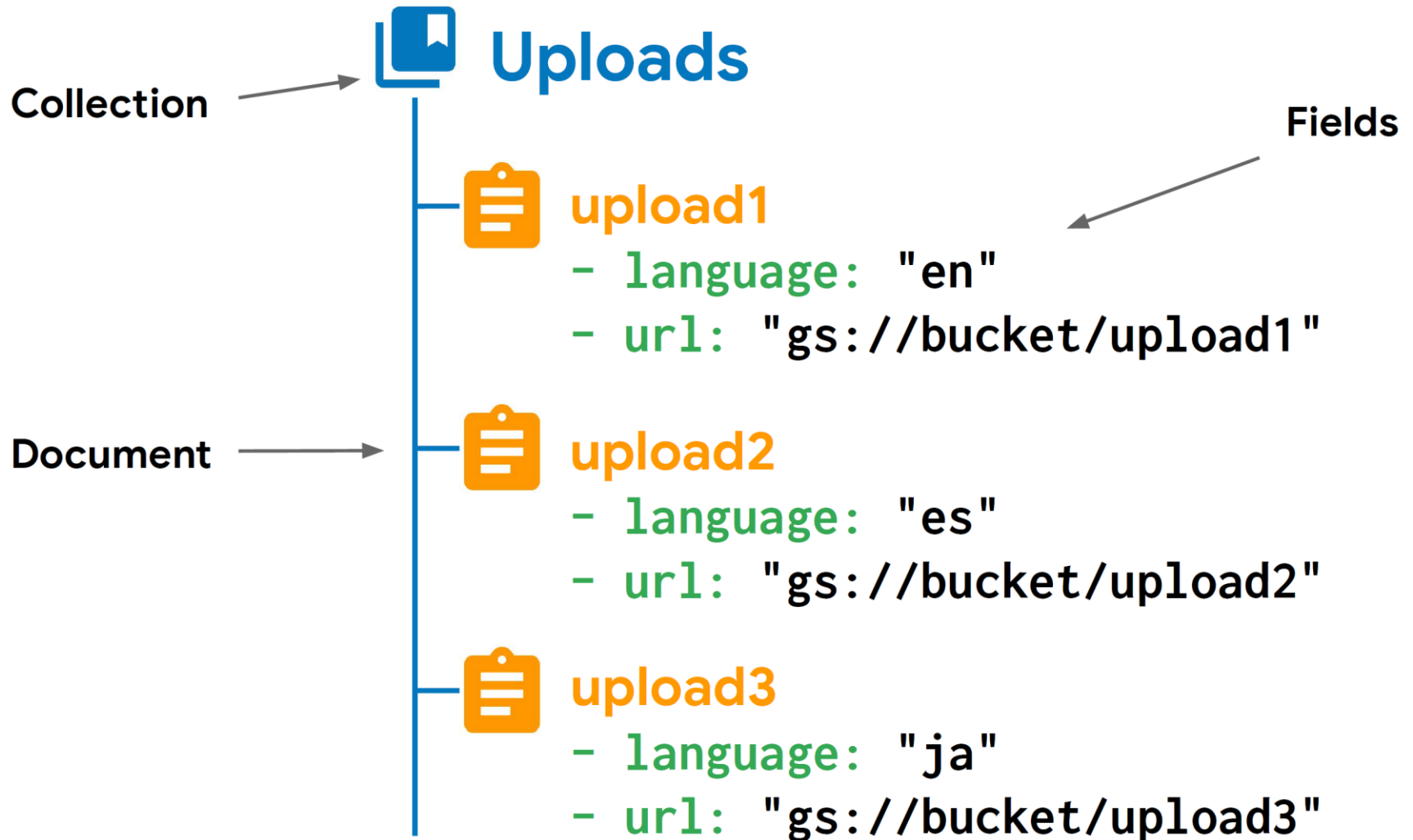
```
bird_type: "swallow"  
airspeed: 42.733  
coconut_capacity: 0.62  
isNative: false  
icon: <binary data>  
vector:  
  {x: 36.4255,  
   y: 25.1442,  
   z: 18.8816}  
distances_traveled:  
  [42, 39, 12, 42]
```


Collection



- **Collection = container** for documents
- Analogous to **table** in a relational database
- **Does not enforce** a schema
- Documents in a collection usually **have similar purpose** but they may have slightly different schema

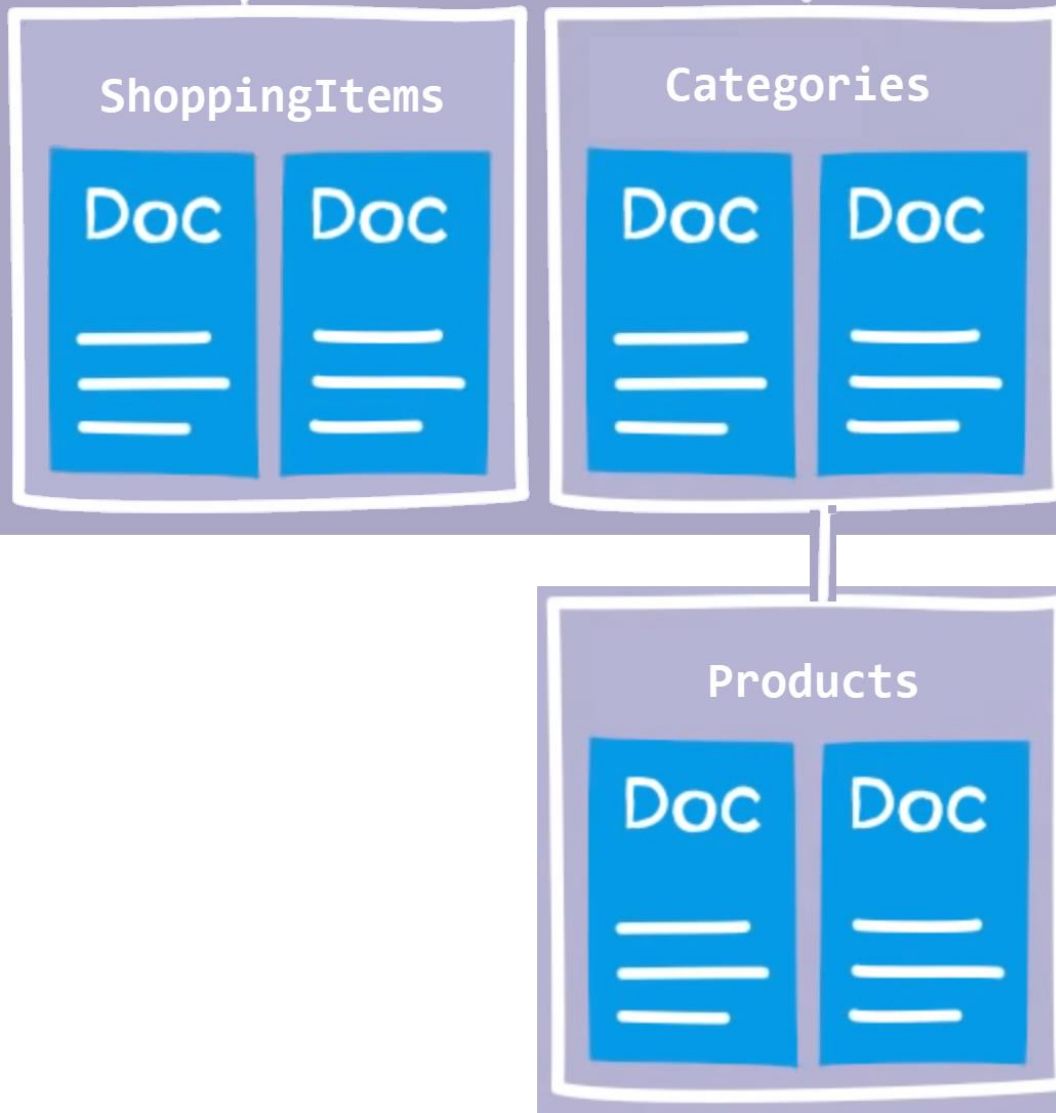
Example Collection & Documents



Firestore Root



Shopping List App



- Database with 2 **top-level** collections: **ShoppingItems** and **Categories**
- Each category document has a **Products** sub-collection

Document Identifiers

- Documents within a collection have unique identifiers
 - You can provide your own keys, such as user IDs, or
 - You can let Cloud Firestore assign a random IDs
- You do not need to "create" or "delete" collections
 - After you create the first document in a collection, the collection exists
 - If you delete all the documents in a collection, it no longer exists
- Access a document using its **collection** and its doc **Id**


```
val u1DocumentRef = db.collection("users").document("u1@test.com")
```


OR using doc path

```
val u1DocumentRef = db.document("users/u1@test.com")
```


Subcollections


- A subcollection is a collection associated with a specific document
 - E.g., A subcollection called messages for every room document in the rooms collection

 rooms

 roomA

name : "my chat room"

 messages


 message1

from : "alex"

msg : "Hello World!"

 message2

...

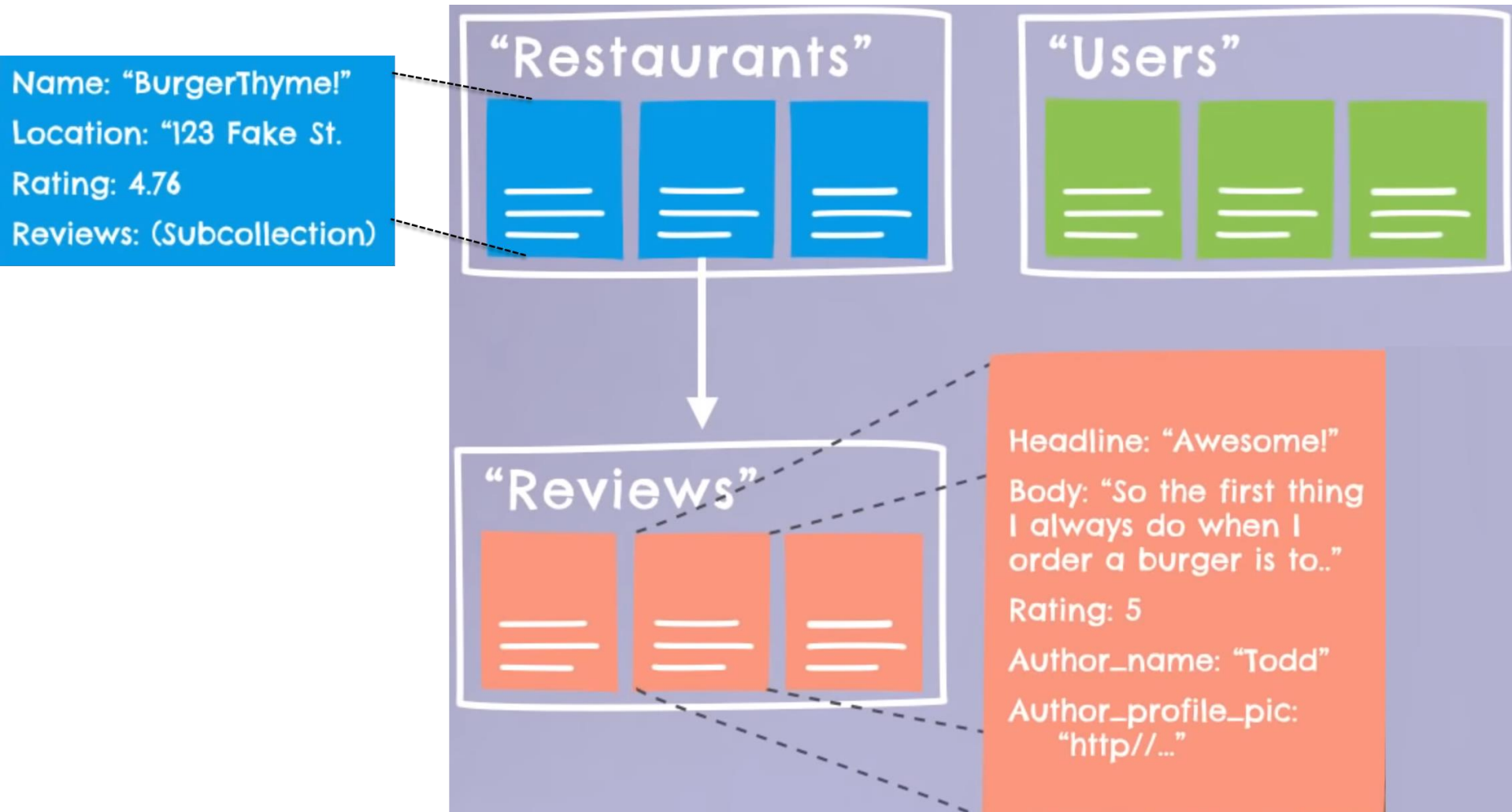
 roomB

...

- Get a reference to a message in the subcollection

```
val messageRef = db
    .collection("rooms").document("roomA")
    .collection("messages").document("message1")
```

Example Restaurant Review App



Add Firebase to your Android project

- Login to <https://console.firebase.google.com/>
- Create a **project** (give it a meaningful name)
 - to keep it simple disable Google Analytics for the project
- From Android Studio use Tools -> Firebase. Then select Firestore and

← **Firebase** > Firestore

Read and write documents with Cloud Firestore

① Connect your app to Firebase

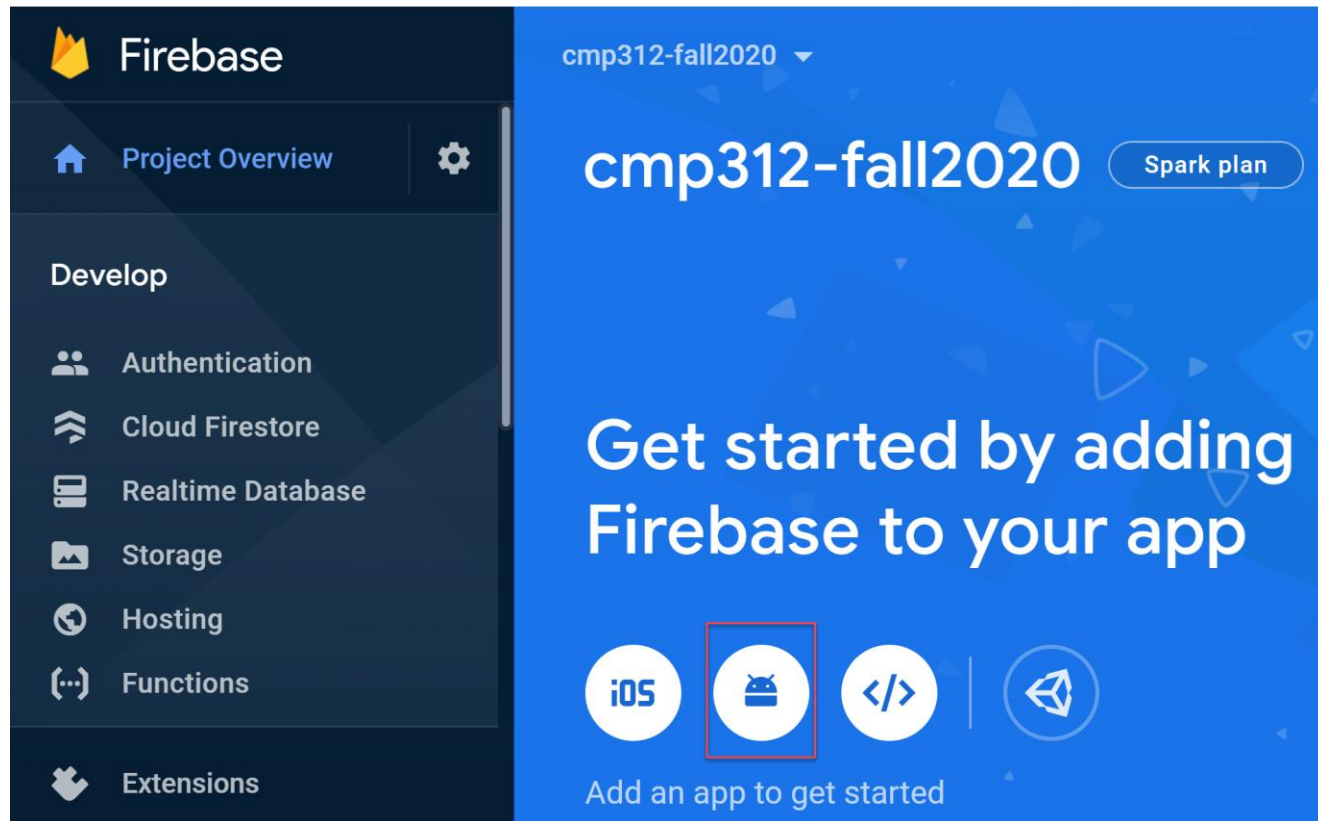
✓ Connected

② Add Cloud Firestore to your app

✓ Dependencies set up correctly

Alternative setup using Firebase console

- Select **Project Overview** and add an Android app



- Download **google-services.json** and place it under **/app** subfolder

Dependencies

- Project-level **build.gradle** (<project>/build.gradle):

```
dependencies { ....  
    // Google services  
    classpath 'com.google.gms:google-services:4.3.4'  
}
```

- App-level **build.gradle** (<project>/<app-module>/build.gradle):

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

```
dependencies { ...  
    // Declare the dependency for the Cloud Firestore Library  
    // When using the BoM, you don't specify versions in Firebase Library dependencies  
    implementation 'com.google.firebase:firebase-firestore-ktx'  
    implementation 'com.google.firebase:firebase-auth-ktx'  
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-play-services:1.2.1'  
  
    // FirebaseUI (for authentication)  
    implementation 'com.firebaseui:firebase-ui-auth:6.4.0'  
    implementation 'com.google.android.gms:play-services-auth:18.1.0'  
}
```

Firestore CRUD Operations



CREATE



READ



UPDATE



DELETE

C

R

U

D

Create Data Classes Mapped to Firebase Docs

- Normal data classes having the same structure as Firebase docs
- Must have a no-argument constructor used by Firebase deserializer
- Doc identifier can be annotated with **@DocumentId**, Firebase will auto-populate it with the doc id
- Can prevent a particular class attribute to Firestore using **@get:Exclude**

```
@get:Exclude val password: String
```


```
data class Category(  
    @DocumentId  
    val id: String = "", val name: String) {  
    // Required by Firebase deserializer other you get exception 'does not define a no-argument constructor'  
    constructor(): this("", "")  
}
```

Query – return all documents

- Using collection reference use the `.get` method to return the collection documents
 - You can sort the results using `.orderBy`
 - Use `.toObjects` to return the query results as a list of objects
 - Use the same technique to get documents from a subcollection associated with a particular document

```
suspend fun getCategories() : List<Category?> {  
    val queryResult = categoryCollectionRef.orderBy("name").get().await()  
    return queryResult.toObjects(Category::class.java)  
}  
  
suspend fun getProducts(categoryId: String) : List<Product?> {  
    val queryResult = categoryCollectionRef.document(categoryId).collection("products")  
        .orderBy("name", Query.Direction.DESENDING)  
        .get().await()  
    return queryResult.toObjects(Product::class.java)  
}
```

Query – filter using .where

- Use **.where*** to filter the documents to return from a collection
- Other filter methods  are available such as
 - whereNotEqualTo
 - whereGreaterThanOrEqualTo
 - whereIn

```
val citiesRef = db.collection("cities")  
citiesRef.whereIn("country", listOf("USA", "Japan"))
```
 - whereArrayContainsAny

```
citiesRef.whereArrayContainsAny("regions", listOf("west coast", "east coast"))
```







```
suspend fun getCategory(category: String) : Category? {  
    val queryResult = categoryCollectionRef.whereEqualTo("category", category)  
        .get().await()  
    return queryResult.firstOrNull()?.toObject(Category::class.java)  
}
```

Add a document to a Collection

- Get a collection reference

```
val collectionRef = Firebase.firestore.collection("colName")
```

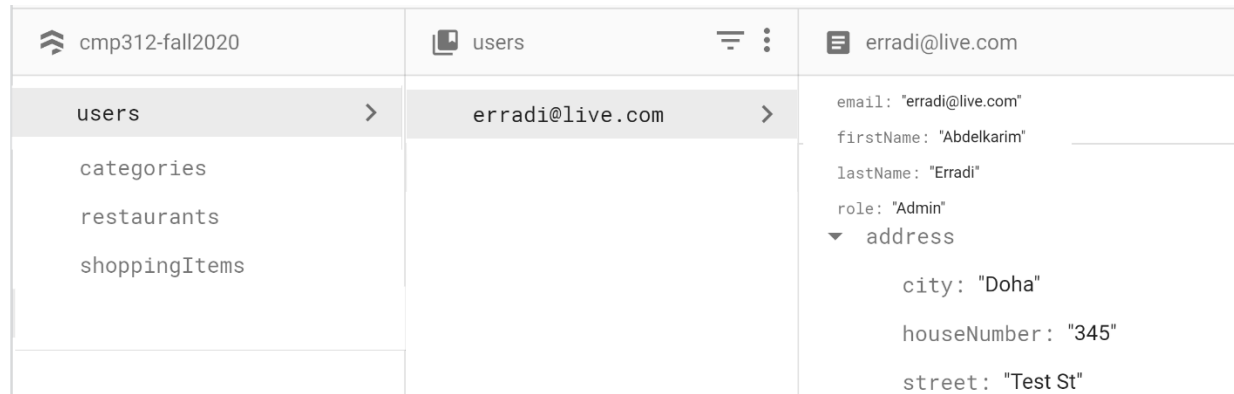
- Call **.add** method and pass the object to add the collection
 - Firebase adds the object to the collection and returns the auto-assigned **docId**

 cmp312-fall2020	 categories  	 9bbraJMpuCt7eFWpbvA6
categories >	9bbraJMpuCt7eFWpbvA6 >	name: "Fruits" 

```
val category = Category("Fruits")
val categoryCollectionRef = Firebase.firestore.collection("categories")
val queryResult = categoryCollectionRef.add(category).await()
val categoryId = queryResult.id
```

Add a document and set DocId

- First specify the desired **docId** to be assigned to the new doc
`collectionRef.document(docId)`
- Call **.set** method and pass the object to add the collection
 - Firebase adds the object to the collection and the id of the new doc is **docId** passed to **.document** method



```
suspend fun addUser(user: User) {  
    val userCollectionRef = Firebase.firestore.collection("users")  
    userCollectionRef.document(user.email).set(user).await()  
}
```

Update a document

- Use **.update** and pass the fields to update and their new values
 - You can pass them as a Map

```
suspend fun updateQuantity(itemId: String, quantity: Int) {  
    shoppingItemCollectionRef.document(itemId)  
        .update("quantity", quantity).await()  
}
```

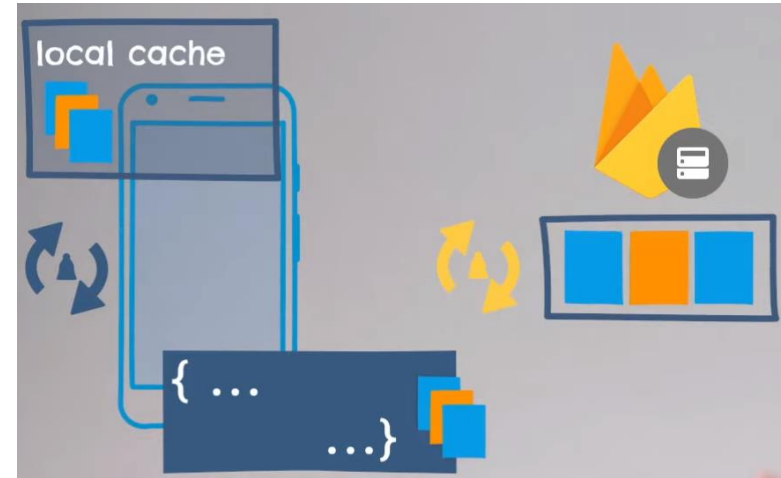

Delete a document

- Use **.delete** method to delete a document

```
suspend fun deleteItem(item: ShoppingItem) {  
    shoppingItemCollectionRef.document(item.id).delete().await()  
}
```

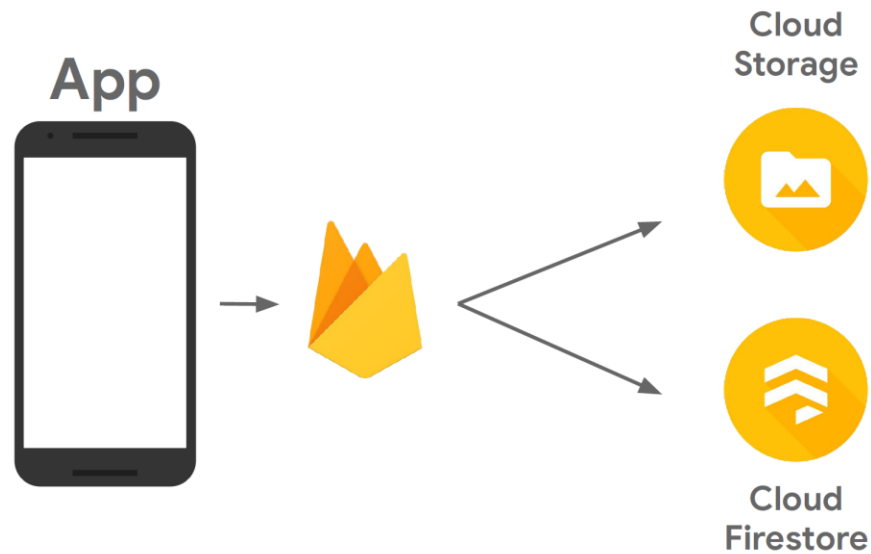
Observe collection/document changes

- Use **.addSnapshotListener** to observe the changes of a collection/document and get near **real-time updates**



```
private val _shoppingList = MutableLiveData<List<ShoppingItem?>>()
fun getShoppingListItems() {
    val query = shoppingItemCollectionRef.whereEqualTo("uid", uid)
    query.addSnapshotListener { snapshot, e ->
        if (e != null) {
            println("Shopping List Update Listener failed. ${e.message}")
            return@addSnapshotListener
        }
        _shoppingList.value = snapshot?.toObjects(ShoppingItem::class.java)
    }
}
```

Firebase Cloud Storage



Firebase Cloud Storage



- Firebase Cloud Storage
 - Store and serve files
 - Robust
 - Secure
 - Access controlled with security rules



- Dependency

implementation `'com.google.firebase:firebase-storage-ktx:19.2.0'`



Cloud Storage File Operations



▼ Upload Operations



  `putBytes(byte[]): UploadTask`

  `putFile(Uri): UploadTask`



▼ Download Operations

  `getBytes(long): Task<byte[]>`



  `getFile(Uri): FileDownloadTask`



  `getFile(File): FileDownloadTask`



▼ Delete

  `delete(): Task<Void>`

▼ List

  `list(int): Task<ListResult>`

  `list(int, String): Task<ListResult>`

  `listAll(): Task<ListResult>`

List

- List files in particular subfolder

```
val images = storageRef.child("images/").listAll().await()
val imageUrls = mutableListOf<String>()
for(image in images.items) {
    val url = image.downloadUrl.await()
    imageUrls.add(url.toString())
}
```

Add

```
storageRef.child("images/$filename").putFile(it).await()
```

Delete

```
storageRef.child("images/$filename").delete().await()
```

Download

```
val maxDownloadSize = 5L * 1024 * 1024

val bytes = storageRef.child("images/$filename").getBytes(maxDownloadSize).await()
val bmp = BitmapFactory.decodeByteArray(bytes, 0, bytes.size)
withContext(Dispatchers.Main) {
    ivImage.setImageBitmap(bmp)
}
```


Firestore Authentication



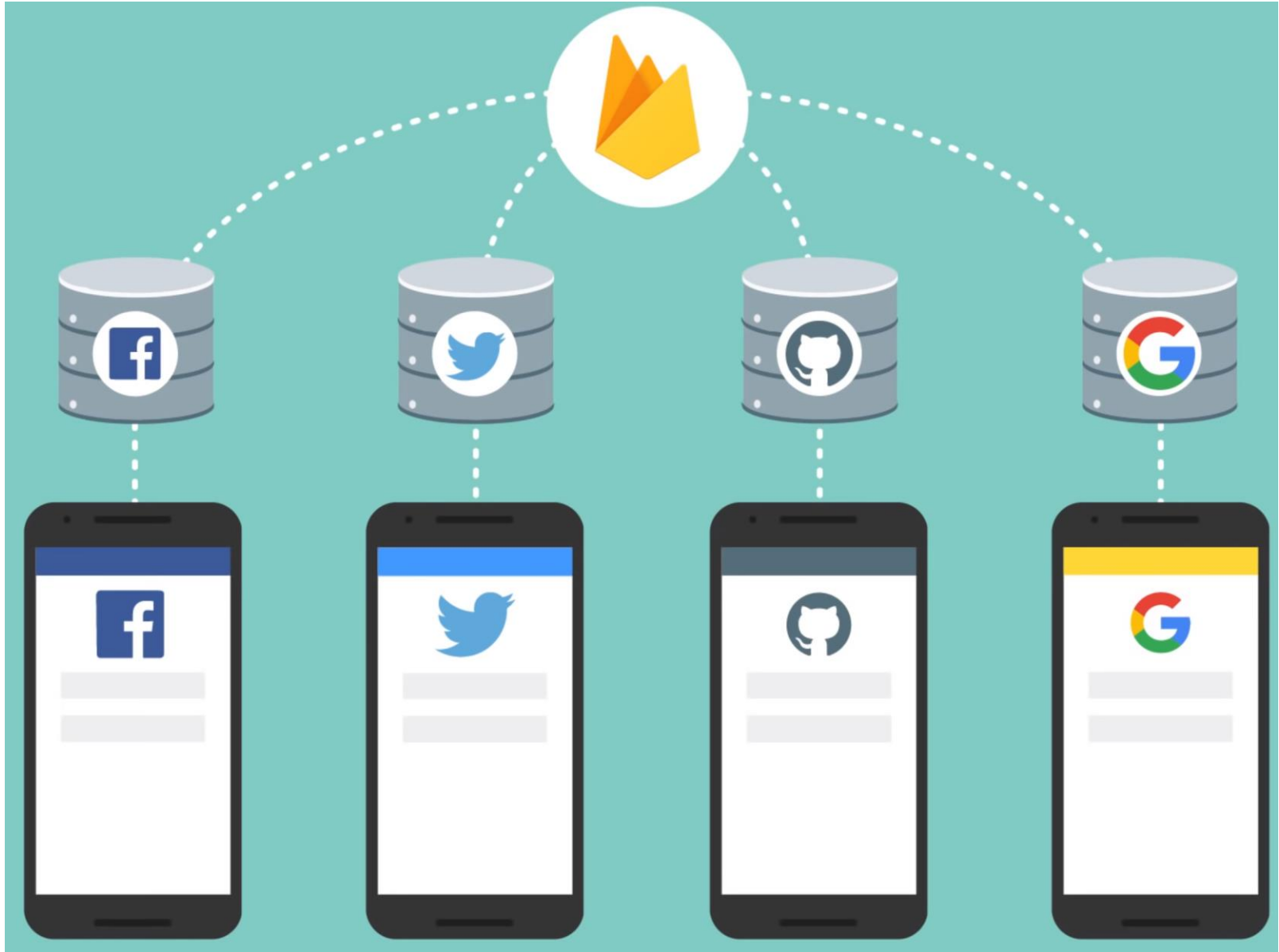


Firebase Authentication

- **Authentication** = **Identity verification**:
 - Verify the identity of the user given the credentials received
 - Making sure the user is who he claims to be
- Every user gets a unique ID
- Restrict who can read and write what data



Multiple Identity Providers can be used for Authentication

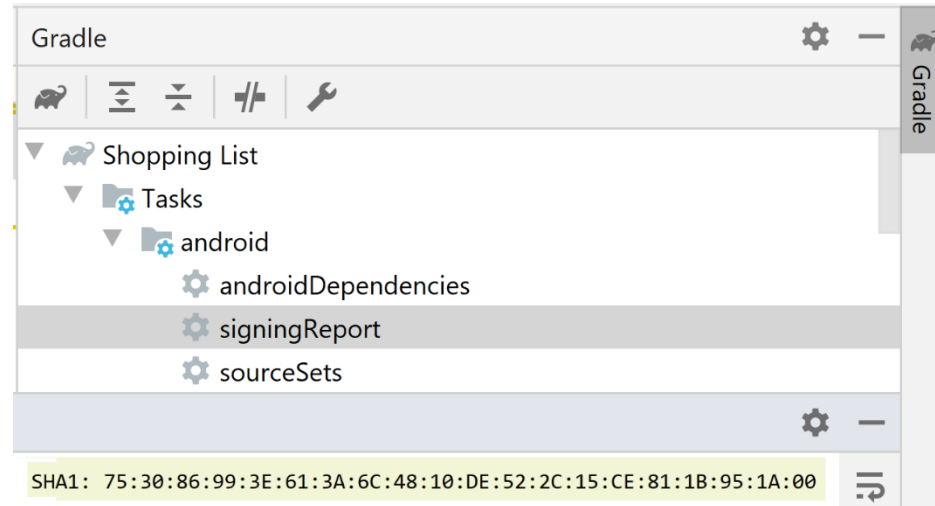


FirebaseUI Auth

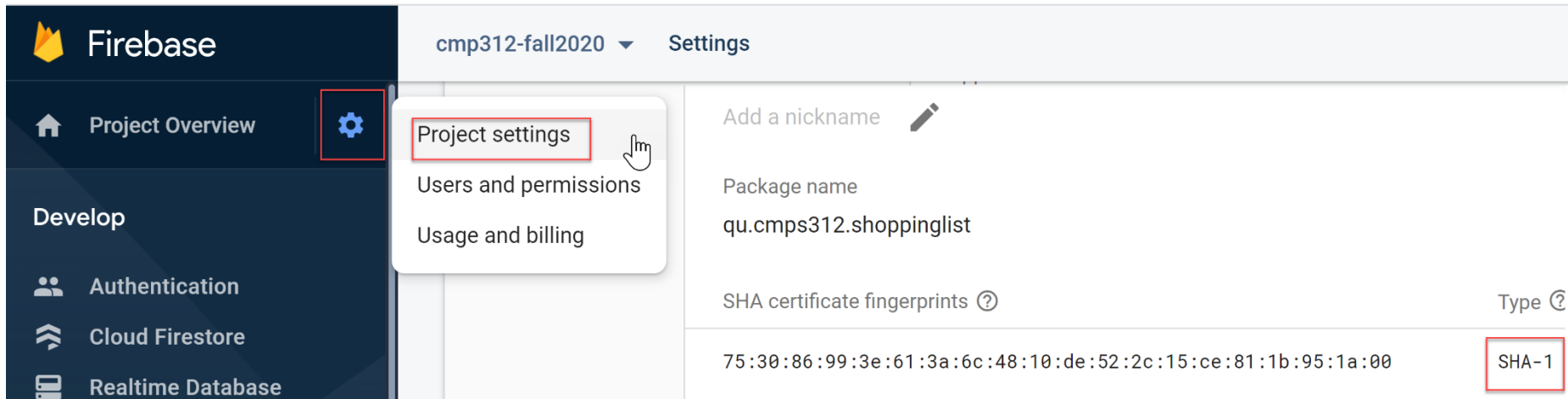
- [FirebaseUI](#) Auth is a library built on top of the Firebase Authentication SDK that provides authentication UI that can be easily integrated with any app
- Supports Multiple Auth Providers - sign-in flows for email/password, email link, phone authentication, Google Sign-In, Facebook Login, Twitter Login, and GitHub Login.

Configure Firebase Auth to use Google Auth

- Select Gradle in android studio from right panel
 - Double-click Tasks -> android-> **signingReport**

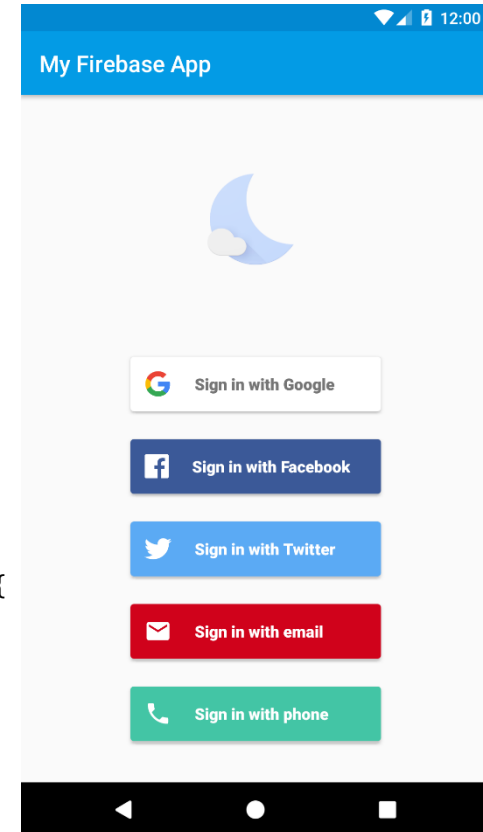


- Copy the generated SHA-1 to your Android app settings on Firebase console



Sign in using FirebaseUI Auth

```
private fun startSignIn() {  
    // You can add more providers such as Facebook, Twitter, Github, etc.  
    val providers = listOf(  
        AuthUI.IdpConfig.EmailBuilder().build(),  
        AuthUI.IdpConfig.GoogleBuilder().build()  
    )  
    // Sign in with FirebaseUI  
    val intent = AuthUI.getInstance()  
        .createSignInIntentBuilder()  
        .setAvailableProviders(providers)  
        .setLogo(R.drawable.img_shopping_list_logo)  
        .setIsSmartLockEnabled(false)  
        .build()  
    startActivityForResult(intent, RC_SIGN_IN)  
}  
  
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {  
    super.onActivityResult(requestCode, resultCode, data)  
  
    if (requestCode == RC_SIGN_IN) {  
        val response = IdpResponse.fromResultIntent(data)  
  
        if (resultCode == Activity.RESULT_OK) {  
            // Successfully signed in  
            val user = Firebase.auth.currentUser  
        }  
    }  
}
```



Sign up

- Sign up and the user details to Firebase authentication

```
suspend fun signUp(user: User) : User? = withContext(Dispatchers.IO) {  
    val authResult = Firebase.auth  
        .createUserWithEmailAndPassword(user.email, user.password).await()  
  
    authResult?.user?.let {  
        val userProfileChangeRequest = userProfileChangeRequest {  
            displayName = "${user.firstName} ${user.lastName}"  
            photoUri = Uri.parse("http://test.com/spongebob.png")  
        }  
        // Add displayName and photoUri to the user  
        // Unfortunately it does not allow adding custom attribute such as role  
        it.updateProfile(userProfileChangeRequest).await()  
    }  
}
```

Sign in

- Sign in using Firebase authentication

```
val authResult = Firebase.auth.signInWithEmailAndPassword(email, password).await()  
println(">> Debug: signIn.authResult : ${authResult.user?.uid}")
```


Sign out

- Sign out from Firebase auth

```
Firebase.auth.signOut()
```

- Anywhere in the app you can access the details of current user

```
Firebase.auth.currentUser
```

- Observe authentication state change

```
Firebase.auth.addAuthStateListener {  
    println("${it.currentUser?.email}")  
}
```

Summary

- **Cloud Firestore** database store/query app's data
 - Data model consists of collections to store documents that contain data as a key-value pair similar to JSON
- **Firebase Authentication** provides built-in backend services that are accessible from the client's SDK to ease user authentication
 - email/password authentication allows users to register and log in to the app
- **Firebase Cloud Storage** is used to store and serve files

Resources

- Cloud Firestore
 - <https://firebase.google.com/docs/firestore/>
- Get to know Cloud Firestore
 - <https://www.youtube.com/playlist?list=PLI-K7zZEsYLIuG5MCVEzXAQ7ACZBCuZgZ>
- Firestore codelab
 - <https://codelabs.developers.google.com/codelabs/firestore-android>
- Firebase Auth
 - <https://firebase.google.com/docs/auth/android/start>