Mobile Applications

4. Persistent storage in Android

Boni García

boni.garcia@uc3m.es

Telematic Engineering Department
School of Engineering

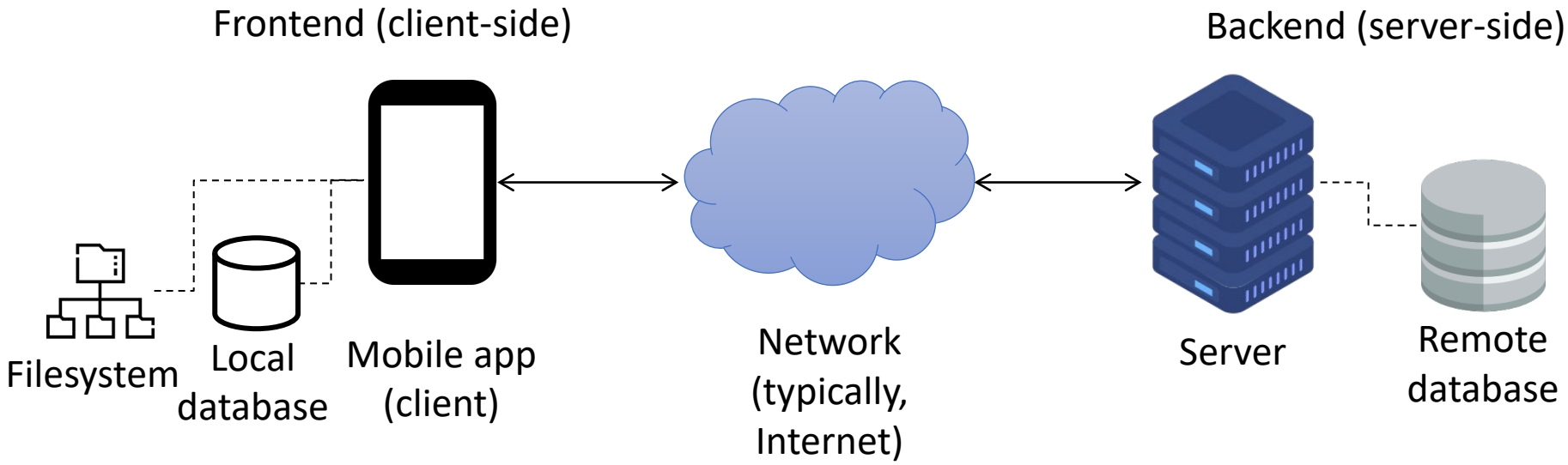
2025/2026

uc3m | Universidad Carlos III de Madrid

Table of contents

1. Introduction

- Mobile applications typically follow a **client-server architecture**:
  - Frontend (client-side): The mobile app running on the user's device
  - Backend (server-side): A remote system that stores data and provides services
- Data can be stored:
  - Locally on the device (file system or local database)
  - Remotely in the backend (remote database)



1. Introduction

- Common **backend** architectures for mobile applications:

1. Serverful. Developers write backend code and manage their own servers (on-premises or hosting)
   - Full control over backend logic. Requires deployment, monitoring, and maintenance
   - Examples: Express (Node.js), Spring Boot (Java), Django (Python)
2. Serverless. Developers write backend code but a cloud provider manages servers and scaling
   - No server management. Backend logic runs on demand. Automatic scaling
   - Examples: Google Cloud Functions, AWS Lambda, Azure Functions
3. Backend-as-a-Service (BaaS). A cloud platform that provides ready-to-use backend services, reducing the need to implement backend logic
   - Typical services: Cloud databases, authentication, file storage, notifications
   - Examples: **Firebase**, AWS Amplify, Supabase

Table of contents

2. Kotlin coroutines

- Android apps must keep the UI responsive, but some operations may take time, such as:
  - Network requests (e.g., Firebase), database queries, file operations
- If these operations run on the main thread (UI), the app may freeze
- Therefore, these tasks should be executed **asynchronously**
  - Without blocking the main thread, so the UI remains responsive while the operation completes in the background
- A common traditional solution is to use callbacks
  - For example: `addOnSuccessListener`, `addOnFailureListener`
- In Kotlin, the modern solution for asynchronous programming is **coroutines**
  - Coroutines let us write asynchronous code in a sequential and readable style

2. Kotlin coroutines

- A **coroutine** is a lightweight mechanism for asynchronous programming
- Key properties:
  - It can suspend execution without blocking the main thread
  - It can resume when the result is ready
  - It allows writing asynchronous code in a sequential style

```
firestore.collection("notes").get()
    .addOnSuccessListener { ... }
    .addOnFailureListener { ... }
```

Without coroutines (callbacks)

```
viewModelScope.launch {
    val result = firestore.collection("notes").get().await()
}
```

With coroutines (easier to read and maintain)

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

2. Kotlin coroutines

- A coroutine runs inside a **scope**, executes on a **dispatcher**, and call **suspending functions**:

1. Scope: defines where a coroutine lives and when it is cancelled, e.g.:
   - `viewModelScope`: used in `ViewModels`
   - `rememberCoroutineScope()`: used in composables to launch coroutines
2. Suspending function: A function that can pause execution while waiting for a result without blocking the thread
   - Defined in Kotlin using the `suspend` keyword
3. Dispatcher: determines which thread executes the coroutine, e.g.:
   - `Dispatchers.Main`: UI operations
   - `Dispatchers.IO`: network and database operations
   - `Dispatchers.Default`: CPU-intensive tasks

Many Android libraries (e.g., Firestore, Room) often handle threading internally

Table of contents

3. 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
- **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

https://cloud.google.com/

https://firebase.google.com/

3. 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

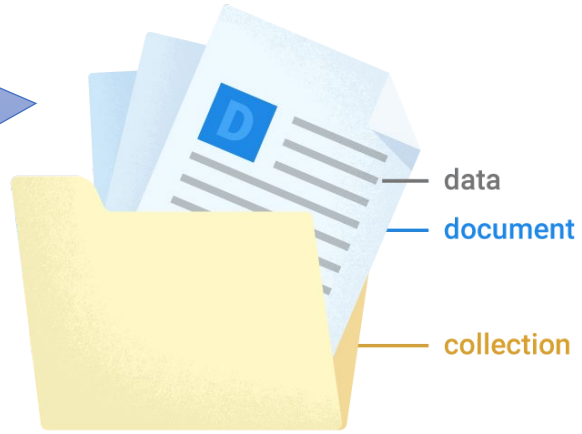
Most Firebase services have quotas and pricing based on the usage

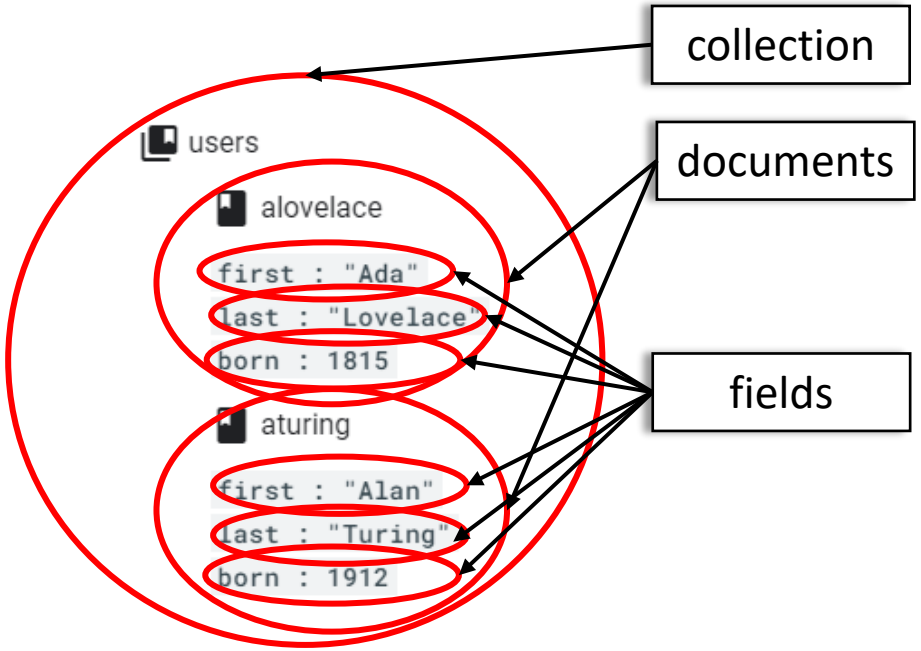
https://firebase.google.com/docs/

3. 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

Each document is uniquely identified by an automatically generated or user-defined id



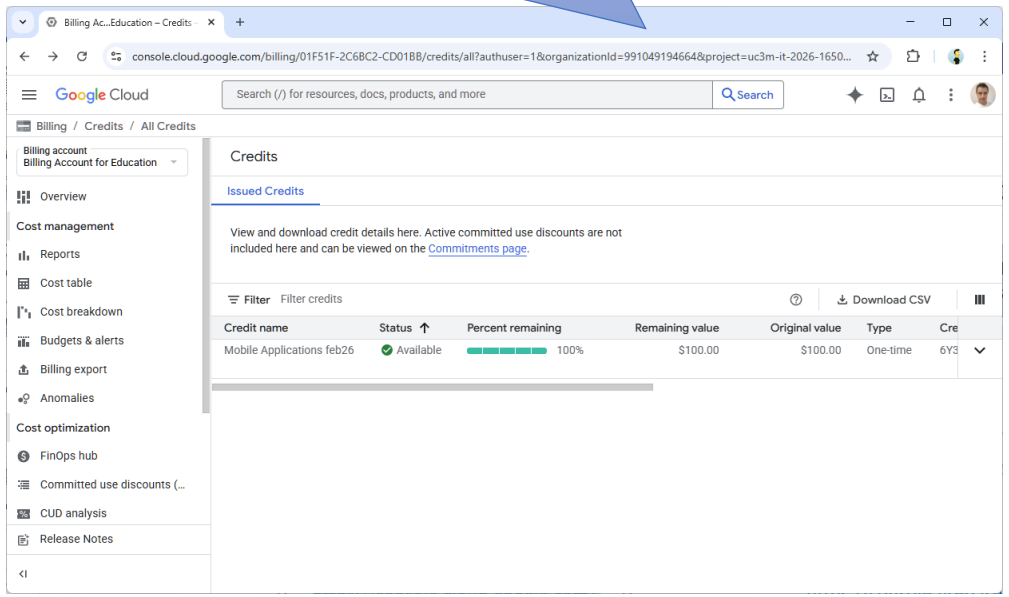


https://firebase.google.com/docs/firestore/data-model

3. Firebase - Cloud Firestore

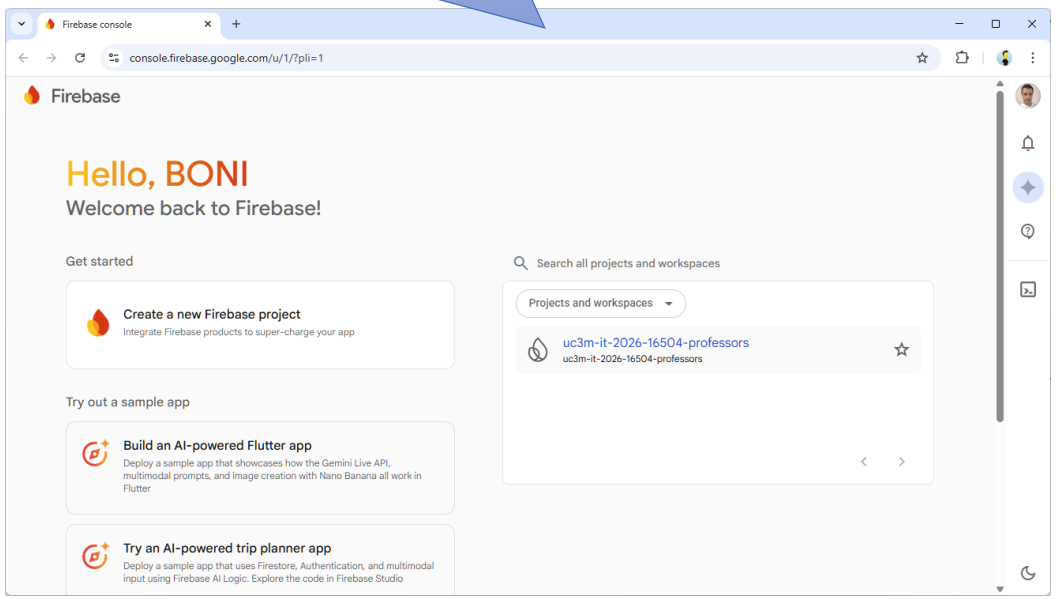
- Some of the Firebase services are also available through the Google Cloud console, for example, Cloud Firestore, Cloud Functions, or Cloud Storage

We used this console in the lab in to redeem the coupons



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

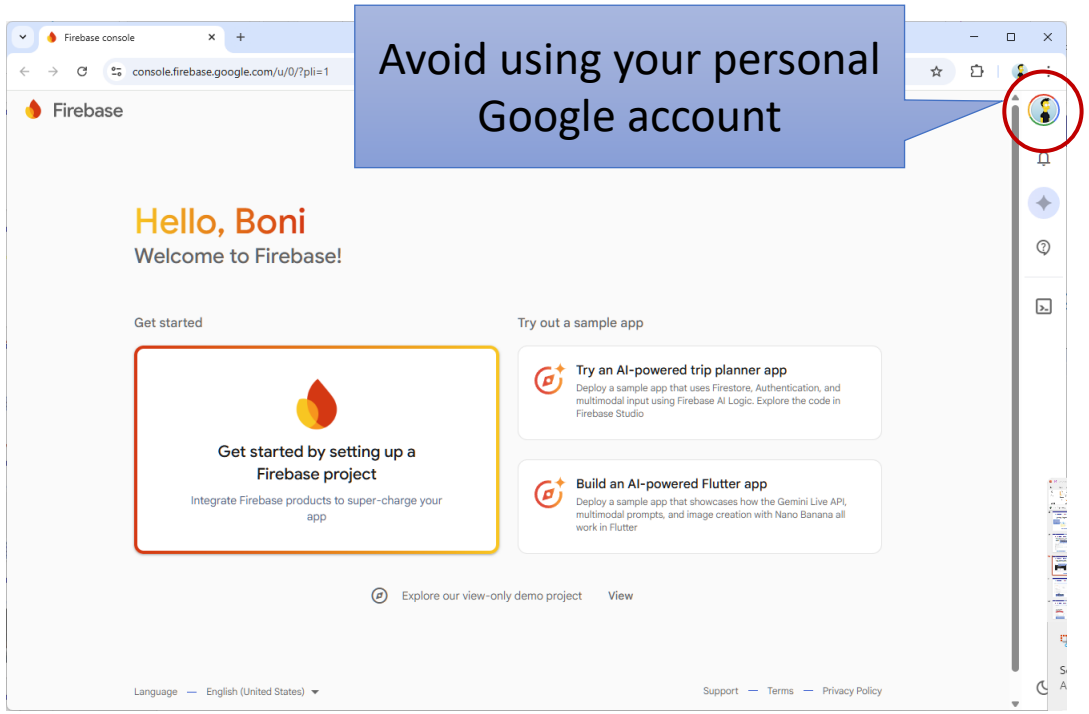
Now, we use the Firebase console in this unit

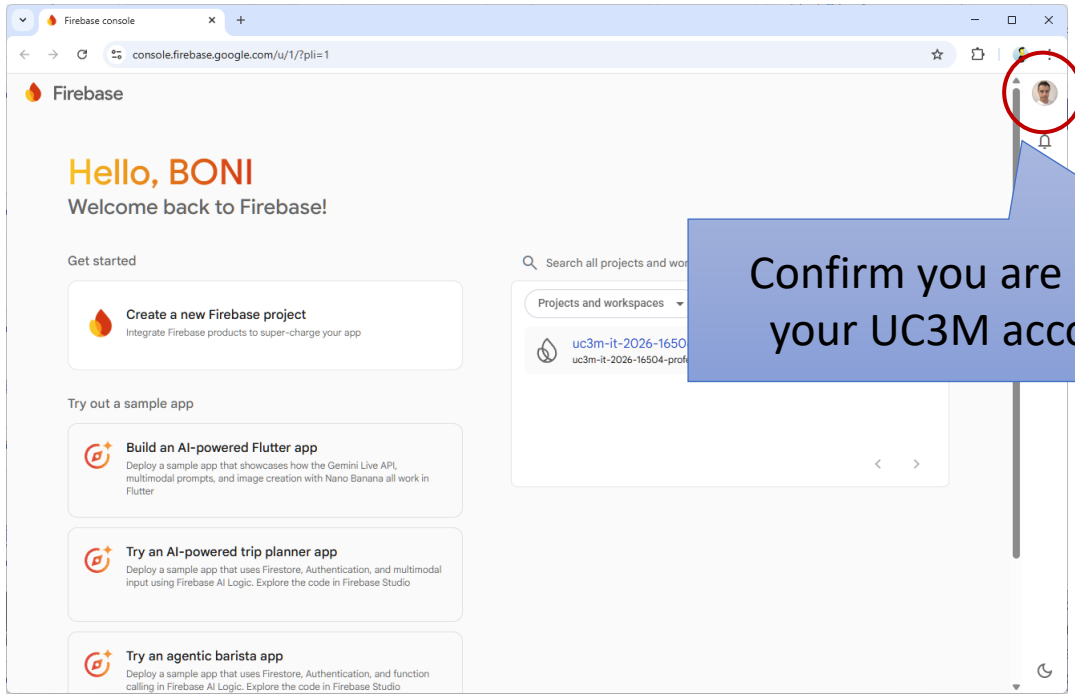


https://console.firebase.google.com/

3. Firebase - Cloud Firestore

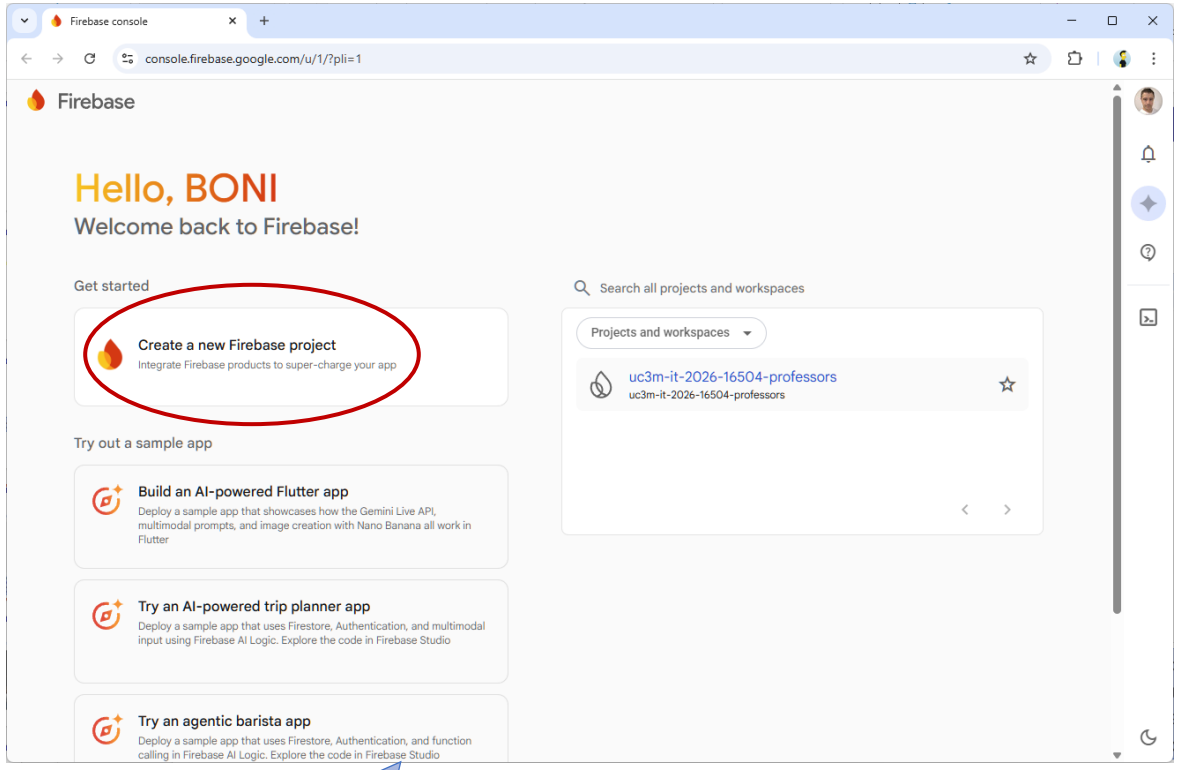
- To use Firebase, first we need a Google account
  - In this course, we should use our UC3M account (e.g. xxxxxxxxx@alumnos.uc3m.es)



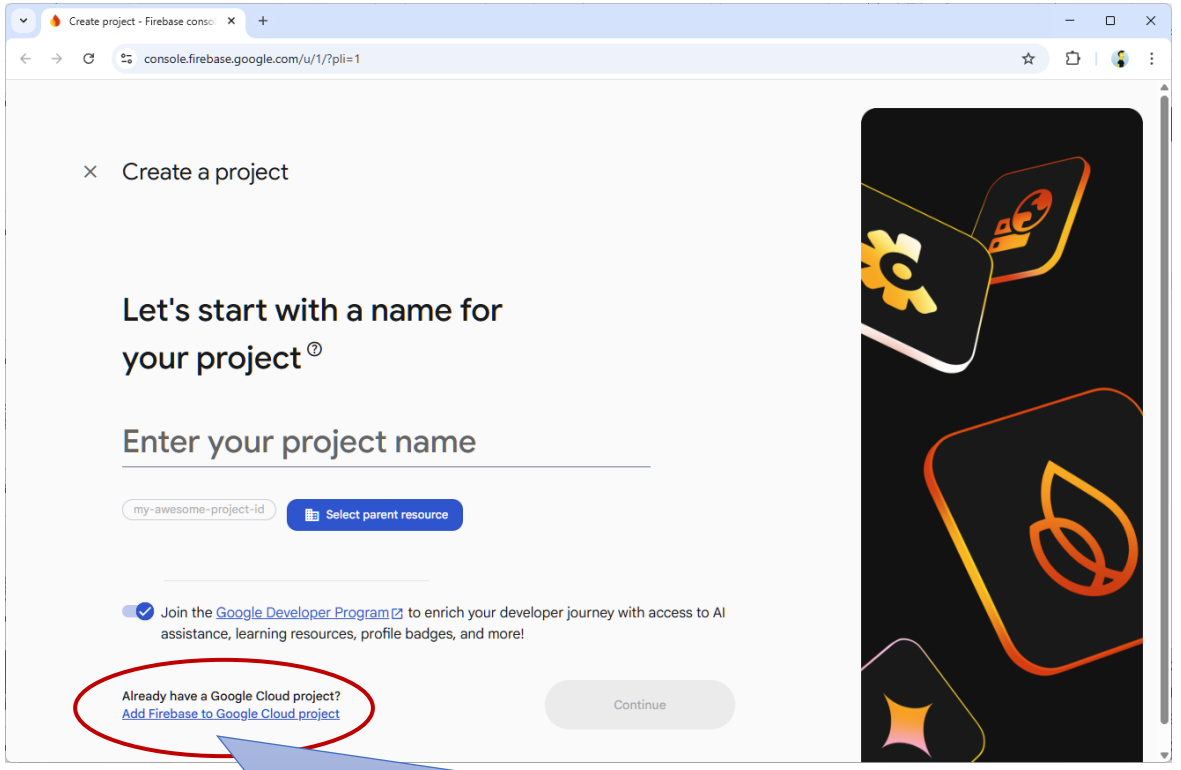


https://console.firebase.google.com/

3. Firebase - Cloud Firestore

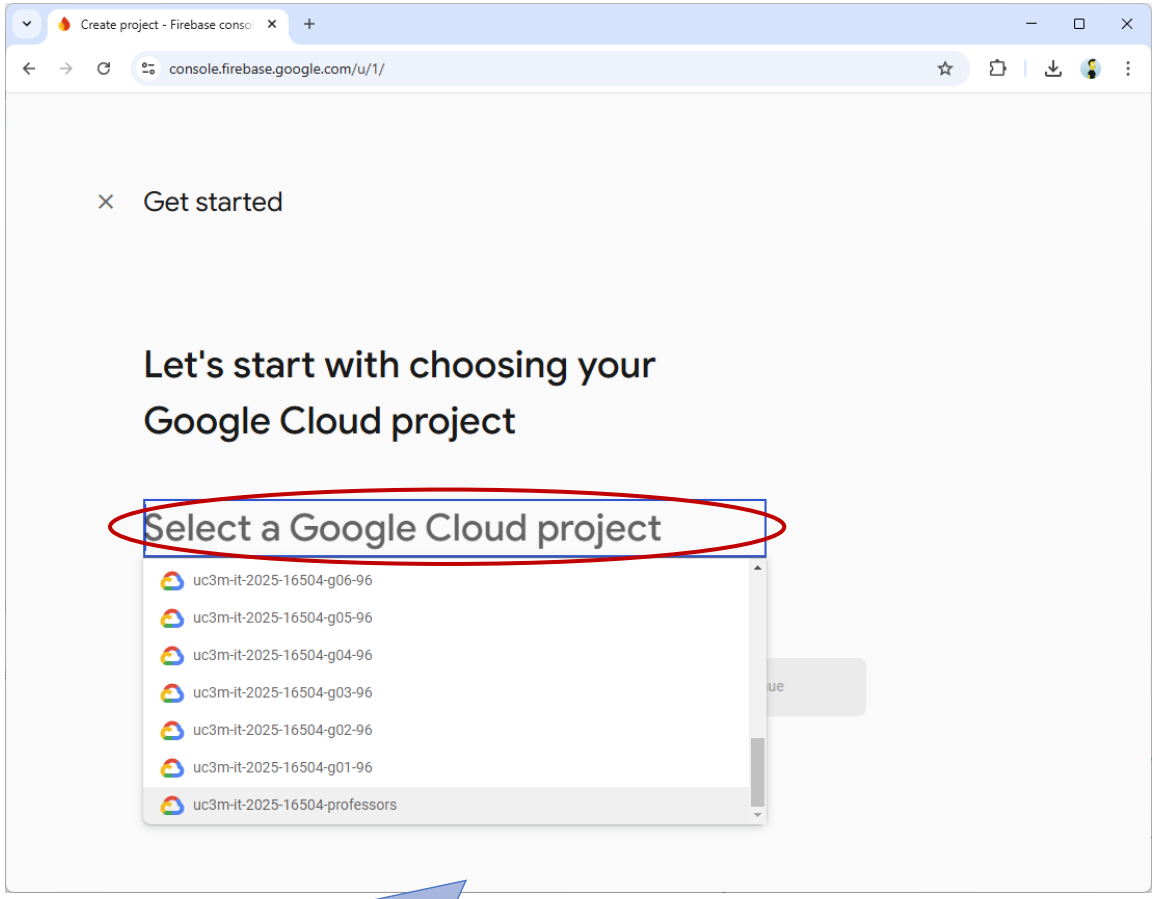


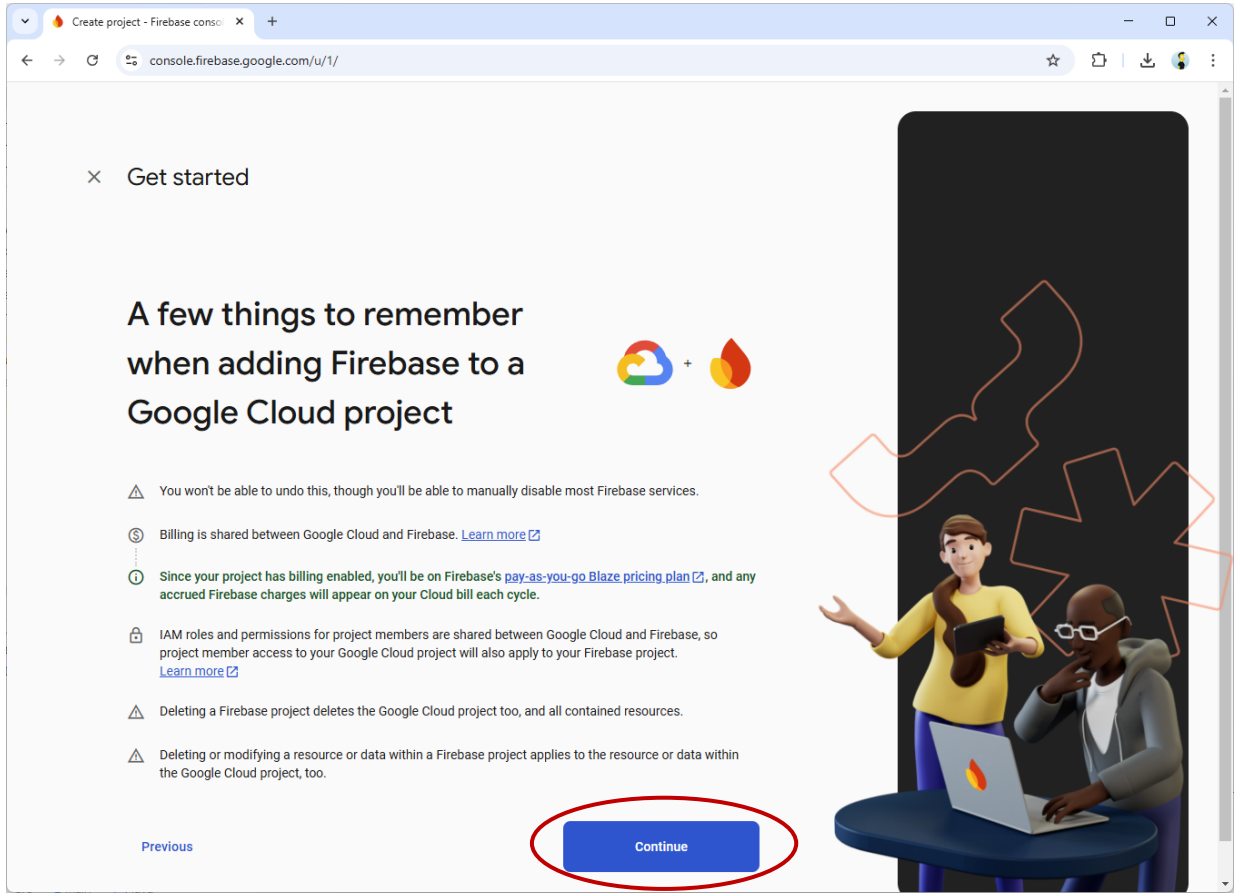
The first step to use Cloud Firestore is to add a project in the console



We need to use the Google cloud project already created (**uc3m-it-2026-16504-g**-lab**), in which you should have redeemed your educational Google coupon ($50) on the lab session on 5-6 March

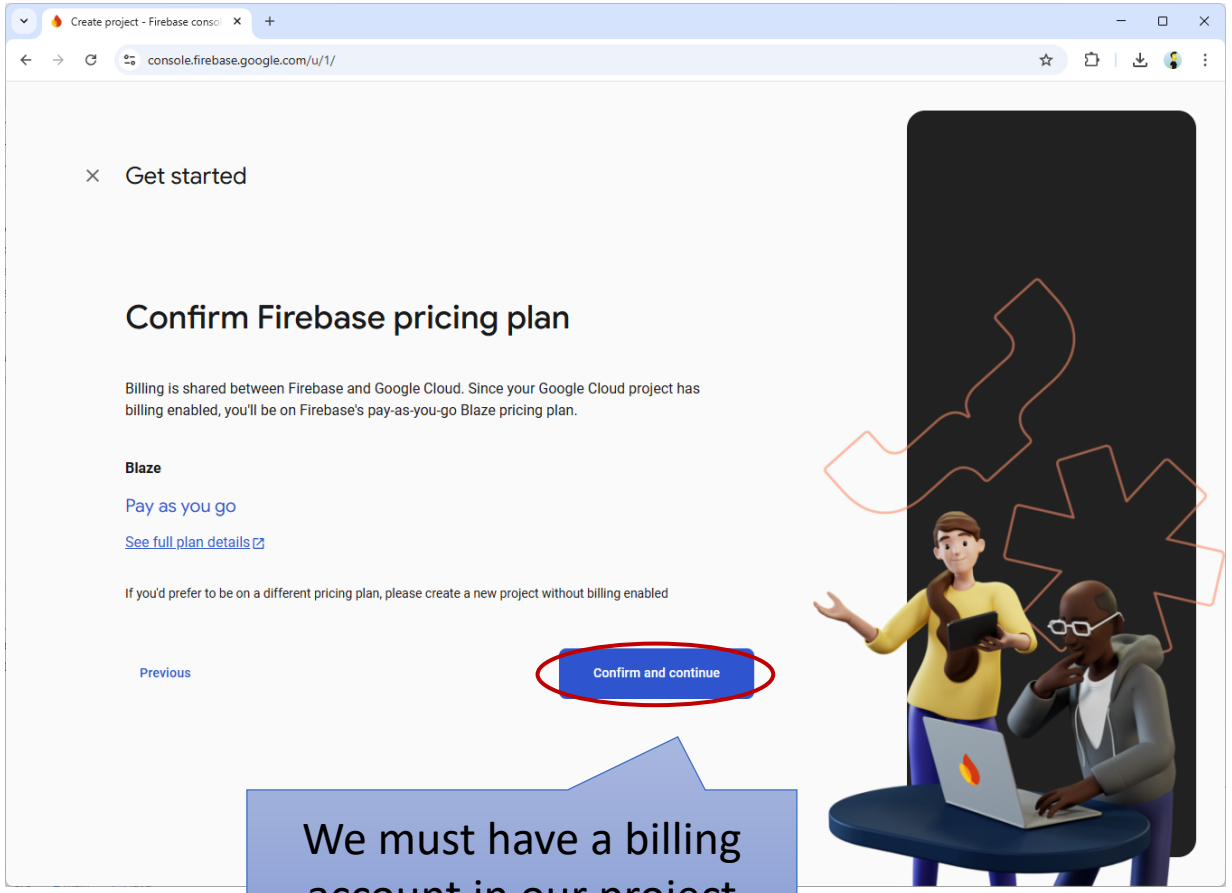
3. Firebase - Cloud Firestore



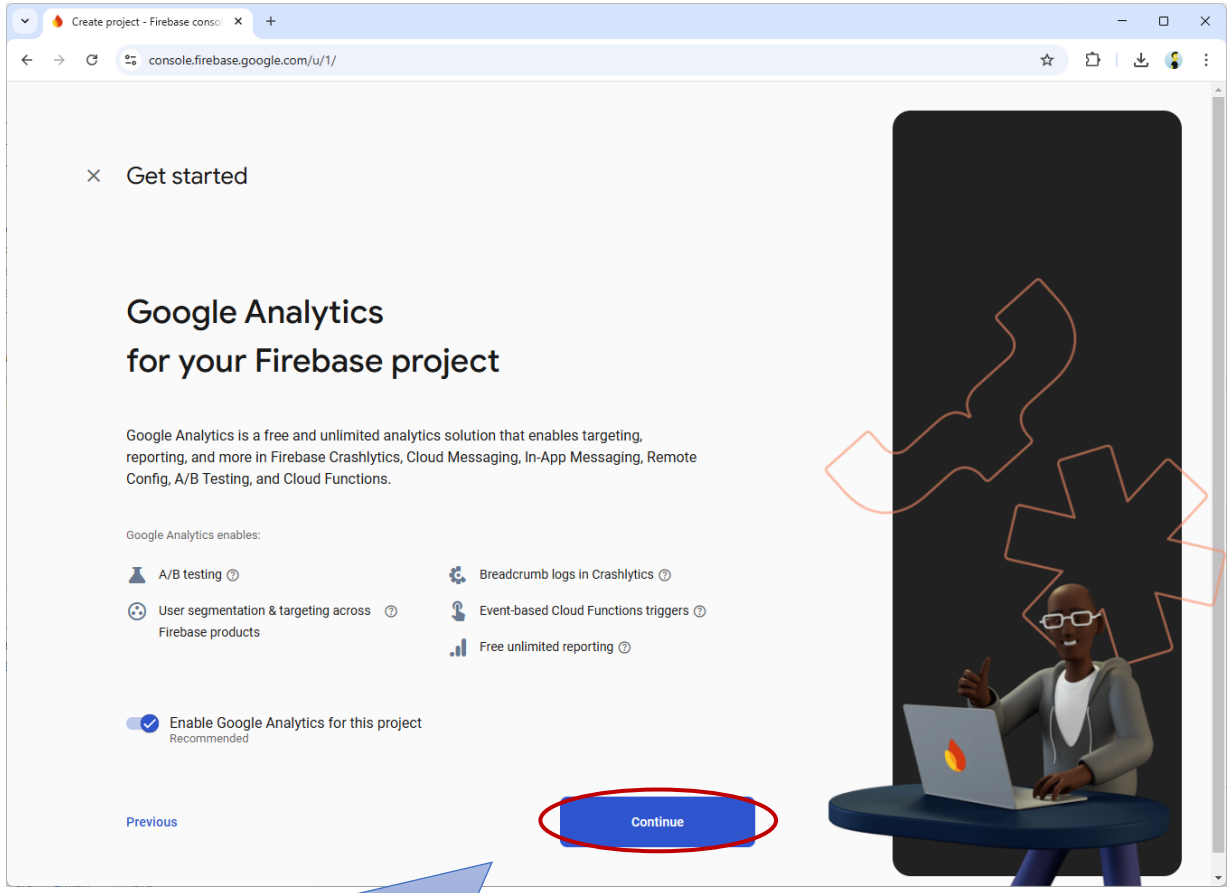


Select your project here and follow the instructions

3. Firebase - Cloud Firestore

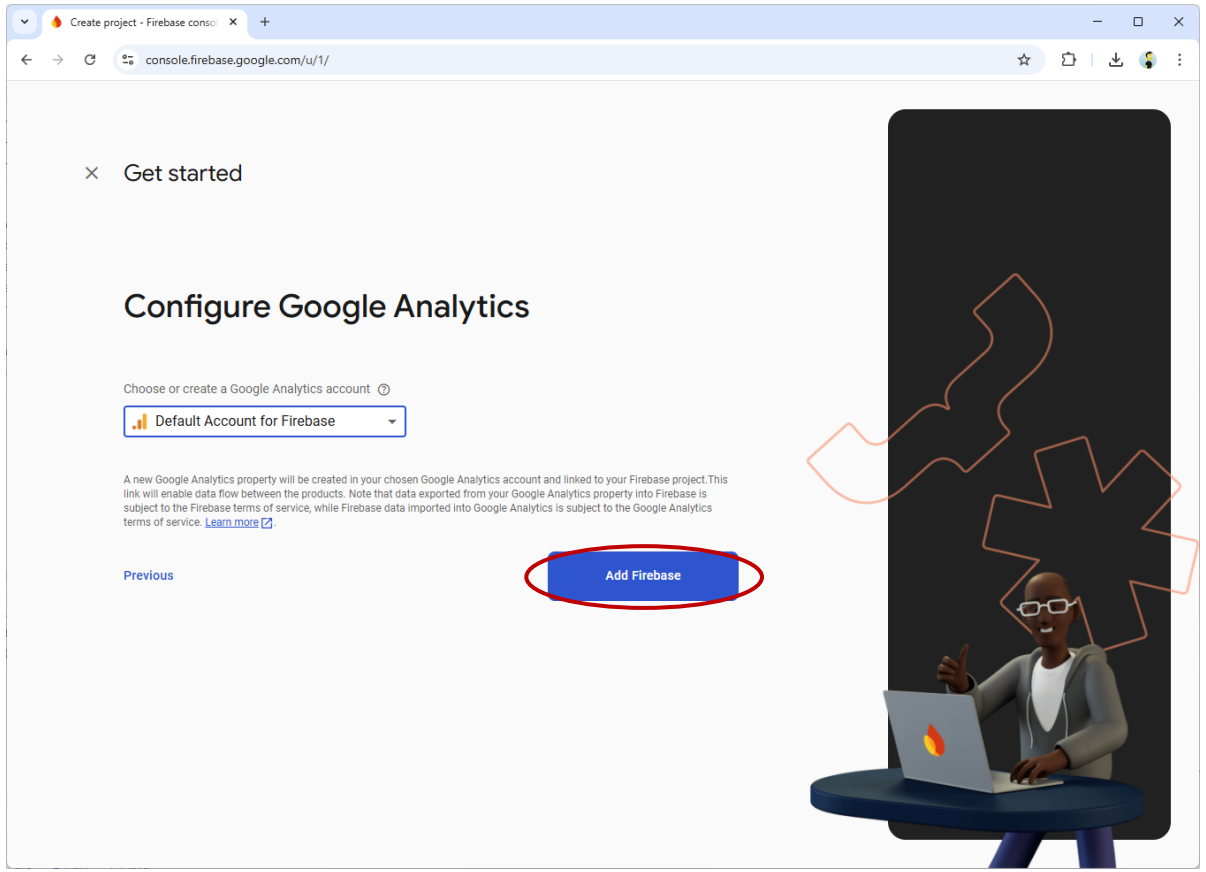


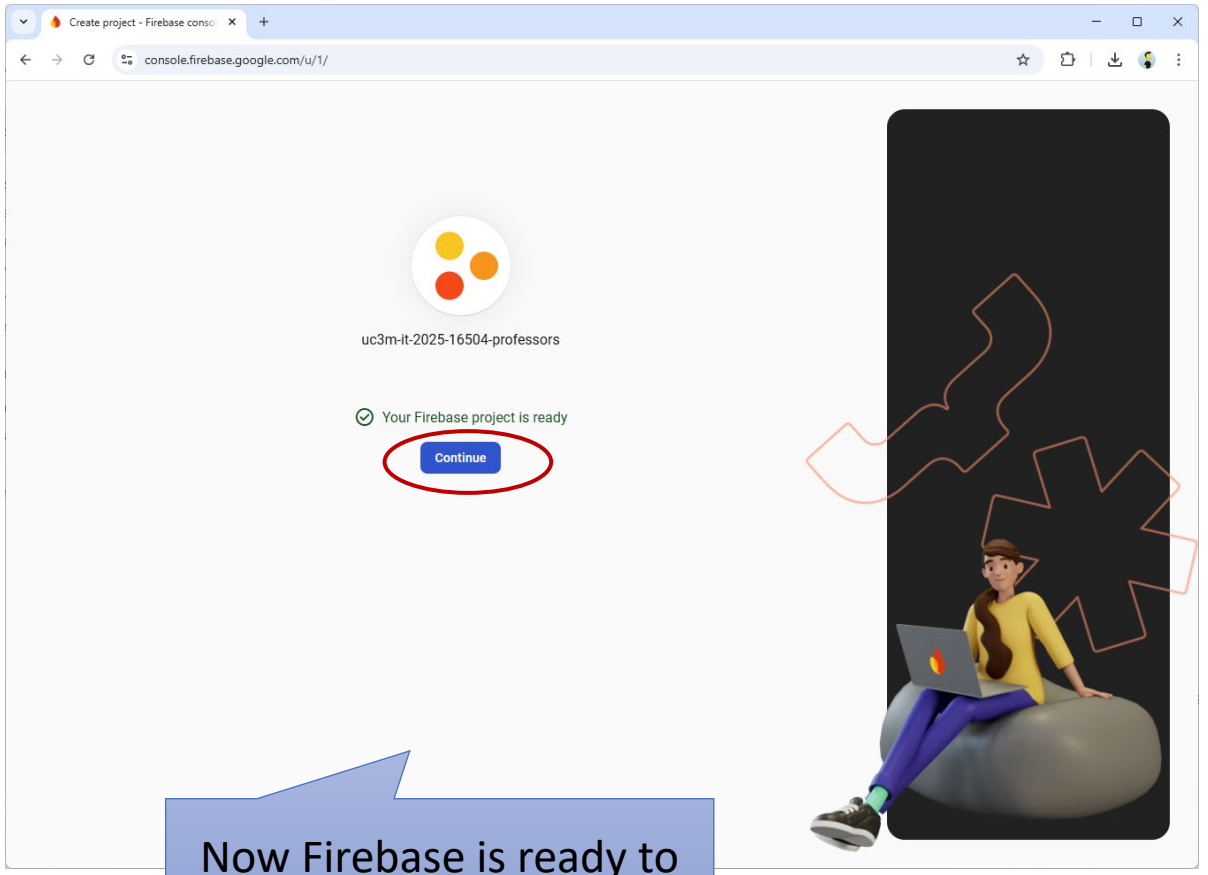
We must have a billing account in our project



Analytics is optional

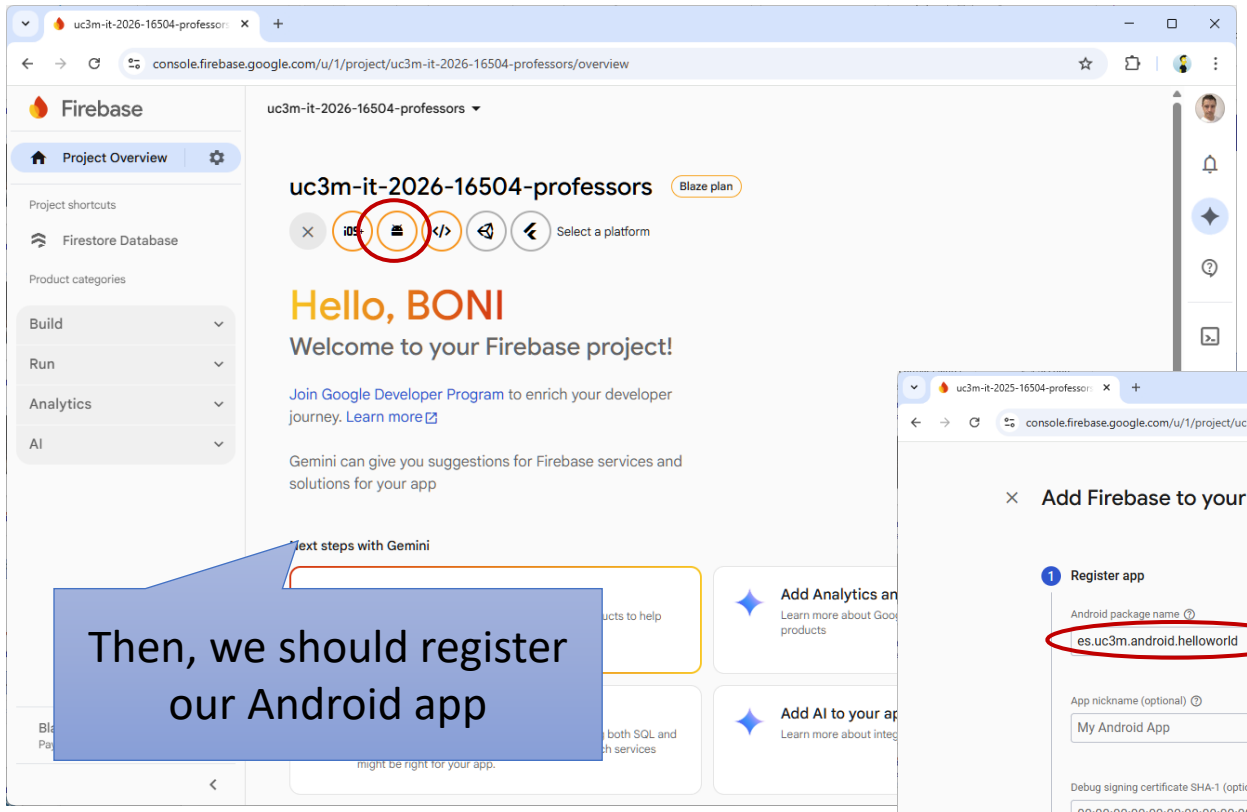
3. Firebase - Cloud Firestore





Now Firebase is ready to be used

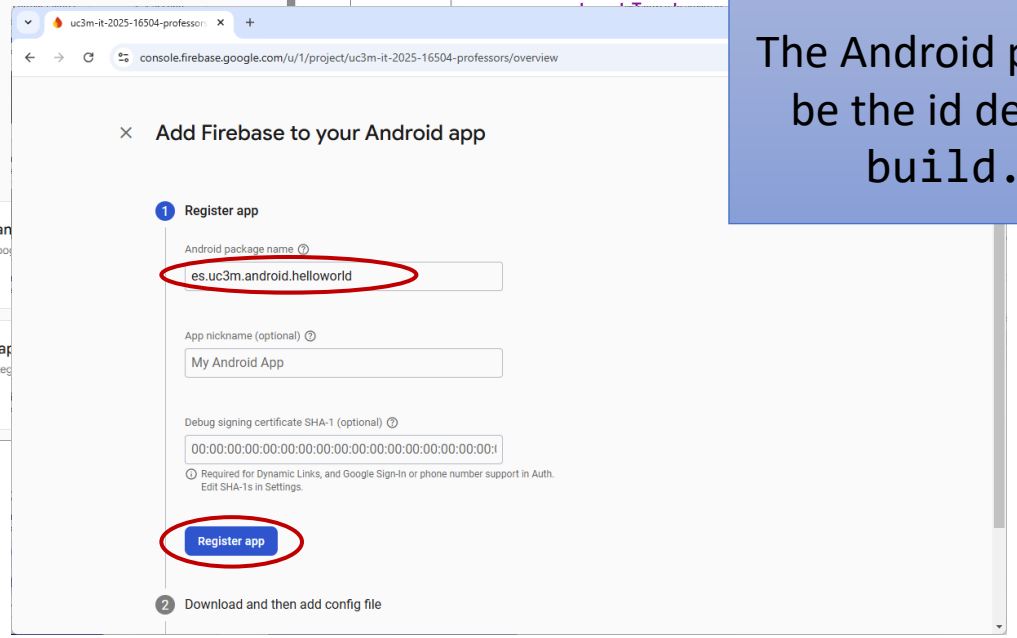
3. Firebase - Cloud Firestore



Then, we should register our Android app

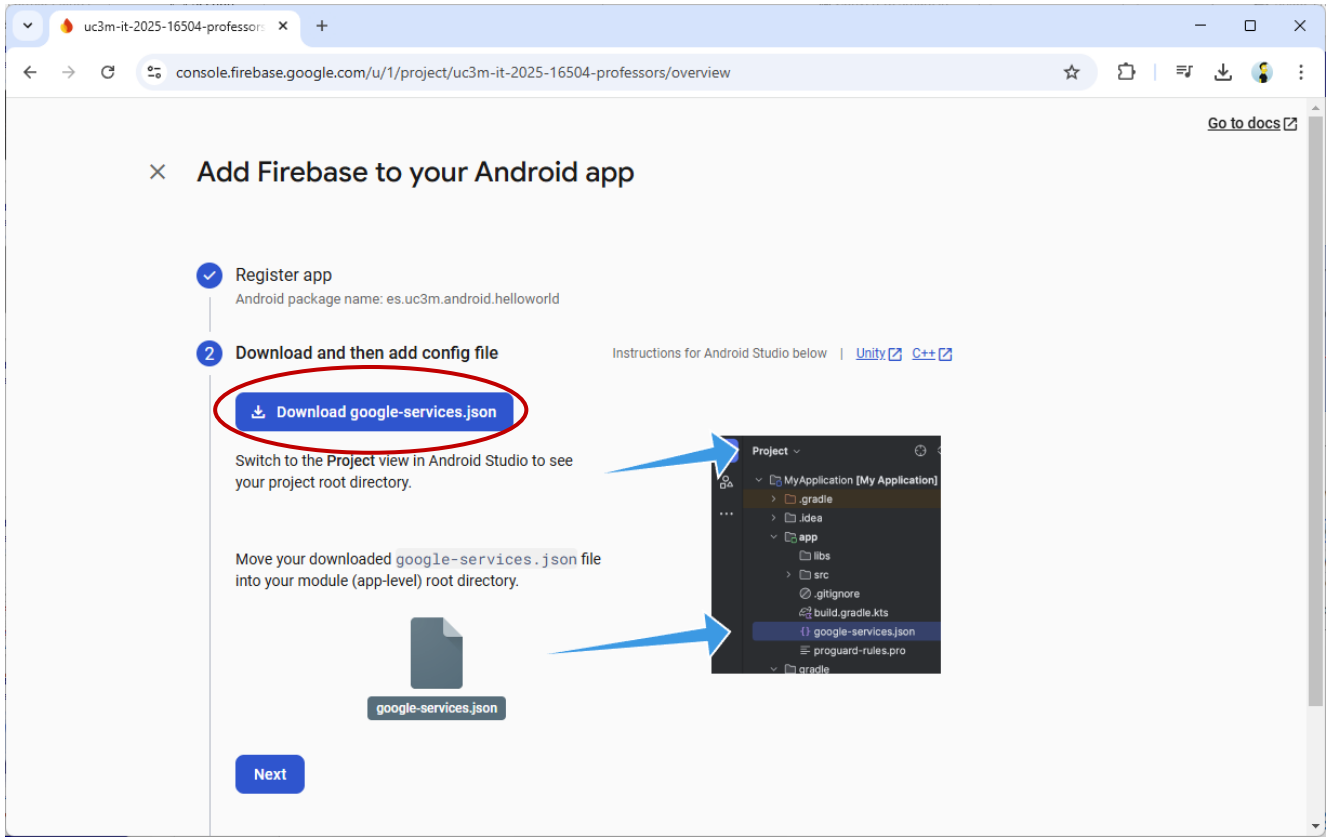
```
android {
    namespace = "es.uc3m.android.firebase"
    compileSdk {
        version = release(36)
    }

    compileSdk = 36

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

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

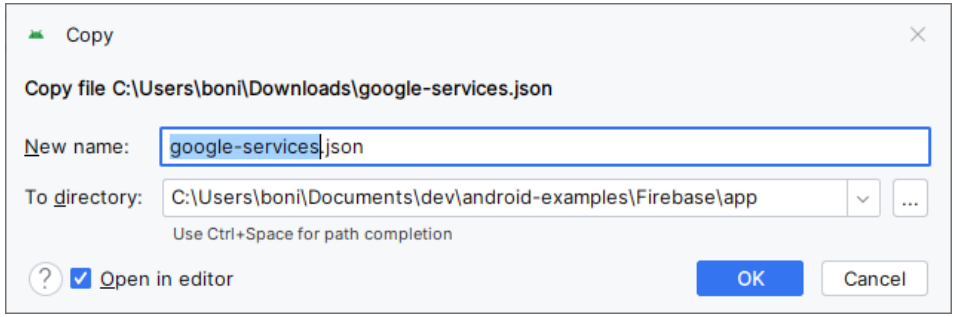


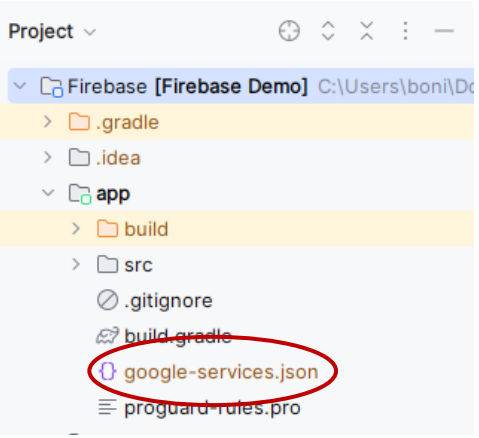
https://firebase.google.com/docs/android/setup

3. 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

3. 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:

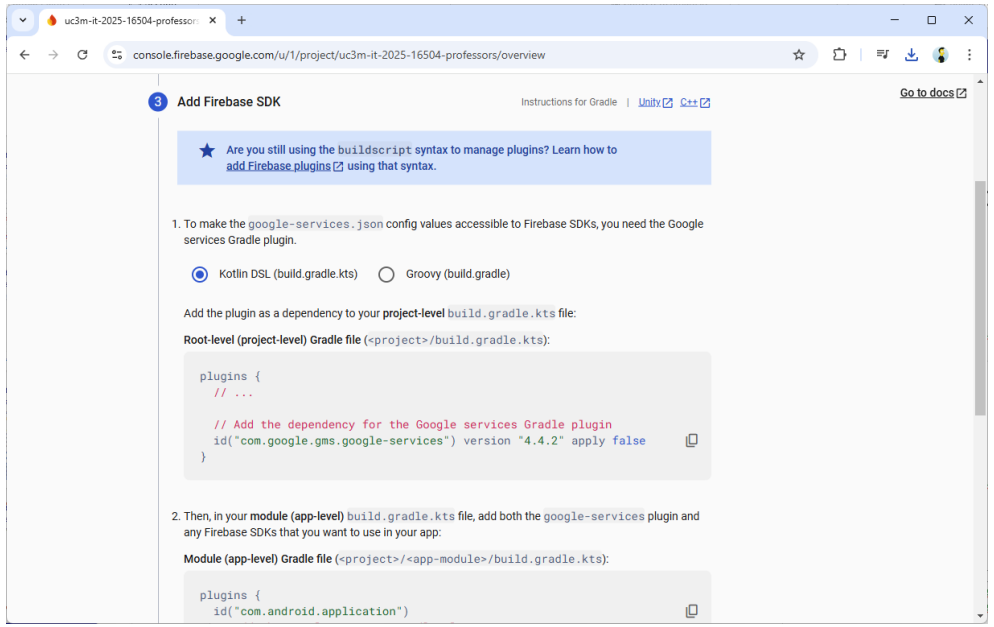
> *For open source projects, we generally do not recommend including the app's Firebase config file or object in source control because, in most cases, your users should create their own Firebase projects and point their apps to their own Firebase resources (via their own Firebase config file or object).*

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

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

3. 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
}
```

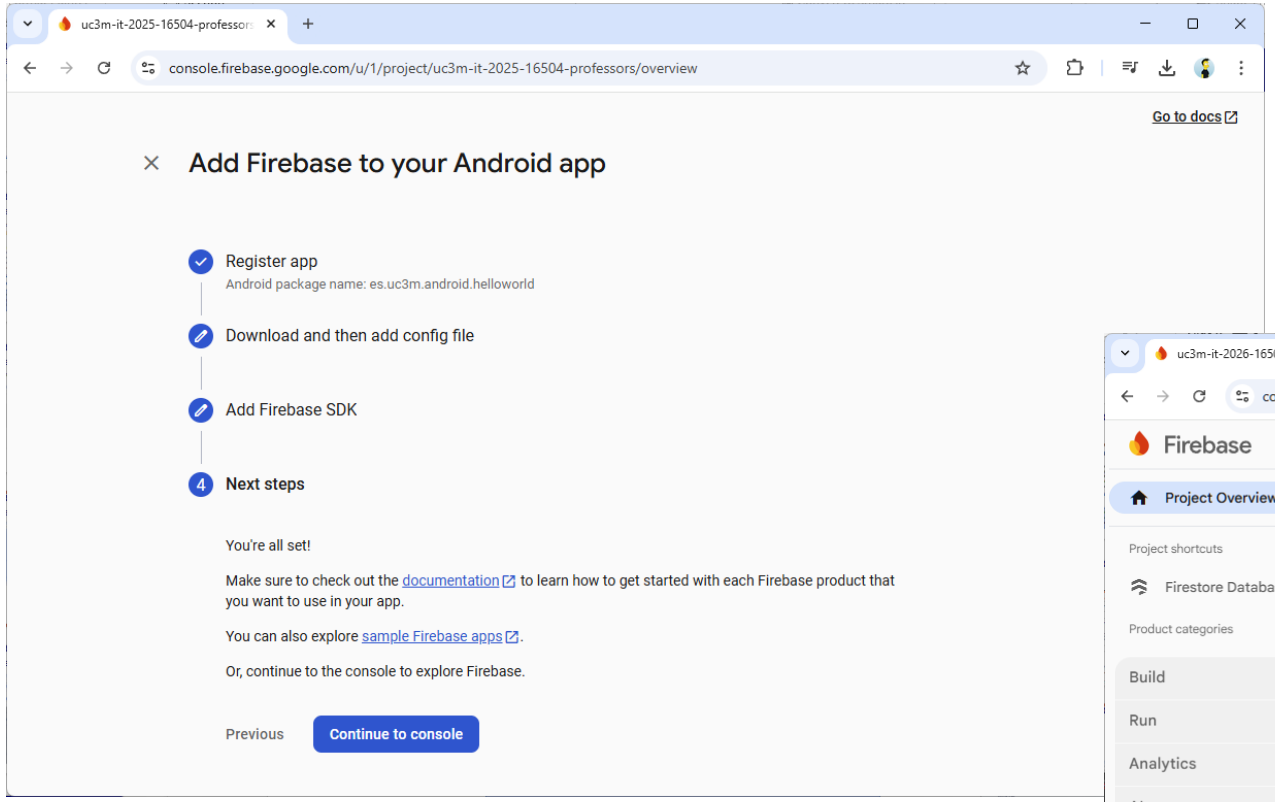
build.gradle.kts (app)

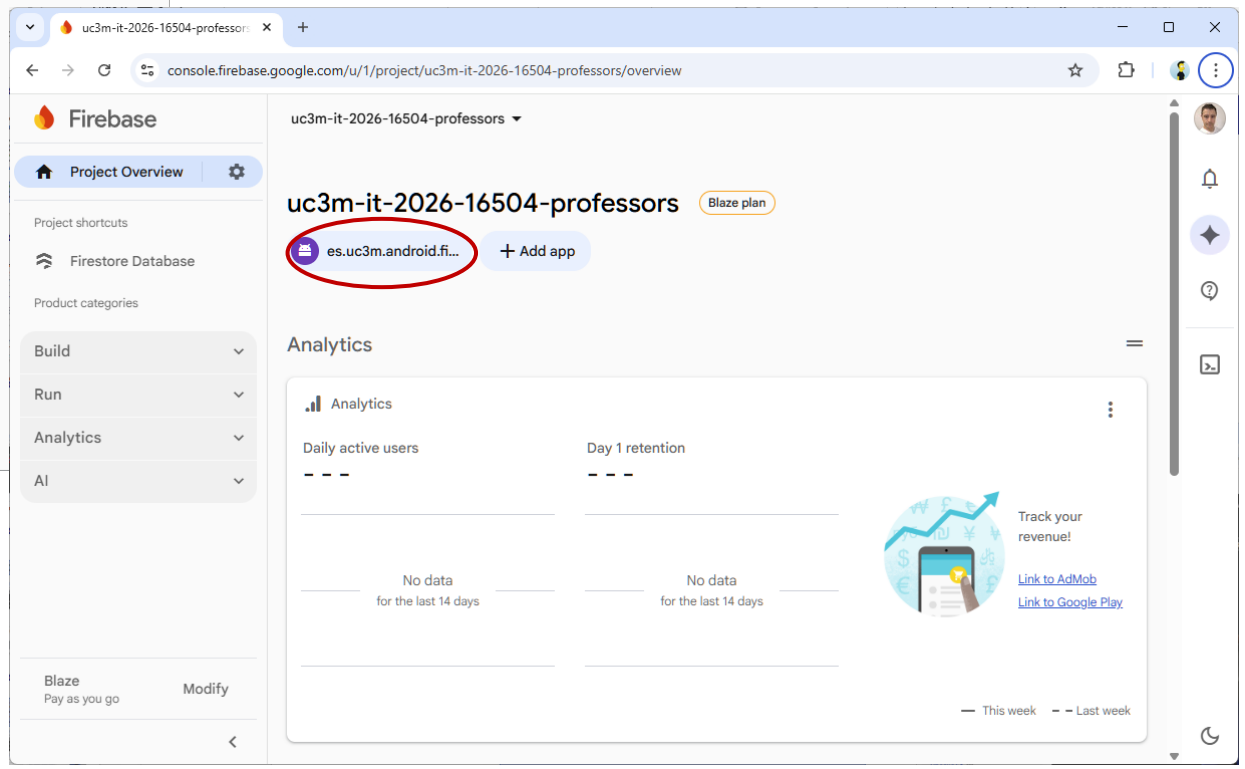
```
plugins {
    alias(libs.plugins.google.services)
}

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

libs.version.toml

```
[versions]
google-services = "4.4.4"
firebaseBom = "34.10.0"

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

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

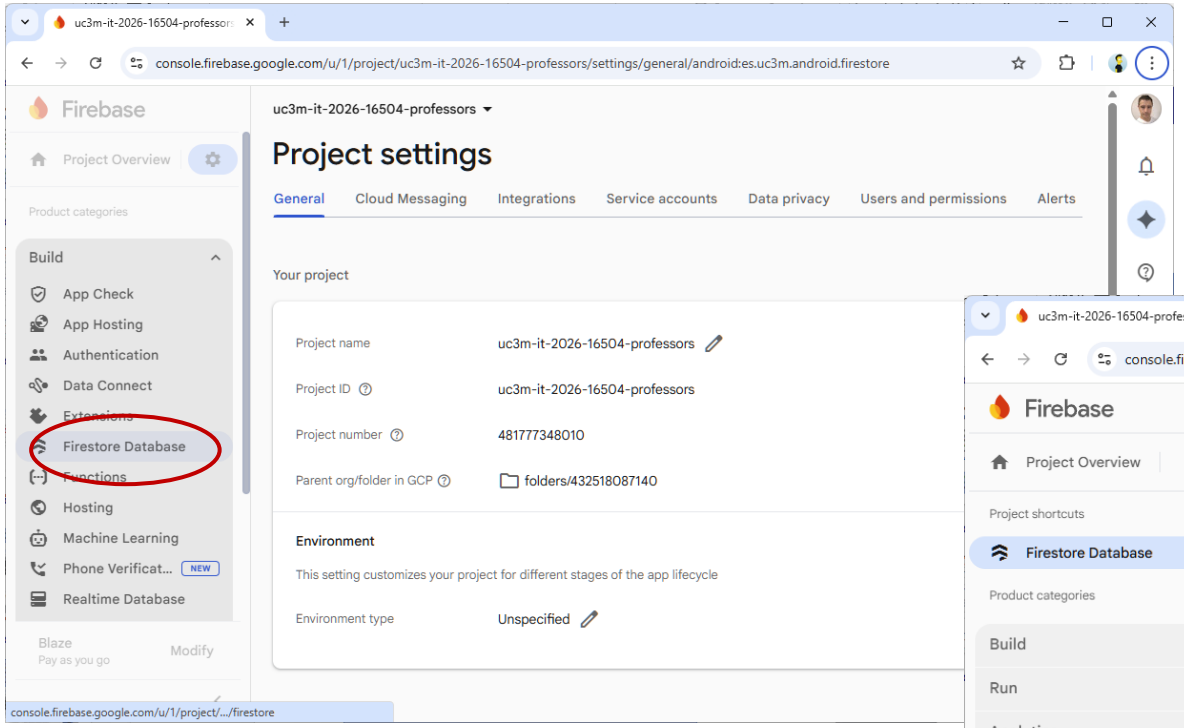
3. Firebase - Cloud Firestore

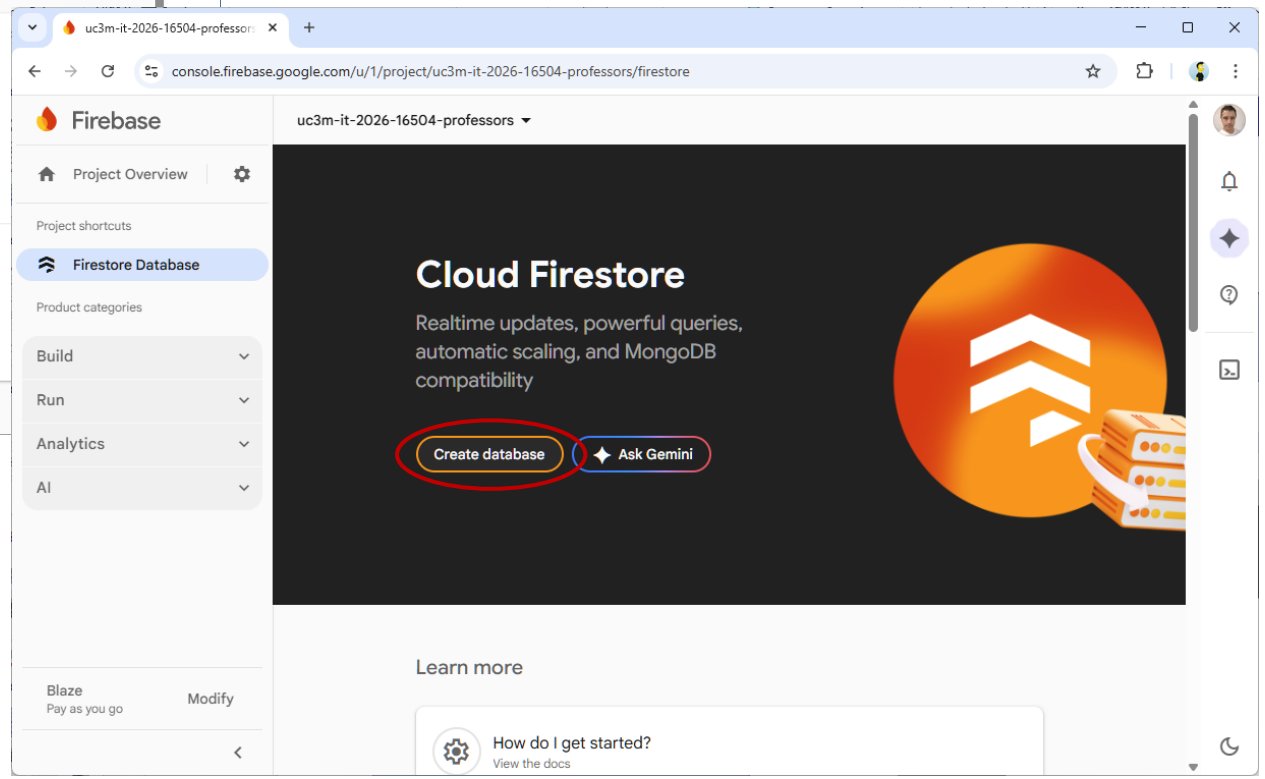




At the end, we can see our app in the Firebase console

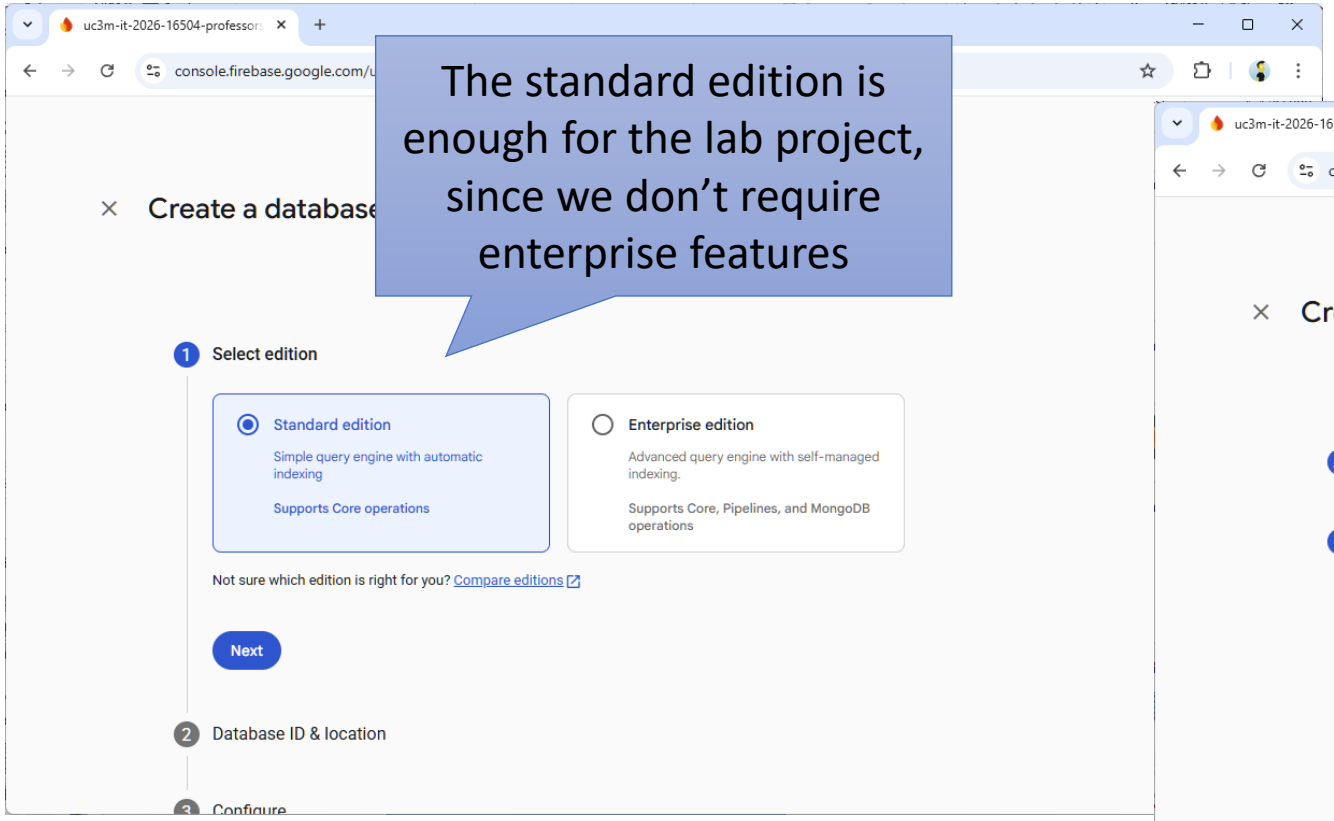
3. Firebase - Cloud Firestore



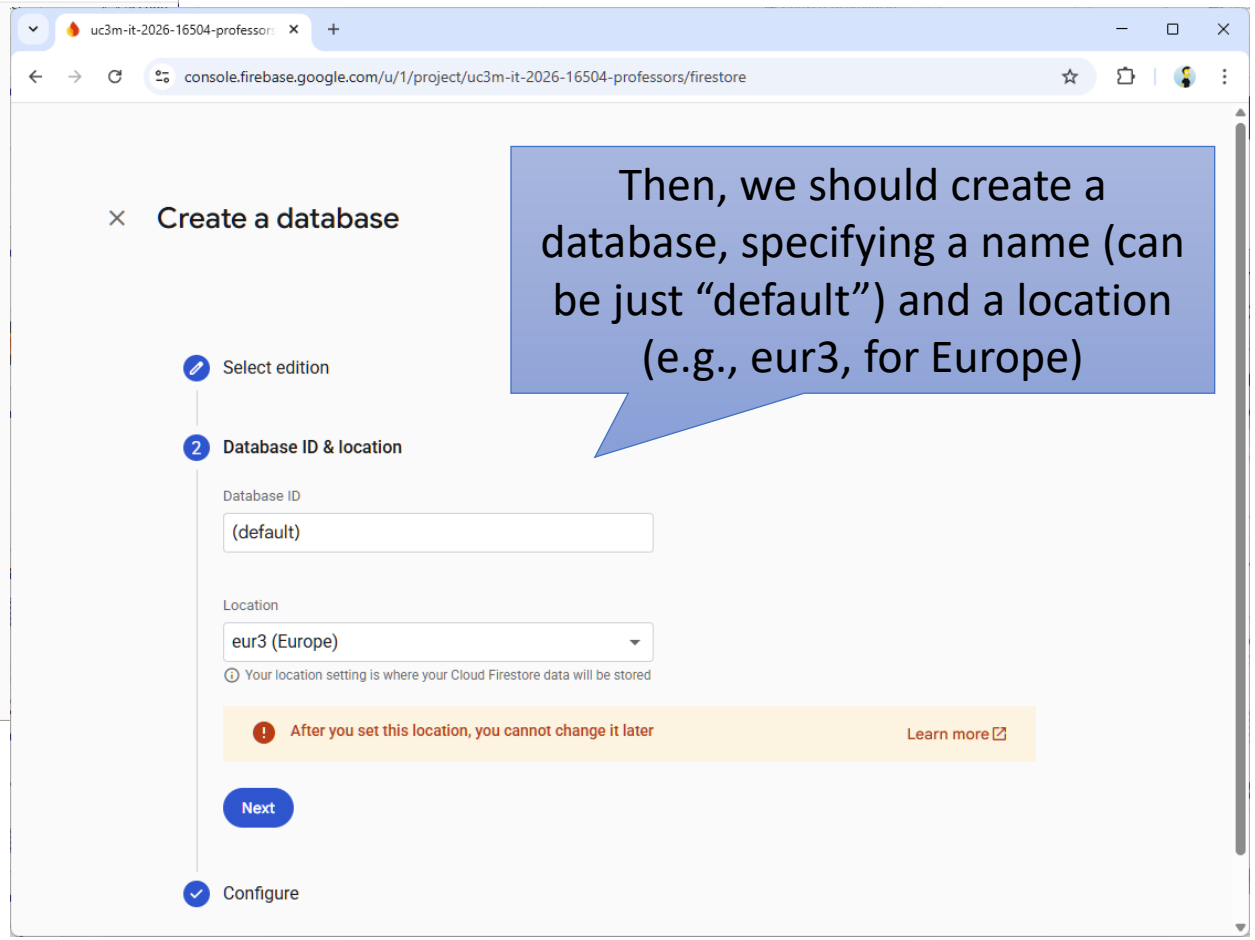


Then, we select “Cloud Firestore” in the left panel, and then “Create database”

3. Firebase - Cloud Firestore

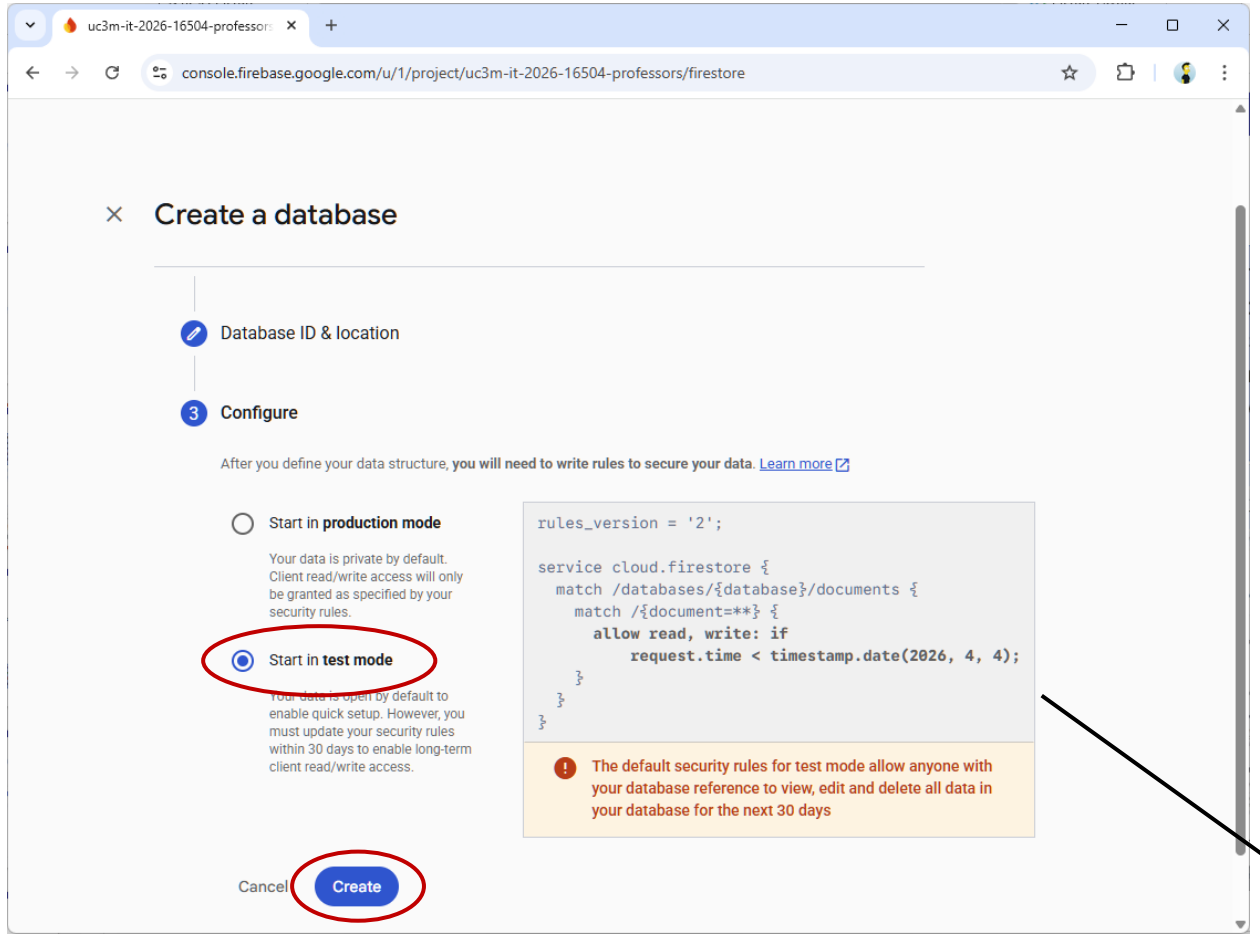


https://firebase.google.com/docs/firestore/editions



https://firebase.google.com/docs/firestore/locations

3. Firebase - Cloud Firestore



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 (temporary open access)

```
rules_version = '2';

service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if
          request.time < timestamp.date(2026, 4, 4);
    }
  }
}
```

3. Firebase - Cloud Firestore

- The Security Rules are a Firestore feature 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

3. Firebase - Cloud Firestore

- Some examples of basic security rules are as follows:

Test mode: temporary open access intended only for development and testing

```
rules_version = '2';

service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if
          request.time < timestamp.date(2026, 4, 4);
    }
  }
}
```

Authenticated users: only allow access to users who are logged in

```
rules_version = '2';

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

Open access: anyone can read and write the database. Never use this in production

```
rules_version = '2';

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

Locked mode: deny all by default (more restrictive option)

```
rules_version = '2';

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

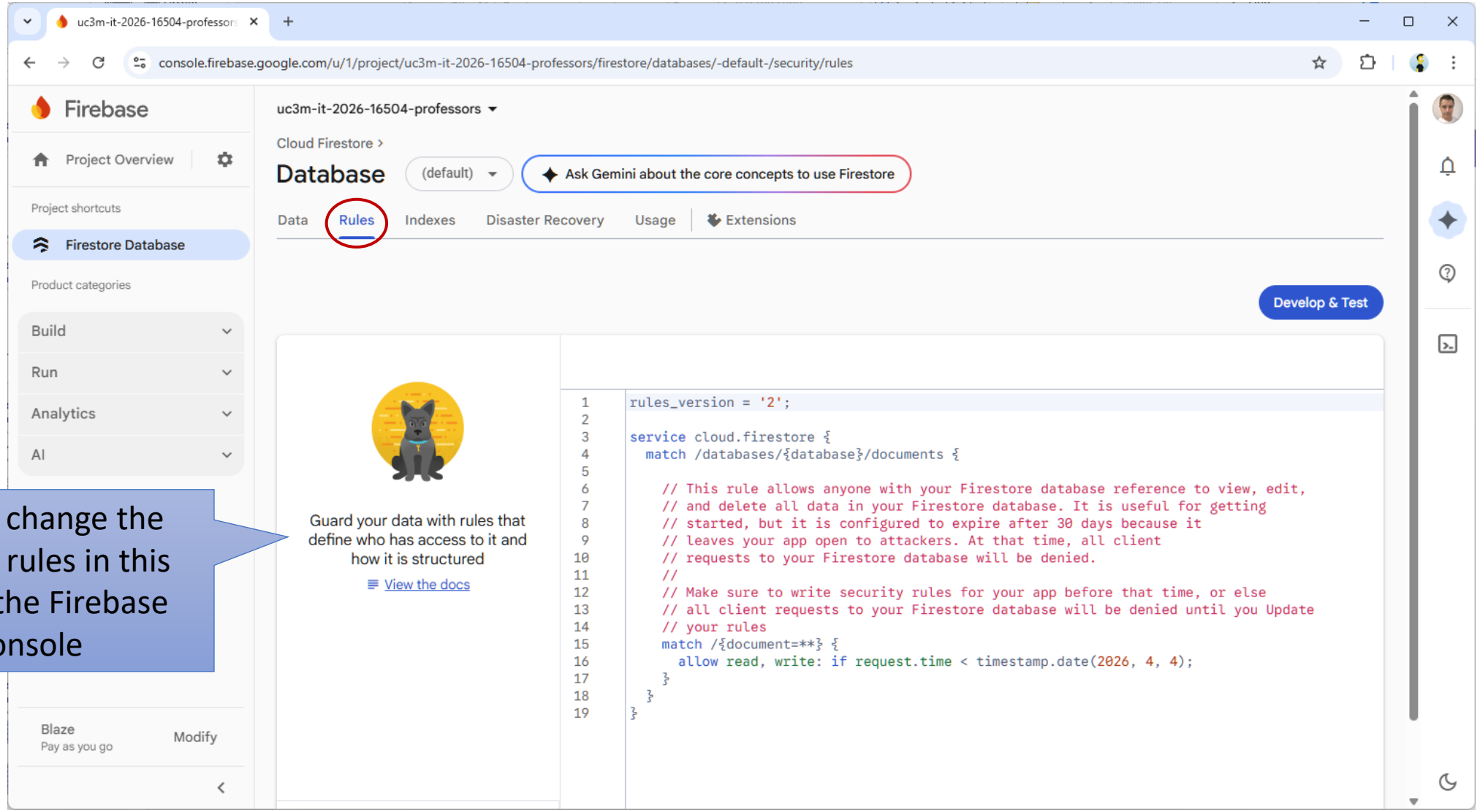
3. Firebase - Cloud Firestore

- In production security rules scenarios, we can match the user ID (UID) to secure documents by author
  - For instance, for a collection called `notes`

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /notes/{noteId} {
      allow read, write: if request.auth != null
                         && request.auth.uid == resource.data.author;
    }
  }
}
```

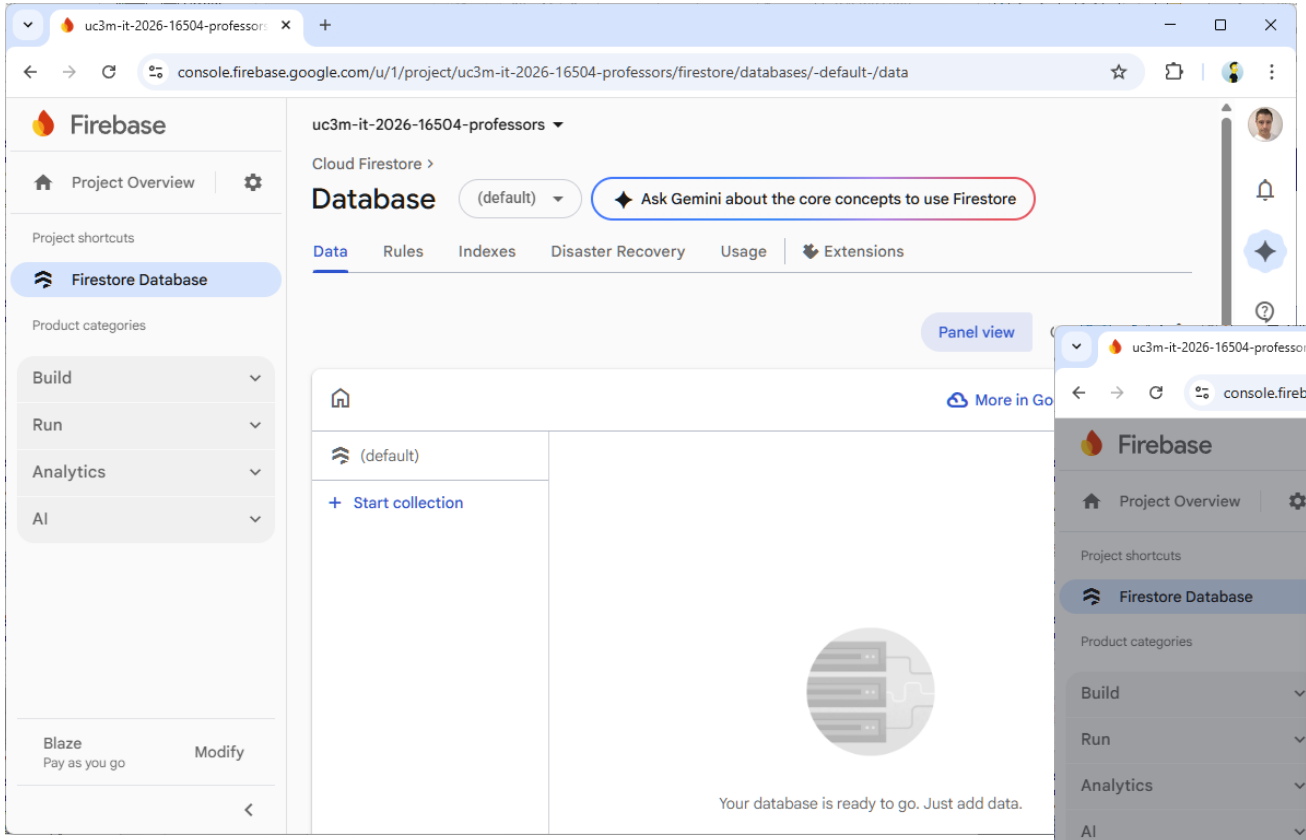
Owner-only access: Allow access only to the author of the document

3. Firebase - Cloud Firestore

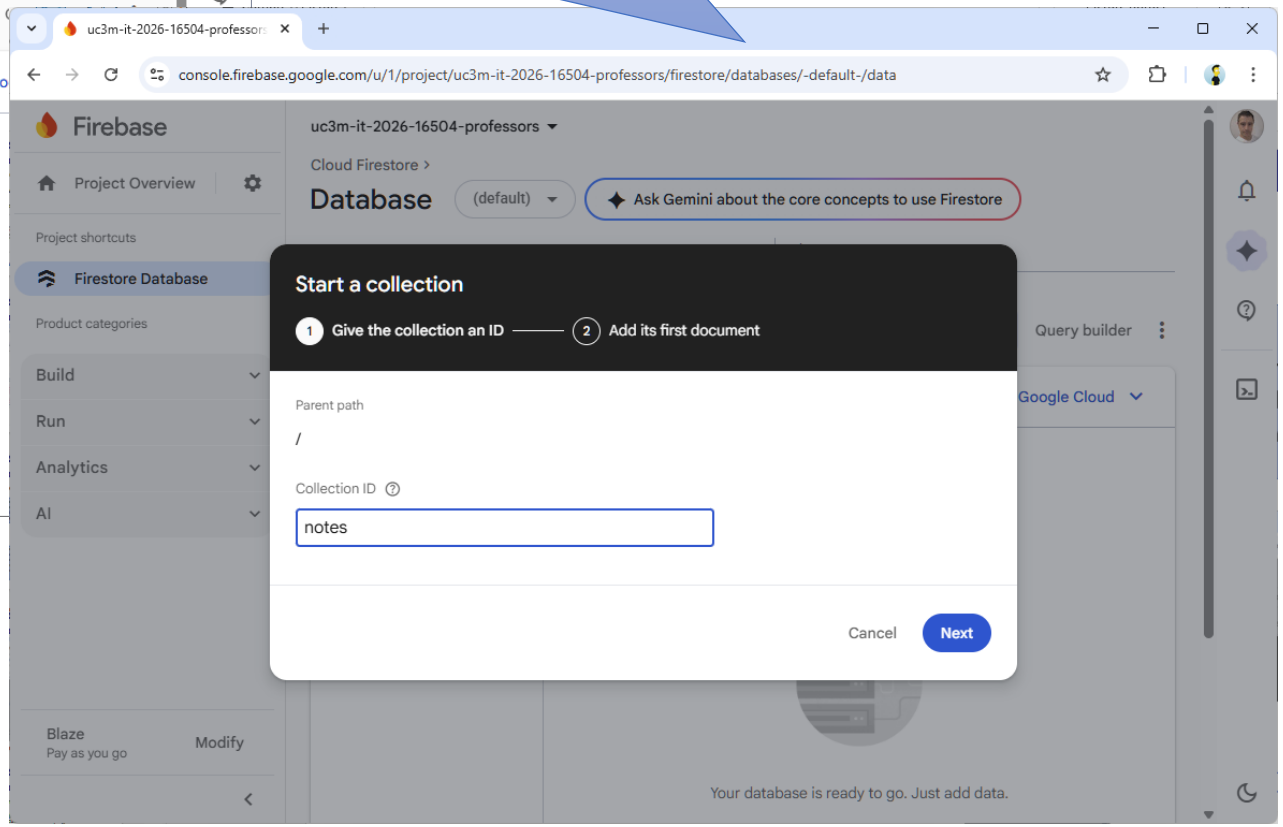


We can change the security rules in this part of the Firebase console

3. Firebase - Cloud Firestore



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



3. Firebase - Cloud Firestore

As we know, for managing complex state that survives configuration changes, we can use an instance of `ViewModel`

```
class MyViewModel() : ViewModel() {

    private val firestore = FirebaseFirestore.getInstance()

    private val _notes = MutableStateFlow<List<Note>>(emptyList())
    val notes: StateFlow<List<Note>> = _notes.asStateFlow()

    private val _snackMessage = MutableStateFlow<String?>(null)
    val snackMessage: StateFlow<String?> = _snackMessage.asStateFlow()

    // ...

}
```

This `ViewModel` make requests to `firestore` and exposes the results to the UI through two properties:

- `notes`: notes list (handled with Firestore CRUD operations)
- `snackMessage`: message to be displayed in the UI and inform about the app state (e.g., for errors)

- `StateFlow` is used to represent a observable state in a ViewModel (it's read-only)
- *`MutableStateFlow`* is a mutable version of `StateFlow` that allows updating its value

3. Firebase - Cloud Firestore

- Read operation:

```
fun fetchNotes() {
    viewModelScope.launch {
        try {
            val snapshot = firestore.collection(NOTES_COLLECTION).get().await()
            val noteList = snapshot.documents.mapNotNull { doc ->
                doc.toObject<Note>()?.copy(id = doc.id)
            }
            _notes.value = noteList
        } catch (e: Exception) {
            e.printStackTrace()
            setSnackMessage(e.message)
        }
    }
}


fun setSnackMessage(message: String?) {
    _snackMessage.value = message
}
```

We use `get()` to retrieve all documents in the notes collection from Firestore

`await()` converts a Firebase task into a suspending operation, so the coroutine is suspended (it waits) until Firestore returns the result (without blocking the thread)

`viewModelScope.launch {...}` starts a new coroutine. It is cancelled automatically when the `ViewModel` is cleared (i.e., when the when the associated activity is permanently removed)

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

3. Firebase - Cloud Firestore

- Write operation:

```
fun addNote(title: String, body: String) {
    viewModelScope.launch {
        try {
            val note = Note(title = title, body = body)
            firestore.collection(NOTES_COLLECTION).add(note).await()
            fetchNotes() // Refresh notes list
        } catch (e: Exception) {
            e.printStackTrace()
            setSnackMessage(e.message)
        }
    }
}
```

The add() method creates a new document in the collection with an automatically generated ID

Firestore creates collections and documents implicitly if they do not already exist

3. Firebase - Cloud Firestore

- Update and delete operations:

```
fun updateNote(id: String, title: String, body: String) {
    viewModelScope.launch {
        try {
            val updatedNote = Note(title = title, body = body)
            firestore.collection(NOTES_COLLECTION).document(id).set(updatedNote).await()
            fetchNotes() // Refresh notes list
        } catch (e: Exception) {
            e.printStackTrace()
            setSnackMessage(e.message)
        }
    }
}
```

We use the `set()` method to modify a document. We should know the document ID we want to update

We use the `delete()` method to remove a document. We should also know the document ID

```
fun deleteNote(id: String) {
    viewModelScope.launch {
        try {
            firestore.collection(NOTES_COLLECTION).document(id).delete().await()
            fetchNotes() // Refresh notes list
        } catch (e: Exception) {
            e.printStackTrace()
            setSnackMessage(e.message)
        }
    }
}
```

3. Firebase - Cloud Firestore

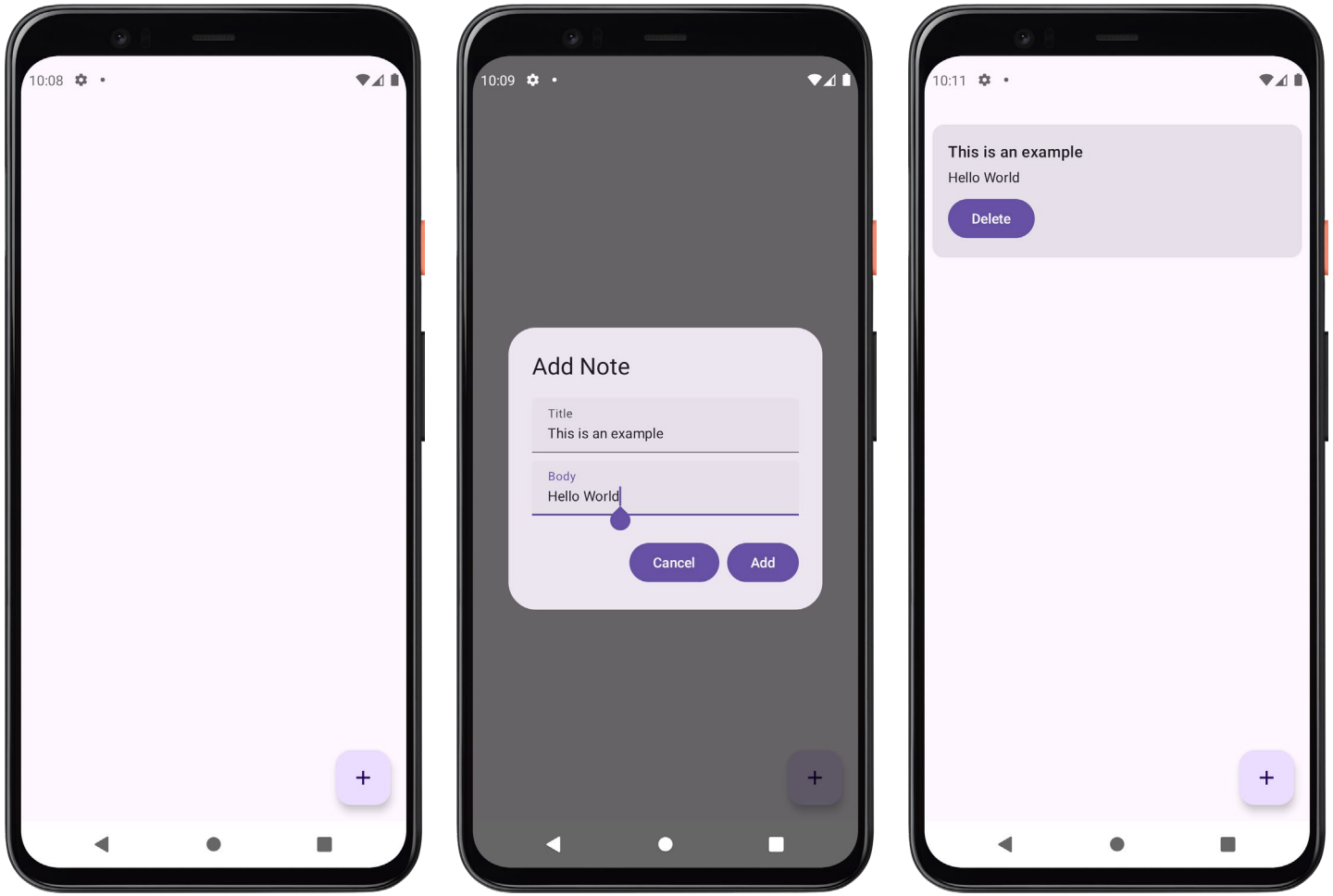

```
@Composable
fun MainScreen(viewModel: FirebaseViewModel = viewModel()) {
    val showAddNoteDialog = remember { mutableStateOf(false) }
    val noteToEdit = remember { mutableStateOf<Note?>(null) }
    val snackHostState = remember { SnackbarHostState() }

    val notes by viewModel.notes.collectAsState()
    val snackMessage by viewModel.snackMessage.collectAsState()

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

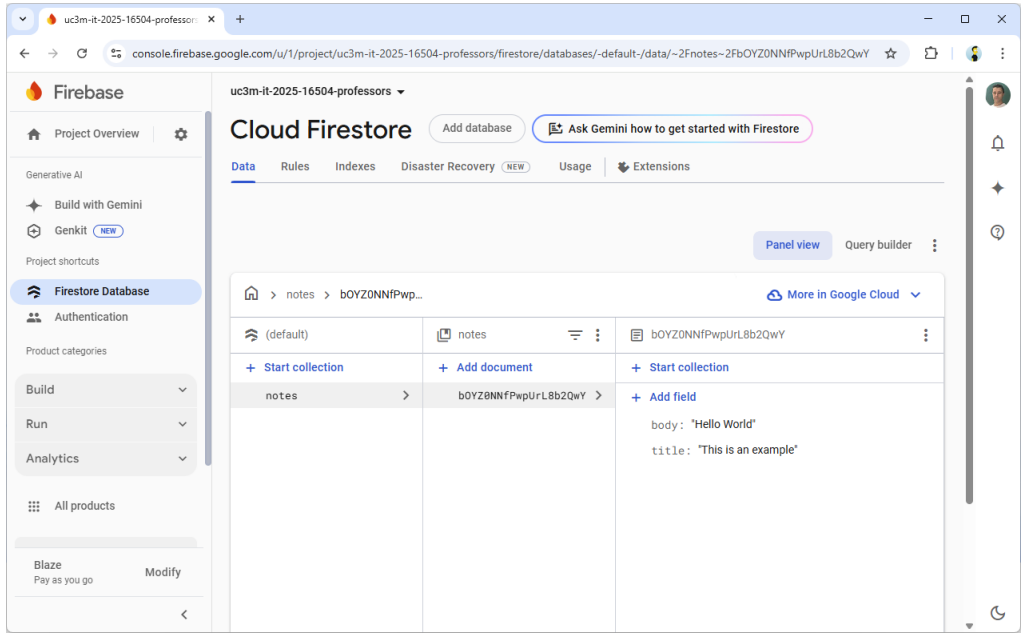
We observe state from the `ViewModel` (`notes`, `snackMessage`) using `collectAsState()`

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

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

    LaunchedEffect(snackMessage) {
        snackMessage?.let { message ->
            snackHostState.showSnackbar(message)
            // Reset message to avoid showing it repeatedly
            // (e.g., on configuration changes)
            viewModel.setSnackMessage(null)
        }
    }

}
```

`LaunchedEffect(key)` is a helper composable that launches a coroutine as a side effect, i.e., whenever the key changes (not on every recomposition). We need to use it here since `showSnackbar(message)` is a suspending function

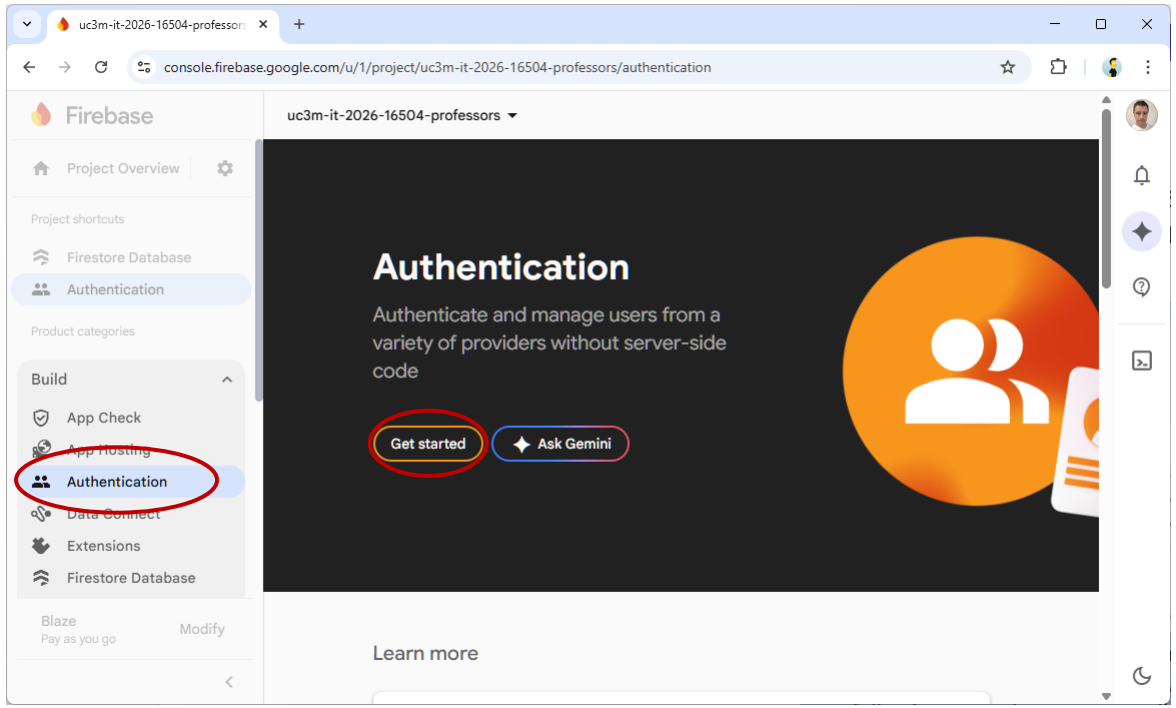
3. Firebase - Cloud Firestore

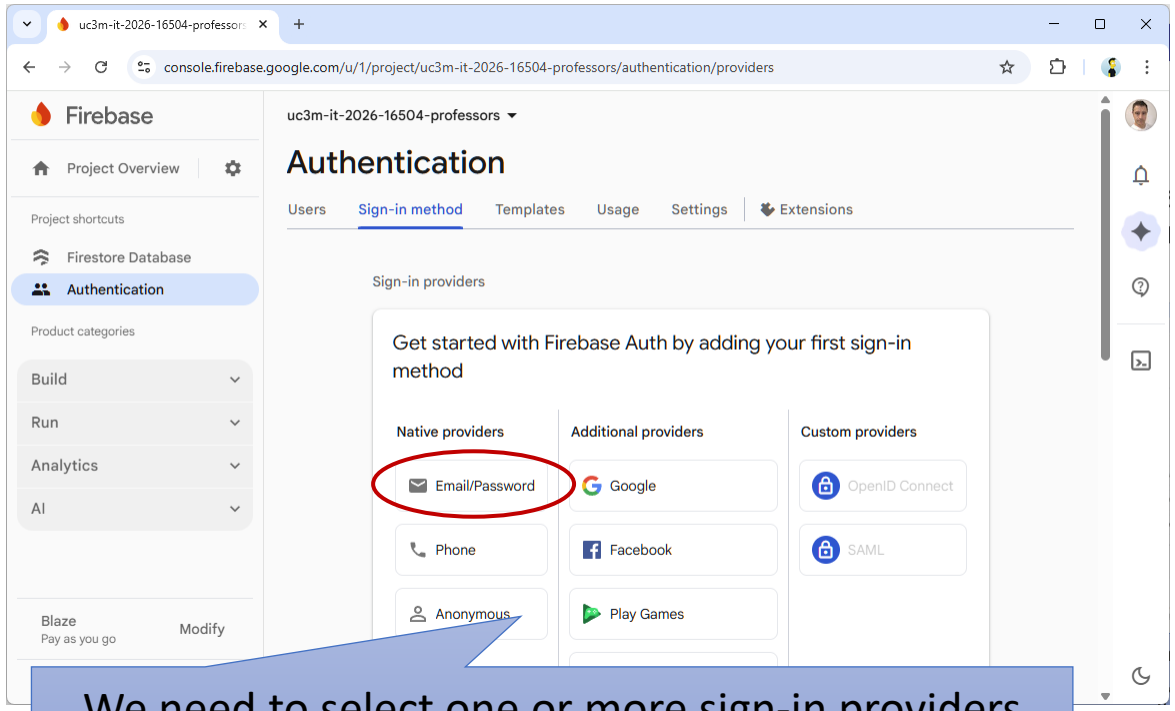




3. Firebase - Authentication

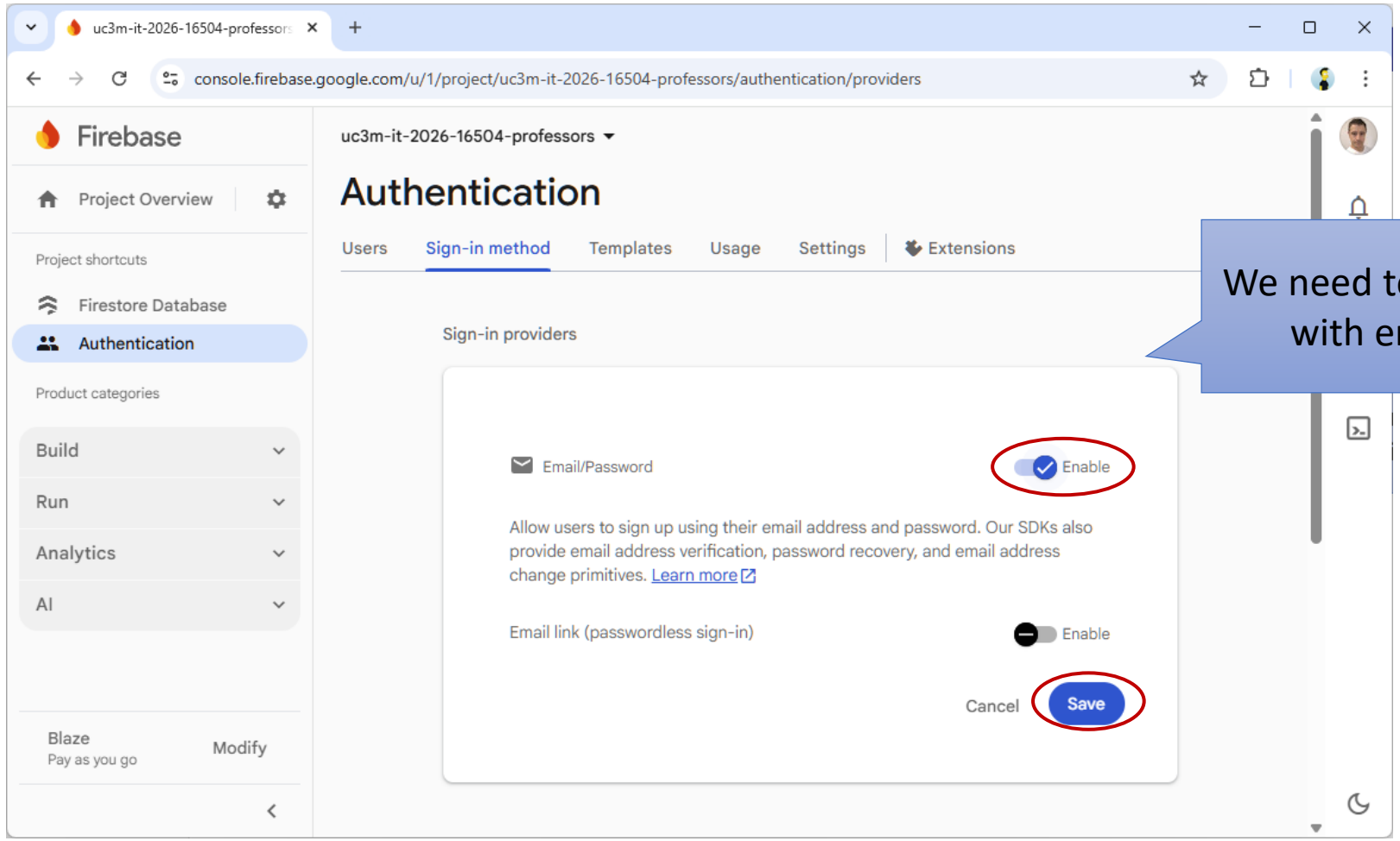
- Authentication in Firebase is a service that allows users to sign in using various methods, such as email/password, phone numbers, or social media accounts, among others





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

3. Firebase - Authentication



3. Firebase - Authentication

- Authentication demo:

This ViewModel extends AndroidViewModel to get the application context, used in this example to read from strings.xml (see getString())

```
class MyViewModel(application: Application) : AndroidViewModel(application) {

    private val firestore = FirebaseFirestore.getInstance()
    private val auth: FirebaseAuth = Firebase.auth

    private val _notes = MutableStateFlow<List<Note>>(emptyList())
    val notes: StateFlow<List<Note>> = _notes.asStateFlow()

    private val _snackMessage = MutableStateFlow<String?>(null)
    val snackMessage: StateFlow<String?> = _snackMessage.asStateFlow()

    private val _route = MutableStateFlow<String?>(null)
    val route: StateFlow<String?> = _route.asStateFlow()

    // Helper method to access strings from resources
    private fun getString(resId: Int, vararg formatArgs: Any): String =
        getApplication<Application>().getString(resId, *formatArgs)

    // ...
}
```

This line retrieves the Firebase Authentication instance

3. Firebase - Authentication

- Then, we use the auth instance for creating users (sign up), authenticating existing users (log in), and terminate sessions (log out):

```
fun signUp(email: String, password: String) {
    viewModelScope.launch {
        try {
            auth.createUserWithEmailAndPassword(email, password).await()
            navigateTo(NavGraph.Home.route) // Go to home screen
            setSnackMessage(getString(R.string.sign_up_ok))
    } catch (e: Exception) {
            e.printStackTrace()
            setSnackMessage(e.message ?: getString(R.string.signup_error))
        }
    }
}
```

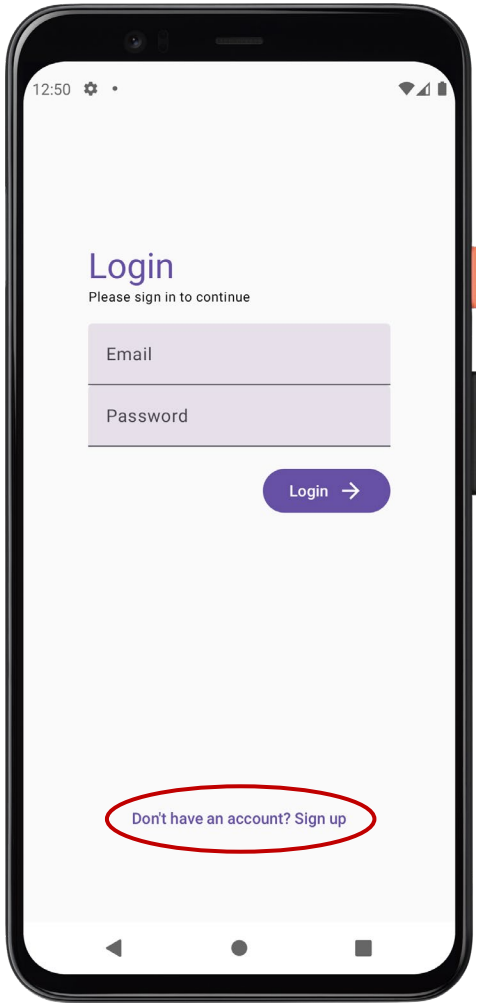
```
fun login(email: String, password: String) {
    viewModelScope.launch {
        try {
            auth.signInWithEmailAndPassword(email, password).await()
            fetchNotes()
            navigateTo(NavGraph.Home.route) // Go to home screen
            setSnackMessage(getString(R.string.login_ok))
        } catch (e: Exception) {
            e.printStackTrace()
            setSnackMessage(e.message ?: getString(R.string.login_error))
        }
    }
}
```

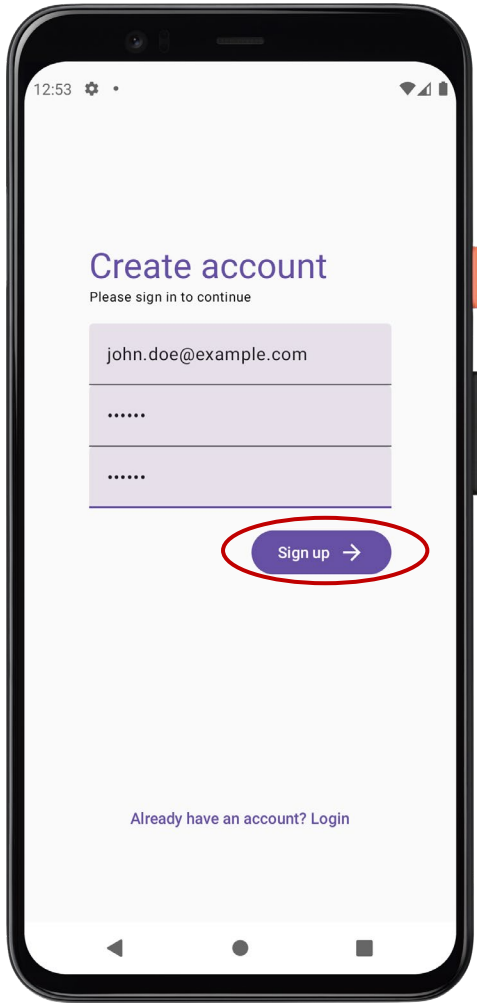
```
fun logout() {
    try {
        auth.signOut() // Synchronous operation, so coroutine is not needed
        navigateTo(NavGraph.Login.route) // Go to login screen
    } catch (e: Exception) {
        e.printStackTrace()
        setSnackMessage(e.message ?: getString(R.string.logout_error))
    }
}
```

We need to launch a coroutine for signing up and login (asynchronous operations), but not for signing out (synchronous operation)

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

3. Firebase - Authentication





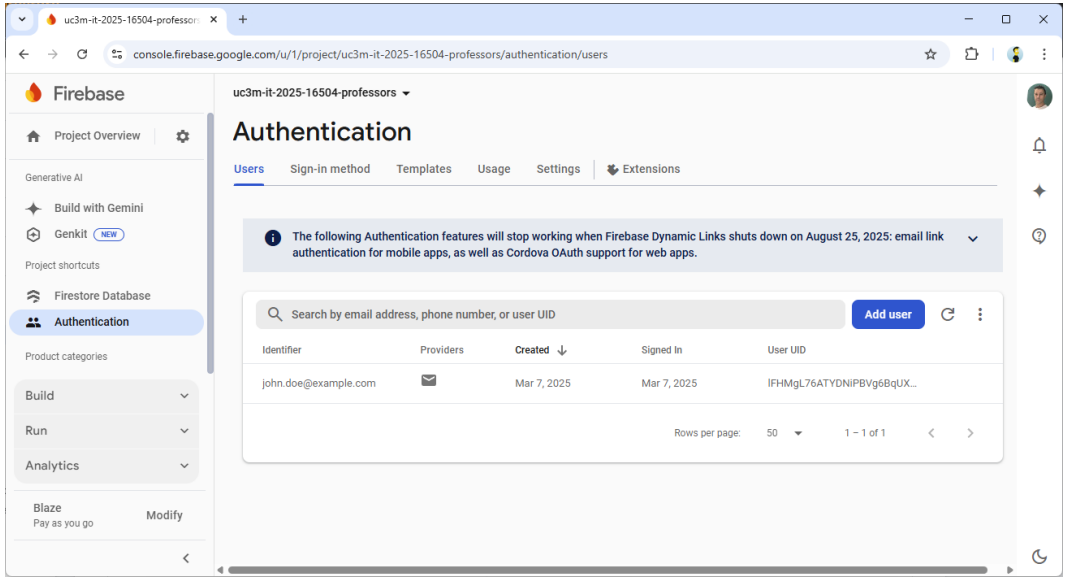


Table of contents

4. Files

- Android applications can store files in two main locations:

1. **Internal storage**
   - Private storage for each app (stored in the app *sandbox*)
     - The sandbox is a security mechanism that isolates each app so it can only access its own data and resources unless explicit permissions are granted. It includes a private directory in internal storage (`/data/data/<applicationId>/`)
   - Typical uses: App configuration files, sensitive information, cached data
2. **External storage**
   - Shared storage accessible by the user and other apps
   - Typical uses: Media files and user-generated content
     - Photos, videos, documents, downloads

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

4. Files - Internal storage

- The **internal storage** includes two main directories:

a) Files: `/data/data/<applicationId>/files/`
   - Characteristics:
     - Persistent storage
     - Used for files that must be kept long-term (removed when the app is uninstalled)
   - Example uses:
     - Configuration files
     - Private files

b) Cache: `/data/data/<applicationId>/cache/`
   - Characteristics:
     - Temporary storage
     - The system may delete these files when storage is low
   - Example uses:
     - Temporary files

4. Files - Internal storage


```
class InternalStorageHelper(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) {
            e.printStackTrace()
            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) {
            e.printStackTrace()
            ""
        }
    }
```

This helper class encapsulates the access to the internal storage

```
    // 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)
    }

}
```

4. Files - Internal storage


```
class MyViewModel(private val internalStorageHelper: InternalStorageHelper) : ViewModel() {

    private val _fileContent = MutableStateFlow("")
    val fileContent: StateFlow<String> = _fileContent.asStateFlow()

    private val _fileList = MutableStateFlow<List<String>>(emptyList())
    val fileList: StateFlow<List<String>> = _fileList.asStateFlow()

    fun writeToFile(fileName: String, content: String) {
        viewModelScope.launch {
            val success = withContext(Dispatchers.IO) {
                internalStorageHelper.writeToFile(fileName, content)
            }
            if (success) {
                refreshFileList()
            }
        }
    }

    fun readFromFile(fileName: String) {
        viewModelScope.launch {
            _fileContent.value = withContext(Dispatchers.IO) {
                internalStorageHelper.readFromFile(fileName)
            }
        }
    }
```

The helper class is used in a `ViewModel`, and their results are observed in the UI

```
    fun refreshFileList() {
        viewModelScope.launch {
            _fileList.value = withContext(Dispatchers.IO) {
                internalStorageHelper.listFiles().toList()
            }
        }
    }

    fun deleteFile(fileName: String) {
        viewModelScope.launch {
            val success = withContext(Dispatchers.IO) {
                internalStorageHelper.deleteFile(fileName)
            }
            if (success) {
                refreshFileList()
            }
        }
    }
}
```

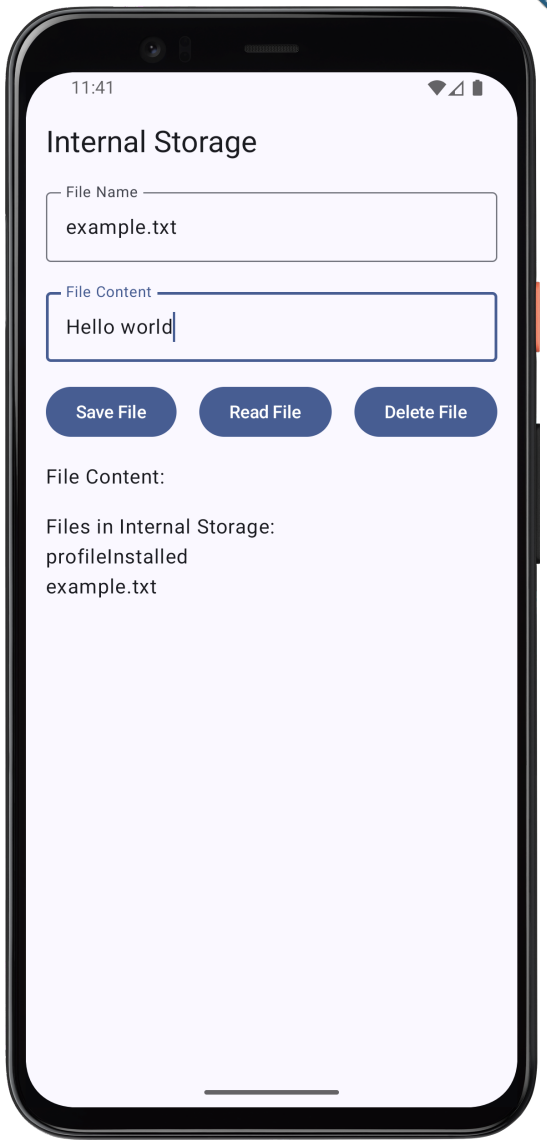
`withContext(Dispatchers.IO) {...}`: is a suspending function, so the coroutine is suspended while the file operation runs on the I/O dispatcher

4. Files - Internal storage


```
@Composable
fun MainScreen(modifier: Modifier = Modifier) {
    val viewModel: MyViewModel = viewModel(
        factory = viewModelFactory {
            initializer {
                val application = this[APPLICATION_KEY] as Application
                val helper = InternalStorageHelper(application)
                MyViewModel(helper)
            }
        })

        // ...

}
```

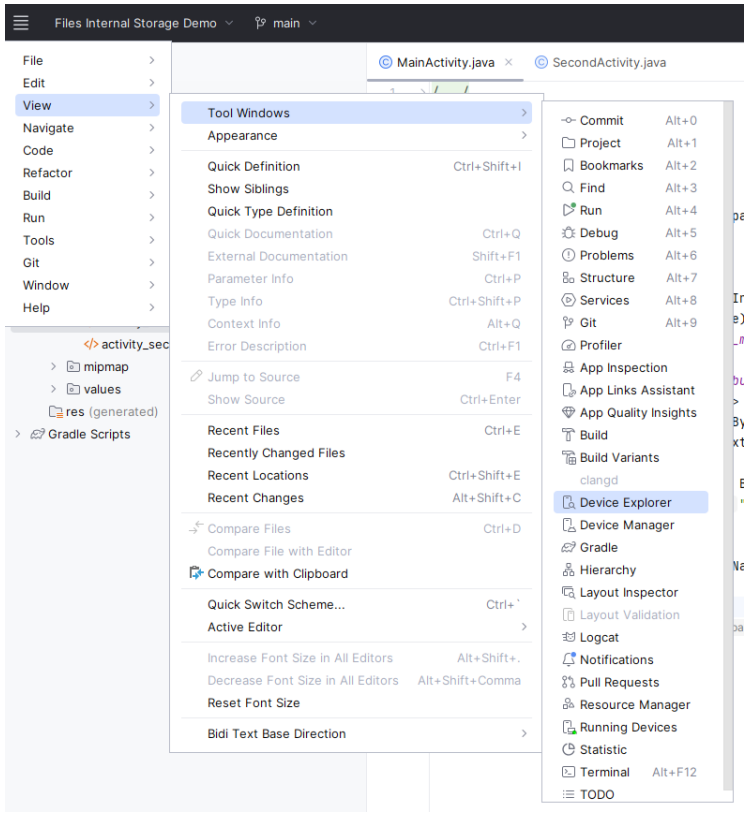
The application context is used because it lives longer than activities and prevents memory leaks

The previous `ViewModel` has a constructor parameter (`internalStorageHelper`). For that reason, the `ViewModel` is instantiated using a `factory`, so that we can pass dependencies

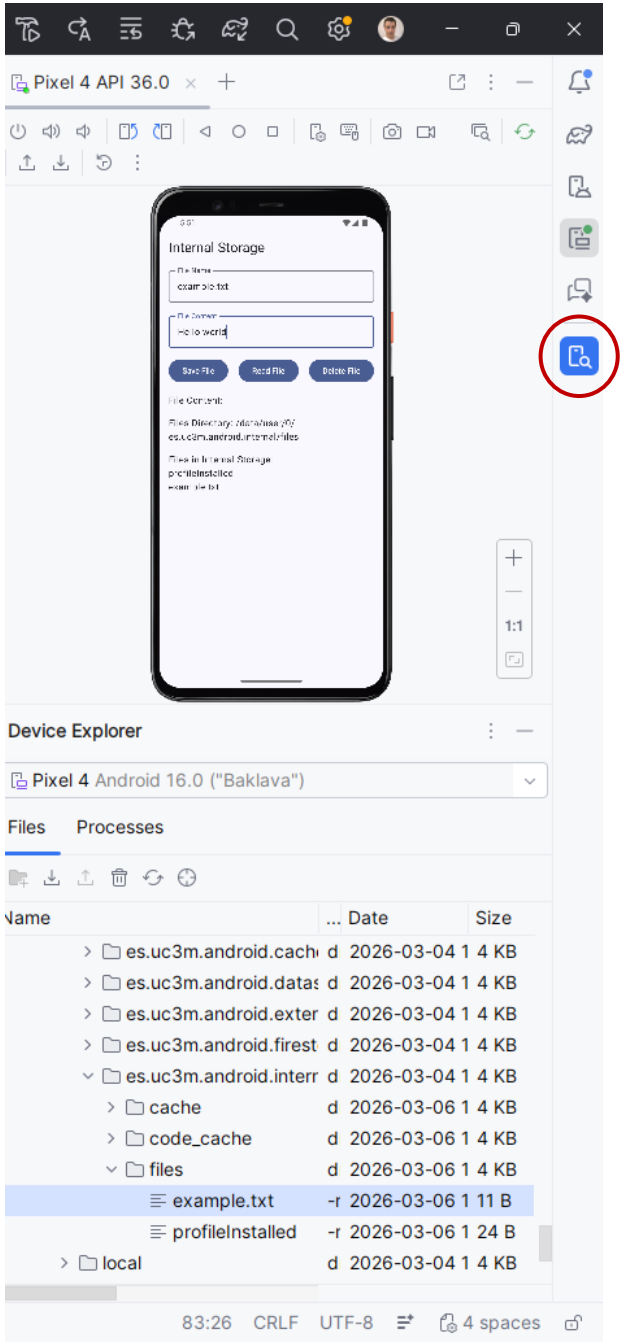


4. Files - Internal storage

- We can use the integrated **Device Explorer** in Android Studio to browser the file system:



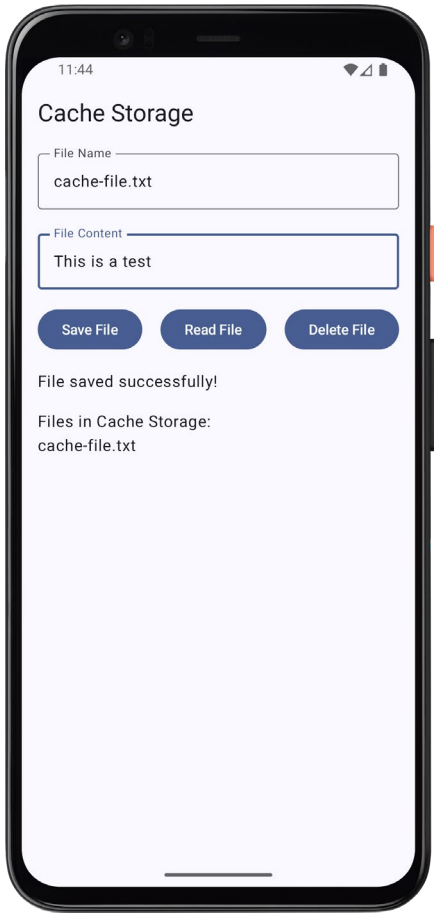
The device explorer can be opened in Android Studio using View → Tool Windows → Device Explorer



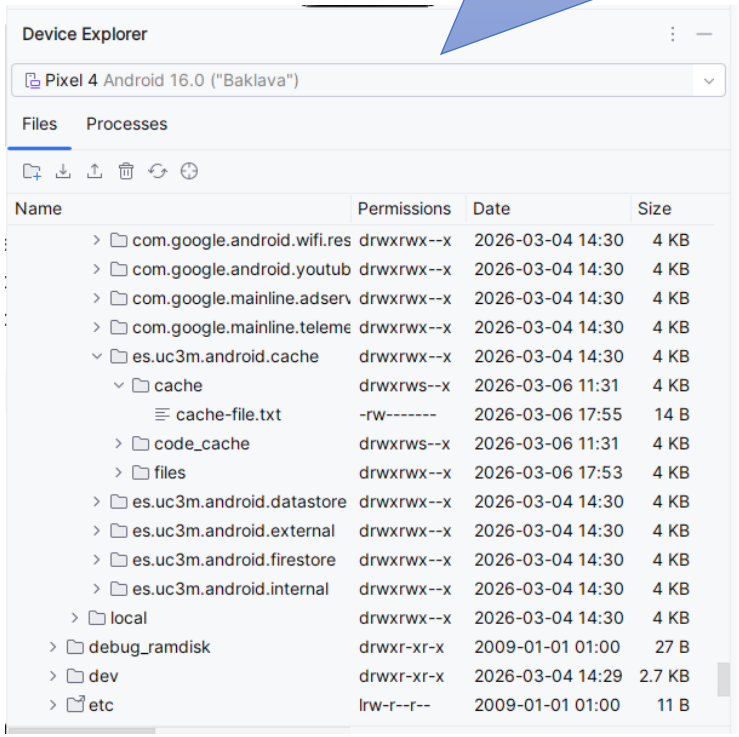
4. Files - Internal storage


```
class CacheFileHelper(private val context: Context) {

    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
        }
    }

    fun readFromCache(fileName: String): String {
        return try {
            val file = File(context.cacheDir, fileName)
            file.bufferedReader().use { reader ->
                reader.readText()
            }
        } catch (e: IOException) {
            e.printStackTrace()
            ""
        }
    }

    fun deleteFromCache(fileName: String): Boolean {
        val file = File(context.cacheDir, fileName)
        return file.delete()
    }

    fun listFiles(): Array<String> {
        return context.cacheDir.list() ?: emptyArray()
    }
}
```

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



We can use the device explorer in Android Studio to handle the cache files:



4. Files - External storage

- **External storage** is shared device storage that is visible to the user
  - Historically, Android devices included external storage in an SD card
- Since Android 10, Android uses **scoped storage** for external storage, using the following mechanisms:

1. App-specific external storage
   - Files stored in a directory dedicated to the app
2. MediaStore API
   - Media files (images from the gallery, videos/audios recorded by the user)
3. Storage Access Framework (SAF)
   - For accessing arbitrary files chosen by the user (e.g., PDFs, text files, etc.)
   - Instead of accessing file paths directly, the user selects the file through a system picker

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

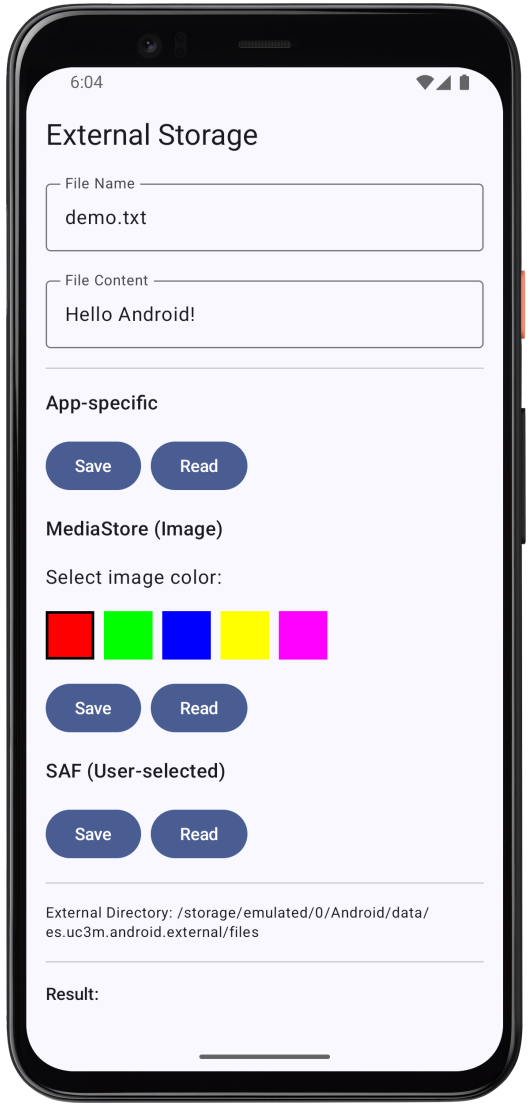
4. Files - External storage


```
class ExternalStorageHelper(private val context: Context) {

    // 1. App-specific external storage
    fun writeToAppSpecificStorage(fileName: String, content: String): Boolean {
        return try {
            val file = File(context.getExternalFilesDir(null), fileName)
            file.writeText(content)
            true
        } catch (e: IOException) {
            e.printStackTrace()
            false
        }
    }

    // 3. SAF (Storage Access Framework) logic
    fun writeToUri(uri: Uri, content: String): Boolean {
        return try {
            context.contentResolver.openOutputStream(uri).use { outputStream ->
                outputStream?.write(content.toByteArray())
            }
            true
        } catch (e: IOException) {
            e.printStackTrace()
            false
        }
    }
```

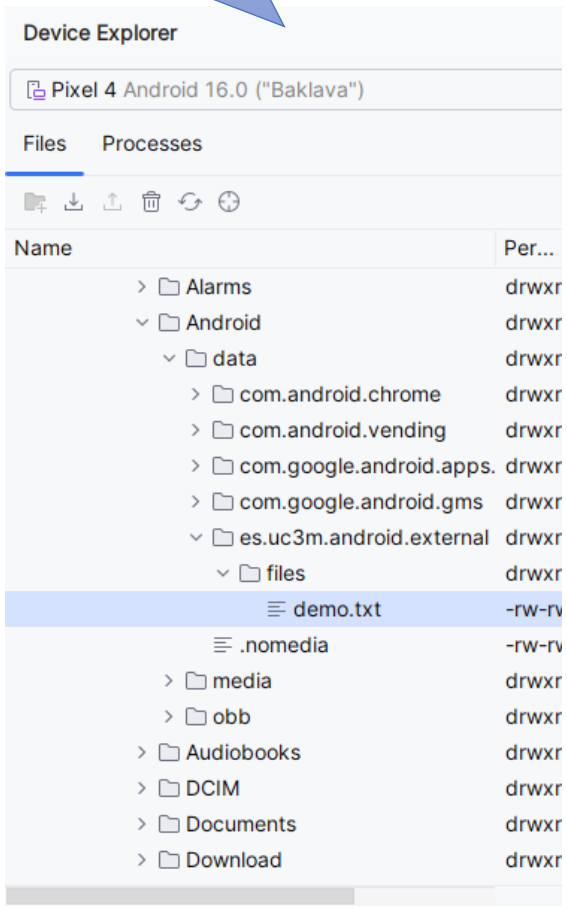
In this example, we use another helper class to encapsulate all the access to the external storage

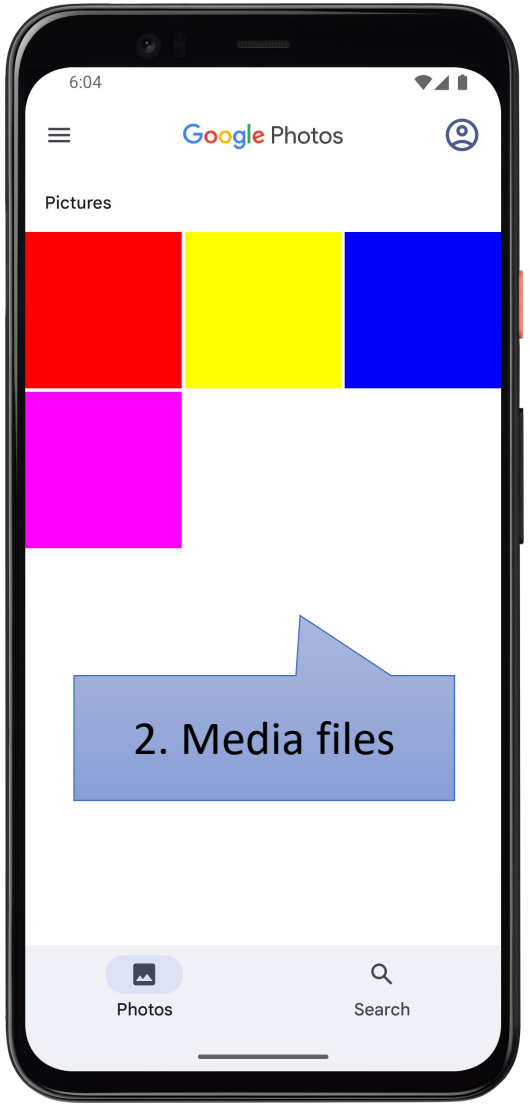
```
    // 2. MediaStore: Saving an image
    fun saveImageToMediaStore(bitmap: Bitmap, displayName: String): Uri? {
        val resolver = context.contentResolver
        val imageCollection = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
            MediaStore.Images.Media.getContentUri(MediaStore.VOLUME_EXTERNAL_PRIMARY)
        } else {
            MediaStore.Images.Media.EXTERNAL_CONTENT_URI
        }

        val contentValues = ContentValues().apply {
            put(MediaStore.Images.Media.DISPLAY_NAME, "$displayName.jpg")
            put(MediaStore.Images.Media.MIME_TYPE, "image/jpeg")
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
                put(MediaStore.Images.Media.RELATIVE_PATH, Environment.DIRECTORY_PICTURES)
            }
        }

        return try {
            val uri = resolver.insert(imageCollection, contentValues)
            uri?.let {
                resolver.openOutputStream(it).use { outputStream ->
                    if (outputStream == null || !bitmap.compress(Bitmap.CompressFormat.JPEG, 100, outputStream)) {
                        throw IOException("Failed to save bitmap")
                    }
                }
            }
            uri
        } catch (e: IOException) {
            e.printStackTrace()
            null
        }
    }

}
```

4. Files - External storage

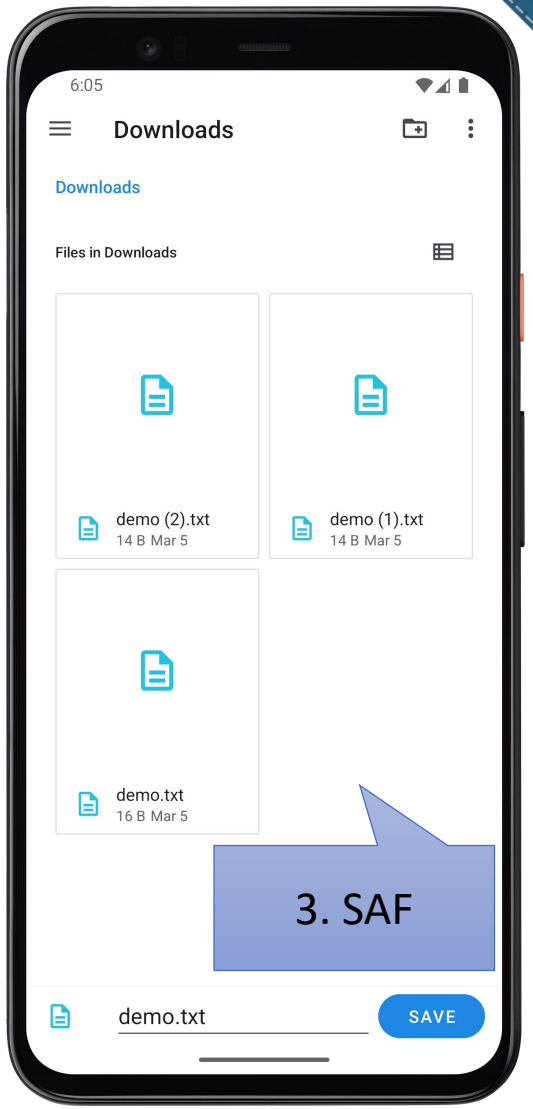


1. App-specific external storage





2. Media files



3. SAF

Table of contents

5. DataStore

- DataStore is an efficient way to store **key-value** pairs
  - It is designed to address the limitations of `SharedPreferences` (used in the legacy Android View system)
- Some advantages of DataStore are:
  - Asynchronous API: Uses Kotlin coroutines for asynchronous operations, avoiding blocking the main thread
  - Modern architecture: integrates naturally with coroutines, `Flow`, and `ViewModel`

build.gradle.kts (app)

```
dependencies {
    implementation(libs.androidx.datastore)
}
```

libs.version.toml

```
[versions]
datastore = "1.2.0"

[libraries]
androidx-datastore = { module =
"androidx.datastore:datastore", version.ref = "datastore" }
```

To use it, first we need to include the following dependency

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

5. DataStore

This example uses a DataStore to store two values:

- a username (string)
- a flag indicating whether the user is enabled (boolean)

```
private val Context.dataStore: DataStore<Preferences> by preferencesDataStore("settings")

private object PreferencesKeys {
    val USER_NAME_KEY = stringPreferencesKey("user_name")
    val IS_ENABLED_KEY = booleanPreferencesKey("enabled")
}

class DataStoreHelper(private val context: Context) {

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

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

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

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

The keyword suspend indicates that the function is a suspending function, which can pause and resume execution without blocking the thread. Suspending functions must be called from a coroutine scope

Flow is used to represent asynchronous values (i.e., it can emit updates over time). For that reason, suspend is not required here

5. DataStore

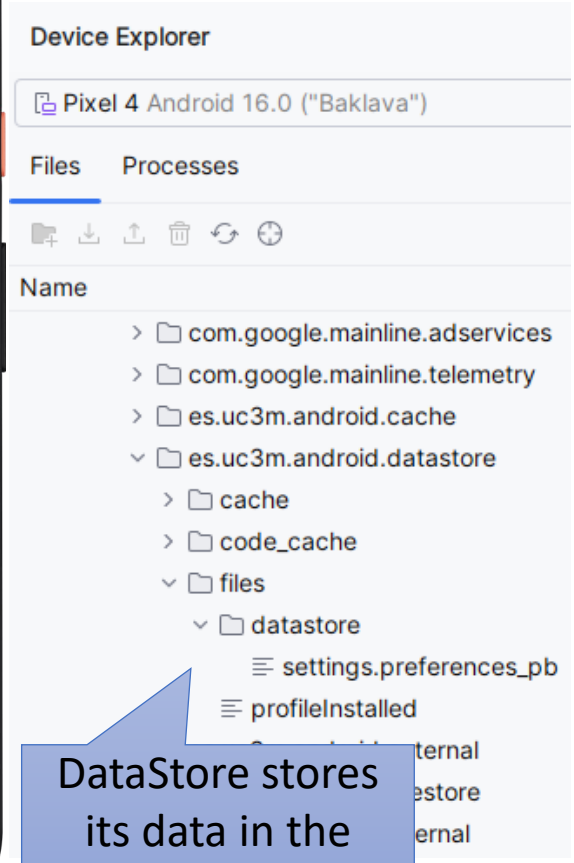
As usual, we use a `ViewModel` to interact with the previous `DataStoreHelper` to manage and persist user settings

```
class MyViewModel(private val dataStoreHelper: DataStoreHelper) : ViewModel() {

    private val _userName = MutableStateFlow("")
    val userName: StateFlow<String> = _userName.asStateFlow()

    private val _isEnabled = MutableStateFlow(false)
    val isEnabled: StateFlow<Boolean> = _isEnabled.asStateFlow()

    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)
        }
    }
}
```

5. DataStore


```
@Composable
fun SettingsScreen(
    modifier: Modifier = Modifier
) {
    val viewModel: MyViewModel = viewModel(
        factory = viewModelFactory {
            initializer {
                val application = this[APPLICATION_KEY] as Application
                val helper = DataStoreHelper(application)
                MyViewModel(helper)
            }
        })

    val userName by viewModel.userName.collectAsState()
    val enabled by viewModel.isEnabled.collectAsState()

    // UI ...

}
```

Finally, we use the ViewModel in our UI

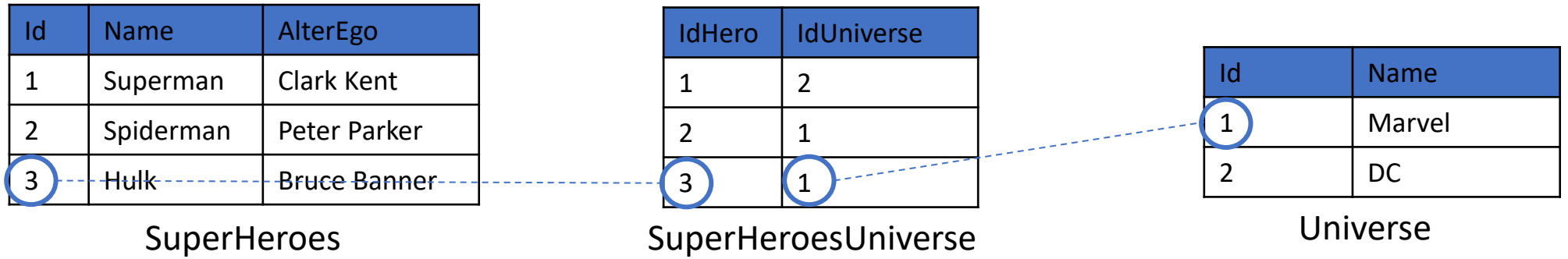


DataStore stores its data in the app's private internal storage

Table of contents

6. Local database - Relational databases

- A **relational database** stores data in tables made of rows and columns
  - Each table represents a type of entity
  - Each row stores one record
  - Each column stores one attribute
  - Rows are typically identified by a primary key
  - Relational databases are managed using SQL (Structured Query Language)



6. Local database - SQLite

- **SQLite** is a lightweight open-source relational database included by default in Android devices
- Each database is stored as a local file inside the app sandbox
  - Path: `/data/data/<applicationId>/databases`
- In Android development, we can manage SQLite through two different APIs:
  - `SQLiteOpenHelper` (low-level)
  - **Room**: (recommended high-level API). Benefits:
    - Compile-time verification of SQL queries
    - Custom annotations to map tables to Java
    - Improved support for database migration and preloaded databases

https://www.sqlite.org/

6. Local database - Room

- Major components in the Room API:

1. **Data entities** that represent tables in the database
   - Classes annotated with `@Entity` , containing attributes annotated with `@PrimaryKey` (for setting the primary key)
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. **Database class**, main access point with the database
   - A single Java/Kotlin class annotated with `@Database`

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

6. Local database - Room

1. Date entities (just one in this example, for the "notes" table):

```
@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

6. Local database - Room

2. DAOs interfaces (just one in this example, for the notes entity):

```
@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(): Flow<List<Note>>
}
```

The `suspend` keyword indicates that this is a suspending function, so it can perform database operations without blocking the thread

`getAllNotes()` does not use `suspend` because it returns a `Flow`, which emits database updates asynchronously over time

6. Local database - Room

3. Database class:

We need to 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

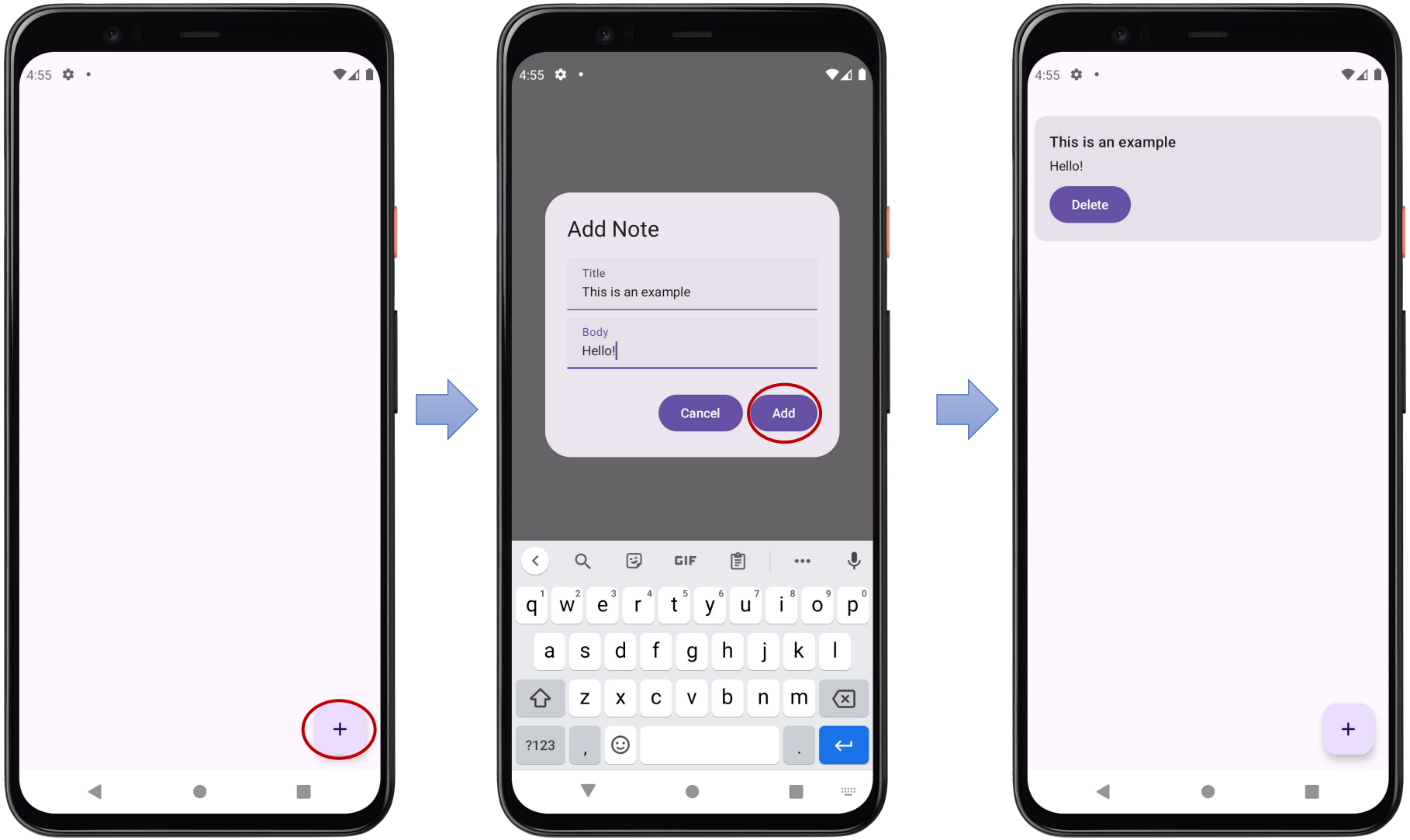
Creation of connection to the database using Room

6. Local database - Room


```
class MyViewModel(application: Application) : AndroidViewModel(application) {
    private val noteDao = NoteDatabase.getDatabase(application).noteDao()
    val notes: Flow<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)
        }
    }
}
```

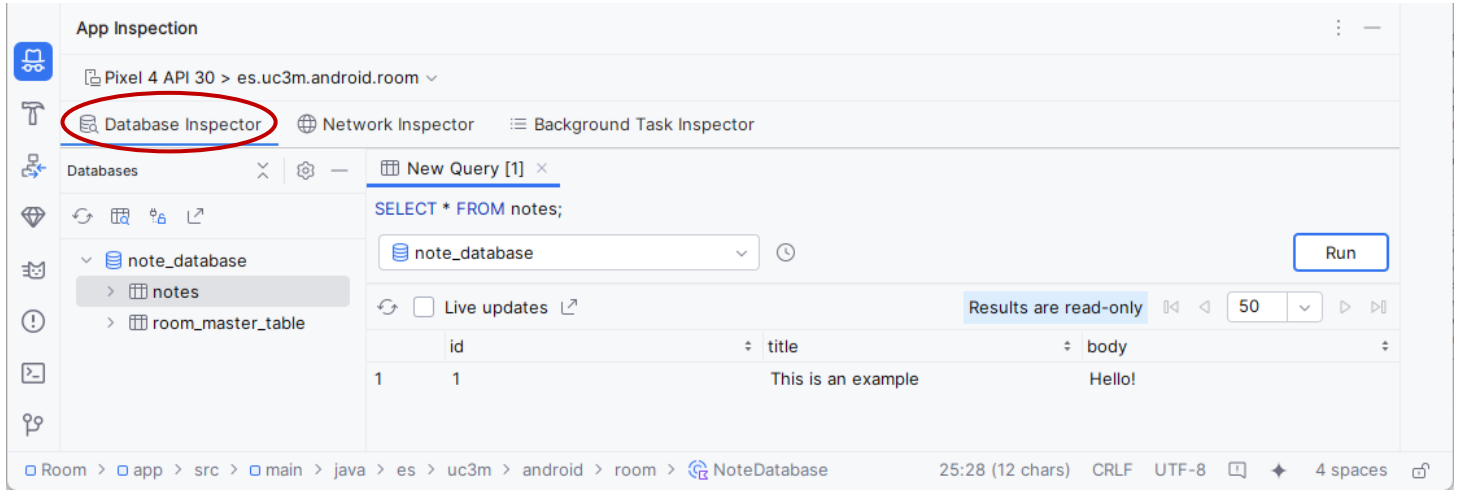
Finally, we access the database in our Kotlin logic (typically in a `ViewModel` class), invoking the DAO methods to manage the data

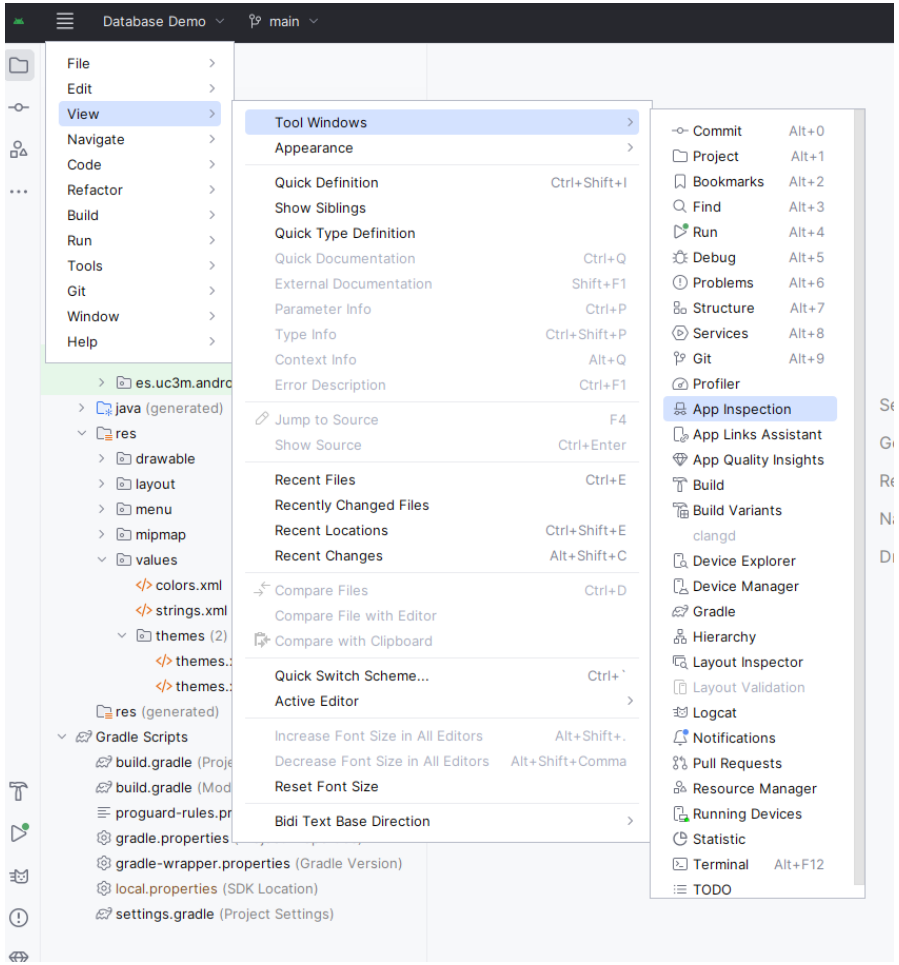
6. Local database - Room



6. Local database - Room

- We can interact with the database using an integrated tool in Android Studio: View → Tool Windows → App Inspection →
  - App Inspection → Database inspector





6. Local database - Preloaded database

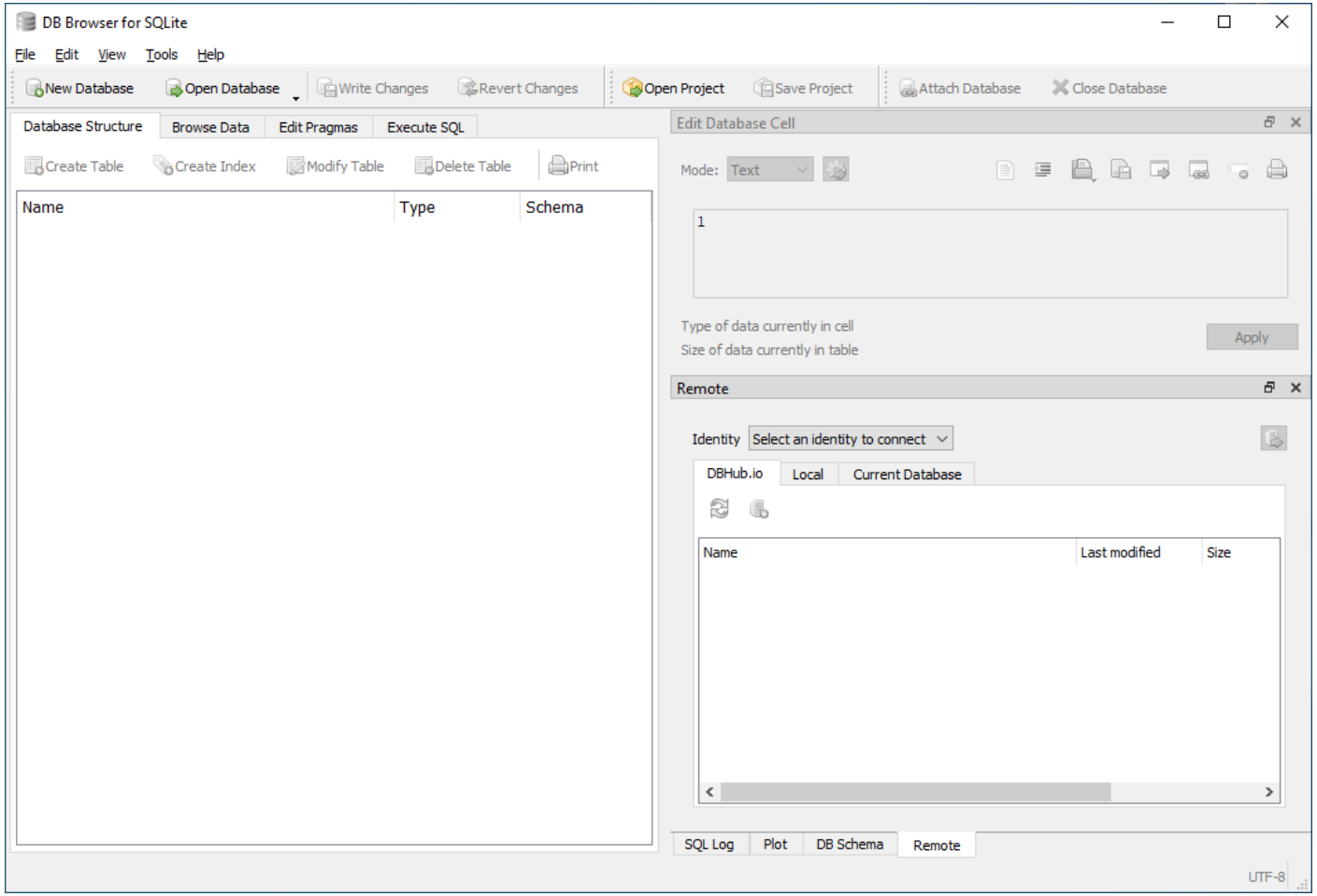
- It is possible to create a SQLite database using an external tool and copy it to our Android app
  - This option can be interesting for having preloaded data in our apps
- The procedure to use this **preloaded database** can be as follows:
  1. Create SQLite database. For instance, using DB Browser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

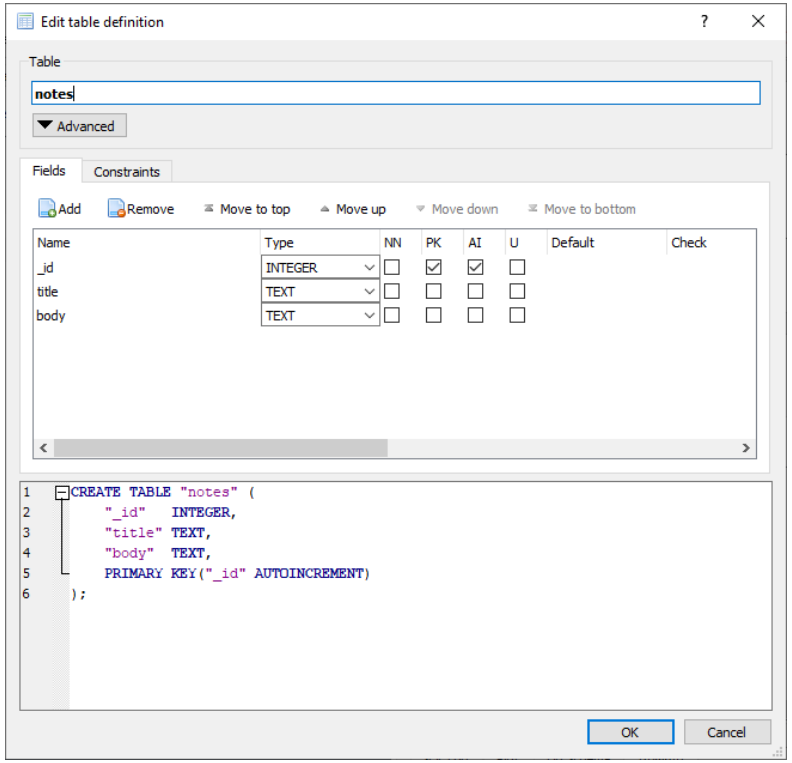
https://sqlitebrowser.org/

  2. Copy the SQLite database in an assets folder
  3. Load database in our project

6. Local database - Preloaded database

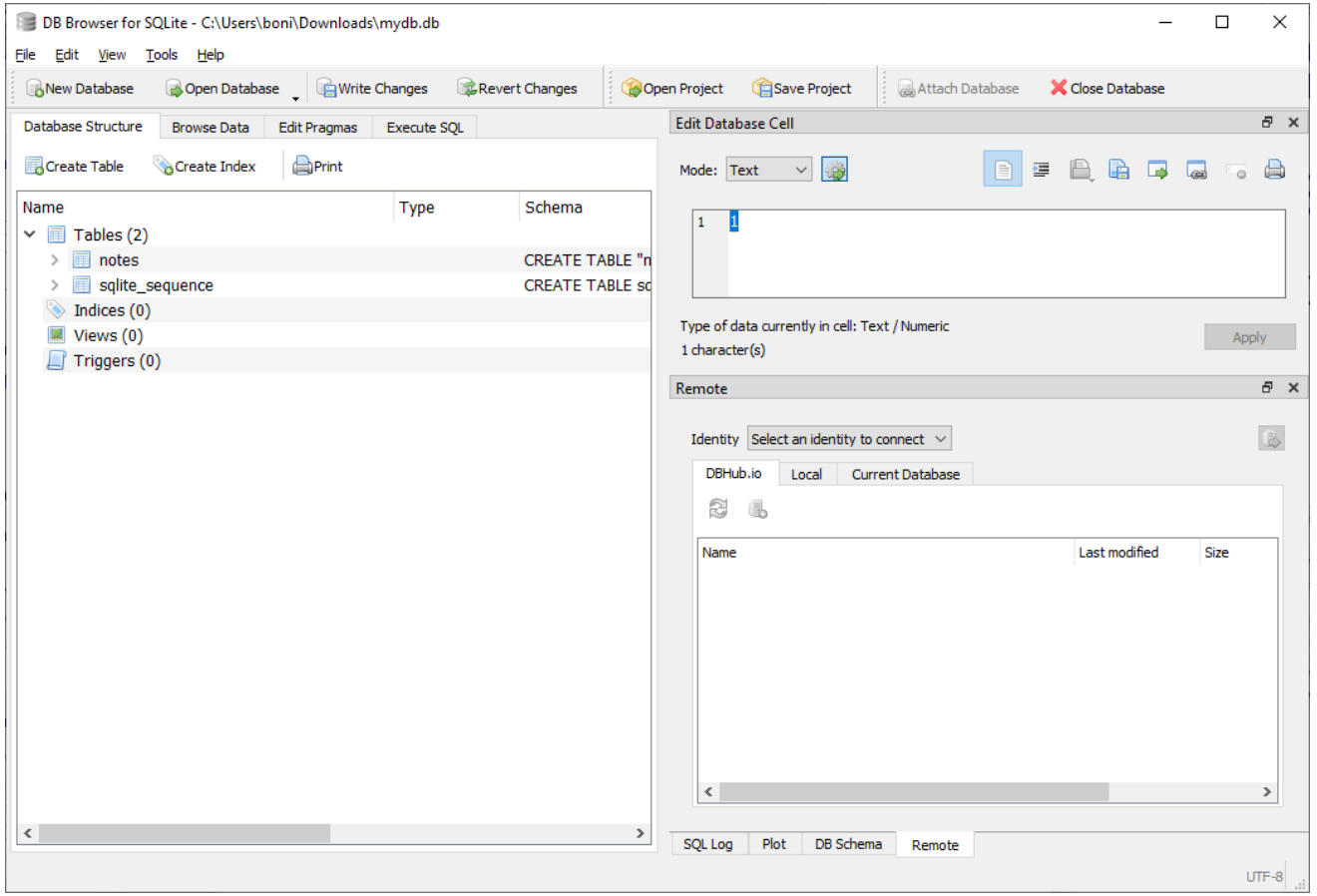
1. Create SQLite database

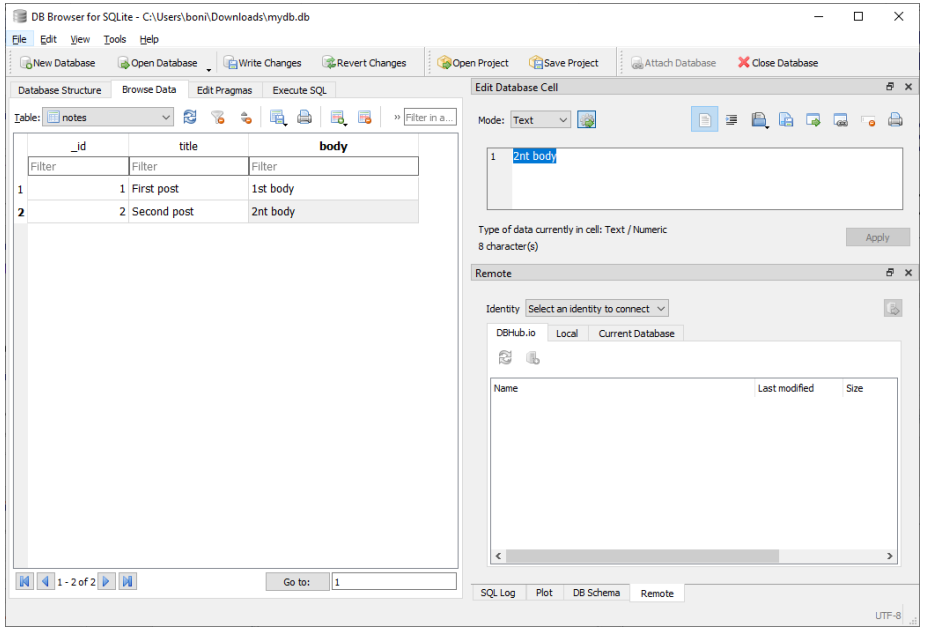




6. Local database - Preloaded database

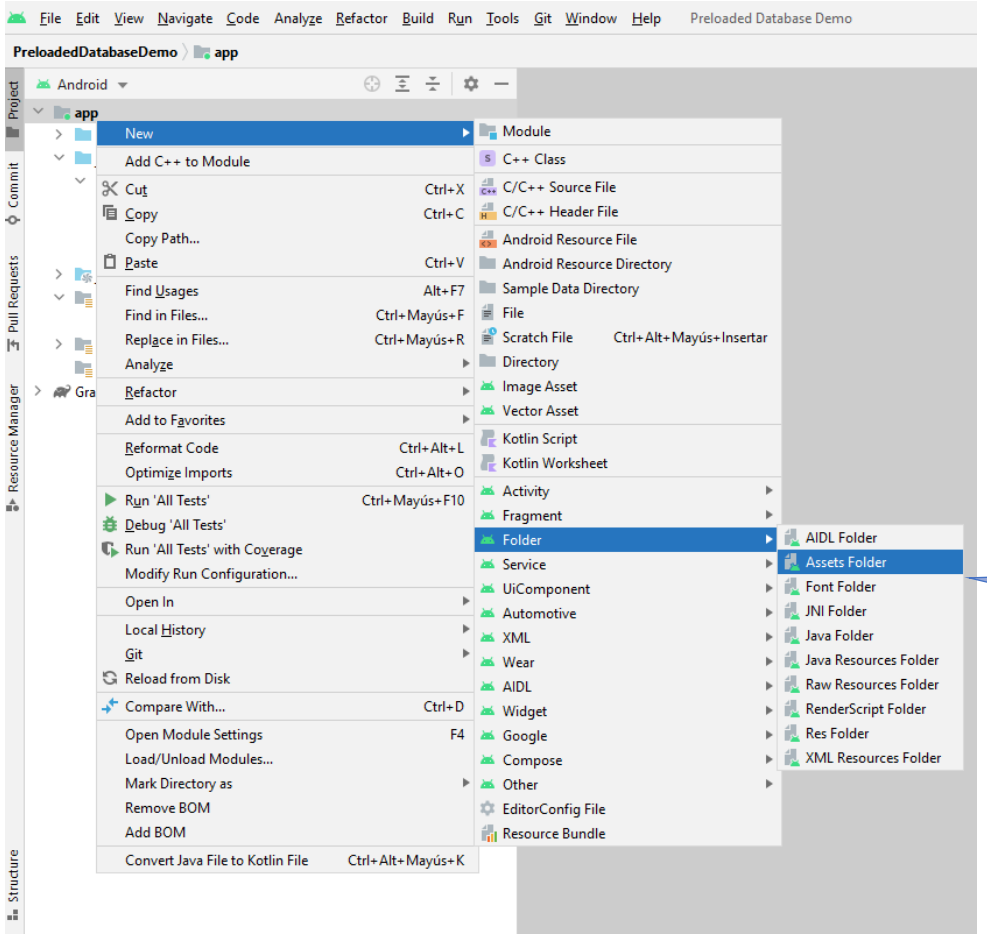
1. Create SQLite database





6. Local database - Preloaded database

2. Copy the SQLite database in an assets folder



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

6. Local database - Preloaded database

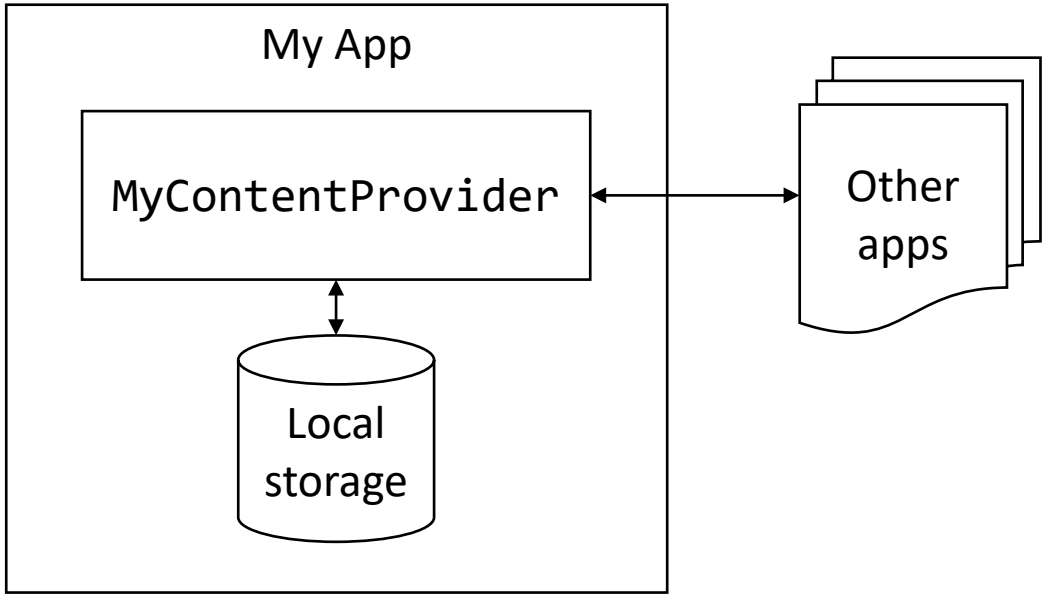
3. Load database in our project
   - We invoke the method `createFromAsset` in the database class

```
@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

7. Content providers

- A **content provider** is a type of app component in Android aimed to share data between different apps



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

7. Content providers

- A content provider exposes app data through a standard interface, and other apps can access that data using CRUD-style methods:
  - `insert()` → create
  - `query()` → read
  - `update()` → update
  - `delete()` → 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`**)
- Once created, the content provider is accessed in Android using a **content URI** (*Uniform Resource Identifier*)
  - The structure of a content URI is:

```
content://provider-authority/path
```

The application ID is typically used as provider authority in the URI

7. 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 {
        val ctx = context ?: return false
        dbHelper = MyDatabaseHelper(ctx)
        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}"
    }
}
```

The demo app ContentProvider provides a basic example using a content provider

Table of contents

8. Takeaways

- Android apps can store data locally or remotely
- Coroutines help perform storage and network operations without blocking the UI
- Firebase provides cloud-based storage and authentication services
- Files can be stored in:
  - Internal storage
  - App-specific external storage
  - Shared storage through MediaStore or SAF
- DataStore is recommended for small key-value settings
- Room is the recommended API for structured local relational data
- Content providers allow apps to expose and share data with other apps

8. Takeaways

- Choose the storage mechanism according to the **type of data**, its **lifetime**, and whether it must be **shared**

Storage	**Use it for**	**Example**
DataStore	Small key-value data	Settings, preferences
Internal storage	Private app files	Config files, sensitive files, cached files
App-specific external storage	Large app-generated files	Exported files, downloads
MediaStore	Shared media files	Photos, videos, audio
SAF	Files explicitly chosen by the user	Documents, PDFs
Local database (SQLite)	Structured local relational data	Offline, non-shared data
Firestore	Remote cloud data	Shared data between users