Hanoi University of Science and Technology
School of Information and Communication Technology

Mobile Programming

Note-taking application

Supervisor: PhD. Nguyen Dinh Thuan
Members:

- Chu Trung Anh 20225564
- Bui Duy Anh 20225563

Class: Mobile Programming (IT4785E)-152475

Semester: 2024.1

Table of Contents

Abstract

This project presents the development of a note-taking mobile application for Android, implemented using Kotlin. The application is designed to enhance productivity by allowing users to create, edit, and manage notes seamlessly. It incorporates a client-server architecture, enabling synchronization with a remote server to retrieve and store data, ensuring data consistency across multiple devices. The app also supports offline functionality by leveraging SQLite as a local database, allowing users to access and modify their notes without an active internet connection. Changes made offline are queued for synchronization with the server once connectivity is restored. This dual storage mechanism ensures data reliability and availability, making the app a versatile tool for personal and professional use. The development emphasizes user-friendly design, efficient data handling, and robust synchronization protocols to deliver a high-performance and practical note-taking solution.

Acknowledgements

We would like to express our heartfelt gratitude to Mr. Nguyen Dinh Thuan, our lecturer, for his dedicated teaching of the Mobile Programming course. His detailed guidance and support played a crucial role in helping us complete our project on the Note taking app project. This project has been an invaluable learning experience, and we truly appreciate his contributions to our understanding of the subject.

We also wish to thank our friends and families for their unwavering support and encouragement throughout this project. Their understanding and motivation have been essential in helping us manage our time and complete this work successfully.

I. Introduction

1. Background and context

In the digital age, note-taking has evolved from pen and paper to powerful digital tools that enhance productivity and organization. Mobile applications have become an essential medium for this evolution, offering users the ability to create, manage, and access notes anytime and anywhere. However, many existing note-taking apps rely heavily on persistent internet connectivity, limiting their usability in scenarios where users have limited or no access to a network.

To address this gap, this project explores the development of a note-taking mobile application for Android using Kotlin, designed to work both online and offline. By integrating server communication capabilities, the app ensures seamless data synchronization across devices, enabling users to retrieve and store notes on a remote server. Simultaneously, the app incorporates SQLite as a local database, ensuring users can access and edit their notes even without internet access.

The implementation of offline capabilities is critical in enhancing user experience, as it eliminates reliance on constant connectivity while ensuring data consistency through effective synchronization mechanisms. Furthermore, Kotlin's modern programming features and Android’s robust development ecosystem make it an ideal choice for building this application, ensuring a clean, maintainable, and scalable codebase.

This project is motivated by the need for a versatile and reliable note-taking solution that caters to both individual and collaborative usage scenarios. It seeks to provide a balance between accessibility, performance, and data security, addressing the challenges commonly associated with offline and online hybrid applications.

2. Objectives

The primary goal of this project is to develop a mobile application that allows users to create, manage, and access notes both online and offline. Key objectives include:

1. Enabling seamless data synchronization with a remote server to ensure consistency across devices.
2. Implementing a local SQLite database to facilitate offline access and editing of notes.
3. Designing an intuitive and user-friendly interface to enhance usability.
4. Ensuring data security and reliability throughout the application.

3. Scope

The application offers core note-taking features, including creating, editing, and deleting notes, with seamless synchronization between local storage and a remote server. It supports offline functionality by saving changes locally and syncing them once internet connectivity is restored. However, the app is limited to text-based notes and does not currently support multimedia content like images or voice recordings. Additionally, the synchronization mechanism assumes a reliable server connection and does not include advanced collaboration features such as real-time editing or multi-user access.

4. Audience

This application is designed for a wide range of users, including students, professionals, and anyone seeking an efficient way to manage their notes. It caters to individuals who require access to their notes across multiple devices while working in environments with intermittent or no internet connectivity. By balancing offline capabilities and online synchronization, the app is particularly beneficial for users who travel frequently or operate in remote areas.

II. Application Overview

1. App Description

The note-taking mobile application is a productivity tool designed to help users create, manage, and organize their notes efficiently. It enables users to access their notes both online and offline, with seamless data synchronization to a remote server. This hybrid approach ensures that users can continue working on their notes even in environments with limited or no internet connectivity.

2. Core Features

- Create and Edit Notes: Users can easily create new notes and update existing ones.
- Offline Access: Notes are stored locally using SQLite, allowing access and modifications without internet connectivity.
- Data Synchronization: Changes made offline are automatically synced with a remote server when connectivity is restored.
- Delete Notes: Users can delete notes locally, and deletions are synchronized with the server.
- User-Friendly Interface: A clean and intuitive UI ensures ease of navigation and usability.
- Sync Status Indicators: Visual cues to show whether notes are synced or pending synchronization.

3. Target Platform

This application is developed specifically for Android devices, ensuring compatibility with a wide range of smartphones and tablets running Android OS.

4. Technology Stack

- **Language**: Kotlin
- **Database**:
  - **Local**: SQLite for offline storage (Using ROOM dependency to interact with the database)
  - **Remote**: Firebase Cloud Storage
- **Development Tools**: Android Studio for coding and debugging.
- **Testing**: JUnit for unit testing, along with instrumentation tests for app-wide validation.

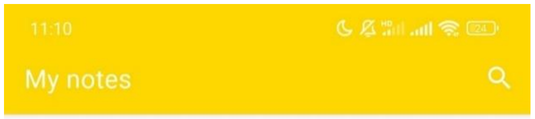
This combination of features and technologies ensures the application is efficient, reliable, and user-friendly.

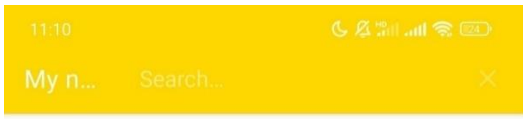
III. Design and Architecture

1. UI/UX Design

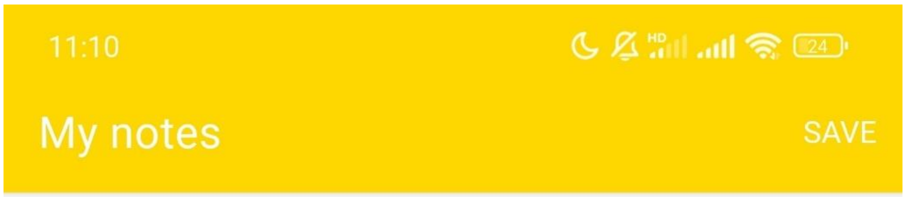
Our application contains create menu: search menu, save menu and delete menu:

- **Search Menu**: (When clicking the search icon) (defined in res/menu/home_menu.xml)

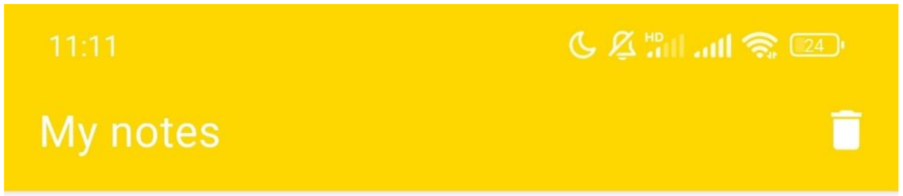




- **Save Menu**: (defined in res/menu/menu_add_note.xml)

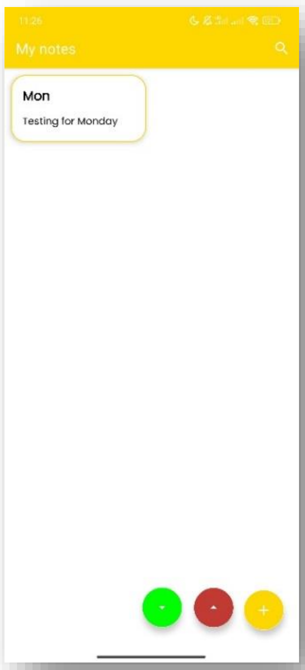


- **Delete menu**: (defined in res/menu/menu_edit_note.xml)



These menus are attached to three corresponding fragments:

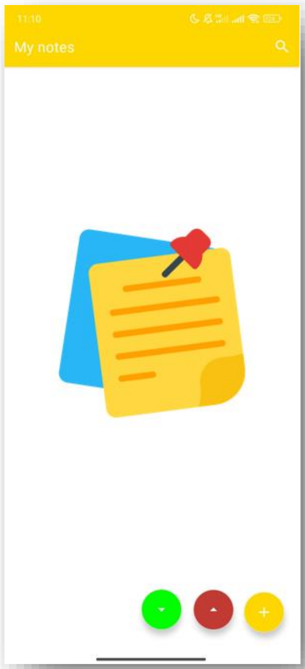
- **The home fragment with Search Menu**: (defined in res/layout/fragment_home.xml)



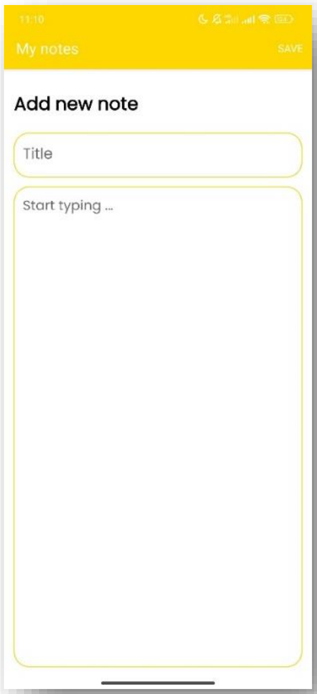
There are three floating buttons near the bottom of the fragment:

- **Download note button**: Download/Fetch all notes data from cloud to local
- **Upload note button**: Send all notes data from local to the cloud
- **Add note button**: use to add a new note to the recycle view. When user click to this button, this will navigate to the add note fragment

When there are no notes available, the RecyclerView's visibility will be set to GONE, while the visibility of the empty_sticky_note.png image will be set to VISIBLE. Conversely, when notes are present, the RecyclerView's visibility will be set to VISIBLE, and the image's visibility will be set to GONE.

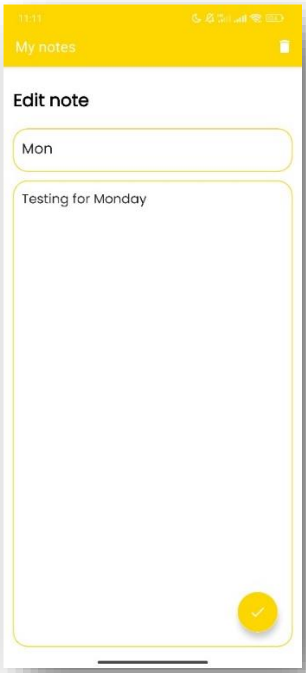


- **The add note fragment**: (defined in res/layout/fragment_add_note.xml)



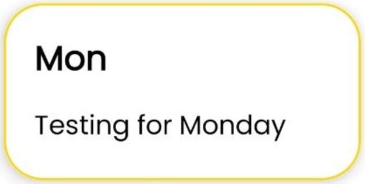
There is a save button at the right top corner to save the new created note

- **Edit note fragment**: (defined in res/layout/fragment_edit_note.xml)



Like the add note fragment, but the difference is the delete button at the right top corner and to floating save note button to update the data for the edited note.

- Notes are displayed using a recycle view, with a note items are defined in res/layout/note_layout.xml. Each note will have a title and a content section, covered in a gold color border:



All these layouts are using binding layout type in Android. Binding layouts are used to facilitate the data binding process, which enables you to bind UI components in your layouts to data sources in your application using a declarative format, rather than programmatically.

The `layout` tag serves as the root element, and it is a requirement for enabling data binding in the layout. For example, in the fragment_home.xml file

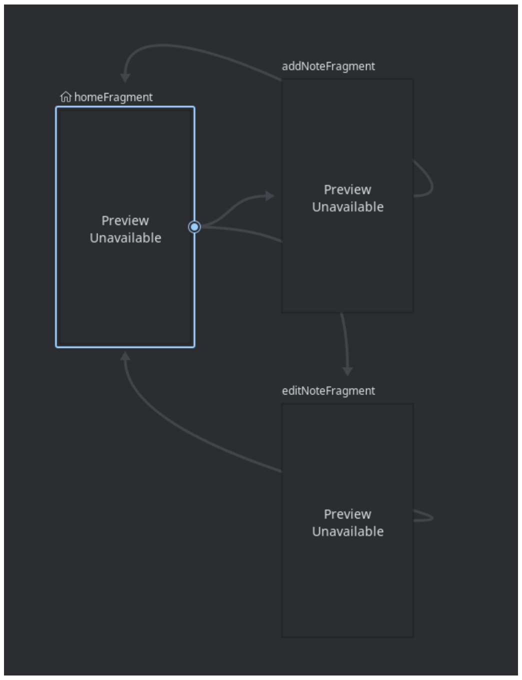
```
<?xml version="1.0" encoding="utf-8"?>
<layout
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".HomeFragment">
        .....
```

The main_layout.xml displays all these three fragments using a FragmentContainerView, which serves as a container for the fragments.

~~~
```xml
 <androidx.fragment.app.FragmentContainerView
            android:id="@+id/fragmentContainerView"
            android:name="androidx.navigation.fragment.NavHostFragment"
            android:layout_width="0dp"
            android:layout_height="0dp"
            android:layout_marginStart="1dp"
            android:layout_marginTop="1dp"
            android:layout_marginEnd="1dp"
            android:layout_marginBottom="1dp"
            app:defaultNavHost="true"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            app:navGraph="@navigation/nav_graph" />
~~~

We also set up as a **NavHostFragment** to handle navigation between three fragments. Navigation is managed via a **navigation graph** (defined in the res/navigation/nav_graph file). The nav graph is used to define all possible paths that the user can take in the app. It is a visual representation of the app's navigation graph.



2. Database

We use the ROOM library to simplify and abstract the process of working with databases in Android applications. ROOM provides a higher-level, object-oriented abstraction over SQLite, making it easier to define and interact with the database while ensuring compile-time verification of SQL queries. It has three main components: the **Entity class**, the **DAO interface**, and the **Database class**. These components work together to establish the structure and functionality of the database.

- **Entity Class:** represents a table in the database. Each instance of the entity corresponds to a row in the table. In our case, we only need a Note object as entity (defined inside /model/Note.kt)

```
@Entity(tableName = "notes")
@Parcelize
data class Note(
    @PrimaryKey(autoGenerate = true)
    var id: Int = 0,
    val noteTitle: String = "",
    val noteDesc: String = ""
) : Parcelable
```

The Note class also includes **Parcelization**, which is a mechanism to convert complex data objects into a simpler format. Parcelization allows these objects to be passed easily between fragments or activities, ensuring seamless data transfer in the application. To do that, we also need to include the Note object as an argument to pass between fragments in the nav_graph:

```
 <argument
            android:name="note"
            app:argType="com.example.snote.model.Note"
            app:nullable="true"/>
```

- **DAO (Data Access Object):** an interface that provides methods to interact with the database. It is used to define SQL queries and map them to method calls. The DAO acts as a bridge between the database and the rest of your app. It abstracts the database logic, allowing developers to focus on high-level operations without worrying about SQL syntax at runtime. In our case, we defined operations like: inserting, updating, deleting (a single note and all notes), and querying note base on note tittle or note content. (defined inside /database/NoteDao.kt)

```
@Dao
interface NoteDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertNote(note: Note)
    // onConflict = OnConflictStrategy.REPLACE is used to replace the existing
note with the new one if the note with the same id already exists.
    // suspend keyword indicates that the function is called from coroutine. It
runs in a background thread.

    @Update
    suspend fun updateNote(note: Note)

    @Delete
    suspend fun deleteNote(note: Note)


    @Query("DELETE FROM notes")
    suspend fun deleteAllNotes()

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

    @Query("SELECT * FROM notes WHERE noteTitle LIKE :query OR noteDesc LIKE
:query")
    fun searchNote(query: String?): LiveData<List<Note>>
    // Query no need to be suspend because since it already run on a special
background thread, specifically designed for Room database operations.
}
```

- **3. Database Class:** annotated with @Database and serves as the main access point to the database. It integrates all the entities and DAOs, enabling Room to handle database creation, setup, and versioning. The database class provides a singleton instance of the database, ensuring efficient resource use and thread safety while managing database connections. (defined inside /database/NoteDatabase.kt)

```
@Database(entities = [Note::class], version = 1)
abstract class NoteDatabase: RoomDatabase() {

    abstract fun getNoteDao(): NoteDao

    companion object{
        @Volatile
        private var instance: NoteDatabase? = null
        private val LOCK = Any() // to make sure only one instance of database
is created

        operator fun invoke(context: Context) = instance ?:
        synchronized(LOCK){
            instance ?:
            createDatabase(context).also{
                instance = it
            }
        }

        private fun createDatabase(context: Context) =
            Room.databaseBuilder(
                context.applicationContext,
                NoteDatabase::class.java,
                "note_db"
            ).build()
    }
}
```

3. Setting Up Recycler View

RecyclerView is used to efficiently display a large list of data items in the UI by reusing views as they scroll in and out of the screen. For this, we need an Adapter and a ViewHolder to bind the data to the respective UI elements dynamically.

Implementing the Adapter and ViewHolder

- The NoteAdapter class extends RecyclerView.Adapter and binds data to the RecyclerView. Here's the detailed implementation: (defined inside /adapter/NoteAdaptor.kt)

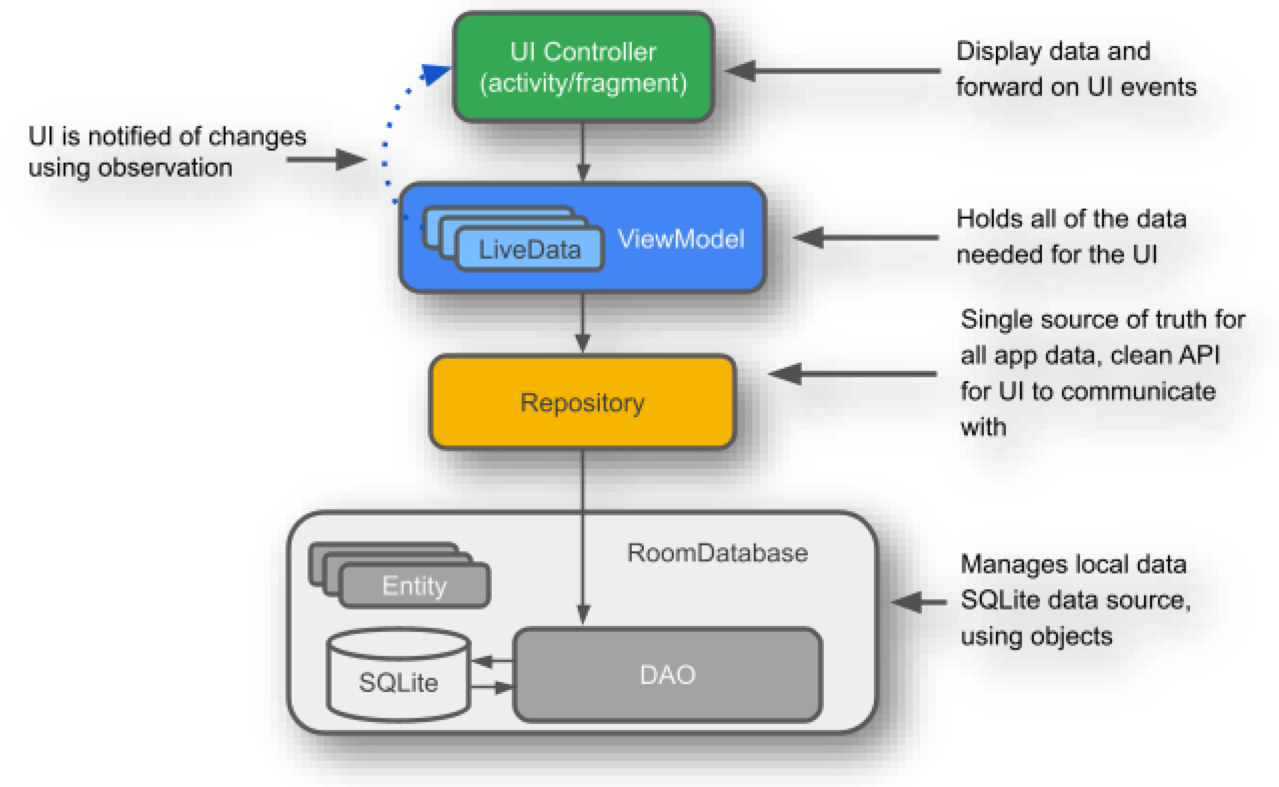
```
class NoteAdapter : RecyclerView.Adapter<NoteAdapter.NoteViewHolder>() {

    class NoteViewHolder(val itemBinding: NoteLayoutBinding):
RecyclerView.ViewHolder(itemBinding.root)
    // Each item in the RecyclerView is represented by a ViewHolder object,
defined each items layout in the NoteLayoutBinding class.

    // These below function are to efficiently update the UI without having to
refresh the entire list., by comparing the old and new list of items.
    private val differCallback = object : DiffUtil.ItemCallback<Note>(){
        override fun areItemsTheSame(oldItem: Note, newItem: Note): Boolean {
            return oldItem.id == newItem.id &&
                    oldItem.noteDesc == newItem.noteDesc &&
                    oldItem.noteTitle == newItem.noteTitle
        }

        override fun areContentsTheSame(oldItem: Note, newItem: Note): Boolean
{
            return oldItem == newItem
        }
    }
    val differ = AsyncListDiffer(this, differCallback)

    // Three functions that must be implemented in the RecyclerView.Adapter
class.
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
NoteViewHolder {
        return NoteViewHolder(
            NoteLayoutBinding.inflate(LayoutInflater.from(parent.context),
parent, false)
        )
    }

    override fun getItemCount(): Int {
        return differ.currentList.size
    }

    override fun onBindViewHolder(holder: NoteViewHolder, position: Int) {
        val currentNote = differ.currentList[position]

        holder.itemBinding.noteTitle.text = currentNote.noteTitle
        holder.itemBinding.noteDesc.text = currentNote.noteDesc

        // When user click on a note, it will navigate to the EditNoteFragment
with the current note as an argument.
        holder.itemView.setOnClickListener {
            val direction =
HomeFragmentDirections.actionHomeFragmentToEditNoteFragment(currentNote)
            it.findNavController().navigate(direction)
        }
    }
}
```

4. Setting Up ViewModel

The **MVVM** (Model-View-ViewModel) architecture is followed in the application to separate concerns:

- **Model**: Represents the data (Note object in this case).
- **ViewModel**: Acts as a bridge between the View (UI) and the Model (data). It manages UI-related data and handles business logic.
- **View**: The UI layer (e.g., Fragment or Activity).

Here is the MVVM structure:


UI Controller (activity/fragment)
Display data and forward on UI events
UI is notified of changes using observation
LiveData
ViewModel
Holds all of the data needed for the UI
Repository
Single source of truth for all app data, clean API for UI to communicate with
RoomDatabase
Entity
SQLite
DAO
Manages local data SQLite data source, using objects

- **Repository:** The repository encapsulates the logic for accessing and manipulating data from the Room database. It ensures that the ViewModel doesn't directly interact with the DAO and provides a clean API to the rest of the app.

```
class NoteRepository(private val db: NoteDatabase) {

    suspend fun insertNote(note: Note) = db.getNoteDao().insertNote(note)
    suspend fun deleteNote(note: Note) = db.getNoteDao().deleteNote(note)
    suspend fun deleteAllNotes() = db.getNoteDao().deleteAllNotes()
    suspend fun updateNote(note: Note) = db.getNoteDao().updateNote(note)

    fun getAllNotes() = db.getNoteDao().getAllNotes()
    fun searchNote(query: String?) = db.getNoteDao().searchNote(query)
}
```

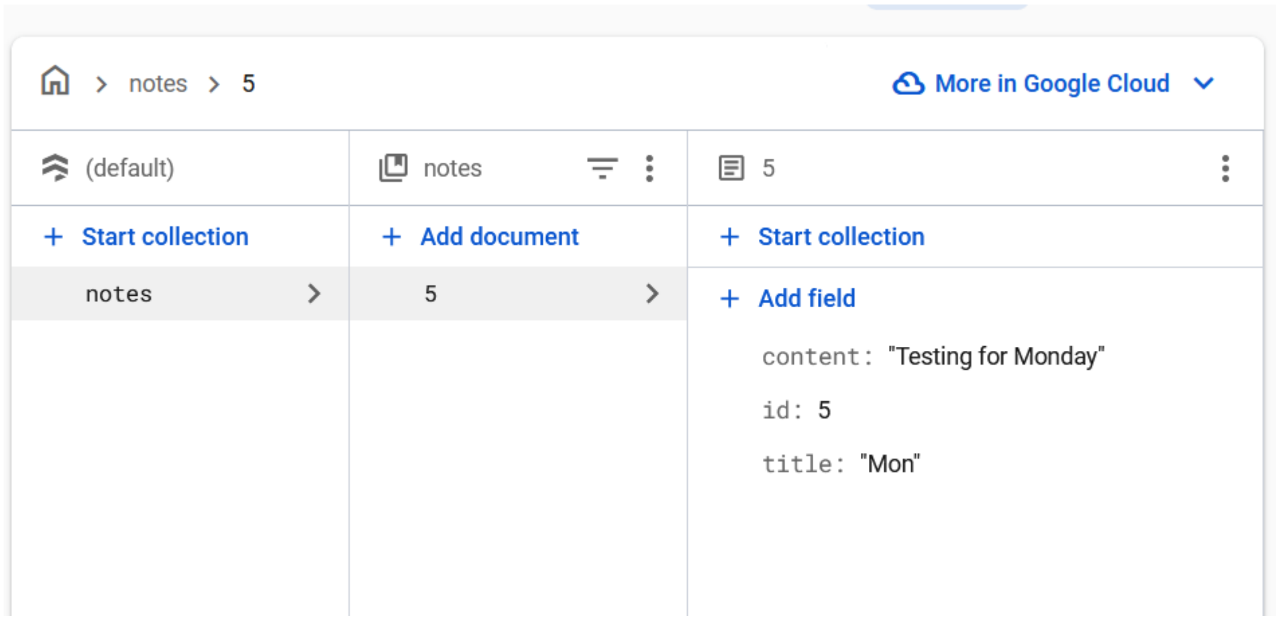
The repository handles both:

    - CRUD operations (Insert, Delete, Update, etc.) using DAO.
    - Data retrieval from Room.

- **ViewModel:** The ViewModel fetches data from the repository and provides it to the UI. It also handles any business logic.

```
class NoteViewModel(app: Application, private val noteRepository:
NoteRepository): AndroidViewModel(app) {

    fun addNote(note: Note) =
        viewModelScope.launch {
            noteRepository.insertNote(note)
        }
    // viewModelScope is a predefined CoroutineScope that is bound to the
lifecycle of the ViewModel.

    fun deleteNote(note: Note) =
        viewModelScope.launch {
            noteRepository.deleteNote(note)
        }

    fun deleteAllNotes() =
        viewModelScope.launch {
            noteRepository.deleteAllNotes()
        }

    fun updateNote(note: Note) =
        viewModelScope.launch {
            noteRepository.updateNote(note)
        }

    fun getAllNotes() = noteRepository.getAllNotes()

    fun searchNote(query: String?) = noteRepository.searchNote(query)

    // Upload a single note object to Firestore CLoud Database
    fun uploadNoteToFirebase(note: Note) {
        val firestore = FirebaseFirestore.getInstance()
        Log.i("MyTag", "Receive single note from home fragment: $note, sending
to Firestore")
        val noteMap = hashMapOf(
            "id" to note.id,
            "title" to note.noteTitle,
            "content" to note.noteDesc
        )
        Log.d("MyTag", "Attempting to save note to Firestore: $noteMap")
        firestore.collection("notes").document(note.id.toString()).set(noteMap)
            .addOnSuccessListener {
                Log.d("MyTag", "Note added/updated in Firestore successfully")
                Toast.makeText(getApplication(), "Note uploaded to Firestore
successfully", Toast.LENGTH_SHORT).show()
            }
            .addOnFailureListener { e ->
                Log.w("MyTag", "Error adding/updating note in Firestore", e)
                Toast.makeText(getApplication(), "Error uploading note to
Firestore", Toast.LENGTH_SHORT).show()
            }
    }

    // Fetch all notes from Firestore Cloud Database then update the local
database, UI
    fun downloadNotesFromFirebase() {
        val firestore = FirebaseFirestore.getInstance()
        viewModelScope.launch {
            try {
                val snapshot = firestore.collection("notes").get().await()
                val notes = snapshot.documents.map { document ->
                    Note(
                        id = document.getLong("id")?.toInt() ?: 0,
                        noteTitle = document.getString("title") ?: "",
                        noteDesc = document.getString("content") ?: ""
                    )
                }
                notes.forEach { note ->
                    noteRepository.insertNote(note)
                }
                Log.d("MyTag", "Notes fetched and updated in local database
successfully")
                Toast.makeText(getApplication(), "Notes downloaded from
Firestore successfully", Toast.LENGTH_SHORT).show()
            } catch (e: Exception) {
                Log.w("MyTag", "Error fetching notes from Firestore", e)
                Toast.makeText(getApplication(), "Error downloading notes from
Firestore", Toast.LENGTH_SHORT).show()
            }
        }
    }
}
```

Besides functions that previously implemented inside the DAO interface, we implement two more functions to work with the Firebase Cloud Storage. When the upload function is triggered, it will send the note object to the Firebase Cloud Storage, then in the Firebase Cloud Storage website we would see something like this:



```
content: "Testing for Monday"
id: 5
title: "Mon"
```

- **ViewModelFactory:** is used to instantiate the ViewModel.

```
class NoteViewModelFactory(val app: Application, private val noteRepository:
NoteRepository) : ViewModelProvider.Factory {

    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return NoteViewModel(app, noteRepository) as T
    }
}
```

Setting Up ViewModel in the Main Activity

The ViewModel is set up in the MainActivity and made available to all fragments in the app.

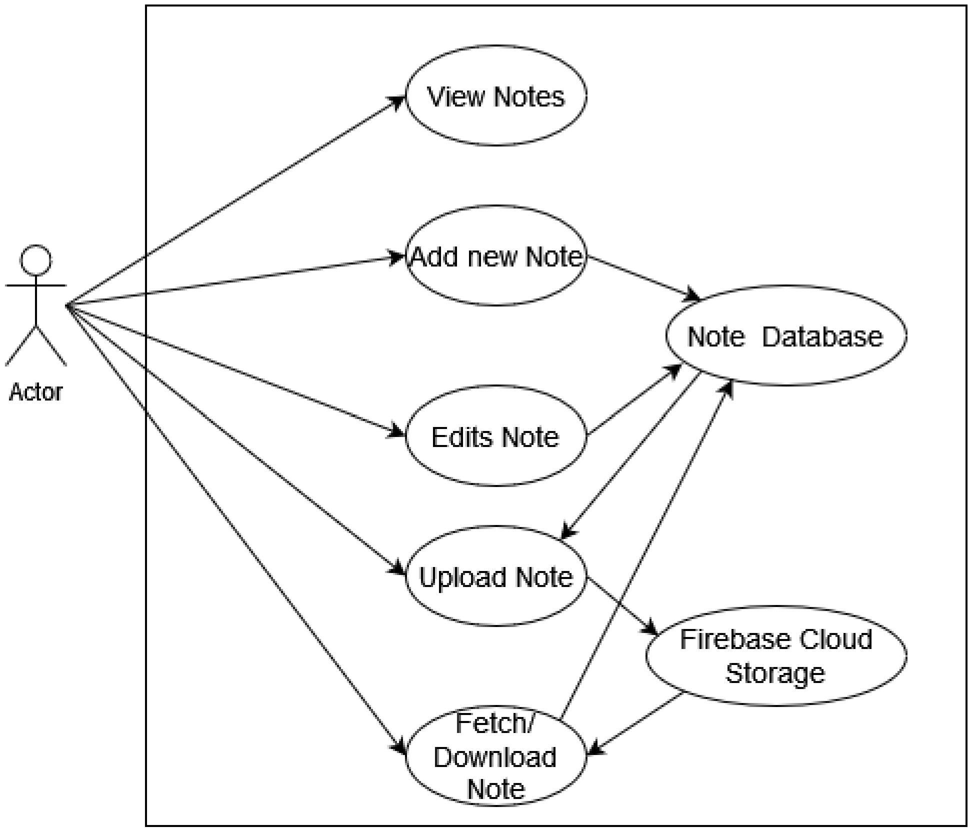
```
class MainActivity : AppCompatActivity() {
    lateinit var noteViewModel: NoteViewModel
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        setupViewModel()
    }

    private fun setupViewModel(){
        val noteRepository = NoteRepository(NoteDatabase(this))
        val viewModelProviderFactory = NoteViewModelFactory(application,
noteRepository)
        noteViewModel = ViewModelProvider(this,
viewModelProviderFactory)[NoteViewModel::class.java]
    }
}
```

The overall project structure is as follow:

```
SNote-IT4785E
    └── app/src/main
        ├── java/com/example/snote/
        │   ├── adapter/
        │   │   └── NoteAdapter.kt
        │   |── database/
        │   │   |── NoteDAO.kt
        │   │   └── NoteDatabase.kt
        │   ├── fragments/
        │   │   |── AddNoteFragment.kt
        │   │   |── EditNoteFragment.kt
        │   │   └── HomeFragment.kt
        │   ├── model/
        │   │   └── Note.kt
        │   ├── repository/
        │   │   └── NoteRepository.kt
        │   ├── viewmodel/
        │   │   |── NoteViewModel.kt
        │   │   └── NoteViewModelFactory.kt
        │   └── MainActivity.kt
        │
        └── res/
           ├── layout/
           │   |── fragment_home.xml
           │   |── fragment_edit_note.xml
           │   |── fragment_add_note.xml
           │   └── note_layout.xml
           └── menu/
               ├── home_menu.xm
               ├── menu_add_note.xm
               └── menu_edit_note.xml
```

IV. Use case and Demo

Here is the application use case diagram:



When user:

- Open the application, the home fragment is initiated, user will see all the notes, display using RecyclerView

- Click on the Add floating button, then they will be navigated to the fragment_add_note, where they can enter the title and content of the note then click the save button to store to database.

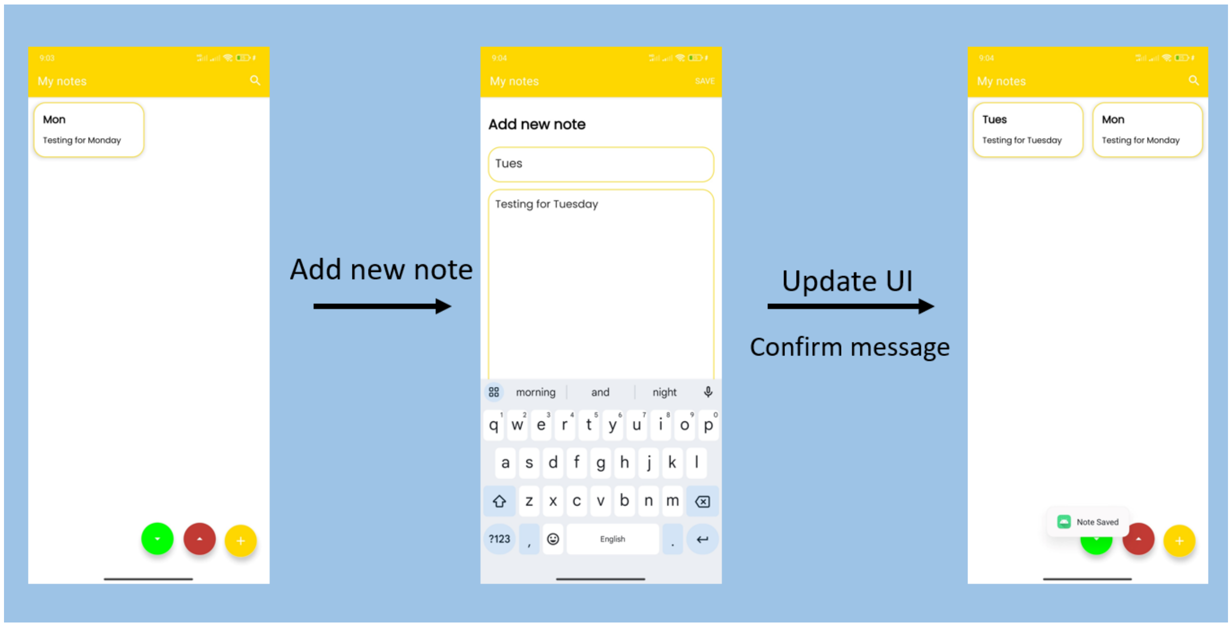
- Click on an existed note item on the home screen, navigate to the fragment_edit_note, where user can modify the title and content of that note, then they can choose to update the content of this note by click the tick floating button on the bottom right corner, or delete this note by clicking the trash symbol on the right top corner of screen.

- When user click on the upload button (red floating button with arrow up symbol), it will upload all the notes in the local database to the Firebase Cloud Storage and show a message confirming of successful or not.

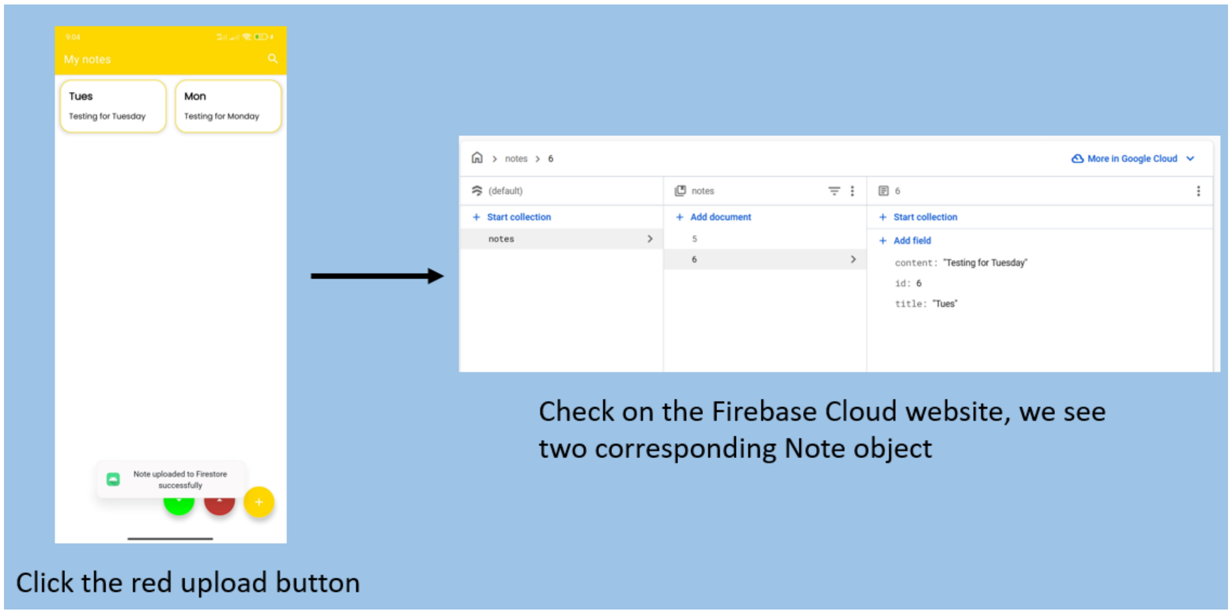
- When the user click on the download button (green floating button with arrow down symbol), it will fetch all the note data currently in the Firebase Cloud Storage to the local database then update the UI RecyclerView.

Here is the visualize for each of functionality:

- **Add Note button**



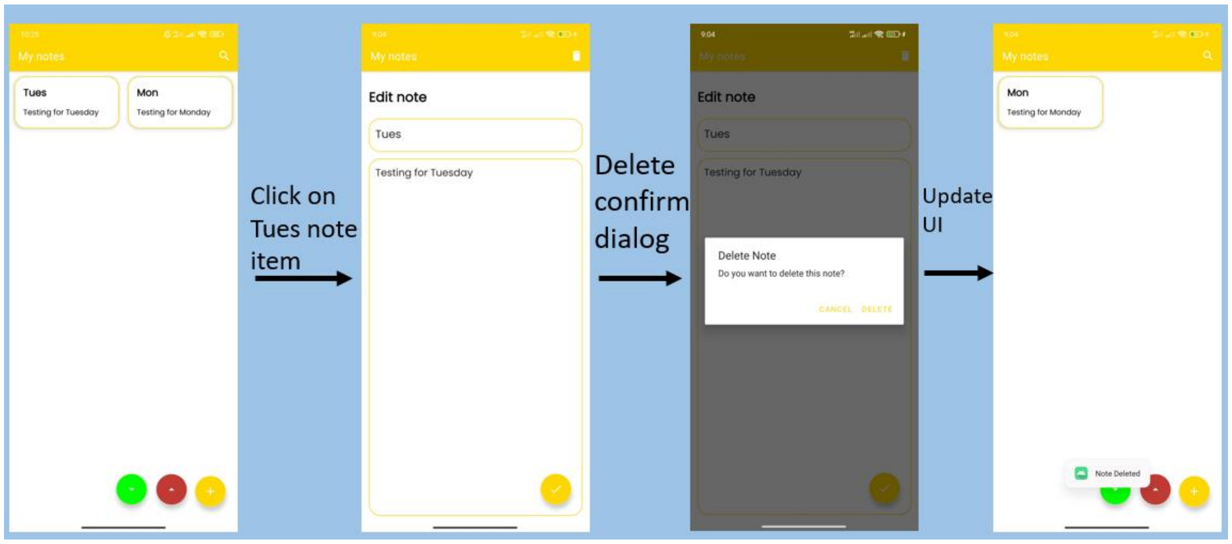
- **Upload Note button:**



Check on the Firebase Cloud website, we see two corresponding Note object

Click the red upload button

- Edit/ Delete Note

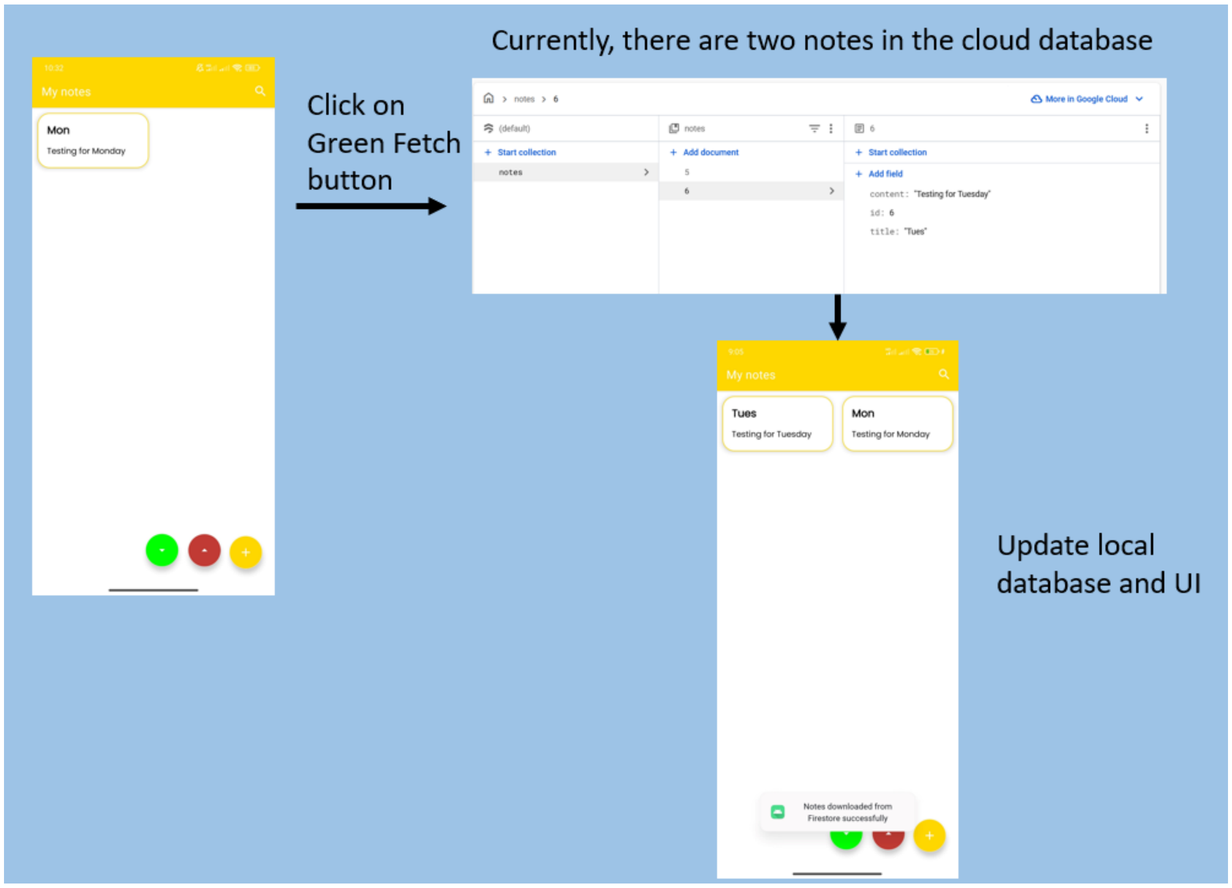


Click on Tues note item

Delete confirm dialog

Update UI

- Download/ Fetch function


Currently, there are two notes in the cloud database
Click on Green Fetch button
Update local database and UI
My notes
Mon
Testing for Monday
notes > 6
More in Google Cloud
(default)
notes
6
Start collection
Add document
Add field
content: "Testing for Tuesday"
id: 6
title: "Tues"
Tues
Testing for Tuesday
Notes downloaded from Firestore successfully

- **Demo uploading and fetching between two devices:**

Two devices must connect to the same Firebase cloud storage, meaning they must use the same Firebase configuration file. When you run the same app from the IDE on multiple devices, they will share the same `google-services.json` file.

google-services.json

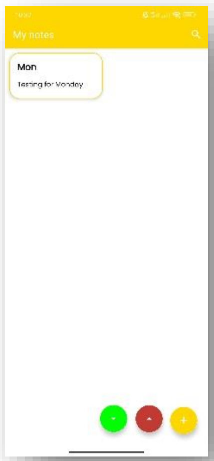
This file typically looks like this:

```
{
  "project_info": {
    "project_number": "510. . .",
    "project_id": "snote-ea7bf",
    "storage_bucket": "snote-ea7bf.firebasestorage.app"
  },
  "client": [
    {
      "client_info": {
        "mobilesdk_app_id": "1:51035080364:android:d. . . ",
        "android_client_info": {
          "package_name": "com.example.snote"
        }
      },
      "oauth_client": [],
      "api_key": [
        {
          "current_key": "AIz. . . . . "
        }
      ],
      "services": {
        "appinvite_service": {
          "other_platform_oauth_client": []
        }
      }
    }
  ],
  "configuration_version": "1"
}
```

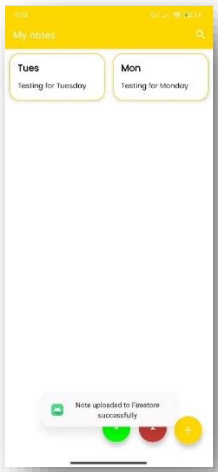
Now, we will test with two devices: a physical device (Redmi K60E) and a virtual device (Pixel 8A).

On the Redmi K60E:

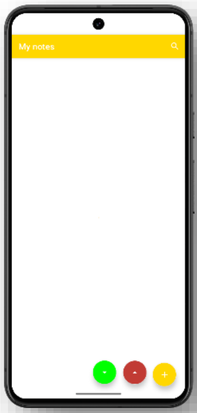
Initially, there is only one note:



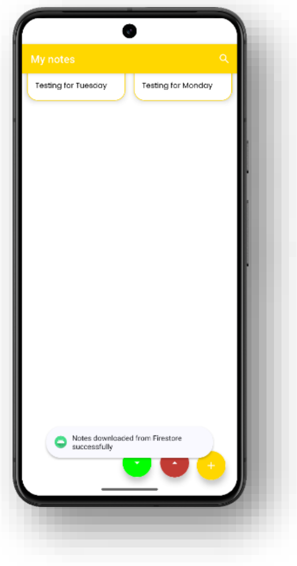
Then we add a new note and click Upload button. Now the Cloud will store two notes



On the Pixel 8A:
Initially, no notes are present.



And when we click the fetching button:

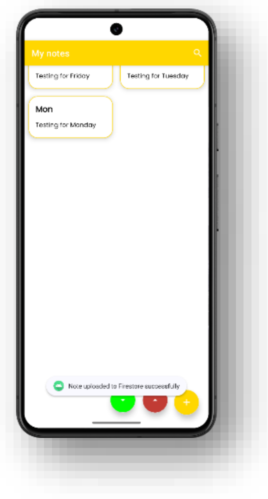


Notes downloaded from Firestore successfully

(Still error when try on different screen size)

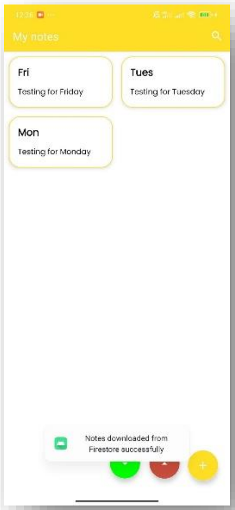
All notes from the K60E device are now displayed. Data is synchronized between two devices successfully

Now, let's test the other direction: On the virtual Pixel 8A, add a new note and save it to the Cloud.



Note uploaded to Firestore successfully

From the K60E side, fetch the new data:



Data from Pixel 8A syncs to k60E successfully

V. Conclusion

1. Summary

The hybrid note-taking application for Android successfully addresses the challenges associated with maintaining productivity in both online and offline environments. The app provides users with the ability to create, manage, and synchronize their notes seamlessly between a local SQLite database and a remote server. Key achievements include:

- A clean and intuitive user interface built using Kotlin and Jetpack Compose.
- Reliable offline functionality supported by SQLite for local storage.
- Effective data synchronization using Retrofit for server communication.
- Secure handling of user data with token-based authentication.

Through rigorous testing, the application demonstrates high usability and reliability, catering to users who require robust note-taking capabilities regardless of internet availability.

2. Future Work

To enhance the application further, the following features and updates are planned:

1. **Multimedia Notes**: Enable users to include images, audio recordings, and other multimedia content in their notes.
2. **Collaboration Tools**: Introduce features for real-time multi-user editing and sharing of notes.
3. **Cloud Backup Options**: Integrate additional cloud storage services such as Google Drive or Dropbox for backup and restore.
4. **Enhanced Security**: Implement end-to-end encryption to ensure greater data privacy and security.
5. **Cross-Platform Availability**: Extend the application to iOS and web platforms for broader accessibility.
6. **Improved Sync Mechanism**: Develop advanced conflict resolution algorithms to handle simultaneous edits across devices.

   These improvements aim to make the application more versatile, user-friendly, and competitive in the note-taking app market.