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

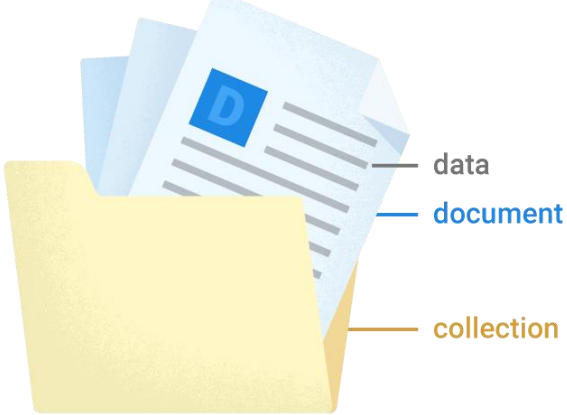
Firestore Data Model

Back

FireStore Database

- Cloud-hosted **scalable** database to manage app data
- 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 as 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** object
- **Document = set of key-value pairs**
- **Document = basic unit of data** in Firestore
- Analogous to **row** in a relational database
- Size limit to **1 MB** per 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

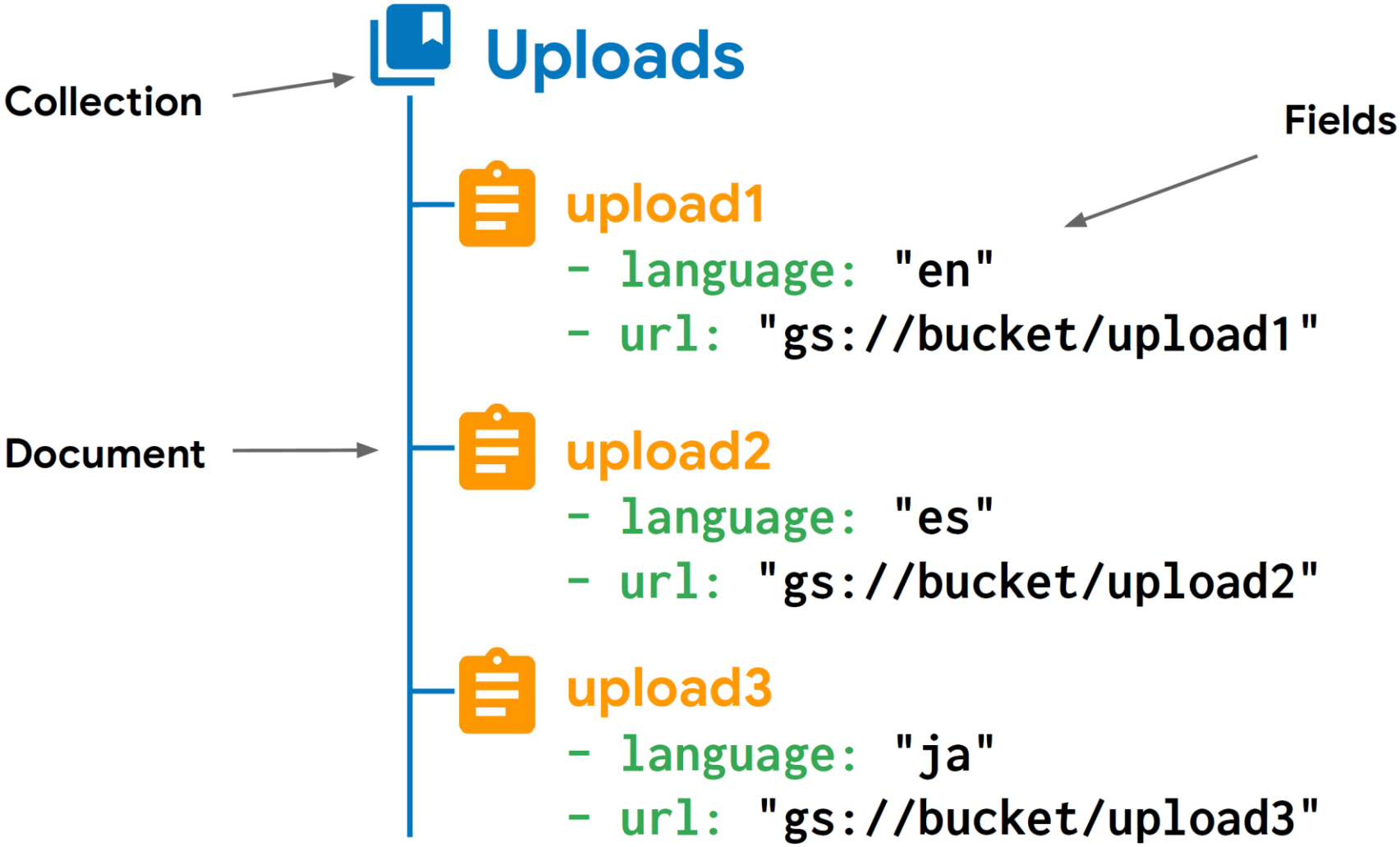
```
bird_type: "swallow"
airspeed: 42.733
coconut_capacity: 0.62
isNative: false
icon: <binary data>
vector:
    {x: 36.4255,
     y: 25.1442,
     z: 18.8816}
distances_traveled:
    [42, 39, 12, 42]
```

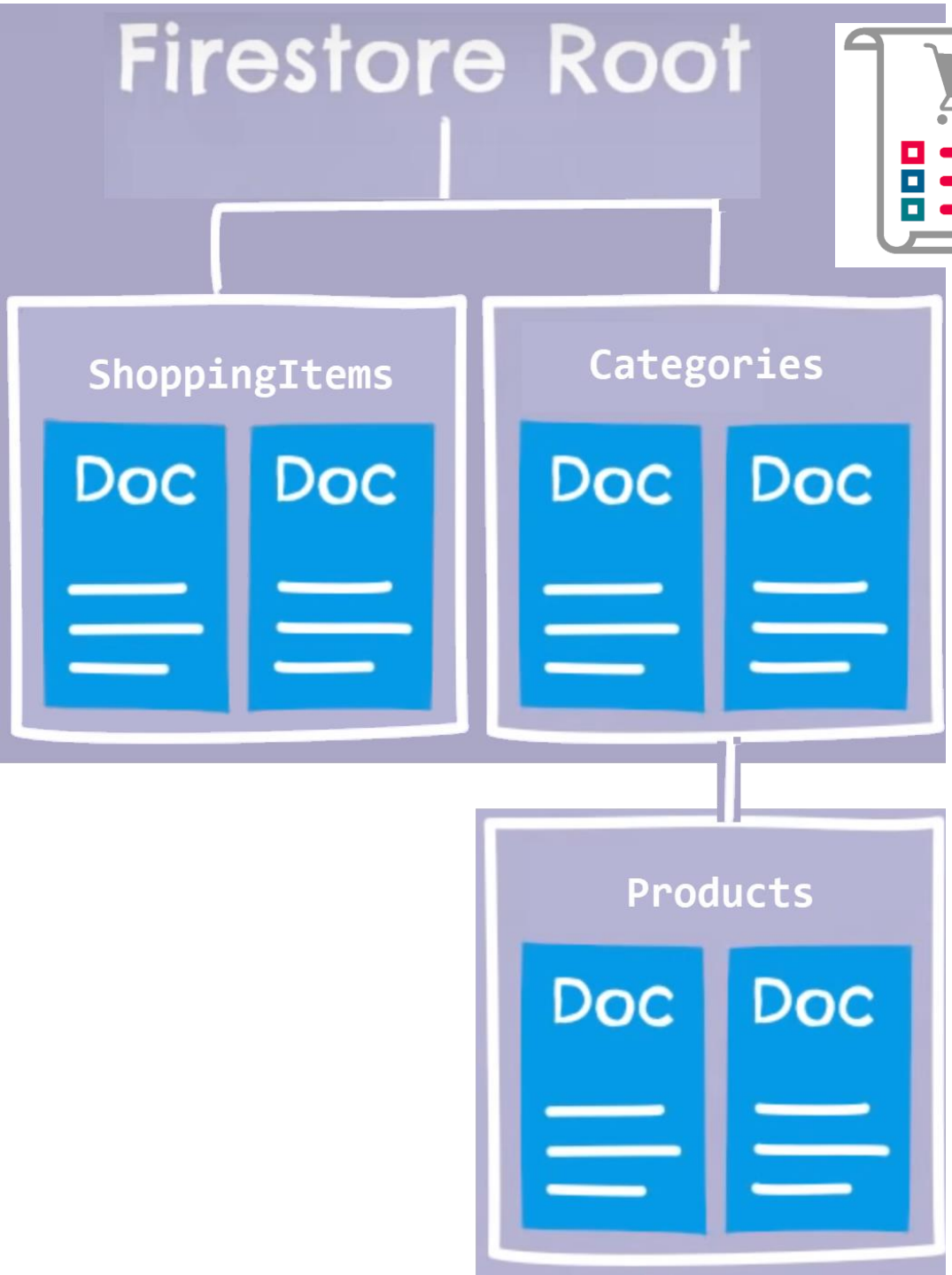
Collection

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

- **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





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

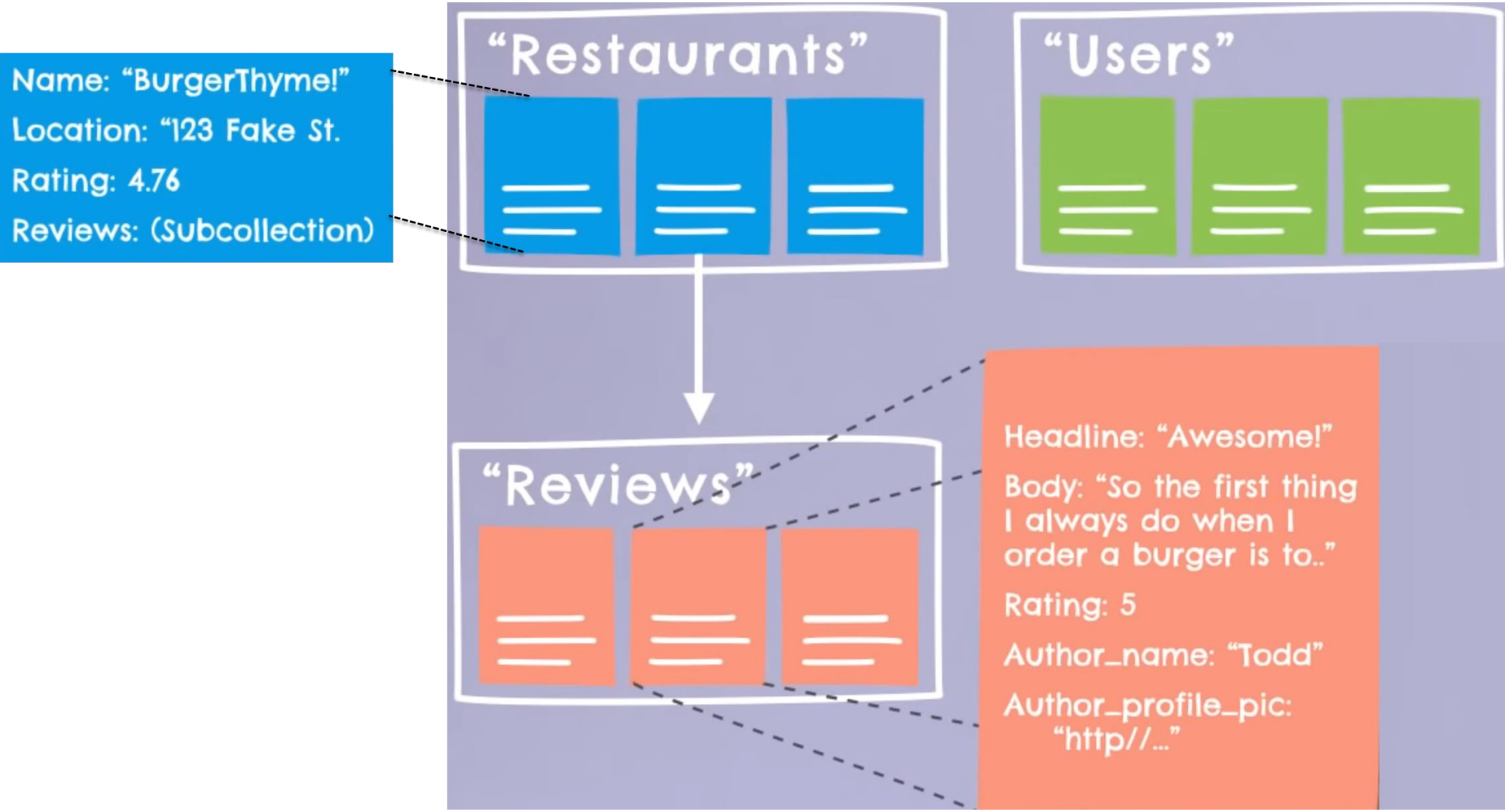
- A subcollection is a collection associated with a specific document
  - E.g., A subcollection called messages for every room document in the rooms collection

```
rooms
    roomA
    name : "my chat room"
        messages
            message1
            from : "alex"
            msg : "Hello World!"
            message2
            ...
    roomB
    ...
```

- Get a reference to a message in the subcollection

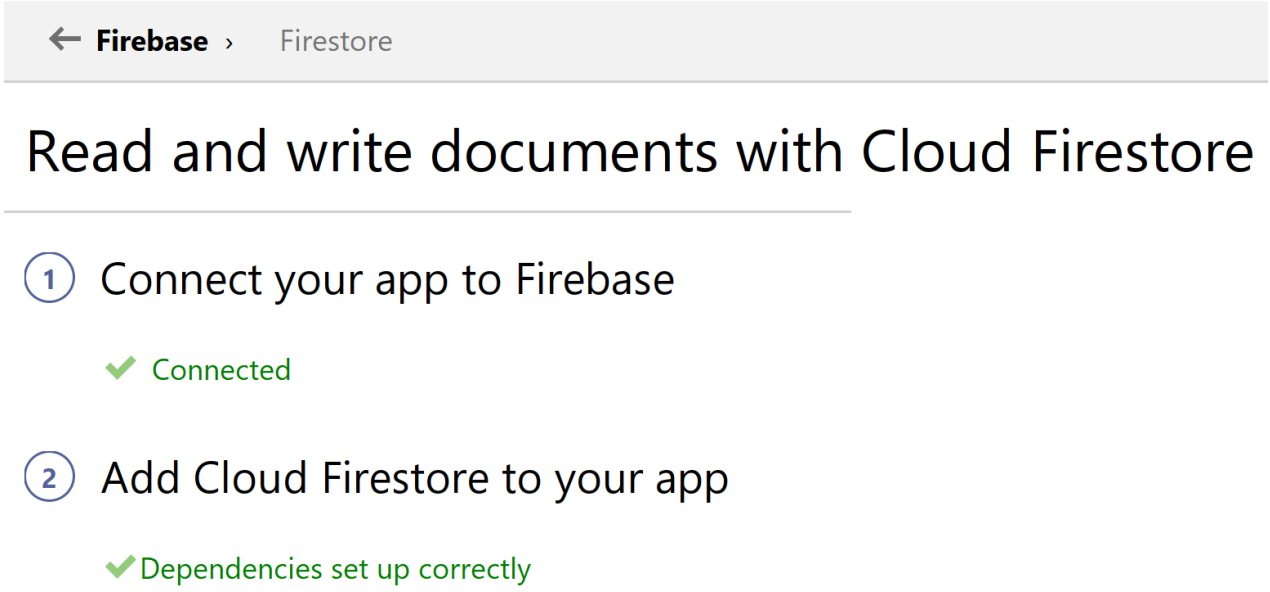
```
val messageRef = db
     .collection("rooms").document("roomA")
     .collection("messages").document("message1")
```

Example Restaurant Review App



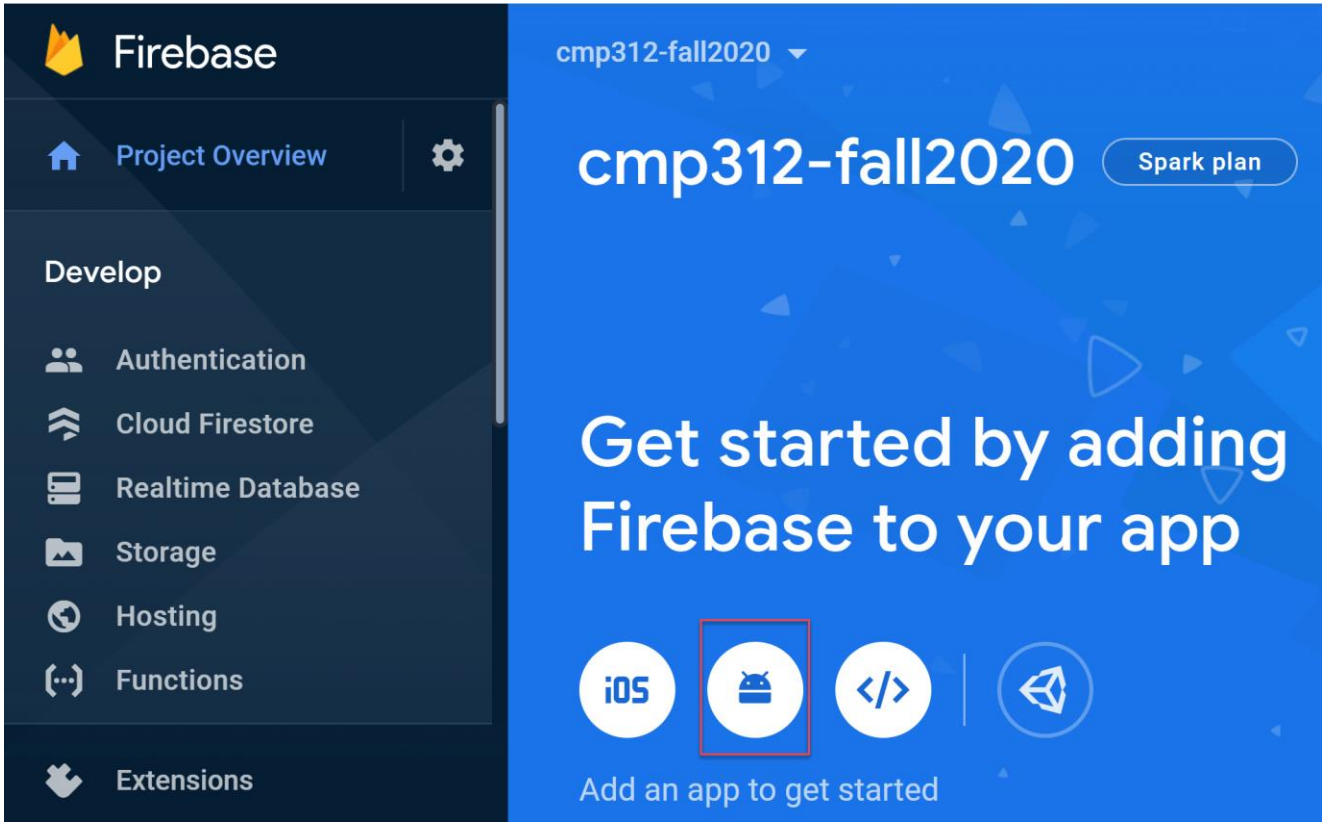
Firebase Cloud Services Setup

- 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



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'
}
```

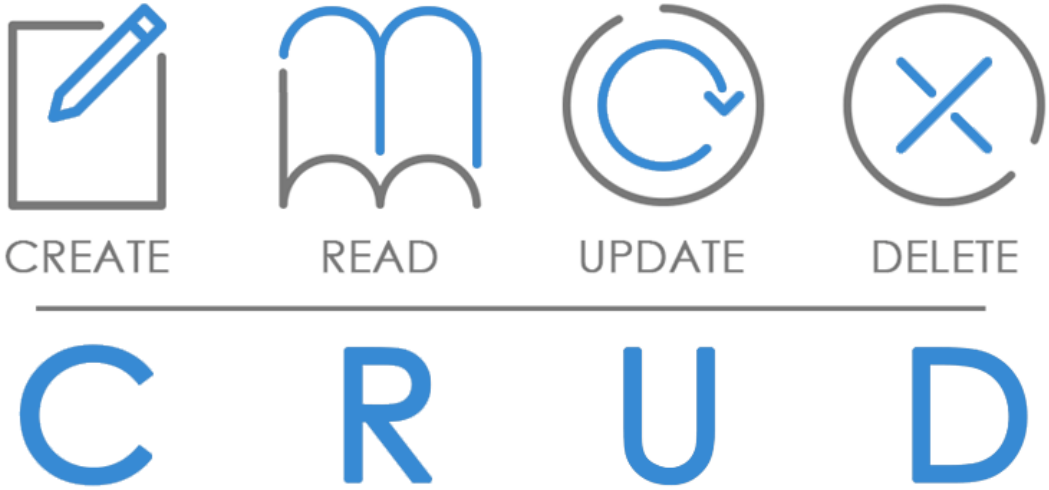
- App-level **build.gradle** (<project>/<app-module>/build.gradle):

```
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



Back

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-assign the doc id to the class property having this annotation
- 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.DESCENDING)
                                            .get().await()
    return queryResult.toObjects(Product::class.java)
}
```

Query – filer 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"))
```

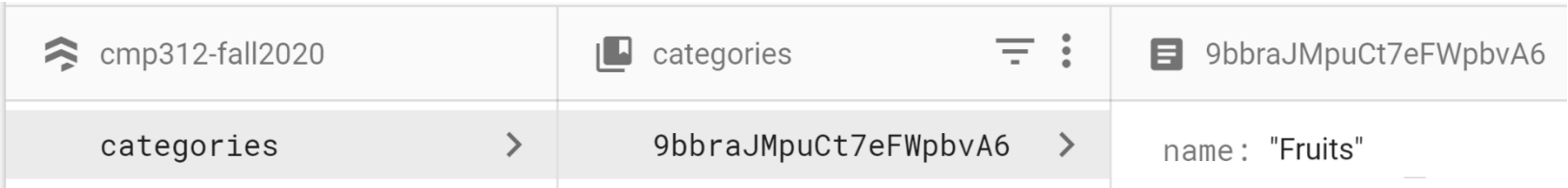
```
suspend fun getCategory(category: String) : Category? {
    val queryResult = categoryCollectionRef.whereEqualTo("category", category)
                                            .get().await()
    return queryResult.firstOrNull()?.toObject(Category::class.java)
}
```

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`**



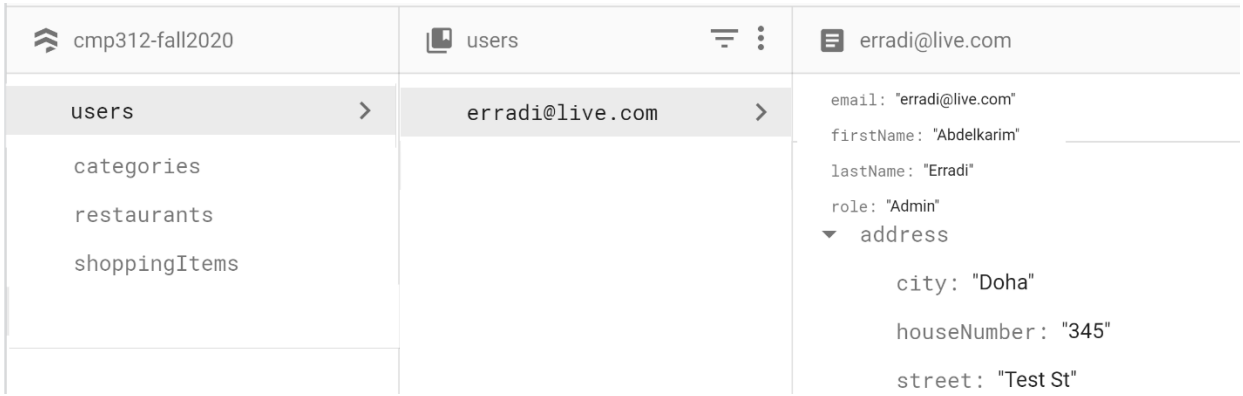

```
val category = Category("Fruits")
val categoryCollectionRef = Firebase.firestore.collection("categories")
val queryResult = categoryCollectionRef.add(category).await()
val categoryId = queryResult.id
```

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

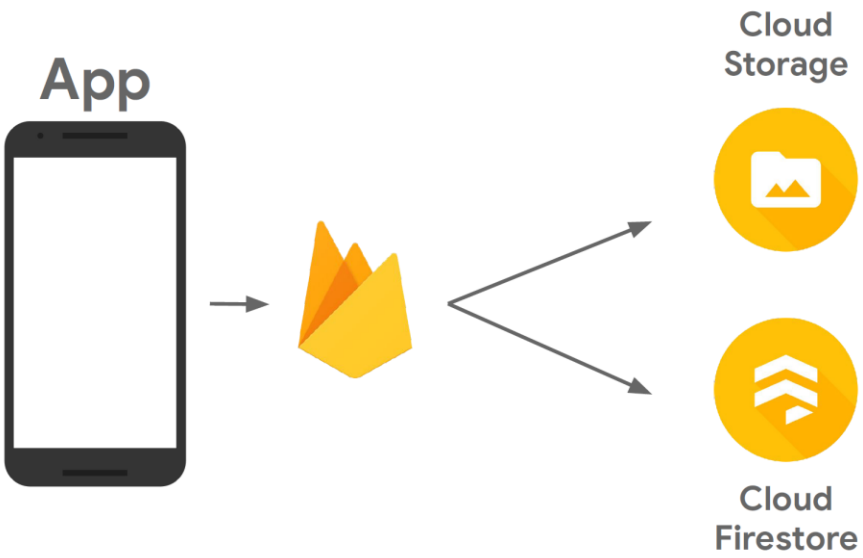
- Use `.addSnapshotListener` to observe the changes of a collection/document

```
private val _shoppingList = MutableLiveData<List<ShoppingItem?>>()
fun getShoppingListItems() {
    shoppingListUpdateListener?.remove()

    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



Back

Firebase Cloud Storage

- Firebase Cloud Storage
  - Uploads and downloads direct from app
  - Robust
  - Secure
  - Access controlled with security rules

Upload file to Cloud Storage

Firebase Authentication

Back

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

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.

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

- ToDo

Resources

- Cloud Firestore
  - https://firebase.google.com/docs/firestore/
- Get to know Cloud Firestore
  - https://www.youtube.com/playlist?list=PLl-K7zZEsYLluG5MCVEzXAQ7ACZBCuZgZ
- Firestore codelab
  - https://codelabs.developers.google.com/codelabs/firestore-android