

CMPS 312

Firebase Cloud Services



Firestore Database



Authentication



Storage




Dr. Abdelkarim Erradi

CSE@QU

Outline

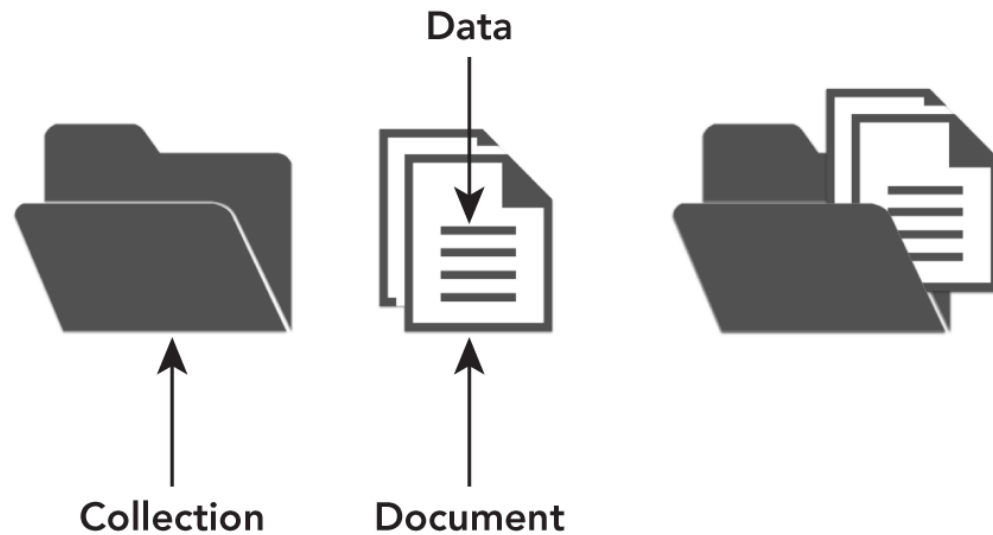
1. Firestore Data Model
2. Firestore CRUD Operations
3. Firebase Storage
4. Access Image Gallery and Camera
5. Firebase Authentication

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:
 -  **Firestore**: store/query documents in collections
 -  **Storage**: store and retrieve files
 -  **Firebase Authentication**: secure user's authentication using various identity providers (e.g., email/password, Google Auth)



Firestore Data Model

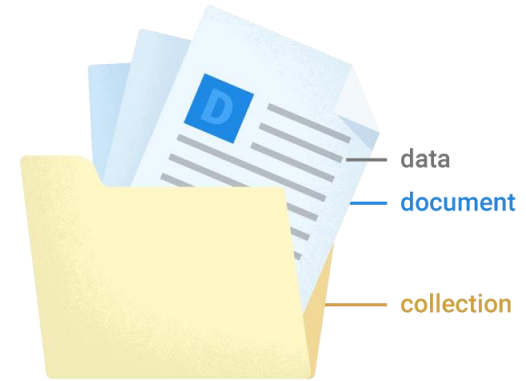




Firestore Database

- 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 GB 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 a complex object (e.g., address) within a document

Document

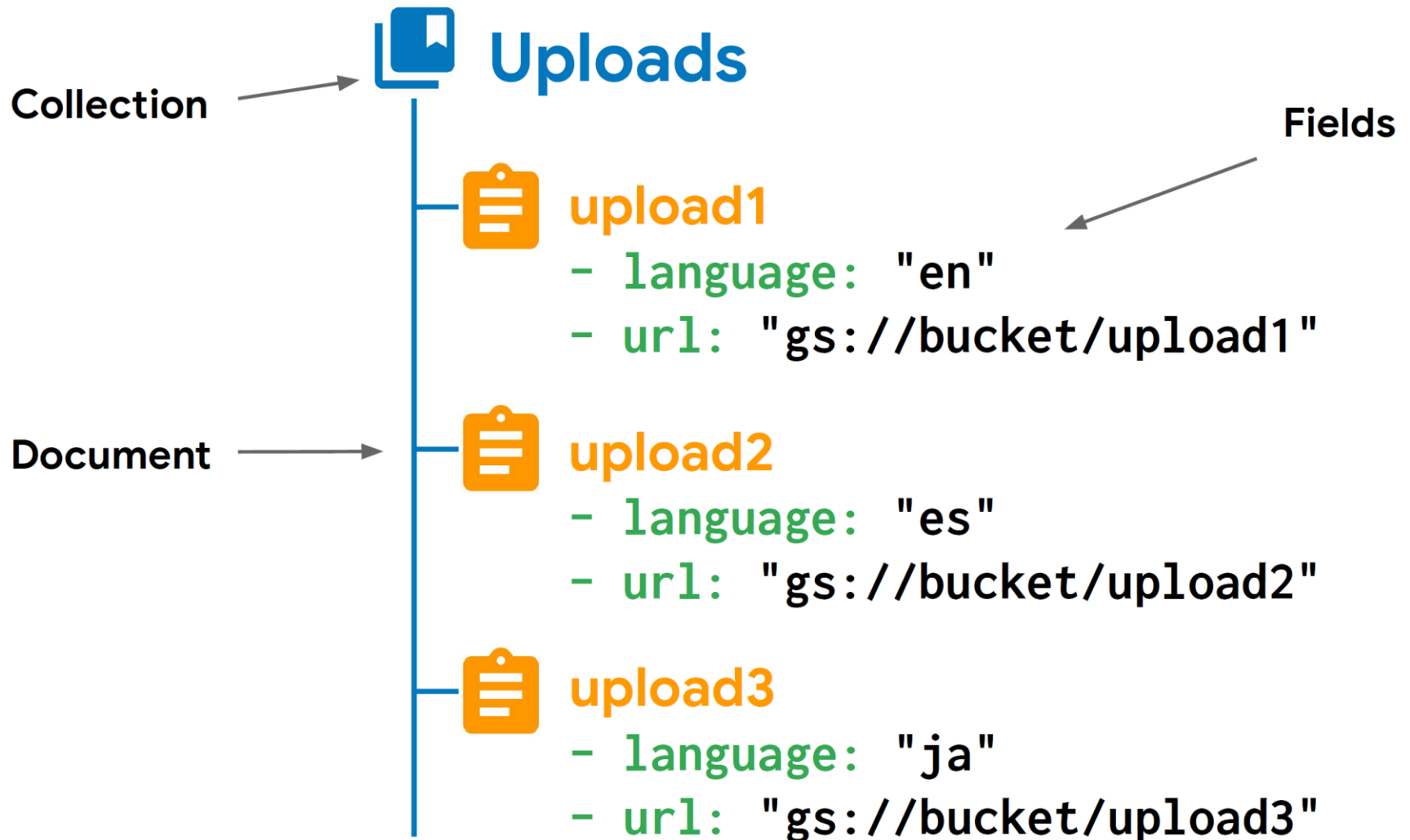
```
bird_type: "swallow"
airspeed: 42.733
coconut_capacity: 0.62
isNative: false
icon: <binary data>
vector:
  {x: 36.4255,
   y: 25.1442,
   z: 18.8816}
distances_traveled:
  [42, 39, 12, 42]
```


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
- Cannot contain other collections

Example Collection & Documents



Firestore Root



Shopping List App

ShoppingItems



Categories



Products



- 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
 - A collection creates creating after you create the first document in a collection
 - A collection is deleted when you delete all the documents in a collection
- Access a document using its **collection** and its doc **Id**
 - `Firebase.firestore.collection(path) => CollectionReference`
 - `Firebase.firestore.document(path) => DocumentReference`


```
val u1DocRef = Firebase.firestore.collection("users").document("u1@test.com")
```

OR using doc path `val u1DocRef = Firebase.firestore.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

...

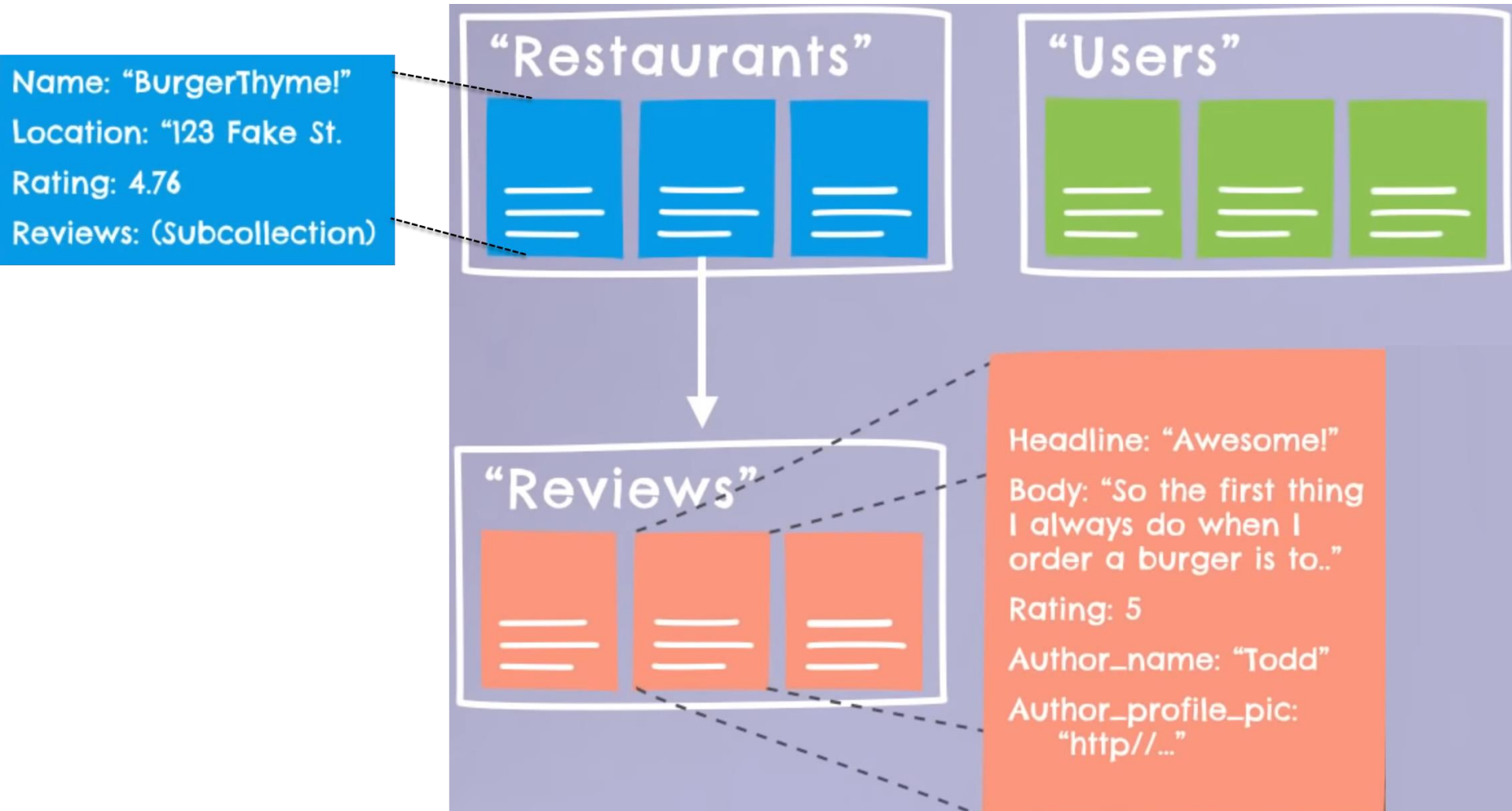
 roomB

...

- Get a reference to a message in the subcollection

```
val message1DocRef = Firestore.firestore  
    .collection("rooms").document("roomA")  
    .collection("messages").document("message1")
```

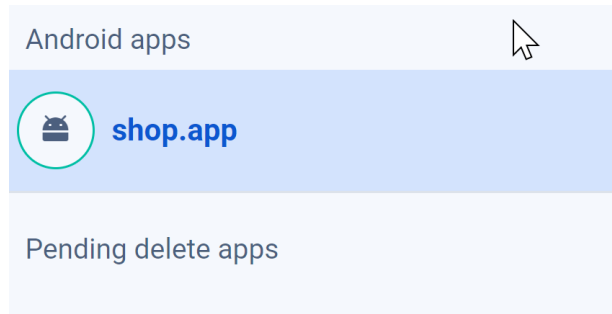
Example Restaurant Review App



Source: https://www.youtube.com/watch?v=v_hR4K4auoQ

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



SDK setup and configuration

Need to reconfigure the Firebase SDKs for your app? Revisit the download the configuration file containing keys and identifiers

 [See SDK instructions](#)

 [google-services.json](#)

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

Dependencies

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

```
plugins { ...  
    id("com.google.gms.google-services") version "4.3.14" apply false  
}
```

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

```
plugins { ...  
    id("com.google.gms.google-services")  
}
```

```
dependencies { ...  
    // Import the BoM for the Firebase platform  
    implementation(platform("com.google.firebase:firebase-bom:32.6.0"))  
  
    // 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:1.7.3")  
  
    // Firebase Authentication  
    implementation("com.google.android.gms:play-services-auth:20.7.0")  
    // Firebase storage  
    implementation("com.google.firebase:firebase-storage-ktx:20.3.0")  
}
```




Firestore CRUD Operations



CREATE

C



READ

R



UPDATE

U



DELETE

D

Create Data Classes Mapped to Firestore 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**

```
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("", "")  
}
```

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


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

```
val categoryCollectionRef = Firebase.firestore.collection("categories")
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(categoryName: String) : Category? {  
    val queryResult = categoryCollectionRef.whereEqualTo("name", categoryName)  
                                .get().await()  
    return queryResult.firstOrNull()?.toObject(Category::class.java)  
}
```

and / or filter condition

- Filter condition connected with **and**

```
citiesRef.whereEqualTo("state", "CO").whereEqualTo("name", "Denver")
citiesRef.whereEqualTo("state", "CA").whereLessThan("population", 1000000)
```

- Filter condition connected with **or**

```
val query = citiesRef.where(Filter.or(
    Filter.equalTo("capital", true),
    Filter.greaterThanOrEqualTo("population", 1000000)
))
```







```
val query = citiesRef.where(Filter.and(
    Filter.equalTo("state", "CA"),
    Filter.or(
        Filter.equalTo("capital", true),
        Filter.greaterThanOrEqualTo("population", 1000000)
    )
))
```

Add a document to a Collection

- Get a collection reference

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

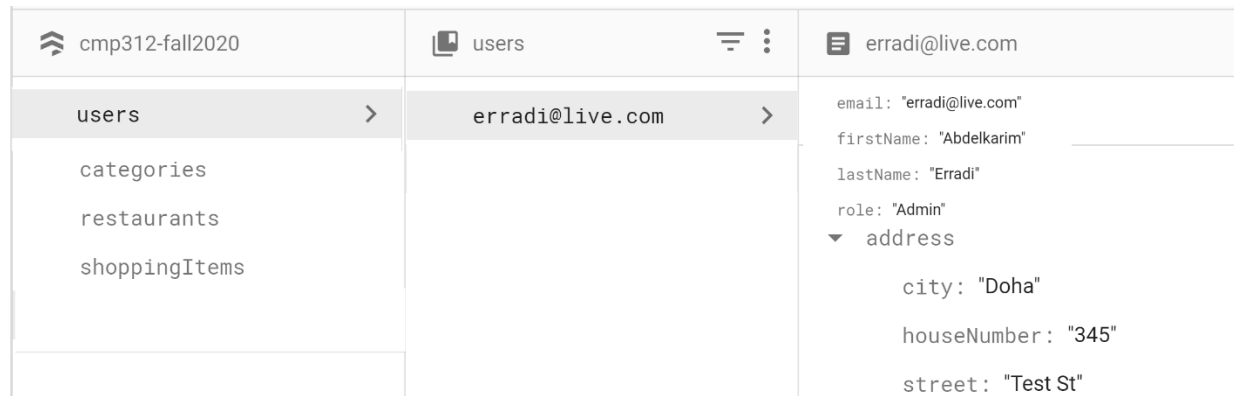
- Call **.add** method and pass the object to add the collection
 - Firebase adds the object to the collection and returns the auto-assigned **docId**

 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()  
}
```

```
suspend fun updateItem(item: ShoppingItem) {  
    shoppingItemCollectionRef.document(item.id).set(item).await()  
}
```


Delete a document

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

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

Subscribing to collection/document Realtime Updates

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



```
fun observeShoppingListItems() : Flow<List<ShoppingItem>> = callbackFlow {  
    val uid = Firebase.auth.currentUser?.uid  
    val query = shoppingItemCollectionRef.whereEqualTo("uid", uid)  
    val snapShotListener = query.addSnapshotListener { items, error ->  
        if (error != null) {  
            println("Shopping List Update Listener failed. ${error.message}")  
            return@addSnapshotListener  
        }  
        val itemObjects = items?.toObjects(ShoppingItem::class.java)?  
            .toList().orEmpty()  
        trySend(itemObjects)  
    }  
    awaitClose { snapShotListener.remove() }}
```

Watch: <https://www.youtube.com/watch?v=3aoxOtMM2rc>

Securing Data

- Cloud Firestore **Security Rules** consist of:
 - **match statements**, which identify documents in the database, and
 - **allow expressions**, which control access to those documents

```
// Allow read/write access on all documents to any user signed in to the app
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if request.auth.uid != null;
    }
  }
}
```

Firebase Storage



Firebase Storage

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

- Dependency



implementation `'com.google.firebase:firebase-storage-ktx:20.0.0'`



- Firebase Cloud Storage reference

```
val storageRef = Firebase.storage.reference
```



Firestore 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

- Get URLs of 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(filePath).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)
}
```

Access Image Gallery and Camera



rememberLauncherForActivityResult

- **rememberLauncherForActivityResult** can be used to **launch** an activity from another app (e.g., launch take photo activity from the camera app), then **provide** a callback to **handle the result** once it is dispatched by Android OS
 - **rememberLauncherForActivityResult** takes an [ActivityResultContract](#) and a [Callback](#) and returns an [ActivityResultLauncher](#) which is used to launch the desired activity
 - Android offers many built-in activity result contracts such as TakePicture, TakePicturePreview
 - e.g., Your app can start the camera app and receive the captured photo as a result using
rememberLauncherForActivityResult(ActivityResultContracts.[TakePicturePreview](#)())

Select image from the Gallery

- Create **ActivityResultLauncher** using **rememberLauncherForActivityResult** to **launch** the image gallery app to allow the user to select an image, then provide a callback to **handle** the selected image
 - The **path** of the selected image is available as a **uri** parameter accessible to the callback function

```
val imagePicker =  
    rememberLauncherForActivityResult(  
        ActivityResultContracts..GetContent()) { uri ->  
        displayMessage(context, "Select image Uri: $uri")  
        uri?.let {  
            viewModel.uploadImage(it)  
        }  
    }
```

```
onUploadPhotoFromGallery = { imagePicker..launch("image/*") }
```

Take a Picture using the Camera

- Create **ActivityResultLauncher** using `rememberLauncherForActivityResult` to **launch** the camera app, then provide a callback to **handle** the taken image once the camera app is closed
 - The image taken is made available as a bitmap parameter to the callback function

```
val takePicture =  
    rememberLauncherForActivityResult(  
        ActivityResultContracts.TakePicturePreview()) { bitmap ->  
        bitmap?.let {  
            viewModel.uploadImage(it)  
            displayMessage(context, "Picture uploaded successfully", Toast.LENGTH_SHORT)  
        }  
    }
```

```
onTakePicture = { takePicture.Launch() }
```

Firebase 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



Sign in

- Sign in using Firebase authentication

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

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 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 **Cloud Storage** is used to store and retrieve files
- **Firebase Authentication** provides built-in backend services to ease user authentication
 - email/password authentication allows users to register and log in to the app
 - Secure user's authentication using various identity providers (e.g., email/password, Google Auth)

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>