**Section 37 – Firebase Firestore Database**

**1. Key Concepts**

1. **What is Cloud Firestore?**
   * A **NoSQL, document-oriented database** by Firebase.
   * Stores data in **collections → documents**.
   * **Schema-less** → documents in the same collection can have different fields.
   * Supports **real-time synchronization** → all connected clients get updates instantly.
   * Scalable, ideal for collaborative or interactive apps.
2. **Core Structure**
   * **Collection**: A group of related documents (similar to a SQL table but without a fixed schema).
   * **Document**: A JSON-like object with key-value pairs (like a row but flexible).
   * **Document ID**: Unique in its collection; can be **auto-generated** or set manually.
   * **Sub-collections**: Collections nested inside documents for hierarchical data.
3. **Data Types in Documents**
   * String, Number, Boolean, Map, Array, Timestamp, GeoPoint, Reference, etc.
4. **Firestore Access Points**
   * **FirebaseFirestore** → entry point to database.
   * **CollectionReference** → reference to a collection (for queries, adding docs).
   * **DocumentReference** → reference to a specific document (for read/update/delete).

**2. Step-by-Step Implementation**

**Step 1 – Create Firebase Project & Firestore Database**

1. Go to **Firebase Console** → Add Project.
2. Name project (e.g., FirestoreApp), choose account, continue.
3. Navigate to **Build → Firestore Database** → **Create Database**.
4. **Test mode** for development (allows public read/write for 30 days).
5. Enable → you’ll see **Data, Rules, Indexes, Usage** tabs.

**Step 2 – Connect Android App to Firebase**

1. **In Firebase Console → Project Settings → Add App → Android**.
2. Enter **package name** (exact from Android Studio).
3. Download **google-services.json** → place in app/ folder.
4. Add **Gradle setup**:

**build.gradle (Project)**

buildscript {

dependencies {

classpath "com.google.gms:google-services:4.3.15" // latest

}

}

**build.gradle (App)**

plugins {

id 'com.android.application'

id 'com.google.gms.google-services'

}

dependencies {

implementation platform('com.google.firebase:firebase-bom:32.3.1') // BOM

implementation 'com.google.firebase:firebase-analytics'

implementation 'com.google.firebase:firebase-firestore'

}

**BOM** ensures all Firebase libs use the same version.

**Step 3 – Layout**

Example activity\_main.xml:

<EditText android:id="@+id/name" ... />

<EditText android:id="@+id/email" ... />

<Button android:id="@+id/saveBtn" android:text="Save Data" ... />

<Button android:id="@+id/readBtn" android:text="Read Data" ... />

<Button android:id="@+id/updateBtn" android:text="Update Data" ... />

<Button android:id="@+id/deleteBtn" android:text="Delete Data" ... />

<TextView android:id="@+id/resultText" ... />

**Step 4 – Model Class**

**Friend.java**

public class Friend {

private String name; private String email;

// Empty constructor REQUIRED by Firestore for object mapping

public Friend() { }

public Friend(String name, String email) {

this.name = name;

this.email = email;

}

// Getters & setters

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getEmail() { return email; }

public void setEmail(String email) { this.email = email; }

}

**Step 5 – Initialize Firestore References**

**MainActivity.java**

FirebaseFirestore db = FirebaseFirestore.getInstance();

// Reference to "users" collection

CollectionReference usersRef = db.collection("users");

// Reference to a specific document in "users" collection

DocumentReference friendRef = usersRef.document("friendDocId"); // Replace with real ID

**Step 6 – Save Data**

private void saveData() {

String nameStr = nameEditText.getText().toString();

String emailStr = emailEditText.getText().toString();

Friend friend = new Friend(nameStr, emailStr);

usersRef.add(friend) // Auto-generates doc ID

.addOnSuccessListener(docRef -> {

String docId = docRef.getId();

resultText.setText("Saved with ID: " + docId);

})

.addOnFailureListener(e -> {

resultText.setText("Error: " + e.getMessage());

});

} .add() → generates a new doc with random ID in the collection.

**Step 7 – Read All Documents**

private void readData() {

usersRef.get()

.addOnSuccessListener(querySnapshot -> {

StringBuilder data = new StringBuilder();

for (QueryDocumentSnapshot doc : querySnapshot) {

Friend friend = doc.toObject(Friend.class);

data.append(friend.getName())

.append(" - ")

.append(friend.getEmail())

.append("\n");

}

resultText.setText(data.toString());

})

.addOnFailureListener(e -> resultText.setText("Read failed: " + e.getMessage()));

}

**Step 8 – Update Specific Document**

private void updateData(String docId) {

DocumentReference docRef = usersRef.document(docId);

Map<String, Object> updates = new HashMap<>();

updates.put("name", nameEditText.getText().toString());

updates.put("email", emailEditText.getText().toString());

docRef.update(updates)

.addOnSuccessListener(aVoid -> resultText.setText("Updated successfully"))

.addOnFailureListener(e -> resultText.setText("Update failed: " + e.getMessage()));

}

**Important**: You must know the document ID to update.

**Step 9 – Delete Specific Document**

private void deleteData(String docId) {

usersRef.document(docId).delete()

.addOnSuccessListener(aVoid -> resultText.setText("Deleted successfully"))

.addOnFailureListener(e -> resultText.setText("Delete failed: " + e.getMessage())); }

**Step 10 – Button Listeners**

saveBtn.setOnClickListener(v -> saveData());

readBtn.setOnClickListener(v -> readData());

updateBtn.setOnClickListener(v -> updateData("SPECIFIC\_DOC\_ID"));

deleteBtn.setOnClickListener(v -> deleteData("SPECIFIC\_DOC\_ID"));

**3. Tools, Libraries, APIs Used**

* **Firebase Firestore SDK**
* **Firebase BOM** for version control
* **Firebase Console** for DB management
* Android **OnSuccessListener / OnFailureListener** callbacks

**4. Best Practices**

* Always use **empty constructors** in model classes for Firestore.
* Use **BOM** to keep Firebase dependencies consistent.
* Keep **document IDs** if you’ll need to update/delete later.
* Minimize reads/writes to reduce **Firestore billing costs**.
* For **real-time updates**, use addSnapshotListener() instead of get().
* Secure your database → **DO NOT use test mode in production**. Set security rules.
* Avoid deeply nested sub-collections unless absolutely needed.

**5. Part B – Extra Knowledge Not Covered**

1. **Real-Time Listeners**:
2. usersRef.addSnapshotListener((snapshots, e) -> {
3. if (e != null) return;
4. for (DocumentChange dc : snapshots.getDocumentChanges()) {
5. // Handle ADDED, MODIFIED, REMOVED
6. }
7. });
8. **Offline Persistence**:
9. FirebaseFirestoreSettings settings = new FirebaseFirestoreSettings.Builder()
10. .setPersistenceEnabled(true)
11. .build();
12. db.setFirestoreSettings(settings);
13. **Pagination & Querying**:
    * .orderBy("field").limit(10)
    * .whereEqualTo("name", "John")
14. **Security Rules**:
    * Enforce user authentication and field-level rules in Firestore.
15. **Batch Writes & Transactions**:
    * Update multiple docs atomically using WriteBatch or runTransaction().
16. **Kotlin & Coroutines**:
    * Prefer Tasks.await() or Kotlin coroutines for cleaner async code.