

Mobile Applications

4. Storing data in Android

Boni García

boni.garcia@uc3m.es

Telematic Engineering Department
School of Engineering

2024/2025

uc3m | Universidad **Carlos III** de Madrid



Table of contents

1. Introduction
2. Firebase
3. DataStore
4. Files
5. Local database
6. Content providers
7. Takeaways

1. Introduction

- **Persistent storage** refers to the mechanisms used to save data permanently on a device, ensuring that the data remains available even after the app is closed or the device is restarted
- Android provides several options for persistent storage, such as:
 - Local database: using **SQLlite**, a lightweight relational database integrated in Android
 - Internal storage: to store private files directly on the device's file system
 - External storage: for storing files that can be shared with other apps
 - DataStore: key-value pairs (e.g., for user preferences)
 - Cloud storage: in this unit, we study **Cloud Firestore**, a cloud-hosted NoSQL document database provided by **Firebase** (Google)

Table of contents

1. Introduction
2. Firebase
 - Cloud Firestore
 - Authentication
3. DataStore
4. Files
5. Local database
6. Content providers
7. Takeaways

2. Firebase

- **Firebase** is a set of backend cloud computing services and application development platforms provided by Google
 - It provides a range of services and tools to assist developers in building, improving, and scaling mobile and web applications
 - It is sometimes called a *backend as a service*
- **Google Cloud** a suite of cloud computing services that provides for data storage, analytics, machine learning, etc.
 - Firebase is considered to be a part of Google Cloud
 - Firebase was initially an independent startup that Google acquired in 2014
 - Firebase has been integrated into the broader Google Cloud ecosystem



Google Cloud

<https://cloud.google.com/>



Firebase

<https://firebase.google.com/>

2. Firebase

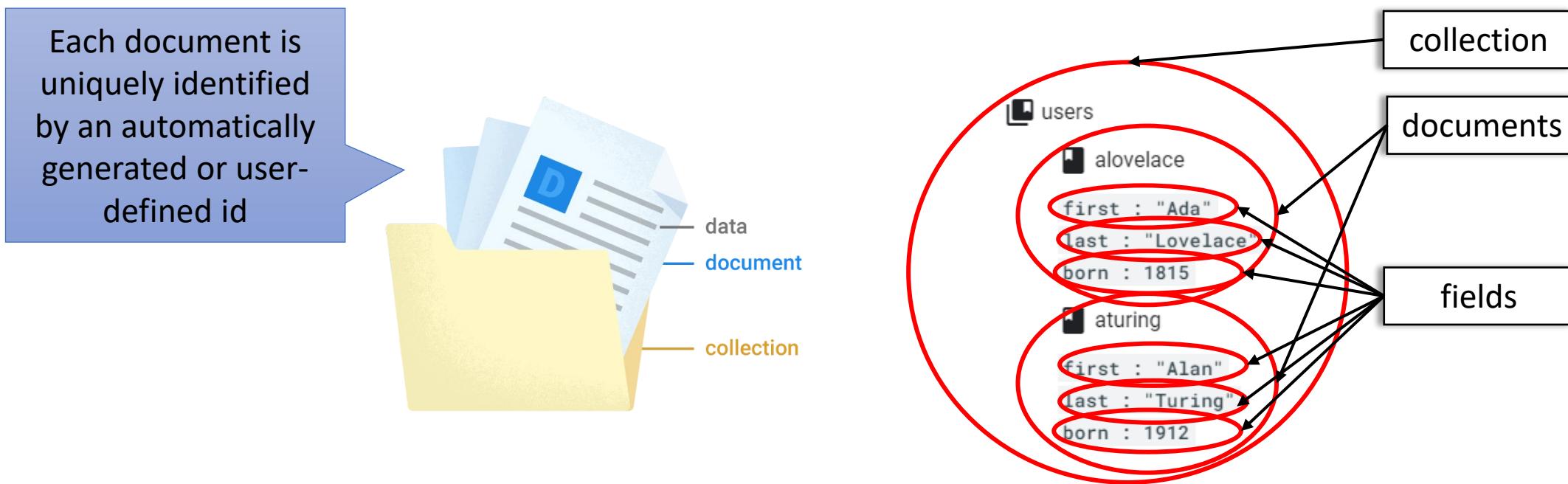
- Some key components of Firebase are the following:
 - Cloud Firestore: a cloud-hosted NoSQL document database
 - Authentication: support service for various authentication methods, including email/password, social media logins (Google, Facebook, Twitter), and more
 - Cloud Functions: serverless computing, allowing to run backend code in response to HTTPS requests and events triggered by Firebase
 - Cloud Storage: scalable and secure cloud storage for user-generated content like images, videos, and other files
 - Analytics: statistics about user engagement, retention, and conversion rates

Most Firebase services have quotas and pricing based on the usage

<https://firebase.google.com/docs/>

2. Firebase - Cloud Firestore

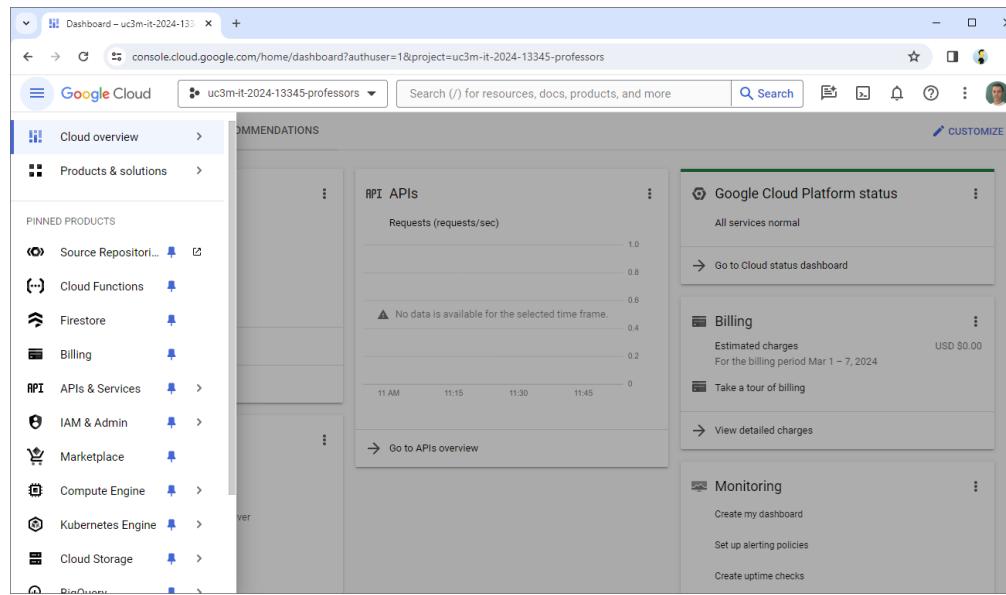
- In Cloud Firestore, the basic unit of storage is the **document**
 - A document is a lightweight record that contains **fields**, which map to values
 - Documents live in **collections**, which are simply containers for documents



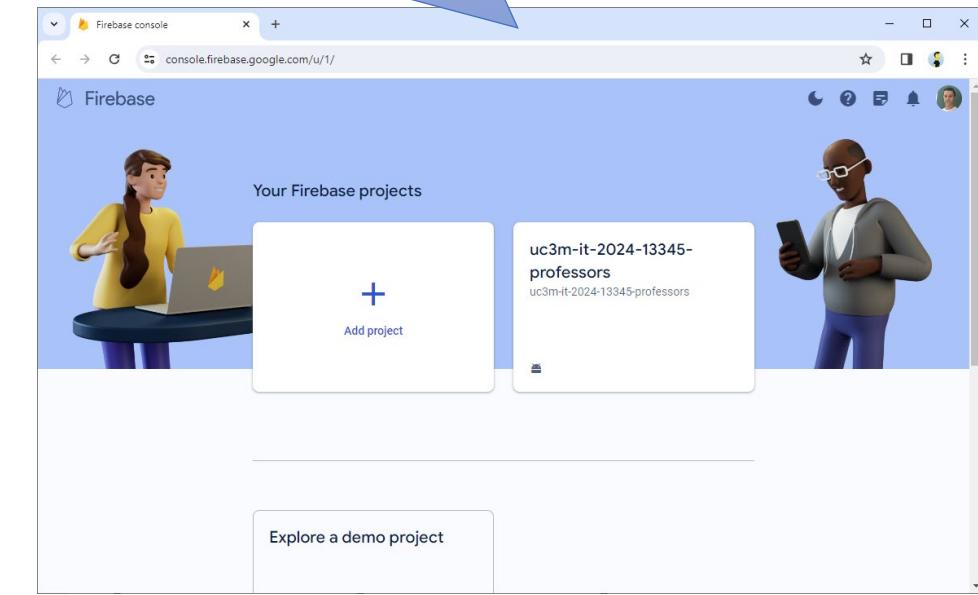
2. Firebase - Cloud Firestore

- Some of the Firebase services are also available through the Google Cloud console, for example, Cloud Firestore, Cloud Functions, or Cloud Storage

We use the Firebase console in this unit



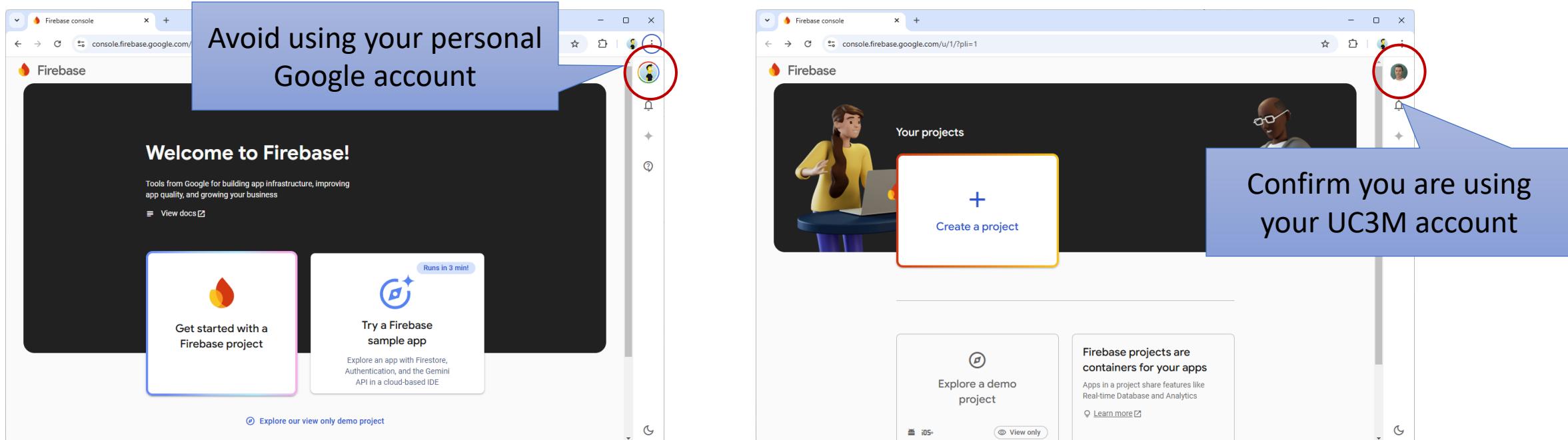
<https://console.cloud.google.com/>



<https://console.firebaseio.google.com/>

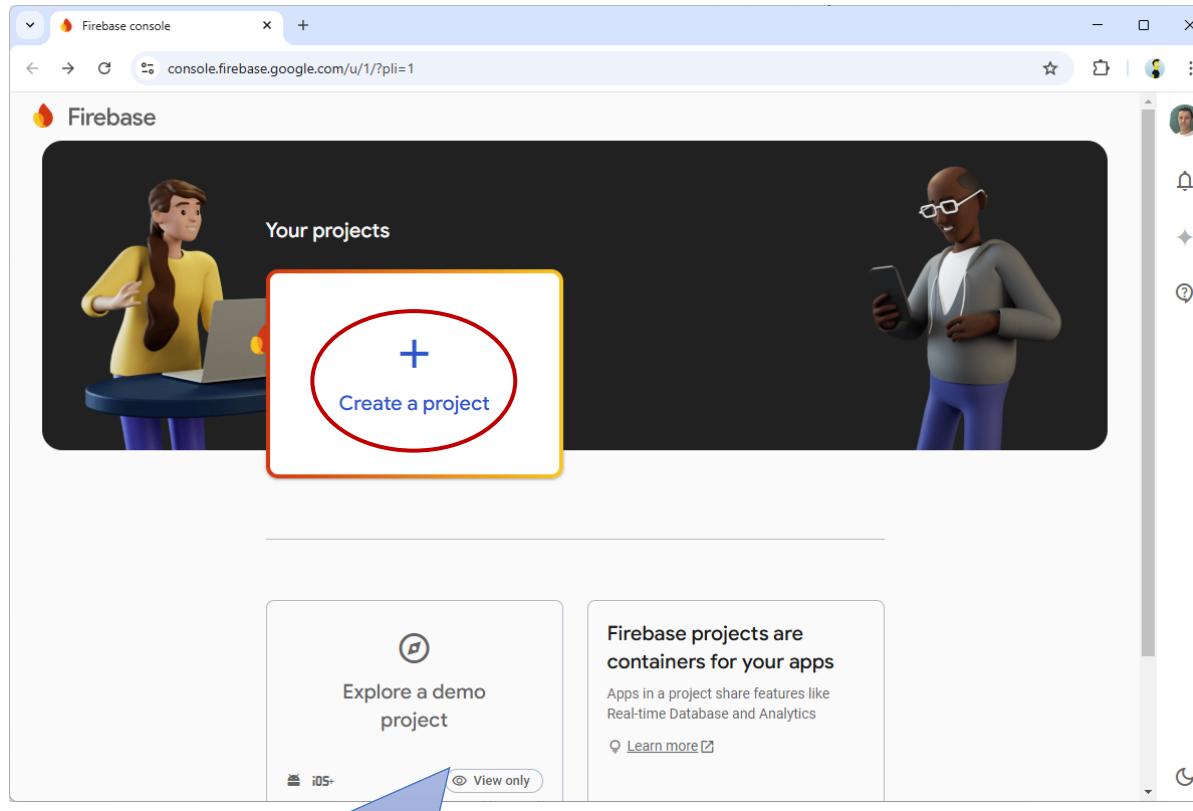
2. Firebase - Cloud Firestore

- To use Firebase, first we need a Google account
 - In this course, we should use our UC3M account (e.g. xxxxxxxxx@alumnos.uc3m.es)

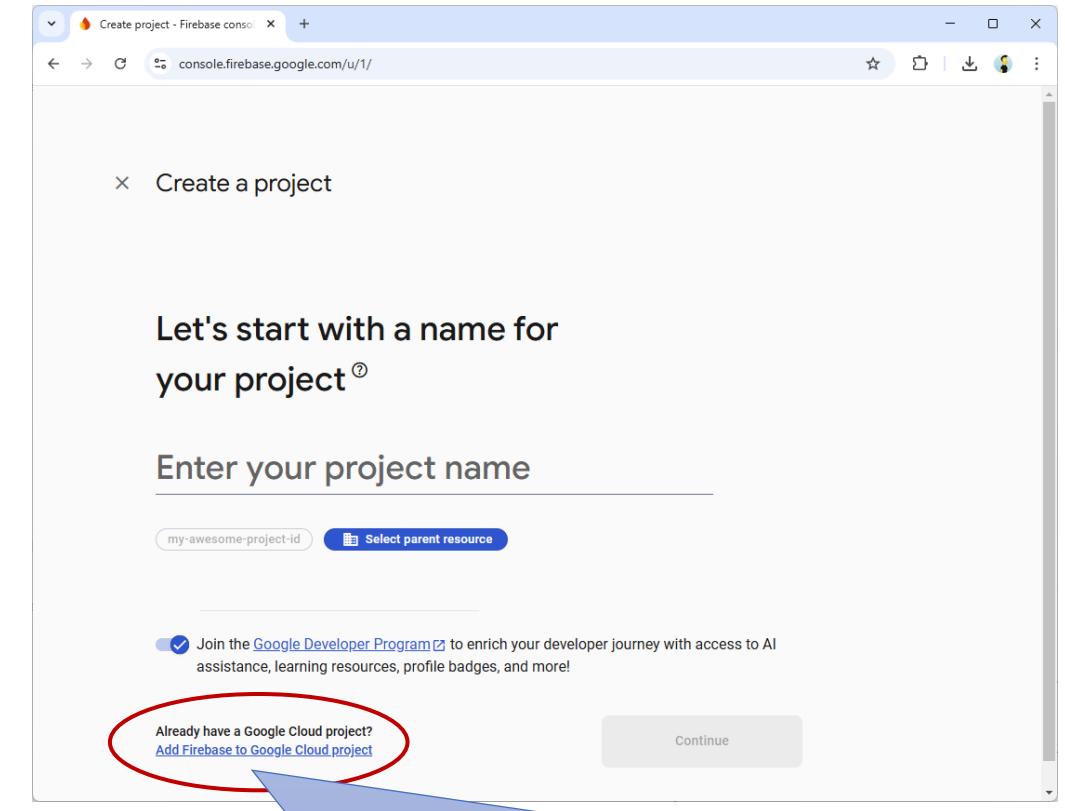


<https://console.firebaseio.google.com/>

2. Firebase - Cloud Firestore



The first step to use Cloud Firestore is to add a project in the console



We need to use the Google cloud project already created (**uc3m-it-2025-16504-g**-lab**), in which you should have redeemed your educational Google coupon (\$50) on the lab session on 7 March

2. Firebase - Cloud Firestore

The screenshot shows a browser window titled "Create project - Firebase console". The URL in the address bar is "console.firebaseio.google.com/u/1/". The main content area has a heading "Get started" and a sub-section "Let's start with choosing your Google Cloud project". Below this is a dropdown menu with the title "Select a Google Cloud project" highlighted by a red oval. The dropdown contains a list of project names:

- uc3m-it-2025-16504-g06-96
- uc3m-it-2025-16504-g05-96
- uc3m-it-2025-16504-g04-96
- uc3m-it-2025-16504-g03-96
- uc3m-it-2025-16504-g02-96
- uc3m-it-2025-16504-g01-96
- uc3m-it-2025-16504-professors

Select your project here and follow the instructions

The screenshot shows a browser window titled "Create project - Firebase console". The URL in the address bar is "console.firebaseio.google.com/u/1/". The main content area has a heading "Get started" and a sub-section "A few things to remember when adding Firebase to a Google Cloud project". This section includes several reminder icons and text:

- You won't be able to undo this, though you'll be able to manually disable most Firebase services.
- Billing is shared between Google Cloud and Firebase. [Learn more](#)
- Since your project has billing enabled, you'll be on Firebase's [pay-as-you-go Blaze pricing plan](#), and any accrued Firebase charges will appear on your Cloud bill each cycle.
- IAM roles and permissions for project members are shared between Google Cloud and Firebase, so project member access to your Google Cloud project will also apply to your Firebase project. [Learn more](#)
- Deleting a Firebase project deletes the Google Cloud project too, and all contained resources.
- Deleting or modifying a resource or data within a Firebase project applies to the resource or data within the Google Cloud project, too.

At the bottom left is a "Previous" button, and at the bottom right is a large blue "Continue" button, which is also circled in red.

2. Firebase - Cloud Firestore

The screenshot shows the 'Create project' wizard in the Firebase console. The current step is 'Confirm Firebase pricing plan'. It states: 'Billing is shared between Firebase and Google Cloud. Since your Google Cloud project has billing enabled, you'll be on Firebase's pay-as-you-go Blaze pricing plan.' Below this, there are two options: 'Blaze' (selected) and 'Pay as you go'. A link 'See full plan details' is also present. At the bottom, it says: 'If you'd prefer to be on a different pricing plan, please create a new project without billing enabled.' A 'Previous' button is at the bottom left, and a 'Confirm and continue' button is at the bottom right, which is circled in red.

We must have a billing account in our project

The screenshot shows the 'Create project' wizard in the Firebase console. The current step is 'Google Analytics for your Firebase project'. It states: 'Google Analytics is a free and unlimited analytics solution that enables targeting, reporting, and more in Firebase Crashlytics, Cloud Messaging, In-App Messaging, Remote Config, A/B Testing, and Cloud Functions.' Below this, it lists what Google Analytics enables: A/B testing, Breadcrumb logs in Crashlytics, User segmentation & targeting across Firebase products, and Free unlimited reporting. A checkbox 'Enable Google Analytics for this project' is checked and labeled 'Recommended'. A 'Previous' button is at the bottom left, and a 'Continue' button is at the bottom right, which is circled in red.

Analytics is optional

2. Firebase - Cloud Firestore

The image consists of two side-by-side screenshots from the Firebase console's 'Create project' wizard.

Screenshot 1 (Left): This screenshot shows the 'Get started' step. It includes a section titled 'Configure Google Analytics' with a dropdown menu set to 'Default Account for Firebase'. Below this is a note about creating a new Google Analytics property. At the bottom left is a 'Previous' link, and at the bottom right is a large blue button labeled 'Add Firebase' which is circled in red.

Screenshot 2 (Right): This screenshot shows the 'Your Firebase project is ready' step. It displays a project icon for 'uc3m-it-2025-16504-professors' and a checked checkbox next to the text 'Your Firebase project is ready'. Below this is a blue 'Continue' button, which is also circled in red. To the right of the button is a blue callout box containing the text 'Now Firebase is ready to be used'.

Illustrations: Both screenshots feature a cartoon character (a man in the first and a woman in the second) sitting at a desk with a laptop that has a small fire icon on its screen. The background of both screenshots is a dark gray gear-like pattern.

2. Firebase - Cloud Firestore

The screenshot shows the Firebase Project Overview page for a project named "uc3m-it-2025-16504-professors". The left sidebar includes options like Generative AI, Build with Gemini, Genkit, Product categories, Build, Run, Analytics, and All products. The main area displays a "Get started by adding Firebase to your app" section with icons for iOS+, Android (highlighted with a red circle), Web, Windows, and Mac. A blue callout box with white text says: "Then, we should register our Android app".

The screenshot shows the "Add Firebase to your Android app" registration page. Step 1: Register app. It asks for the "Android package name" which is filled with "es.uc3m.android.helloworld". A red circle highlights this field. Below it, there's an "App nickname (optional)" field with "My Android App" and a "Debug signing certificate SHA-1 (optional)" field with placeholder text. Step 2: Download and then add config file. A red circle highlights the "Register app" button at the bottom.

```
android {
    namespace = "es.uc3m.android.firebaseio"
    compileSdk = 35

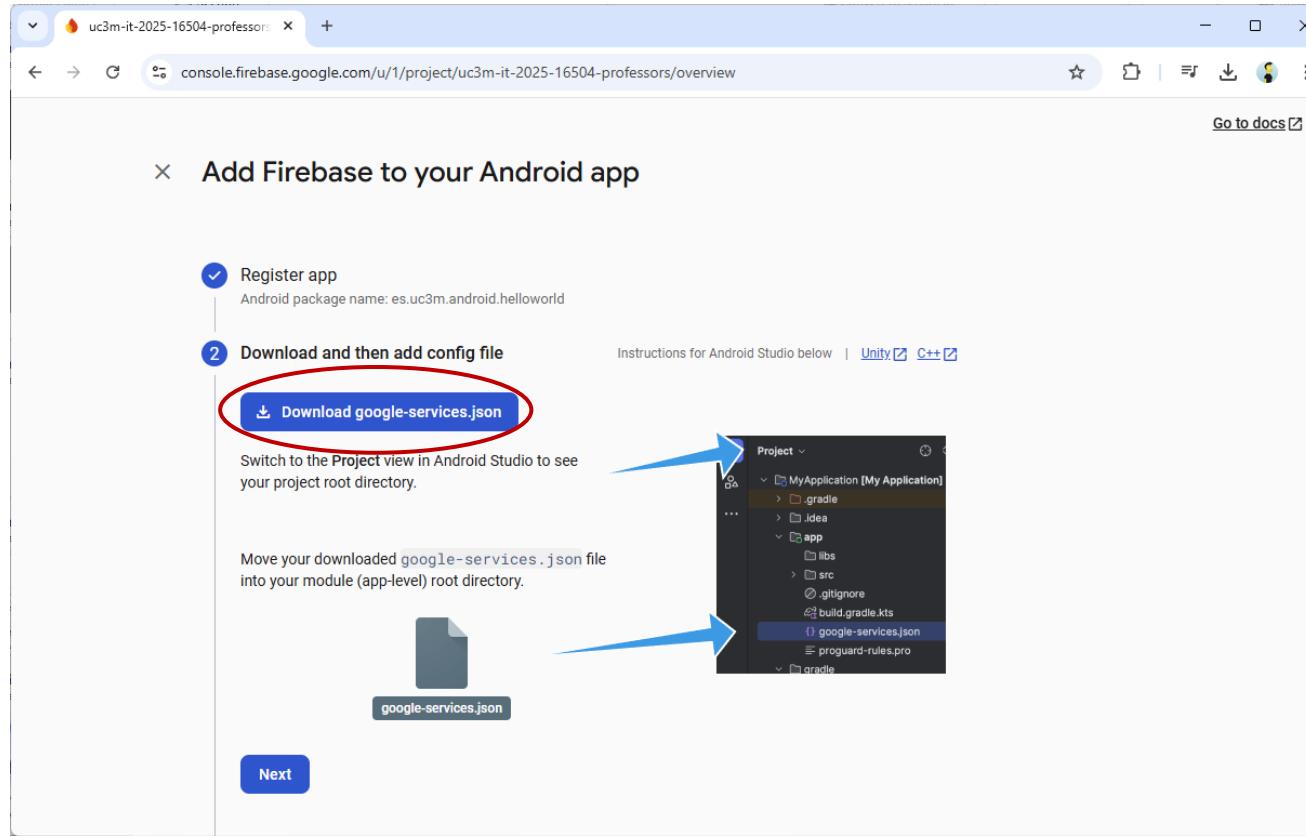
    defaultConfig {
        applicationId = "es.uc3m.android.firebaseio"
        minSdk = 24
        targetSdk = 35
        versionCode = 1
        versionName = "1.0"

        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
    }
}
```

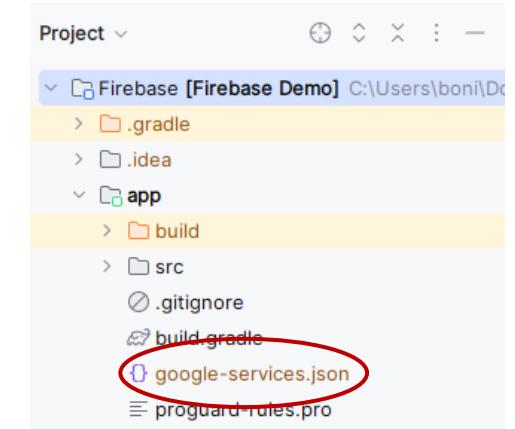
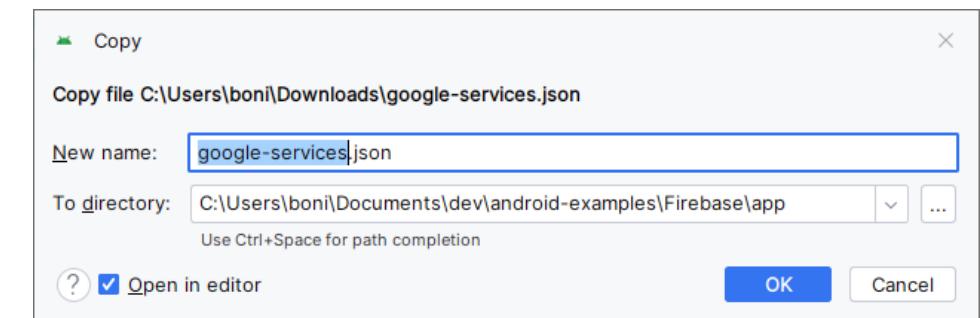
The Android package name must be the id defined in our app's build.gradle.kts

<https://firebase.google.com/docs/android/setup>

2. Firebase - Cloud Firestore



Then we need to download a configuration file called **google-services.json** and copy it to the app folder of our Android project



<https://developers.google.com/android/guides/google-services-plugin>

2. Firebase - Cloud Firestore

- For security reasons, it is not recommended to publish **google-services.json** on open repositories (e.g., in GitHub)
 - As the doc says:

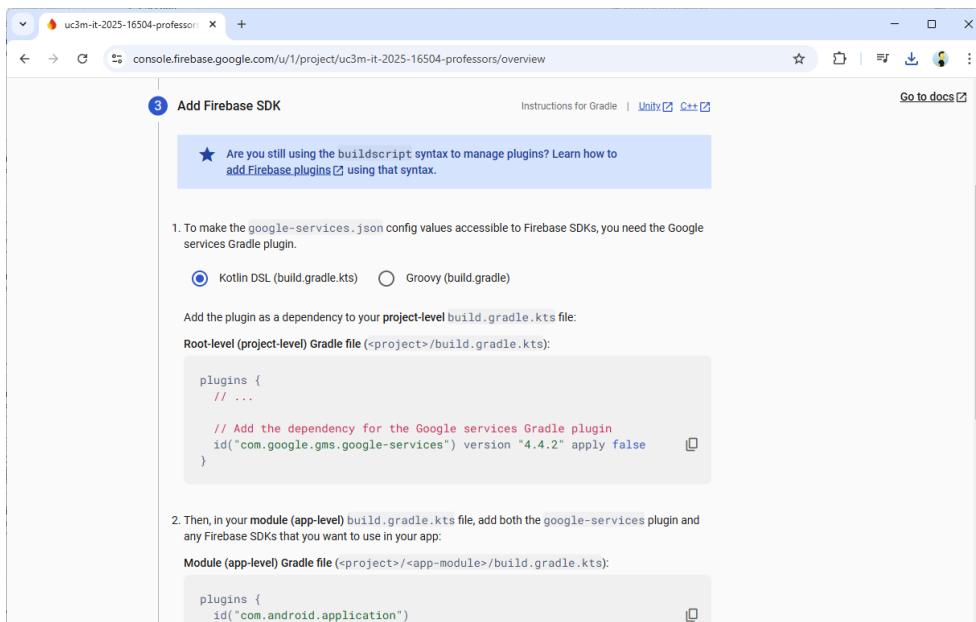
For open source projects, we generally do not recommend including the app's Firebase config file or object in source control because, in most cases, your users should create their own Firebase projects and point their apps to their own Firebase resources (via their own Firebase config file or object).

<https://firebase.google.com/docs/projects/learn-more#config-files-objects>

So, if you are using an open GitHub repository, a good practice is to include this file name in .gitignore

2. Firebase - Cloud Firestore

- Then, we need to configure our Android project to use Firebase:



```
build.gradle.kts (project)
plugins {
    alias(libs.plugins.google.services) apply false
}

build.gradle.kts (app)
plugins {
    alias(libs.plugins.google.services)
}

dependencies {
    implementation(platform(libs.firebaseio.bom))
    implementation(libs.firebaseio.firestore)
    implementation(libs.firebaseio.auth)
}

libs.version.toml
[versions]
google-services = "4.4.2"
firebaseBom = "33.10.0"

[libraries]
firebase-firestore = { module = "com.google.firebaseio:firebase-firestore" }
firebase-bom = { module = "com.google.firebaseio:firebase-bom", version.ref = "firebaseBom" }
firebase-auth = { module = "com.google.firebaseio:firebase-auth" }

[plugins]
google-services = { id = "com.google.gms.google-services", version.ref = "google-services" }
```

2. Firebase - Cloud Firestore

The image shows two screenshots of the Firebase console. The left screenshot is a step-by-step guide titled 'Add Firebase to your Android app'. It includes four steps: 'Register app' (completed, Android package name: es.uc3m.android.helloworld), 'Download and then add config file', 'Add Firebase SDK', and 'Next steps'. The 'Next steps' section contains links to documentation, sample apps, and the console. The right screenshot shows the 'Project Overview' page for the project 'uc3m-it-2025-16504-professors'. It features a 'Blaze plan' button, a 'Getting started? Tell Gemini about your project' button, and a list of products including 'es.uc3m.android.f...'. A blue arrow points from the text 'At the end, you can see your app in the Firebase console' to the Firebase logo in the project title bar.

At the end, you can see your app in the Firebase console

uc3m-it-2025-16504-professors

Project Overview

Generative AI

Build with Gemini

Genkit (NEW)

Product categories

Build

Run

Analytics

All products

Related development tools

IDX Checks

Blaze Pay as you go Modify

Waiting for Analytics data...

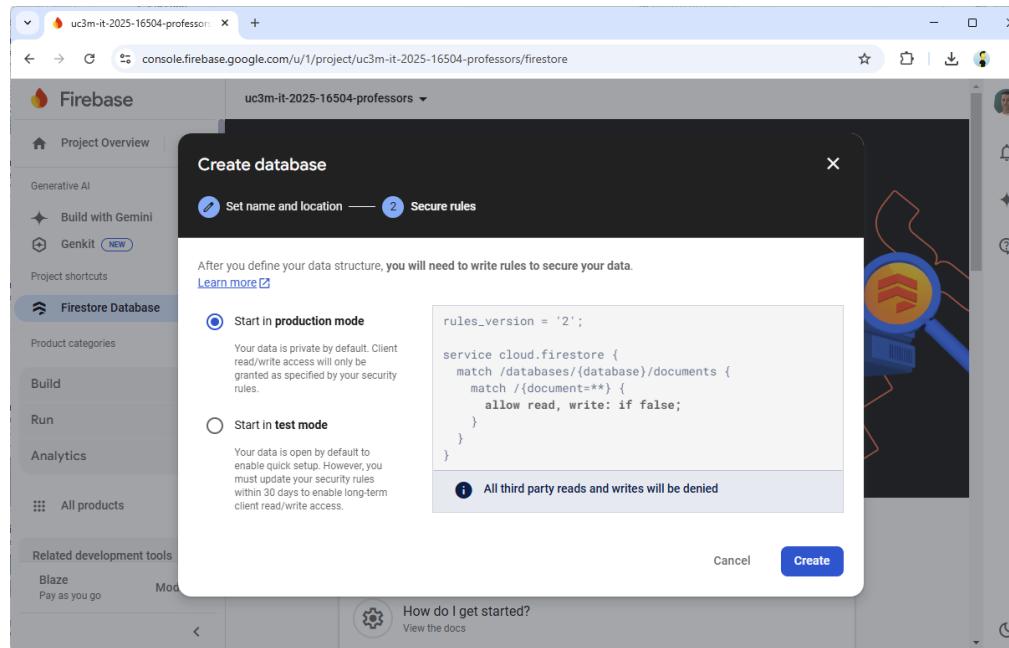
2. Firebase - Cloud Firestore

The image shows two screenshots of the Firebase console. The left screenshot displays the 'Project Overview' page for the project 'uc3m-it-2025-16504-professors'. It features four main service cards: 'Authentication' (User icon), 'Cloud Firestore' (Database icon, circled in red), 'Storage' (Image icon), and 'Hosting' (Website icon). A callout bubble points to the 'Cloud Firestore' card with the text 'Scroll down and click on "Cloud Firestore"'. The right screenshot shows a modal dialog titled 'Create database' under the 'Firestore Database' section. It has two steps: 'Set name and location' (selected) and 'Secure rules'. The 'Database ID' field contains '(default)', and the 'Location' dropdown is set to 'eur3 (Europe)'. A note at the bottom states, 'After you set this location, you cannot change it later.' A 'Next' button is visible at the bottom right of the modal.

Then, we should create a database, specifying a name (can be just “default”) and a location

Scroll down and click on “Cloud Firestore”

2. Firebase - Cloud Firestore

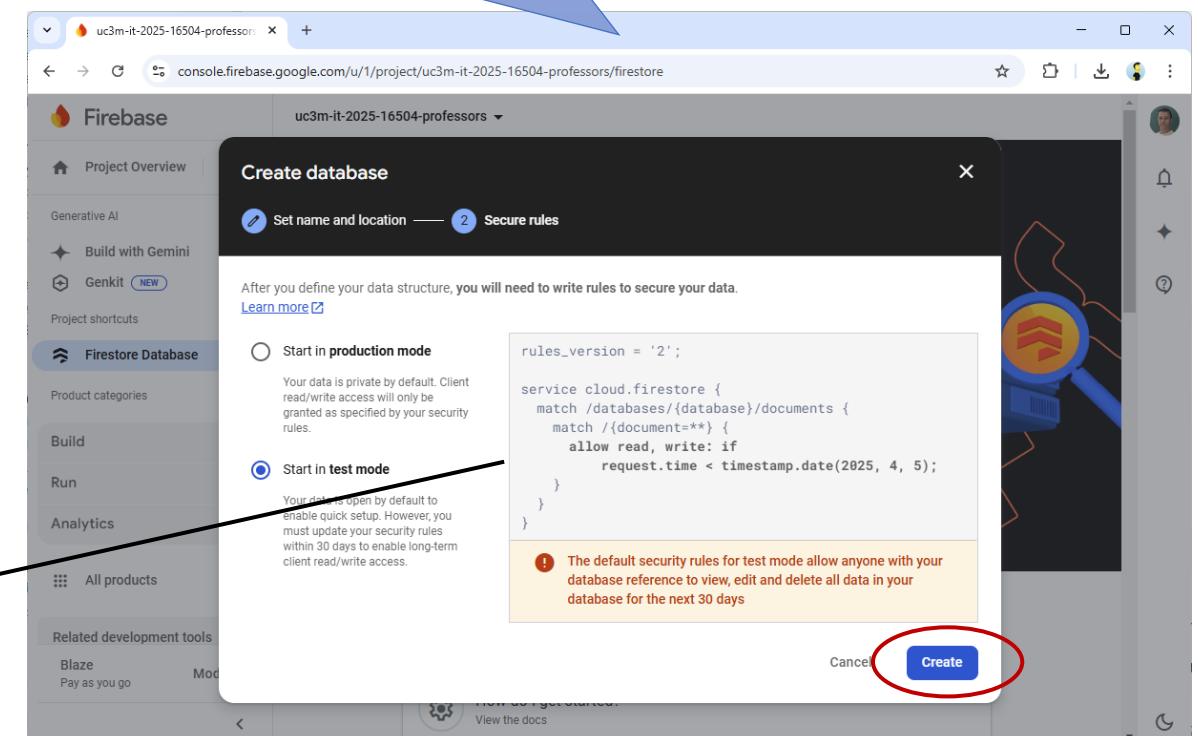


```
rules_version = '2';

service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if
        request.time < timestamp.date(2025, 4, 5);
    }
  }
}
```

<https://firebase.google.com/docs/rules>

Then, we need to specify the **Security Rules**, which are a set of declarative statements that define how Firestore should handle read and write operations. We can start in test mode, although this should be changed in the future



2. Firebase - Cloud Firestore

- The Security Rules is a feature in Firestore that allows us to control access to our database
 - These rules determine who can read, write, update, or delete data in your Firestore collections and documents
 - They are organized hierarchically and follow this structure:

This line specifies the database and documents to which the rules apply

This line defines rules for a specific collection or document

```
rules_version = '2';

service cloud.firestore {
    match /databases/{database}/documents {
        // Define rules for specific collections or documents
        match /collection/{document} {
            allow read, write: if <condition>;
        }
    }
}
```

This line specifies the operations (read, write, create, update, delete) allowed under certain conditions

<https://firebase.google.com/docs/firestore/security/get-started>

2. Firebase - Cloud Firestore

We can change the security rules in this part of the Firebase console

The screenshot shows the Firebase Cloud Firestore Rules page. On the left, there's a sidebar with project settings like Generative AI, Build with Gemini, Genkit, and Project shortcuts. Below that is a navigation bar with 'Firestore Database' selected. The main content area has tabs for Data, Rules (which is highlighted with a red circle), Indexes, Disaster Recovery, Usage, and Extensions. A 'Develop & Test' button is at the bottom right. On the left side of the main area, there's a log of recent rule changes with timestamps: 'Today • 6:18 PM', 'Today • 1:42 PM', and 'Today • 11:49 AM'. To the right of the log is the actual security rules code:

```
rules_version = '2';

service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if request.auth != null;
    }
  }
}
```

At the bottom, there's a 'Rules Playground' section with a note: 'Experiment and explore with Security Rules'.

2. Firebase - Cloud Firestore

- Some examples of basic security rules are as follows:

```
rules_version = '2';

service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if
        request.time < timestamp.date(2025, 4, 5);
    }
  }
}
```

Test mode: open access in our database for a limited period (usually 30 days)

```
rules_version = '2';

service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if true;
    }
  }
}
```

Unlimited open access. This approach is not recommended since it completely disables security

```
rules_version = '2';

service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if request.auth != null;
    }
  }
}
```

2. Firebase - Cloud Firestore

The screenshot shows the Firebase Cloud Firestore console interface. On the left, there's a sidebar with project settings like Generative AI, Build with Gemini, Genkit, Project shortcuts, and the currently selected **Firebase Database**. The main area is titled "Cloud Firestore" and shows a "Data" tab. It includes sections for budgets, security rules, indexes, disaster recovery, usage, and extensions. A "Start collection" button is visible. A modal window is open in the foreground, titled "Start a collection". It asks for a "Collection ID" which is set to "notes". There are "Cancel" and "Next" buttons at the bottom of the modal.

After provisioning the database, we can manage the documents in the database manually, although our objective is to do it programmatically from our Android app

2. Firebase - Cloud Firestore

- For handling Cloud Firestore programmatically, we need some form of **asynchronous programming**
 - Asynchronous programming is a paradigm that allows tasks to be executed concurrently, without blocking the execution of the main program
 - This is particularly useful for tasks that may take a significant amount of time to complete, such as network requests
- Using **Kotlin Coroutines** can simplify asynchronous programming in Android app
 - Coroutines are functions that can be paused and resumed without blocking the thread, making it ideal for long-running tasks like network operations or database access
 - Coroutines allow us to write asynchronous code in a sequential style, making it easier to read and maintain

<https://developer.android.com/kotlin/coroutines>

2. Firebase - Cloud Firestore

Fork me on GitHub

For managing complex state that survives configuration changes, we can use an instance of ViewModel

The ViewModel in this example expose a notes list (`notes`) and functions to handle it with CRUD operations

```
class MyViewModel : ViewModel() {  
    private val _notes = MutableStateFlow<List<Note>>(emptyList())  
    val notes: StateFlow<List<Note>> get() = _notes  
  
    private val _toastMessage = MutableStateFlow<String?>(null)  
    val toastMessage: StateFlow<String?> get() = _toastMessage  
  
    private val firestore = FirebaseFirestore.getInstance()  
  
    // ...  
}
```

The ViewModel should remain UI-agnostic, therefore we expose a message (`toastMessage`) to be displayed in the UI in case of errors

```
build.gradle.kts (app)  
dependencies {  
    implementation(Libs.androidx.runtime.livedata)  
}
```

libs.version.toml

```
[versions]  
runtimeLiveData = "1.7.8"
```

```
[libraries]  
androidx-runtime-livedata = { module =  
    "androidx.compose.runtime:runtime-livedata",  
    version.ref = "runtimeLiveData" }
```

We observe changes in the ViewModel using the library LiveData

2. Firebase - Cloud Firestore

Read data. We can use the `get()` method to retrieve an entire collection

```
private fun fetchNotes() {
    viewModelScope.launch {
        firestore.collection(NOTES_COLLECTION).get()
            .addOnSuccessListener { result ->
                val noteList = result.map { document ->
                    document.toObject<Note>().copy(id = document.id)
                }
                _notes.value = noteList
            }
            .addOnFailureListener { exception ->
                _toastMessage.value = exception.message
            }
    }
}
```

Using `viewModelScope` is a way to manage coroutines in a ViewModel. When the ViewModel is cleared (e.g., when the associated Activity is destroyed), the `viewModelScope` is automatically canceled, ensuring that no coroutines run unnecessarily

Add data. We use the `add()` method to add a document in a collection

```
fun addNote(title: String, body: String) {
    viewModelScope.launch {
        val note = Note(title = title, body = body)
        firestore.collection(NOTES_COLLECTION)
            .add(note)
            .addOnSuccessListener {
                fetchNotes() // Refresh the List after adding
            }
            .addOnFailureListener { exception ->
                _toastMessage.value = exception.message
            }
    }
}
```

Firebase creates collections and documents implicitly the first time we add data to the document

https://firebase.google.com/docs/firestore/quickstart#java_1

2. Firebase - Cloud Firestore

Update data. We can use the update() method to modify a document. We should know the document id (typically autogenerated)

```
fun updateNote(id: String, title: String, body: String) {  
    viewModelScope.launch {  
        val updatedNote = Note(title = title, body = body)  
        firestore.collection(NOTES_COLLECTION).document(id)  
            .set(updatedNote)  
            .addOnSuccessListener {  
                fetchNotes() // Refresh the list after updating  
            }  
            .addOnFailureListener { exception ->  
                _toastMessage.value = exception.message  
            }  
    }  
}
```

Delete data. We can use the delete() method to modify a document. We should also know the document id

```
fun deleteNote(id: String) {  
    viewModelScope.launch {  
        firestore.collection(NOTES_COLLECTION).document(id)  
            .delete()  
            .addOnSuccessListener {  
                fetchNotes() // Refresh the list after deleting  
            }  
            .addOnFailureListener { exception ->  
                _toastMessage.value = exception.message  
            }  
    }  
}
```

2. Firebase - Cloud Firestore

Fork me on GitHub

```
@Composable
fun MainScreen(viewModel: MyViewModel = viewModel()) {
    var showAddNoteDialog by remember { mutableStateOf(false) }
    var noteToEdit by remember { mutableStateOf<Note?>(null) }
    val context = LocalContext.current
    val notes by viewModel.notes.collectAsState()
    val toastMessage by viewModel.toastMessage.collectAsState()

    Scaffold(
        floatingActionButton = {
            FloatingActionButton(onClick = { showAddNoteDialog = true }) {
                Icon(Icons.Default.Add, contentDescription = stringResource(R.string.add_note))
            }
        }
    ) { padding ->
        Column(modifier = Modifier.padding(padding)) {
            LazyColumn {
                items(notes) { note ->
                    NoteItem(
                        note = note,
                        onNoteClick = { noteToEdit = it },
                        onDeleteClick = { viewModel.deleteNote(it.id!!) }
                    )
                }
            }
        }
    }
}
```

LaunchedEffect is a helper composable that allows us to run background tasks (like fetching data, waiting for something, or listening for updates) safely (i.e., without blocking the UI)

We observe changes in **notes** and **toastMessage** from the ViewModel

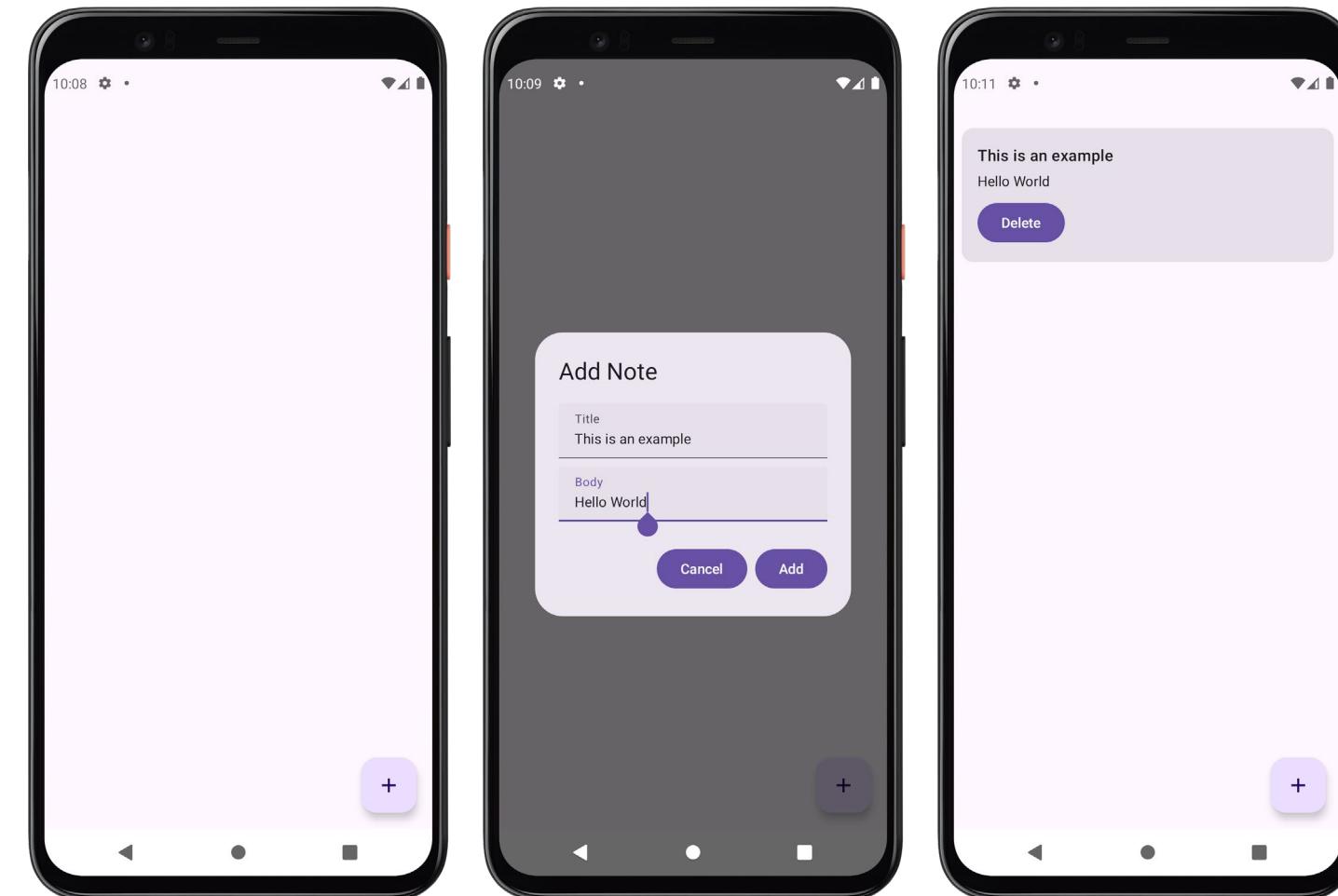
```
if (showAddNoteDialog) {
    AddNoteDialog(
        onDismiss = { showAddNoteDialog = false },
        onAddNote = { title, body ->
            viewModel.addNote(title, body)
            showAddNoteDialog = false
        }
    )
}

noteToEdit?.let { note ->
    EditNoteDialog(
        note = note,
        onDismiss = { noteToEdit = null },
        onUpdateNote = { title, body ->
            viewModel.updateNote(note.id!!, title, body)
            noteToEdit = null
        }
    )
}

LaunchedEffect(toastMessage) {
    toastMessage?.let { message ->
        Toast.makeText(context, message, Toast.LENGTH_LONG).show()
        // Reset message to avoid showing it repeatedly (e.g., config changes)
        viewModel.showToast(null)
    }
}
```

2. Firebase - Cloud Firestore

Fork me on GitHub

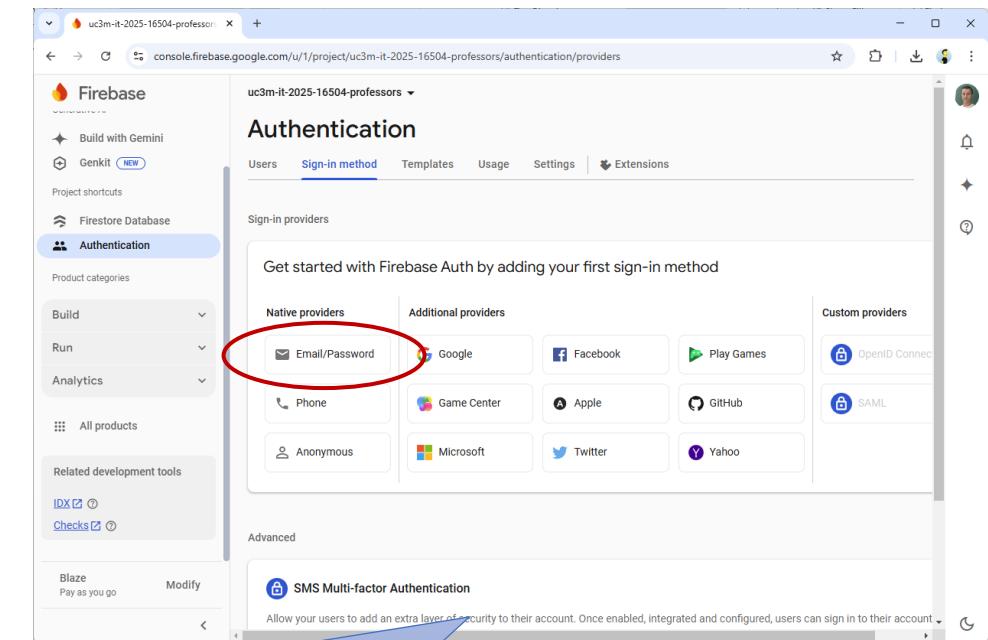
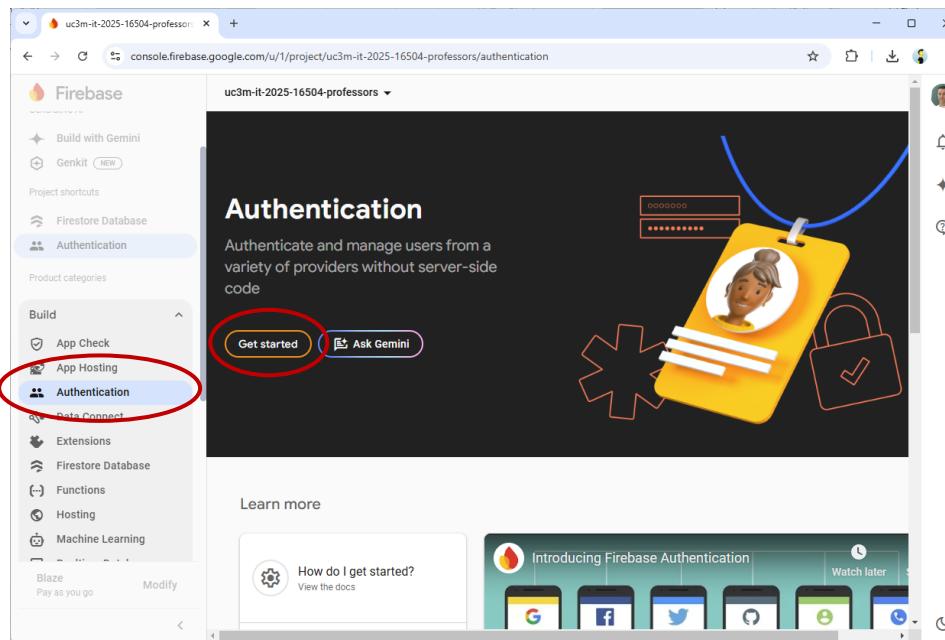


The image shows the Firebase Cloud Firestore console interface:

- Left sidebar:** Shows project navigation, "Generative AI", "Build with Gemini", "Genkit", "Project shortcuts", and "Cloud Firestore" (which is selected).
- Top bar:** Displays the project name "uc3m-it-2025-16504-professors" and tabs for "Data", "Rules", "Indexes", "Disaster Recovery", "Usage", and "Extensions".
- Central area:** Shows the "notes" collection. A specific document is selected, showing its ID "bOYZ0NNfPwpUrL8b2QwY". The document data is listed as follows:
 - body: "Hello World"
 - title: "This is an example"

2. Firebase - Authentication

- Authentication in Firebase is a service that allows users to sign in using various methods, such as email/password, phone numbers, or social media accounts, among others



We need to select one or more sign-in providers (email/password, phone, social ids, etc.). In the following example, we use email/password

2. Firebase - Authentication

The screenshot shows the Firebase Authentication screen in the Firebase console. The left sidebar has 'Authentication' selected. The main area is titled 'Authentication' with tabs for 'Users', 'Sign-in method', 'Templates', 'Usage', 'Settings', and 'Extensions'. Under 'Sign-in providers', there are two sections: 'Email/Password' and 'Email link (passwordless sign-in)'. The 'Email/Password' section has an 'Enable' button with a checkmark, which is circled in red. Below it is a descriptive text: 'Allow users to sign up using their email address and password. Our SDKs also provide email address verification, password recovery, and email address change primitives.' A blue 'Save' button at the bottom right of this section is also circled in red.

We need to enable authentication with email/password here

2. Firebase - Authentication

- In the Java/Kotlin logic, we implement the logic for creating users (sing up) and authenticating existing users (log in):

Sing up

```
fun signUp(email: String, password: String) {
    viewModelScope.launch {
        try {
            auth.createUserWithEmailAndPassword(email, password).await()
            _route.value = NavGraph.Home.route
        } catch (e: Exception) {
            _toastMessage.value = e.message
        }
    }
}
```

Login

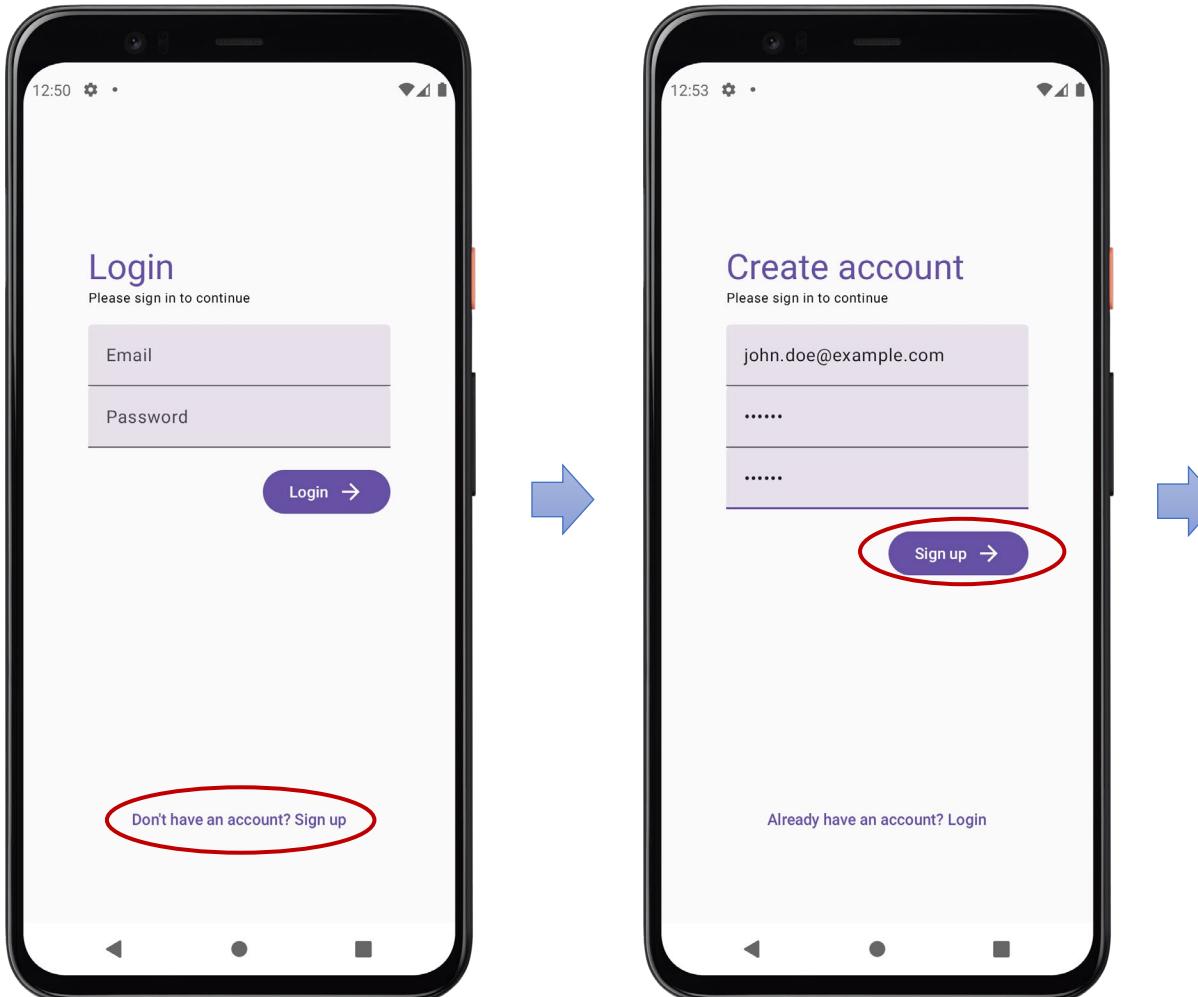
```
fun login(email: String, password: String) {
    viewModelScope.launch {
        try {
            auth.signInWithEmailAndPassword(email, password).await()
            _route.value = NavGraph.Home.route
        } catch (e: Exception) {
            _toastMessage.value = e.message
        }
    }
}
```

In both cases, the strings email and password have been read from the UI

<https://firebase.google.com/docs/auth/android/start>

Fork me on GitHub

2. Firebase - Authentication



The image shows the Firebase Authentication section of the Firebase console:

- Project Overview** (selected)
- Authentication** (selected)
- Users** (selected)
- Sign-in method**
- Templates**
- Usage**
- Settings**
- Extensions**

A message at the top states: "The following Authentication features will stop working when Firebase Dynamic Links shuts down on August 25, 2025: email link authentication for mobile apps, as well as Cordova OAuth support for web apps."

Identifier	Providers	Created	Signed In	User UID
john.doe@example.com	✉️	Mar 7, 2025	Mar 7, 2025	IFHMGl76ATYDNIPBvg6BqU...

Table of contents

1. Introduction
2. Firebase
3. DataStore
4. Files
5. Local database
6. Content providers
7. Takeaways

3. DataStore

- DataStore is a modern solution introduced by Android Jetpack to provides an efficient way to store **key-value** pairs
 - It is designed to address the limitations of SharedPreferences (used in the legacy Android View system)
- The advantages of DataStore are the following:
 - Asynchronous API: Uses Kotlin coroutines for asynchronous operations, avoiding blocking the main thread
 - Type safety: Reduces runtime errors by leveraging Kotlin's type system
 - Data consistency: Ensures atomicity for read-write operations
 - Modern architecture: designed to work seamlessly with Jetpack components like ViewModel and LiveData

<https://developer.android.com/topic/libraries/architecture/datastore>

3. DataStore

The following example stores two values (and String and Boolean) in a data store called "settings"

The keyword **suspend** indicates that the function is a coroutine, so it can be paused and resumed, allowing it to perform asynchronous operations without blocking the main thread

```
const val DATASTORE_NAME = "settings"
val USER_NAME_KEY = stringPreferencesKey("user_name")
val IS_ENABLED_KEY = booleanPreferencesKey("enabled")

val Context.dataStore: DataStore<Preferences> by preferencesDataStore(name = DATASTORE_NAME)

class DataStoreHelper(private val context: Context) {

    suspend fun saveUserName(name: String) {
        context.dataStore.edit { preferences ->
            preferences[USER_NAME_KEY] = name
        }
    }

    val userName: Flow<String> = context.dataStore.data
        .map { preferences ->
            preferences[USER_NAME_KEY] ?: ""
        }

    suspend fun saveEnabled(enabled: Boolean) {
        context.dataStore.edit { preferences ->
            preferences[IS_ENABLED_KEY] = enabled
        }
    }

    val enabled: Flow<Boolean> = context.dataStore.data
        .map { preferences ->
            preferences[IS_ENABLED_KEY] ?: false
        }
}
```

3. DataStore

We use a `ViewModel` to interact with the previous `DataStoreHelper` to manage and persist user settings. It uses Kotlin's `StateFlow` to expose these settings to the UI in a reactive way

```
class SettingsViewModel(private val dataStoreHelper: DataStoreHelper) : ViewModel() {  
  
    private val _userName = MutableStateFlow("")  
    val userName: StateFlow<String> get() = _userName  
  
    private val _isEnabled = MutableStateFlow(false)  
    val isEnabled: StateFlow<Boolean> get() = _isEnabled  
  
    init {  
        // Observe DataStore changes  
        viewModelScope.launch {  
            dataStoreHelper.userName.collectLatest { name ->  
                _userName.value = name  
            }  
            viewModelScope.launch {  
                dataStoreHelper.enabled.collectLatest { enabled ->  
                    _isEnabled.value = enabled  
                }  
            }  
        }  
  
        fun saveUserName(name: String) {  
            viewModelScope.launch {  
                dataStoreHelper.saveUserName(name)  
            }  
        }  
  
        fun saveEnabled(enabled: Boolean) {  
            viewModelScope.launch {  
                dataStoreHelper.saveEnabled(enabled)  
            }  
        }  
    }  
}
```

When a `ViewModel` has dependencies (e.g., `dataStoreHelper`), we need a way to pass those dependencies to the `ViewModel` during its creation

3. DataStore

A way to provide ViewModel dependencies (e.g. `dataStoreHelper` in this example) is through a `ViewModelProvider.Factory` to customize the creation of ViewModel instances

```
class SettingsViewModelFactory(
    private val dataStoreHelper: DataStoreHelper
) : ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(SettingsViewModel::class.java)) {
            @Suppress("UNCHECKED_CAST")
            return SettingsViewModel(dataStoreHelper) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

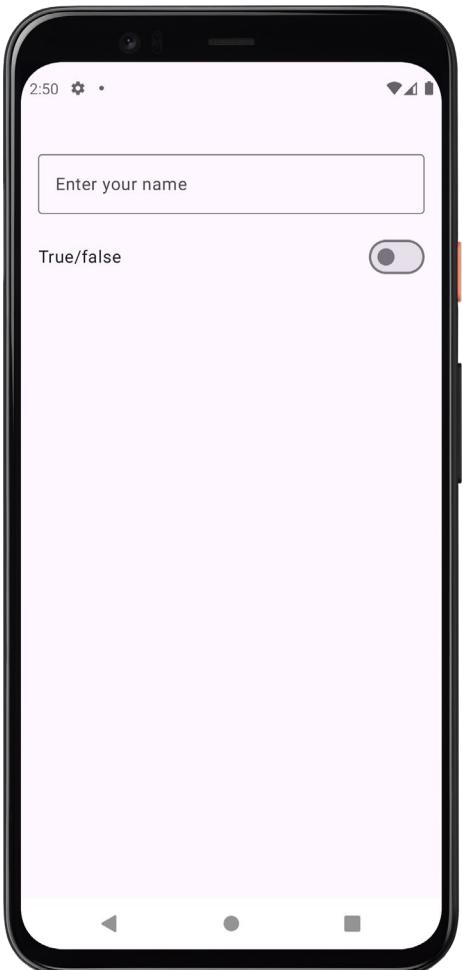
- While custom factories work, they can become verbose when having many ViewModel classes with dependencies. Modern alternatives include:
 - [Hilt](#): A dependency injection library that simplifies ViewModel creation
 - [Koin](#): A lightweight dependency injection framework for Kotlin

3. DataStore

```
@Composable
fun SettingsScreen(
    modifier: Modifier = Modifier, viewModel: SettingsViewModel = viewModel(
        factory = SettingsViewModelFactory(
            DataStoreHelper(LocalContext.current)
        )
    )
) {
    val userName by viewModel.userName.collectAsState()
    val enabled by viewModel.isEnabled.collectAsState()

    Column(
        modifier = modifier
            .fillMaxSize()
            .padding(16.dp)
    ) {
        OutlinedTextField(
            value = userName,
            onValueChange = { viewModel.saveUserName(it) },
            label = { Text(stringResource(R.string.enter_name)) },
            modifier = Modifier.fillMaxWidth()
        )
        Spacer(modifier = Modifier.height(16.dp))
        Row(
            verticalAlignment = Alignment.CenterVertically, modifier = Modifier.fillMaxWidth()
        ) {
            Text(text = stringResource(R.string.enabled), modifier = Modifier.weight(1f))
            Switch(
                checked = enabled, onCheckedChange = { viewModel.saveEnabled(it) }
            )
        }
    }
}
```

Finally, we use
the ViewModel
in our UI



Fork me on GitHub

Table of contents

1. Introduction
2. Firebase
3. DataStore
4. Files
 - Internal storage
 - External storage
5. Local database
6. Content providers
7. Takeaways

4. Files

- Managing files in Android using Kotlin and Jetpack Compose involves interacting with the Android file system to read and write files
- Android provides different APIs for file management, such as:
 1. **Internal storage:** private to the app
 2. **External storage:** shared storage accessible by other apps (requires permissions)

<https://developer.android.com/training/data-storage>

4. Files - Internal storage

- For each app, the Android system provides directories within internal storage where an app can organize its files:
 - One directory is designed for our app's persistent files
 - Other directory contains our app's temporal files (cache)
- In either case, the app doesn't require any system permissions to read and write to files in these directories
- There are different ways to manage files in the internal storage:
 - Using the File API
 - Using context methods (e.g., `openFileInput()` to read, `openFileOutput()` to write, etc.)

<https://developer.android.com/training/data-storage/app-specific>

4. Files - Internal storage

```
class FileHelper(private val context: Context) {  
  
    // Write to a file in internal storage  
    fun writeToFile(fileName: String, content: String): Boolean {  
        return try {  
            context.openFileOutput(fileName, Context.MODE_PRIVATE).use { stream ->  
                stream.write(content.toByteArray())  
            }  
            true  
        } catch (e: IOException) {  
            Toast.makeText(context, e.message, Toast.LENGTH_LONG).show()  
            false  
        }  
    }  
  
    // Read from a file in internal storage  
    fun readFromFile(fileName: String): String {  
        return try {  
            context.openFileInput(fileName).bufferedReader().use { reader ->  
                reader.readText()  
            }  
        } catch (e: IOException) {  
            Toast.makeText(context, e.message, Toast.LENGTH_LONG).show()  
            ""  
        }  
    }  
  
    // List all files in internal storage  
    fun listFiles(): Array<String> {  
        return context fileList()  
    }  
  
    // Delete a file in internal storage  
    fun deleteFile(fileName: String): Boolean {  
        return context.deleteFile(fileName)  
    }  
}
```

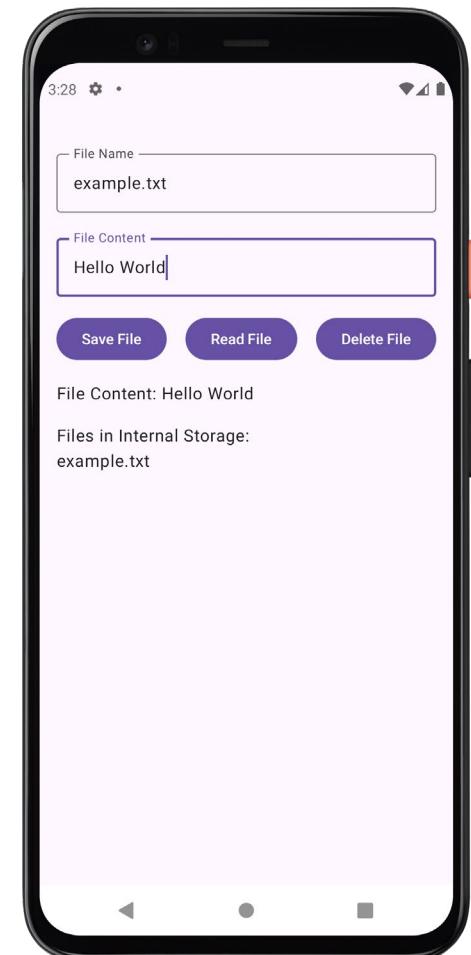
This example encapsulates the access to the internal storage in a helper class

Fork me on GitHub

4. Files - Internal storage

The helper class is used in a ViewModel, which is then observed in the UI

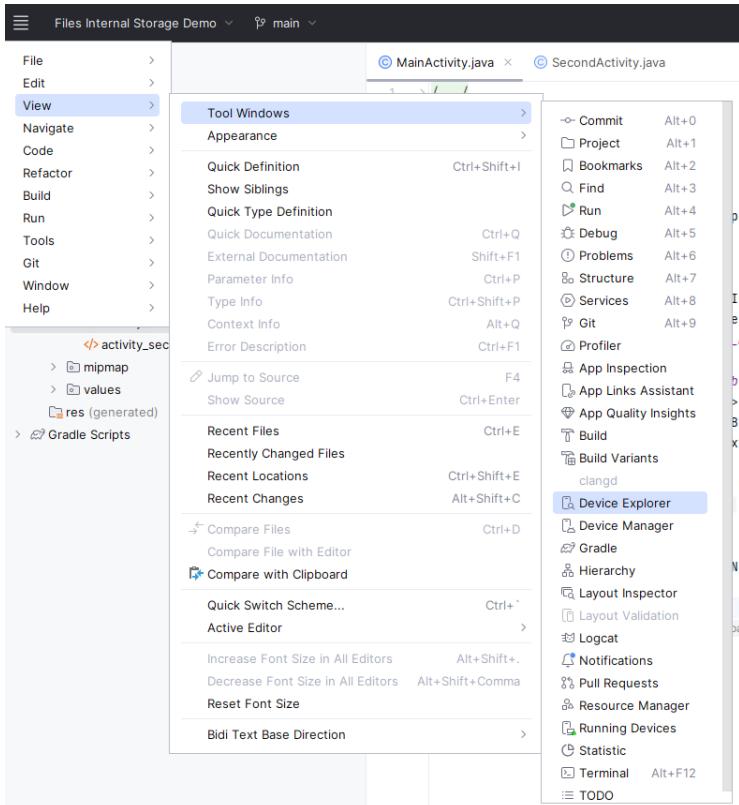
```
class FileViewModel(private val fileHelper: FileHelper) : ViewModel() {  
  
    private val _fileContent = MutableStateFlow("")  
    val fileContent: StateFlow<String> get() = _fileContent  
  
    private val _fileList = MutableStateFlow<List<String>>(emptyList())  
    val fileList: StateFlow<List<String>> get() = _fileList  
  
    fun writeToFile(fileName: String, content: String) {  
        viewModelScope.launch {  
            val success = fileHelper.writeToFile(fileName, content)  
            if (success) {  
                refreshFileList()  
            }  
        }  
    }  
  
    fun readFromFile(fileName: String) {  
        viewModelScope.launch {  
            _fileContent.value = fileHelper.readFromFile(fileName)  
        }  
    }  
  
    fun refreshFileList() {  
        viewModelScope.launch {  
            _fileList.value = fileHelper.listFiles().toList()  
        }  
    }  
  
    fun deleteFile(fileName: String) {  
        viewModelScope.launch {  
            val success = fileHelper.deleteFile(fileName)  
            if (success) {  
                refreshFileList()  
            }  
        }  
    }  
}
```



Fork me on GitHub

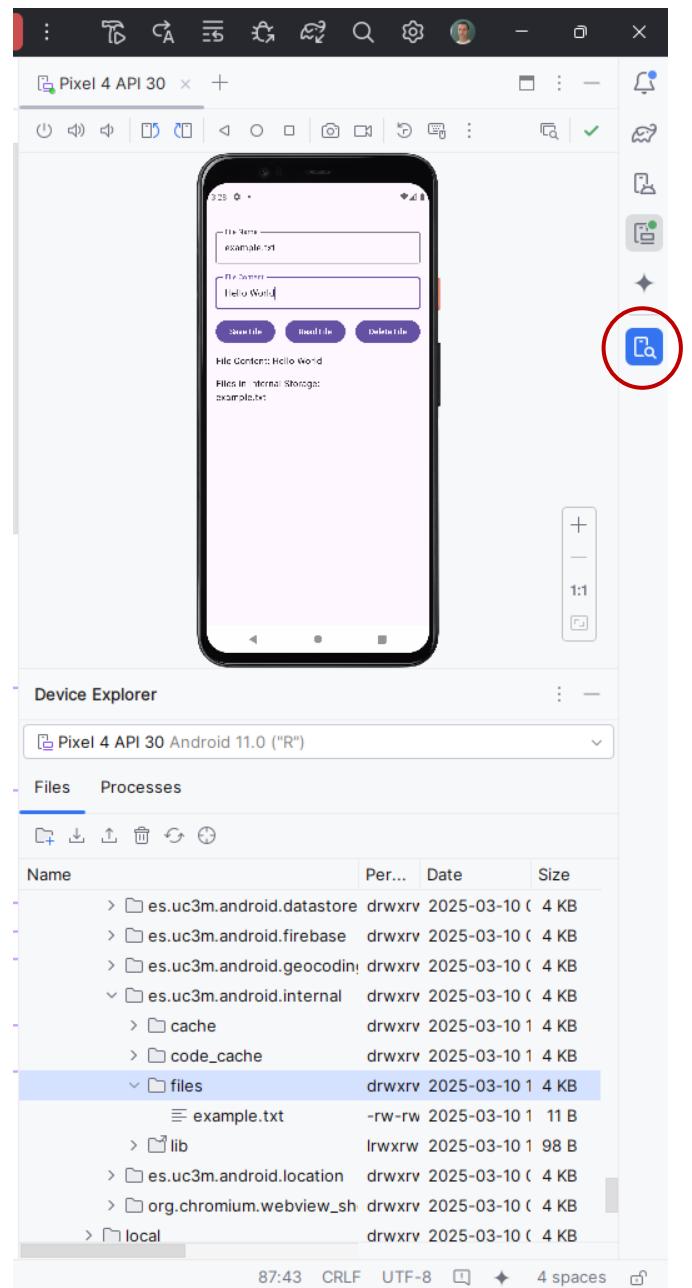
4. Files - Internal storage

- The path for storing files in the internal storage is: /data/data/<app_id>/files



This explorer can be opened
using View→Tool
Windows→Device Explorer

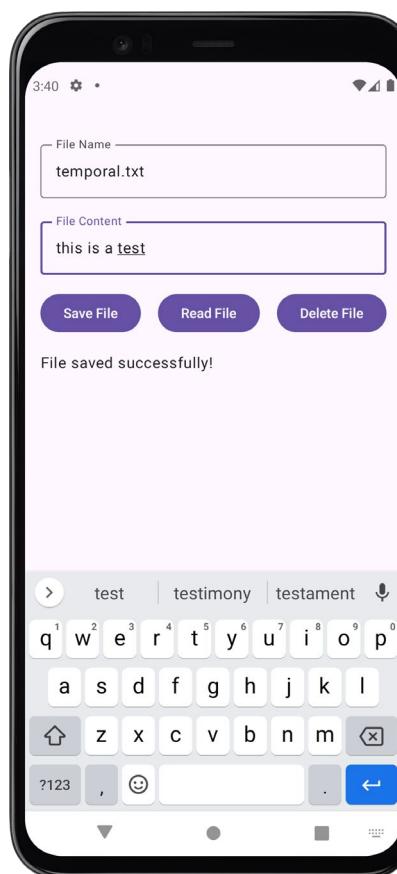
We can use the integrated
Device Explorer in Android
Studio to browser these files



4. Files - Internal storage

```
class CacheFileHelper(private val context: Context) {  
  
    // Write to a file in the cache directory  
    fun writeToCache(fileName: String, content: String): Boolean {  
        return try {  
            val file = File(context.cacheDir, fileName)  
            FileOutputStream(file).use { stream ->  
                stream.write(content.toByteArray())  
            }  
            true  
        } catch (e: IOException) {  
            e.printStackTrace()  
            false  
        }  
    }  
  
    // Read from a file in the cache directory  
    fun readFromCache(fileName: String): String {  
        return try {  
            val file = File(context.cacheDir, fileName)  
            file.bufferedReader().use { reader ->  
                reader.readText()  
            }  
        } catch (e: IOException) {  
            e.printStackTrace()  
            ""  
        }  
    }  
  
    // Delete a file from the cache directory  
    fun deleteFromCache(fileName: String): Boolean {  
        val file = File(context.cacheDir, fileName)  
        return file.delete()  
    }  
}
```

In the examples repository, you can find another app that uses the temporal internal storage (cache)



The path for storing temporal files is:
`/data/data/<app_id>/cache`

Device Explorer

Pixel 4 API 30 Android 11.0 ("R")

Files Processes

Name	P...	Date	Size
> com.google.mainline.telen	drw:	2025-03-10 00:00:00	4 KB
< es.uc3m.android.cache	drw:	2025-03-10 00:00:00	4 KB
< cache	drw:	2025-03-10 15:3	4 KB
temporal.txt	-rw-	2025-03-10 15:3	14 B
code_cache	drw:	2025-03-10 15:3	4 KB
files	drw:	2025-03-10 15:3	4 KB
lib	lrwx	2025-03-10 15:3	98 B
> es.uc3m.android.datastore	drw:	2025-03-10 00:00:00	4 KB
> es.uc3m.android.firebaseio	drw:	2025-03-10 00:00:00	4 KB
> es.uc3m.android.geocodin	drw:	2025-03-10 00:00:00	4 KB
> es.uc3m.android.internal	drw:	2025-03-10 00:00:00	4 KB
> es.uc3m.android.location	drw:	2025-03-10 00:00:00	4 KB

Fork me on GitHub

4. Files - External storage

- For reading/writing in the external storage of an Android device, the first thing we need to do in our app project is to specify a system permission that the user must grant
- This is done using the tag **uses-permission** in the manifest file:

```
<uses-permission  
    android:name="android.permission.READ_EXTERNAL_STORAGE"  
    android:maxSdkVersion="32" />  
<uses-permission  
    android:name="android.permission.WRITE_EXTERNAL_STORAGE"  
    android:maxSdkVersion="32"  
    tools:ignore="ScopedStorage" />
```

Starting from Android 13 (API level 33), these permissions are no longer required for accessing shared files due to the introduction of scoped storage and more granular permissions

<https://developer.android.com/training/data-storage/shared>

4. Files - External storage

```
class ExternalStorageHelper(private val context: Context) {  
  
    // Get the public documents directory  
    fun getPublicDocumentsDir(): File? {  
        return Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOCUMENTS)  
    }  
  
    // Write to a file in external storage  
    fun writeToExternalStorage(fileName: String, content: String): Boolean {  
        return try {  
            val documentsDir = getPublicDocumentsDir()  
            if ((documentsDir != null) && !documentsDir.exists()) {  
                documentsDir.mkdirs()  
            }  
            val file = File(documentsDir, fileName)  
            FileOutputStream(file).use { stream ->  
                stream.write(content.toByteArray())  
            }  
            true  
        } catch (e: IOException) {  
            Toast.makeText(context, e.message, Toast.LENGTH_LONG).show()  
            false  
        }  
    }  
  
    // Read from a file in external storage  
    fun readFromExternalStorage(fileName: String): String {  
        return try {  
            val documentsDir = getPublicDocumentsDir()  
            val file = File(documentsDir, fileName)  
            Toast.makeText(context, file.getAbsolutePath, Toast.LENGTH_LONG).show()  
            file.bufferedReader().use { reader ->  
                reader.readText()  
            }  
        } catch (e: IOException) {  
            Toast.makeText(context, e.message, Toast.LENGTH_LONG).show()  
            ""  
        }  
    }  
}
```

In this example, we use another helper class to encapsulate all the access to the external storage

Fork me on GitHub

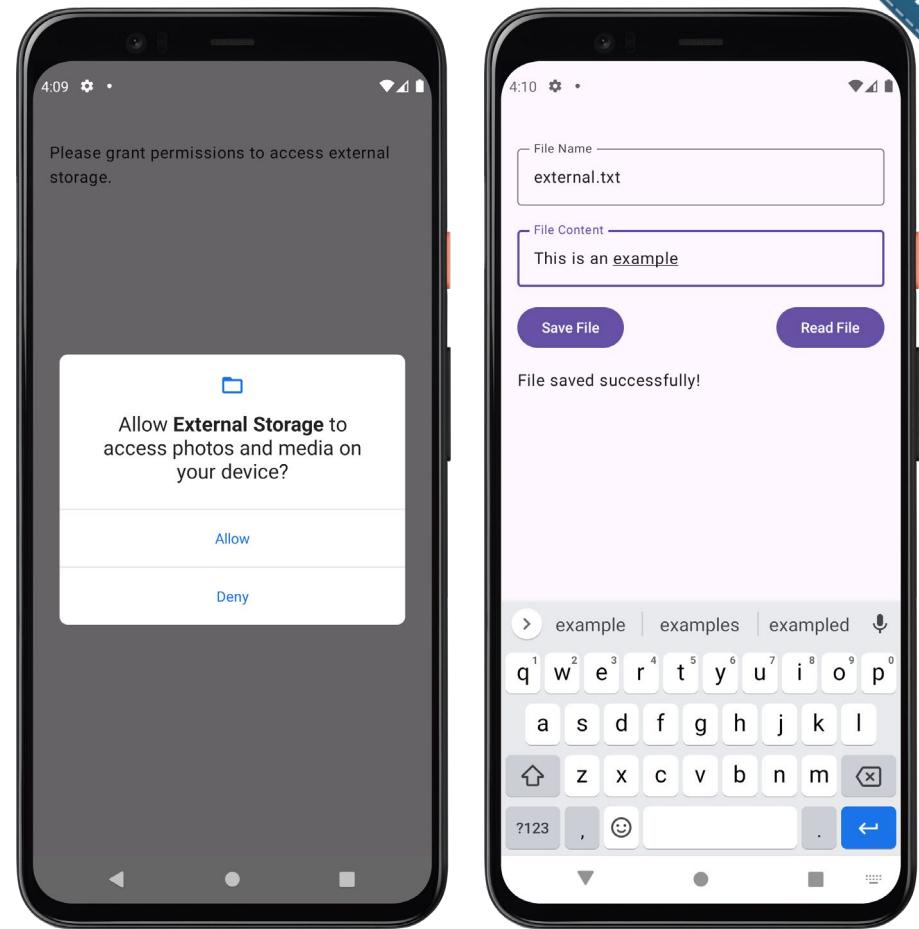
4. Files - External storage

```
@Composable
fun RequestPermissions(
    onPermissionsGranted: () -> Unit, onPermissionsDenied: () -> Unit
) {
    val permissions = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        arrayOf(Manifest.permission.READ_MEDIA_IMAGES, Manifest.permission.READ_MEDIA_VIDEO)
    } else {
        arrayOf(
            Manifest.permission.READ_EXTERNAL_STORAGE, Manifest.permission.WRITE_EXTERNAL_STORAGE
        )
    }

    val launcher = rememberLauncherForActivityResult(
        contract = ActivityResultContracts.RequestMultiplePermissions()
    ) { permissionsMap ->
        val allGranted = permissionsMap.values.all { it }
        if (allGranted) {
            onPermissionsGranted()
        } else {
            onPermissionsDenied()
        }
    }

    LaunchedEffect(Unit) {
        launcher.launch(permissions)
    }
}
```

We need to implement the permissions grant in our composable logic

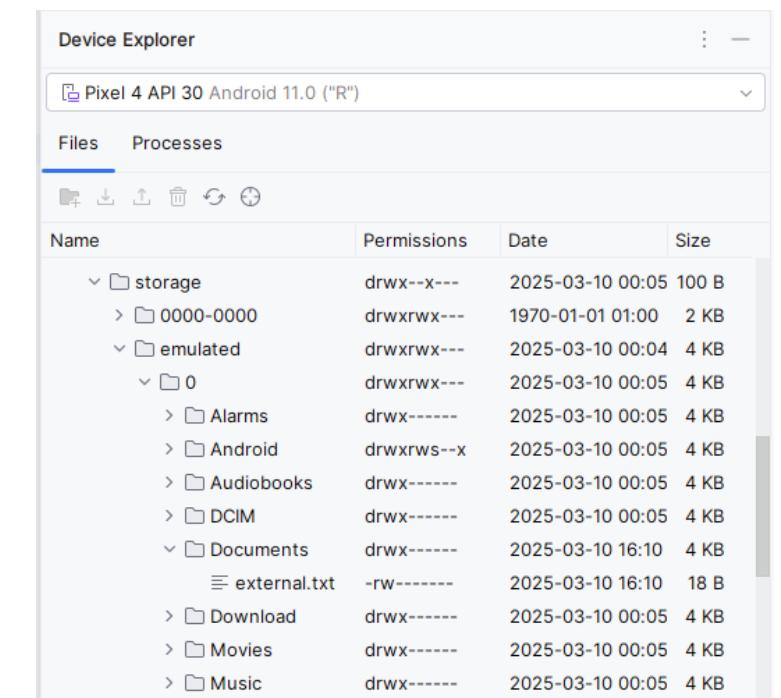


Fork me on GitHub

4. Files - External storage

- The paths for public directories are standardized and can be accessed using the Environment class:

Directory	Path
Downloads	/storage/emulated/0/Download
Pictures	/storage/emulated/0/Pictures
Music	/storage/emulated/0/Music
Movies	/storage/emulated/0/Movies
Documents	/storage/emulated/0/Documents
Camera	/storage/emulated/0/DCIM



Fork me on GitHub

Table of contents

1. Introduction
2. Firebase
3. DataStore
4. Files
5. Local databases
 - Relational databases
 - SQLite
 - Room persistence library
 - Database inspector
 - Preloaded database
 - Foreign keys
6. Content providers
7. Takeaways

5. Local database - Relational databases

- A relational database (RDBMS) stores data as a set of **tables** with columns and rows following the relational model (RM), which was postulated in 1970 by Edgar Frank Codd at IBM laboratories:
 - Tables store information about the objects being represented
 - Each column of a table stores a certain type of data
 - The rows (or record) of the table represent a collection of related values of an object or entity
 - A field stores the value of a row and column
 - Each row in a table usually have a unique identifier called a primary key
 - Rows from different tables can be related to foreign keys

Id	Name	AlterEgo
1	Superman	Clark Kent
2	Spiderman	Peter Parker
3	Hulk	Bruce Banner

SuperHeroes

IdHero	IdUniverse
1	2
2	1
3	1

SuperHeroesUniverse

Id	Name
1	Marvel
2	DC

Universe

5. Local database - Relational databases

- **SQL** (Standard Query Language) is a standard language that allows us managing a DBMS
- SQL commands fall into two categories:
 - Data Manipulation Language (DML). Allows performing basic operations on data (CRUD = create, read, update, delete): `SELECT`, `INSERT`, `DELETE`, `UPDATE`
 - Data Definition Language (DDL). Allows you to create, delete and modify tables, users, views, or indexes: `CREATE TABLE`, `DROP TABLE`, `ALTER TABLE`

5. Local database - SQLite

- **SQLite** is a lightweight open-source relational database written in C
 - SQLite is included by default in Android devices
- As usual in RDBMS, we use SQL to manage the data and data definition in SQLite
- Each SQLite database is stored as a single binary file. In Android, these files are store in the following path:
`/data/data/<package_name>/databases`
- There are two main ways to manage SQLite databases in Android
 - Low-level: Through the class [SQLiteOpenHelper](#)
 - High-level: Through the **Room API**



<https://www.sqlite.org/>

5. Local database - Room persistence library

- A convenient way to manage SQLite databases from Android apps is through the **Room persistence library**
- This library provides different benefits, such as:
 - Compile-time verification of SQL queries
 - Custom annotations to map tables to Java
 - Improved support for database migration and preloaded databases

<https://developer.android.com/training/data-storage/room>

5. Local database - Room persistence library

- There are three major components in Room:
 1. Data **entities** that represent tables in the database
 - Classes annotated with `@Entity`, containing attributes annotated with `@PrimaryKey` (for setting the primary key), `@ForeignKey` (for specifying foreign keys), and `@ColumnInfo` (for further columns)
 2. **Data Access Objects (DAOs)** that provide methods that an app can use to query, update, insert, and delete data in the database
 - Interfaces annotated with `@Dao`, containing methods annotated with `@Query` (for reading data), `@Insert` (for inserting data), `@Update` (for updating data), and `@Delete` (for deleting)
 3. The **database class** that serves as the main access point with the local database
 - A single Java/Kotlin class annotated with `@Database`

5. Local database - Room persistence library

- The entity class used in the [Room](#) demo is as follows:

```
@Entity(tableName = "notes")
data class Note(
    @PrimaryKey(autoGenerate = true) val id: Int = 0, // Auto-generated ID
    val title: String,
    val body: String
)
```

This entity will be used to
manage a SQLite table like this

id	title	body

5. Local database - Room persistence library

- The **DAO interface** used in this demo is as follows:

```
@Dao
interface NoteDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(note: Note)

    @Update
    suspend fun update(note: Note)

    @Query("DELETE FROM notes WHERE id = :id")
    suspend fun delete(id: Int)

    @Query("SELECT * FROM notes ORDER BY id DESC")
    fun getAllNotes(): LiveData<List<Note>>
}
```

5. Local database - Room persistence library

- The database class used this demo is as follows:

We need specify all the entities (Kotlin classes)

We need specify abstract method using the DAO interfaces

```
@Database(entities = [Note::class], version = 1, exportSchema = false)
abstract class NoteDatabase : RoomDatabase() {
    abstract fun noteDao(): NoteDao

    companion object {
        @Volatile
        private var INSTANCE: NoteDatabase? = null

        fun getDatabase(context: Context): NoteDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    NoteDatabase::class.java,
                    "note_database.db"
                ).build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

We need to extend RoomDatabase

Creation of connection to the database using Room

5. Local database - Room persistence library

```
class NoteViewModel(application: Application) : AndroidViewModel(application) {
    private val noteDao = NoteDatabase.getDatabase(application).noteDao()
    val allNotes: LiveData<List<Note>> = noteDao.getAllNotes()

    fun addNote(title: String, body: String) {
        viewModelScope.launch {
            val note = Note(title = title, body = body)
            noteDao.insert(note)
        }
    }

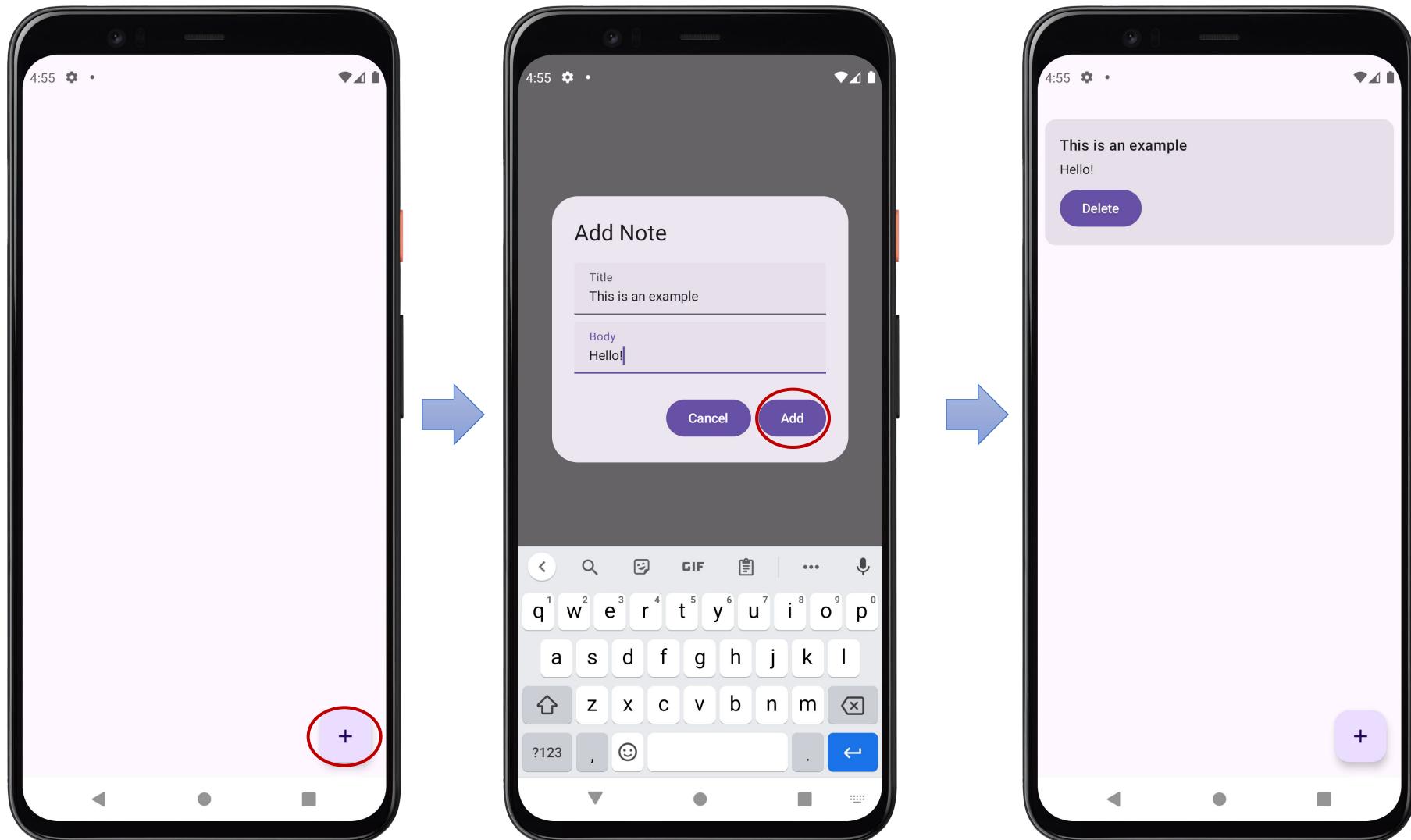
    fun updateNote(id: Int, title: String, body: String) {
        viewModelScope.launch {
            val note = Note(id = id, title = title, body = body)
            noteDao.update(note)
        }
    }

    fun deleteNote(id: Int) {
        viewModelScope.launch {
            noteDao.delete(id)
        }
    }
}
```

Finally, we access the database in our Kotlin logic (typically in a ViewModel class), invoking the DAO methods to manage the data

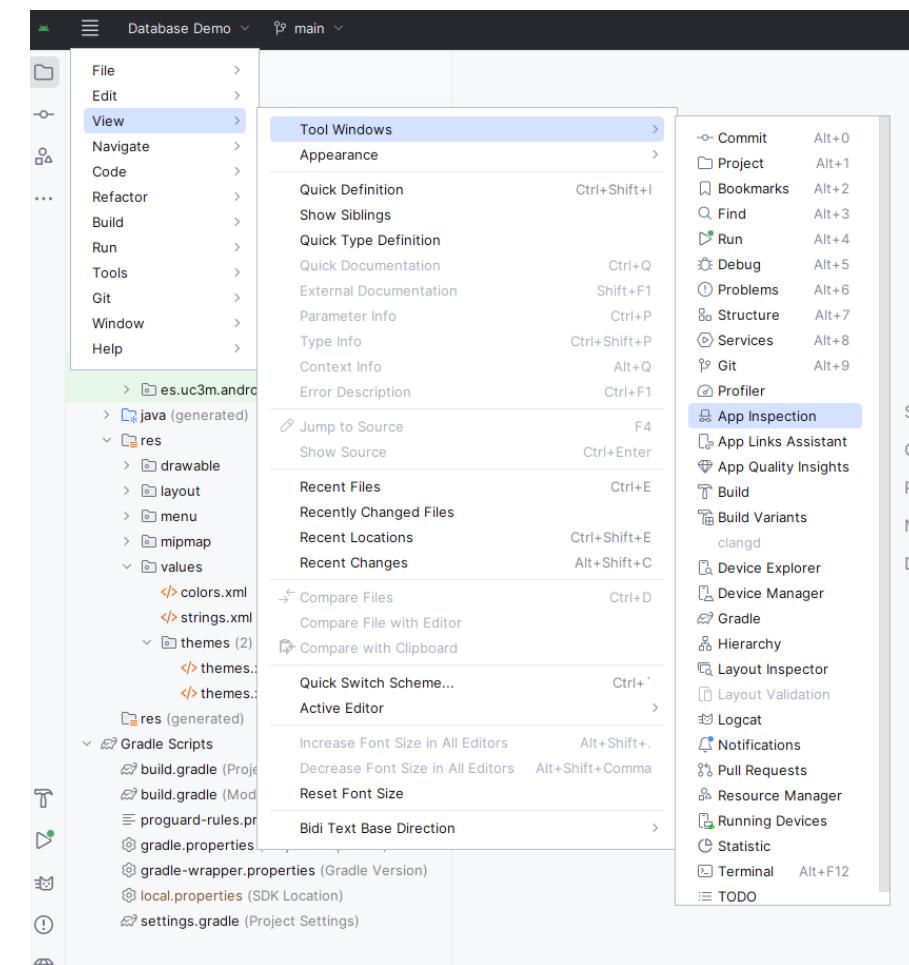
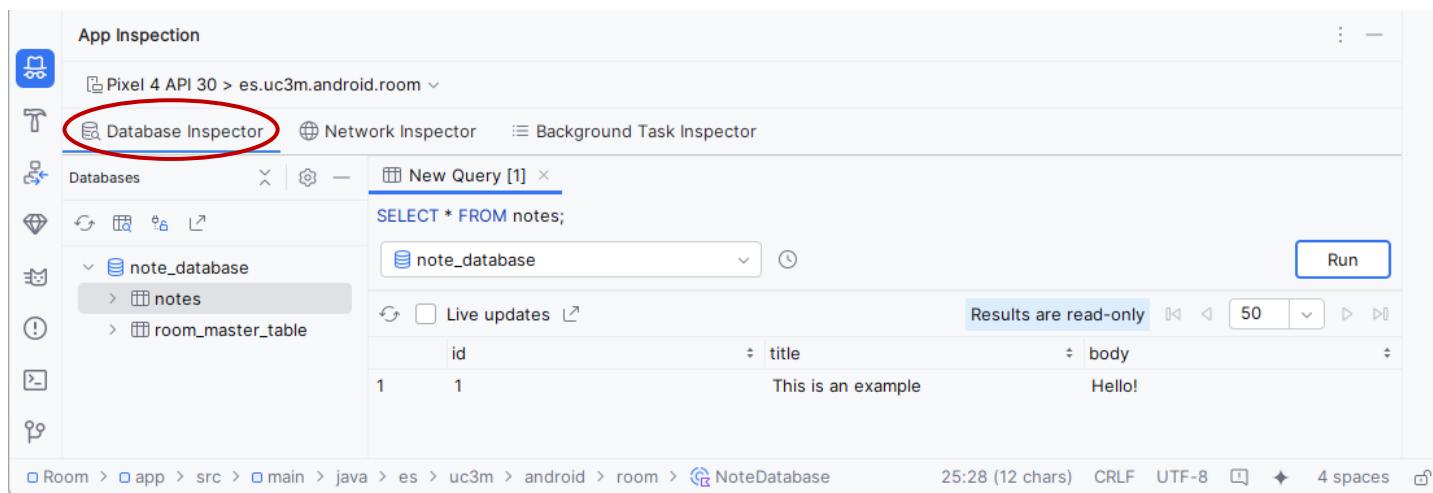
5. Local database - Room persistence library

Fork me on GitHub



5. Local database - Database inspector

- It is possible to browse the database using an integrated database inspector available in Android Studio (View → Tool Windows → App Inspection)
 - App Inspection → Database inspector



5. Local database - Preloaded database

- It is possible to create a SQLite database using an external tool and copy it to our Android app
 - This option can be interesting for having preloaded data in our apps, for example for the final project app
- The procedure to use this preloaded database can be as follows:
 1. Create SQLite database. For instance, using DB Browser for SQLite



<https://sqlitebrowser.org/>

2. Copy the SQLite database in an assets folder
3. Load database in our project (we can use shared preferences to do it only one time)

5. Local database - Preloaded database

1. Create SQLite database. For instance, using [DB Browser for SQLite](#):

The image shows two windows from the DB Browser for SQLite application.

Main Interface: The left window displays the application's main interface. It includes a toolbar with 'File', 'Edit', 'View', 'Tools', and 'Help' menus, and buttons for 'New Database', 'Open Database', 'Write Changes', 'Revert Changes', 'Open Project', 'Save Project', 'Attach Database', and 'Close Database'. Below the toolbar is a menu bar with 'Database Structure', 'Browse Data', 'Edit Pragmas', and 'Execute SQL'. A central pane shows a table structure with columns 'Name', 'Type', and 'Schema'. A modal dialog titled 'Edit Database Cell' is open, showing a text input field containing '1' and a toolbar below it. A 'Remote' connection dialog is also visible, showing a list of databases under 'DBHub.io'.

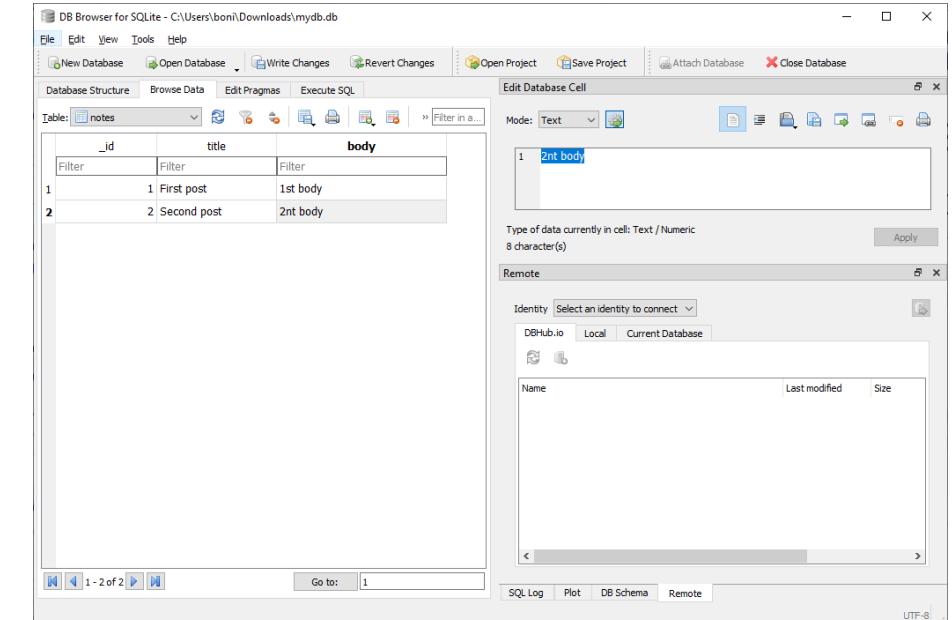
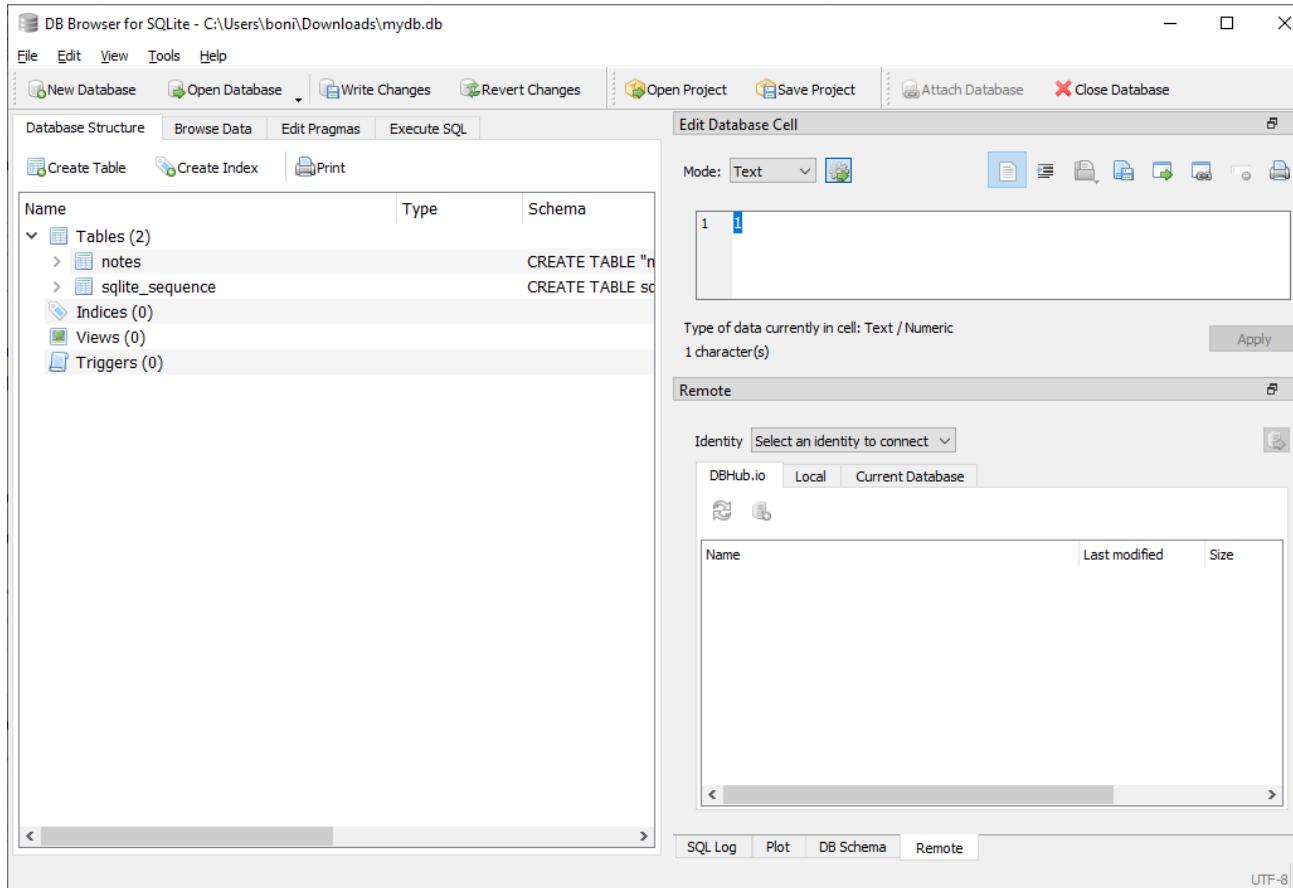
Edit Table Definition Dialog: The right window is a modal dialog titled 'Edit table definition' for a table named 'notes'. It has tabs for 'Fields' and 'Constraints'. Under 'Fields', there are three entries: '_id' (Type: INTEGER, PK: checked, AI: checked), 'title' (Type: TEXT), and 'body' (Type: TEXT). Below the table structure is a code editor showing the generated SQL code:

```
1 CREATE TABLE "notes" (
2     "_id" INTEGER,
3     "title" TEXT,
4     "body" TEXT,
5     PRIMARY KEY("_id" AUTOINCREMENT)
6 );
```

At the bottom of the dialog are 'OK' and 'Cancel' buttons.

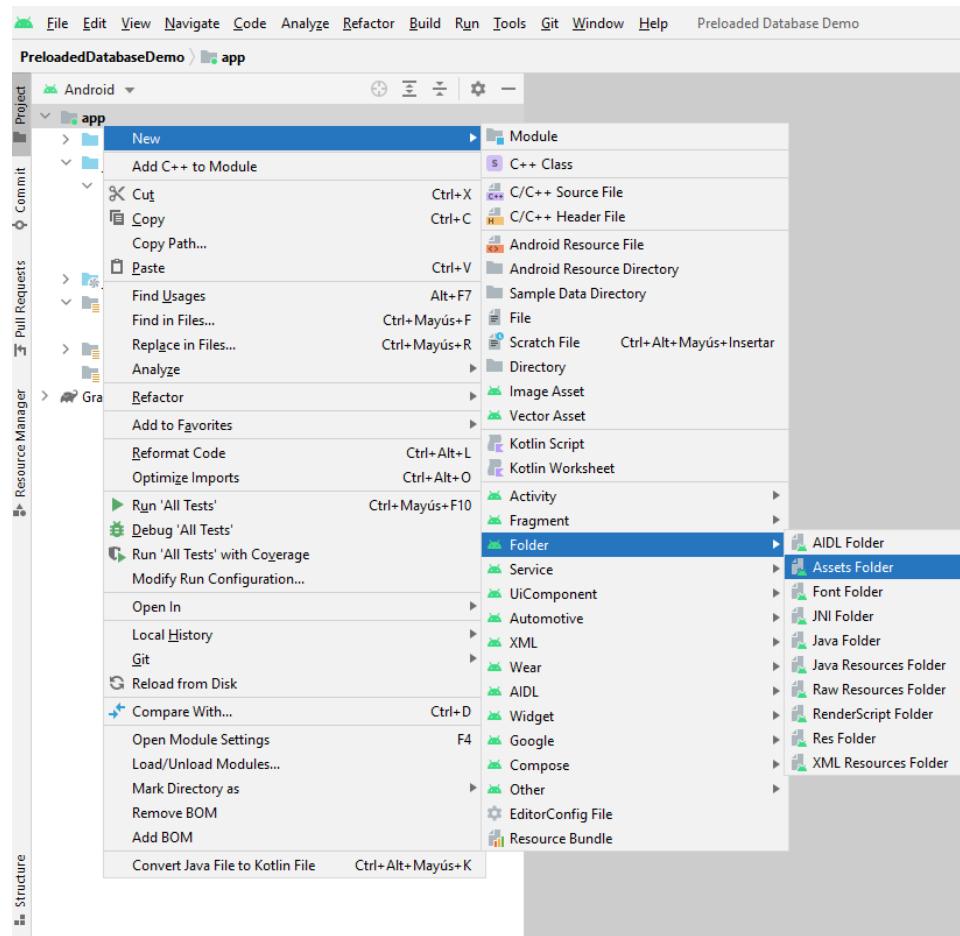
5. Local database - Preloaded database

1. Create SQLite database. For instance, using [DB Browser for SQLite](#):



5. Local database - Preloaded database

2. Copy the SQLite database in an assets folder



We can use the Android Studio wizard New→Folder→Assets folder

5. Local database - Preloaded database

3. Load database in our project

- The room library eases the manipulation of a preloaded databases
- We need to put the preloaded database in the assets folder and invoke the method `createFromAsset` in the database class

```
@Database(entities = [Note::class], version = 1, exportSchema = false)
abstract class NoteDatabase : RoomDatabase() {
    abstract fun noteDao(): NoteDao

    companion object {
        @Volatile
        private var INSTANCE: NoteDatabase? = null

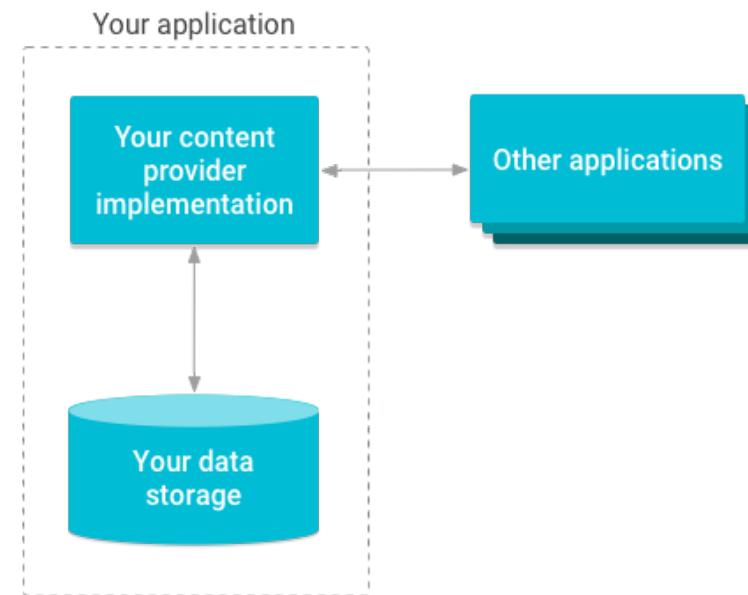
        fun getDatabase(context: Context): NoteDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    NoteDatabase::class.java,
                    "note_database.db"
                ).createFromAsset("preloaded_notes.db")
                    .build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

Table of contents

1. Introduction
2. Firebase
3. DataStore
4. Files
5. Local database
6. Content providers
7. Takeaways

6. Content providers

- A content provider is type of app component in Android (together with activities, broadcast receivers, and services) aimed to share data between different apps



<https://developer.android.com/guide/topics/providers/content-providers>

6. Content providers

- A content provider behaves like a distributed database where other apps can carry out CRUD (create, read, update, and delete) operations
 - Using the methods: `insert()`, `query()`, `update()`, and `delete()`
- To create a content provider in Android, we need:
 1. Create a subclass of `ContentProvider`
 2. Register it in the Manifest (using the tag **provider**)
- The demo app [ContentProvider](#) provides a basic example using a content provider
 - It is an app that exposed CRUD operations on a basic table through a content provider

6. Content providers

- A content provider is identified in Android using a content **URI** (*Uniform Resource Identifier*)
 - An URI is a unique sequence of characters that identifies a logical or physical resource
 - URL (Uniform Resource Locators) is subsets of URIs aimed to identify web resources
- The structure of a content URI in Android is as follows:

content://provider-name/path



6. Content providers

Fork me on GitHub

Method to initialize the provider

Method to **read** data from the provider

Method to **write** data into the provider

Method to **update** data from the provider

Method to **delete** data from the provider

Method to return the MIME type corresponding to a content URI

```
class MyContentProvider : ContentProvider() {  
  
    private lateinit var dbHelper: MyDatabaseHelper  
  
    override fun onCreate(): Boolean {  
        dbHelper = MyDatabaseHelper(context!!)  
        return true  
    }  
  
    override fun query(  
        uri: Uri,  
        projection: Array<String>?,  
        selection: String?,  
        selectionArgs: Array<String>?,  
        sortOrder: String?  
    ): Cursor {  
        val db = dbHelper.readableDatabase  
        return db.query(TABLE_NAME, projection, selection, selectionArgs, null, null, sortOrder)  
    }  
  
    override fun insert(uri: Uri, values: ContentValues?): Uri {  
        val db = dbHelper.writableDatabase  
        val id = db.insert(TABLE_NAME, null, values)  
        return ContentUris.withAppendedId(uri, id)  
    }  
  
    override fun update(  
        uri: Uri, values: ContentValues?, selection: String?, selectionArgs: Array<String>?  
    ): Int {  
        val db = dbHelper.writableDatabase  
        return db.update(TABLE_NAME, values, selection, selectionArgs)  
    }  
  
    override fun delete(uri: Uri, selection: String?, selectionArgs: Array<String>?): Int {  
        val db = dbHelper.writableDatabase  
        return db.delete(TABLE_NAME, selection, selectionArgs)  
    }  
  
    override fun getType(uri: Uri): String {  
        return "vnd.android.cursor.dir/vnd.com.example.provider.${TABLE_NAME}"  
    }  
}
```

Table of contents

1. Introduction
2. Firebase
3. DataStore
4. Files
5. Databases
6. Content providers
7. Takeaways

7. Takeaways

- Firebase is a set of backend cloud computing services provided by Google
 - Cloud Firestore is a cloud-hosted NoSQL document database within Firebase
 - Firebase Authentication is a Firebase service provided that simplifies the process of adding user authentication to apps
- DataStore is lightweight mechanism for storing name/value data
- Also, we can use regular files for writing and reading persistent data in the internal or external storage
 - The user must grant permissions to access the external storage
- Android provides access to SQLite, which is a lightweight open-source relational database
 - The Room persistence library eases the management of SQLite for Android
 - We can use an external tool (like DB Browser for SQLite) to create a preloaded database to be included in our Android app project
- Content providers is a type of app component which allows to share data between different apps