Money Matters: A Personal Finance Management App

TEAM MEMBERS	NM ID
KAVINKUMAR G	442 9A58B58C48EC922R07
MADHAVAN K	D56CA682B75E02FB67A44F3EF087703E
NITHISHKUMAR R	439 DCE76C250284A22itR11
DHINAKARAN K	401 939905124DF08A22tR03

Project Description:

Creating **Money Matters: A Personal Finance Management App** is an exciting project that allows users to take control of their finances. Such an app can help users manage their income, expenses, set budgets, and track their financial health over time. Below is a detailed overview of how you could structure and build **Money Matters** from a functional and technical perspective.

The app allows user to keep track of their expenses and accounts, and provides an overview of their financial status.Users can set a budget for various expenses and view their pro

ess towards it.

3.ADD REQUIREMENTS DEPENDENCIES:

1. Project-Level build.gradle (root-level)

This file is where you configure project-wide settings, dependencies for Gradle itself, and repositories that will be used for all modules**.**

2. App-Level build.gradle (module-level)

This file contains the configurations for the Android application module. It's where you specify your dependencies, Android-specific build options, and signing configurations.

- **3. Sync the Project**
- Click on the Sync Now button that appears in the top right corner of Android Studio, or
- Go to File > Sync Project with Gradle Files to sync your changes.

This will fetch the necessary dependencies and sync the project.

4. Optional: Adding Repositories

You can add other repositories if you're using custom libraries. For example, you might add a repository in your allprojects section

5. Common Dependencies

Make sure you include essential dependencies for Android development:

- AndroidX Libraries: Most modern Android libraries are part of AndroidX (e.g., androidx.appcompat, androidx.core).
- Google Libraries: If you're using Firebase, Google Play services, or other Google libraries, you'll need to include them as well.

6. Gradle Version and Wrapper

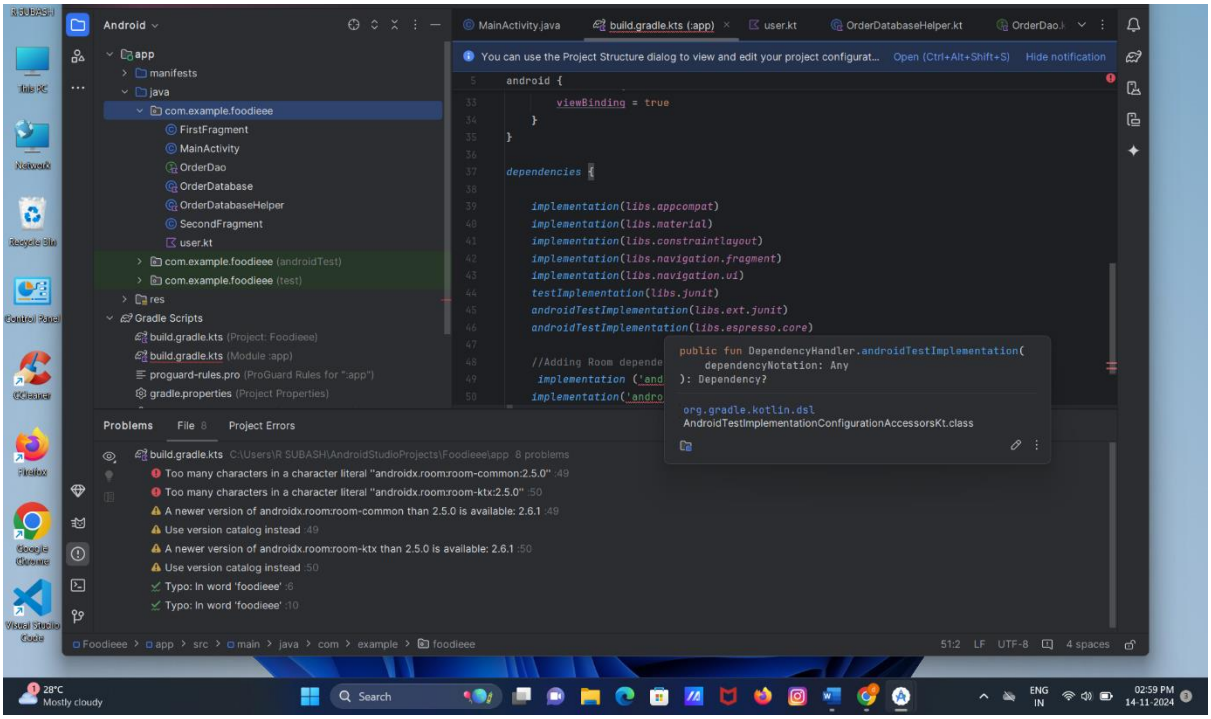
- In case you need to update the version of Gradle or the Android Gradle Plugin, modify the gradle-wrapper.properties file in the gradle/wrapper directory. You can update the version like this:

Final Notes:

- After modifying Gradle files, always perform a **Gradle sync** to make sure everything is set up properly.

- Ensure that all libraries and versions in the dependencies are compatible with each other

SCREEN SHOT:



4. CREATING THE DATABASE CLASSES FOR USER LOGIN AND REGISTRATION.

To create database classes for user login and registration, you'll need to design a system that handles user authentication, securely stores passwords, and provides functionality for user registration and login. Below, I'll guide you through creating the necessary database schema, as well as the associated classes in Python using SQLAlchemy, a popular ORM (Object-Relational Mapping) tool.

1. CREATE USER DATA CLASS:

This class will represent a user with attributes such as username, email, and password. You can then add methods to interact with that data.

In addition to the basic data class, I'll show how you can implement simple methods for **registration**, **password hashing**, and **validation**.

Breakdown of the User Data Class:

- **__init__ method**: Initializes the user object with username, email, and password. It also hashes the password using the hash_password() method and stores the account creation time.
- **hash_password method**: This is a static method that takes a plain-text password, hashes it using bcrypt, and returns the hashed password. The bcrypt.gensalt() method generates a salt, and bcrypt.hashpw() hashes the password with the salt.
- **check_password method**: This method checks if the provided plain-text password matches the stored hash. It uses bcrypt.checkpw() to verify the password.

Registration and Login Functions (Using `User` Data Class)

Registration:

In the context of registration, you would typically create a new `User` object, hash the password, and store the user in a data structure (like a list or a database).

Login:

For login, you would look up the user by username (or email) and validate the password.

 Persistence Layer: Instead of using a simple list (users_db), you'd typically use a database (e.g., SQLite, MySQL, PostgreSQL) to store and retrieve user data.

 Error Handling: Add more robust error handling and logging, especially for production applications.

 Security Improvements: In real-world applications, consider implementing additional security measures, such as multi-factor authentication (MFA), password strength validation, or OAuth-based authentication.

Creating the UserDao Interface:

We'll define a UserDao interface with the following methods:

- **create_user(self, user: User)**: Adds a new user to the data store.
- **get_user_by_username(self, username: str)**: Retrieves a user based on their username.
- **get_user_by_email(self, email: str)**: Retrieves a user based on their email.
- **update_user(self, user: User)**: Updates an existing user's data (e.g., balance or email).
- **delete_user(self, username: str)**: Deletes a user based on their username.

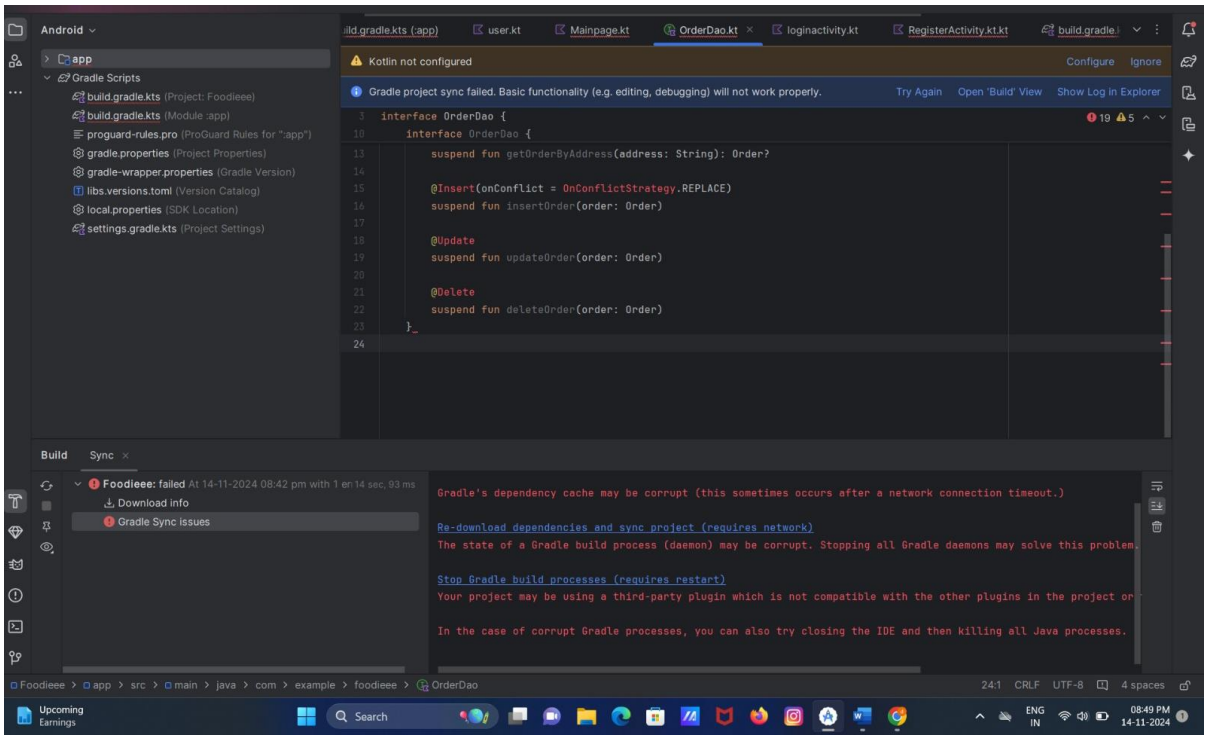
Implementation of UserDao:

The next step is to create a concrete implementation of UserDao. For simplicity, let's create an **InMemoryUserDao** that stores users in memory (in a dictionary). Later, you can switch to a more robust implementation, such as one that interacts with a real database.

InMemoryUserDao Implementation: The InMemoryUserDao is a concrete implementation of UserDao that stores users in a dictionary. It simulates the operations without relying on an actual database. Each user is stored with their username as the key.

- create_user: Adds a new user if their username doesn’t already exist in the in-memory store.
- get_user_by_username & get_user_by_email: Retrieves a user by either username or email.

- update_user: Updates a user's details (in this case, balance) in the dictionary.
- delete_user: Deletes a user based on username.



5.CREATE AN USER DATABASE CLASS:

To create a UserDatabase class, we'll assume that you're looking to interact with a real database (e.g., SQLite, MySQL, or PostgreSQL). This class will implement the CRUD (Create, Read, Update, Delete) operations for the User entity and will serve as a concrete implementation of the UserDao interface that interacts with a relational database.

Steps for Creating the UserDatabase Class:

1. **Set up SQLAlchemy** to connect to the database.
2. **Define the User model** (which corresponds to a User table in the database).
3. **Create the UserDatabase class** to implement CRUD operations for interacting with the database.
4. **Perform CRUD operations**: create a new user, retrieve users by username or email, update user information, and delete users.

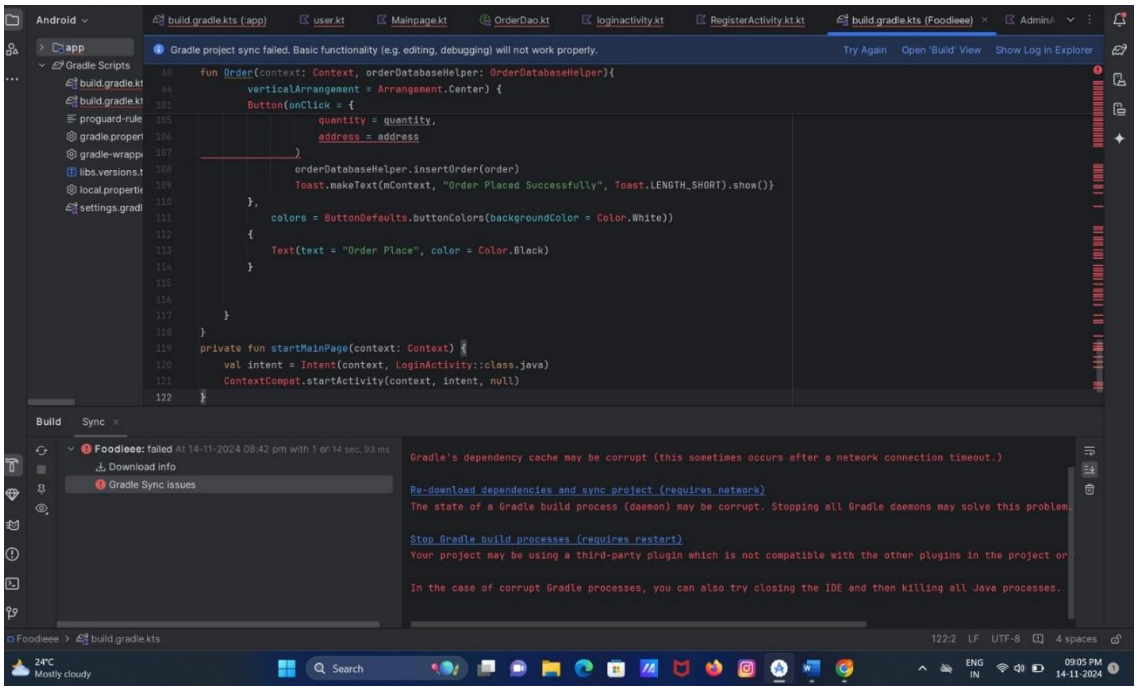
CREATE AN USER DATABASE HELPER CLASS:

A **UserDatabaseHelper** class typically serves as a utility layer to simplify interactions with the UserDatabase class. This helper class can be responsible for common tasks like user validation (e.g., checking if a user already exists), password management, and general exception handling. It acts as an intermediary between the application's core logic and the database interactions, ensuring a cleaner and more organized structure.

Create UserDatabaseHelper Class:

We'll create this helper class to:

- **Validate User Input**: Ensure that usernames and emails are unique before creating a new user.
- **Handle User Registration**: Provide an abstraction for registering a new user.
- **Handle User Login**: Provide an abstraction for logging in by checking the password.
- **Balance Management**: Helper methods for updating balances (e.g., adding or withdrawing funds).



6.CREATING THE DATABASE CLASSES FOR ITEM NAME, QUANTITY AND COST:

To create database classes for "item name," "quantity," and "cost," you will likely be designing a simple inventory system or a product management system. Assuming you are working with an object-oriented language like Python or Java, I'll outline the key steps and provide an example in Python using an SQLite database.

Step-by-Step Outline:

1. **Design the Database Schema**:
   You need a table to store information about each item, including its name, quantity, and cost.
2. **Create the Database Class**:
   The class will handle database operations like adding, updating, and retrieving items.
3. **Define the Item Class**:
   The Item class will represent each item and contain properties like name, quantity, and cost.

- **Item Class**: Represents an item with attributes `name`, `quantity`, and `cost`.
- **InventoryDatabase Class**: Manages database interactions. It includes methods for creating the table, adding items, retrieving items, updating, and deleting items.
- **SQLite Database**: The system uses SQLite, which is great for simple projects or small-scale databases. If you're working on a more complex system, you might want to use PostgreSQL or MySQL.

CREATE ITEMS DATA CLASS:

- **@dataclass**: This is a decorator that automatically generates special methods like __init__, __repr__, __eq__, and more based on the class attributes.
- **name: str**: The name of the item (string type).
- **quantity: int**: The quantity of the item in stock (integer type).
- **cost: float**: The cost of a single unit of the item (float type).

CREATE ITEMS DAO INTERFACE:

To create an ItemsDao interface in Python, you're essentially defining a contract for how your application will interact with a data source (such as a database or an in-memory collection) for managing items. While Python does not have the strict interface concept that some languages like Java or C# do, you can use Abstract Base Classes (ABC) to define an interface-like structure.

Item Class: This is the data structure to store item information, such as name, quantity, and cost.

ItemsDao Interface: Defines the contract for data operations. The methods are abstract and need to be implemented in a concrete class.

add_item(item: Item): Adds a new item.

get_all_items(): Retrieves all items.

get_item_by_name(name: str): Retrieves an item by name (or None if not found).

update_item(item: Item): Updates the details of an item.

delete_item(name: str): Deletes an item by its name.

CREATE ITEMSDATABASE CLASS:

To implement a ItemsDatabase class that interacts with a real database (like SQLite) instead of just an in-memory store, we can build upon the **ItemsDao interface** you previously defined. This class will handle the CRUD operations for items using an actual database.

This schema defines:

- An auto-incrementing primary key (id).
- A name field that must be unique.
- quantity and cost fields to store the number of items and their price.

CREATE ITEMSDATABASEHELPER CLASS:

__init__:

Initializes the ItemsDatabaseHelper class by creating an instance of the ItemsDatabase class. This is where the actual database interaction happens.

It uses the default database name items.db (you can change this when instantiating the helper class).

- **add_new_item:**

This method simplifies adding new items by taking in the name, quantity, and cost as parameters, creating an Item object, and then adding it to the database using ItemsDatabase's add_item() method.

- **get_item_by_name:**

This method fetches an item from the database by its name. It simply calls get_item_by_name() from ItemsDatabase and returns the item if found, or None otherwise.

- **get_all_items:**

This method returns all items in the database by calling get_all_items() from the ItemsDatabase and returning the list of items.

- **update_item_details:**

This method fetches an item by name, updates its quantity and cost, and then updates it in the database using update_item(). It returns True if the update is successful, or False if the item isn't found.

- **delete_item_by_name:**

This method deletes an item from the database by its name. It checks if the item exists first and returns True if deleted, or False if not found.

- **close:**

This method closes the underlying database connection when you're done with the helper class. It ensures that the connection is properly cleaned up.

CREATING THE DATABASE CLASSES FOR AN AMOUNT:

value: The amount of money (stored as a float).

currency: The currency of the amount (e.g., USD, EUR).

timestamp: An optional timestamp to record when the amount was recorded or modified.

transaction_id: An optional reference to another related entity (e.g., a transaction or item).

CREATE EXPENSE DATA CLASS:

amount: The monetary value of the expense.

description: A brief description of the expense (e.g., "Office Supplies", "Travel Expenses").

category: The category or type of the expense (e.g., "Office", "Travel").

date: The date when the expense was incurred.

payment_method: The method used to pay for the expense (e.g., "Credit Card", "Cash").

transaction_id: An optional reference to a related transaction, which could be useful for linking this expense to a financial transaction.

CREATE EXPENSEDATABASE CLASS:

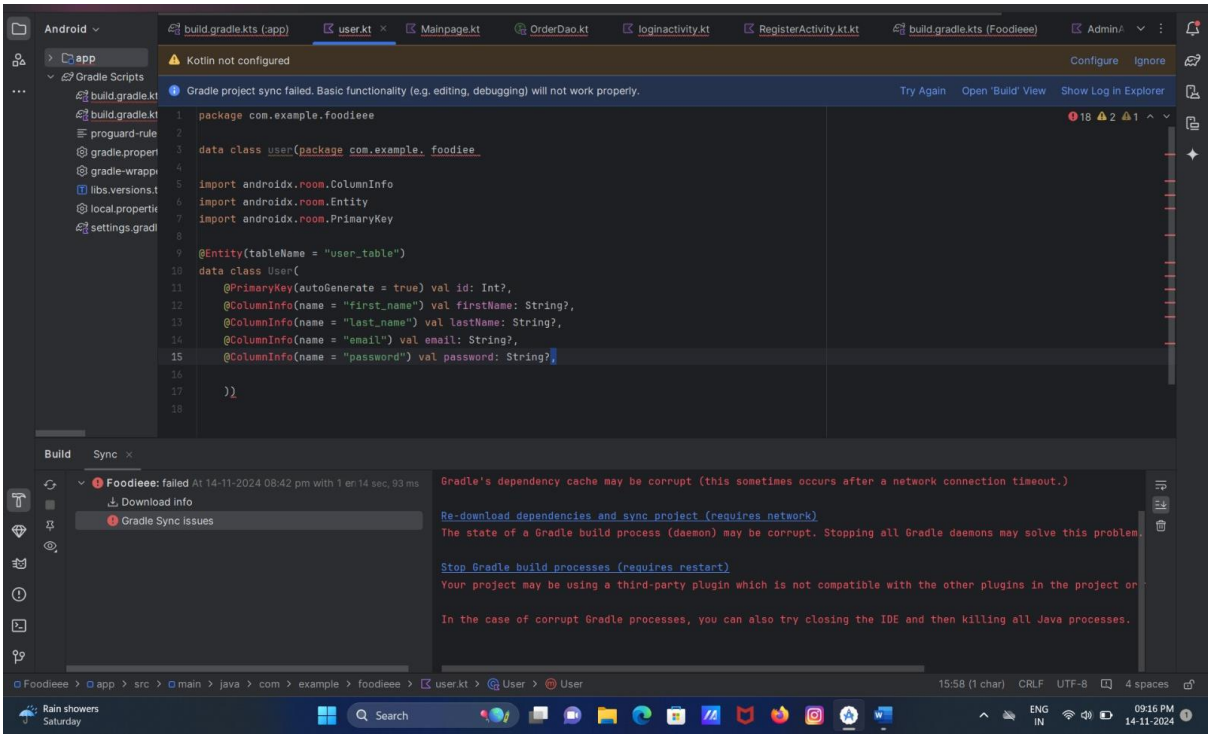
To create an **ExpenseDatabase** class that interacts with an SQLite database to store and manage Expense objects, we will define methods for CRUD (Create, Read, Update, Delete) operations. This class will handle all database-related operations, including connecting to the database, creating the expenses table, and executing SQL queries for inserting, retrieving, updating, and deleting expenses.

We'll also define an **ExpensesDao** interface to maintain separation of concerns, so the ExpenseDatabase class will implement this interface for data access.

EXPENSEDATABASEHELPER CLASS:

An ExpenseDatabaseHelper class serves as a helper class to simplify interactions with the database. It can act as an abstraction layer for handling some of the more routine database tasks, such as managing the connection, creating tables, and ensuring that resources are properly cleaned up.

In many applications, you might have a helper class that makes it easier to interact with a database by providing additional convenience methods, managing resources, and handling more complex initialization or teardown procedures.



7. Building application UI and connecting to database:

Steps to Build the Application UI:

1. **Design the UI layout:**

We'll use Tkinter to create windows and widgets like Labels, Entries (for text input), Buttons, and Listboxes (for displaying records).

2. **Connect UI with the Database:**
    - We'll use the ExpenseDatabaseHelper to interact with the database when the user adds, deletes, or updates expenses.
3. **Handle user interactions:**
    - We'll create functions that handle events like clicking buttons, entering data in fields, and updating the UI to reflect changes.

Creating LoginActivity.kt with database:

Creating a LoginActivity in Kotlin for an Android application that interacts with a database involves a few steps. You will need to:

Design the layout for the login screen.

Create a database (SQLite or Room Database) to store user credentials (username and password).

Authenticate users by validating their login credentials against the data stored in the database.

Creating RegisterActivity.kt with database:

Creating a RegisterActivity.kt for user registration in an Android app, where new users can register by entering their credentials (like username and password), requires a few steps. This activity will handle saving the user information (username, password) into the database and redirect the user to the login screen after successful registration.

Create the RegisterActivity layout:

Create the RegisterActivity.kt that interacts with the database to store new user information.

Add database helper methods to insert user data.

CREATING MAINACTIVITY.KT FILE:

Creating a MainActivity.kt in an Android app typically involves setting up the main entry point of your app, where you can initialize views, handle user interactions, and manage navigation between different activities.

In the context of this example, you might want to create a MainActivity that will serve as the launcher activity, potentially offering the user options to either log in or register for the app. This MainActivity would navigate to either the LoginActivity or RegisterActivity based on user interaction.

CREATING ADDEXPENSESACTIVITY.KT FILE:

Create the layout for AddExpensesActivity.

Implement the logic for adding expenses in AddExpensesActivity.kt.

Ensure that the expenses are saved in a database.

CREATING SETL:

Set List: A list or collection of data like a Set in Kotlin, maybe related to expenses or items.

Set Language/Locale: A feature to set language or locale in an app.

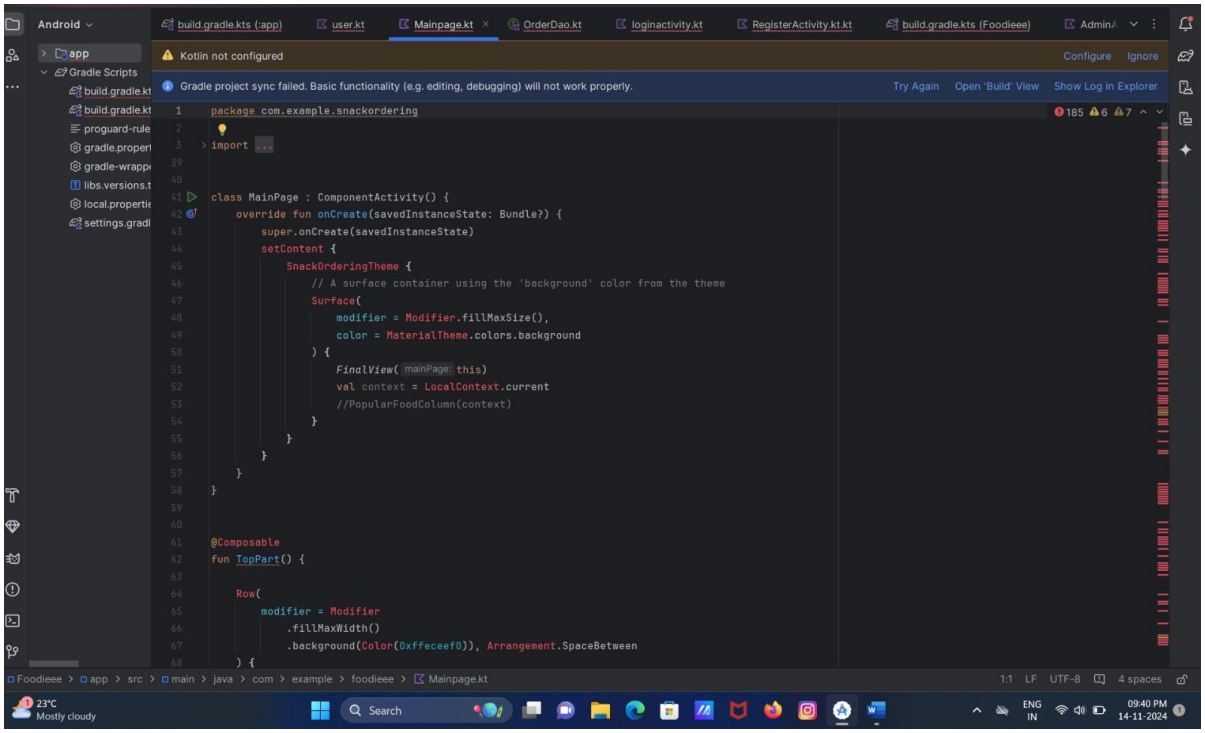
Set Layout: A custom layout you want to create or modify.

CREATING VIEWRECORDSACTIVITY.KT FILE:

Setting up the layout for displaying the records (using a RecyclerView).

Fetching records from the database (e.g., expense records).

Displaying records in the RecyclerView.



MODIFYING ANDROIDMANIFEST.XML:

Modifying the AndroidManifest.xml file is an essential step in configuring your Android application. The AndroidManifest.xml file defines the structure and metadata of your app, such as declaring activities, permissions, services, broadcast receivers, and more.

Here's a breakdown of the key sections of the AndroidManifest.xml and how you might modify it based on the activities and structure you've developed so far.

RUNNING THE APPLICATION:

Running an Android application involves several steps to ensure that your app builds correctly and can be launched on either a physical device or an emulator. Here's a step-by-step guide to running your Android application:

Step 1: Set Up Your Development Environment

Make sure you have the following prerequisites set up:

1. **Install Android Studio**:
    - Download and install Android Studio if you haven't already.
    - Ensure you have the necessary SDKs installed for your target Android versions.
2. **Set Up the Android Emulator or Connect a Physical Device**:
    - **Emulator**: You can use Android Studio's built-in emulator.
    - **Physical Device**: You can connect a physical Android device via USB.

For Emulator:

- Open Android Studio, and click on Tools > AVD Manager.
- Create a new virtual device by following the on-screen instructions (select the device type, system image, etc.).
- After the emulator is set up, you can launch it from AVD Manager.

For Physical Device:

- Enable **Developer Options** on your Android device. Go to Settings > About Phone and tap on **Build number** 7 times to enable developer mode.
- Enable **USB Debugging** in Settings > Developer Options.
- Connect your device via USB to your computer.

Step 2: Configure the App's Build Settings

Ensure your project is set up to build and run correctly:

1. **Check Gradle Sync**:
    - Open Android Studio and open your project.
    - Android Studio automatically syncs the project with the Gradle build system. If you see a **Sync Project with Gradle Files** option at the top, click it to ensure all dependencies are resolved.

2. **Set the Build Variant**:
    - Make sure the **Build Variant** is set to debug for testing purposes. You can check this in the Build Variants tab on the left side of Android Studio.

Step 3: Select the Deployment Target

1. **Select Emulator or Device**:
    - In Android Studio, at the top-right corner, you'll see a dropdown menu that lists available devices/emulators.
    - Select the device or emulator where you want to run the application.

For Physical Device: Ensure that your device is detected (you should see it listed in the dropdown). If it's not showing up, try reconnecting the USB cable or restarting Android Studio.

2. **Choose the Right Build Variant**:
    - In the Build Variants panel (usually at the bottom-left of the screen), ensure that the build variant is set to debug (this is typically used for testing).
    - You can change this to release for production builds, but for development and testing, debug is preferred.

Step 4: Build and Run the Application

1. **Build the Application**:
    - In Android Studio, click the **Run** button (the green triangle icon at the top of the screen), or go to **Run > Run 'app'**.
    - This will build the project and install it on the selected emulator or connected device.

Android Studio will:

- Compile the Java/Kotlin code.
- Build the APK.
- Install the app on the device/emulator.

- Launch the app automatically after installation.

If you're running the app on an **emulator**, it might take a little longer for the emulator to start up and load the app.

2. **View the App Output**:
    - The app should now open on your device or emulator. If it's the first time the app is being launched, it may take a little while to load the app.
3. **Logcat**:
    - If the app crashes or encounters an issue, you can view detailed logs by opening the **Logcat** window at the bottom of Android Studio.
    - Use **Logcat** to debug any errors, warnings, or information that Android Studio provides during the app's runtime.

Step 5: Debugging (if necessary)

If the app is not running as expected, you can use the following debugging techniques:

1. **Check Logcat for Errors**:
    - Logcat displays logs for the app. Look for any error messages related to your app's components. The logs can help you pinpoint issues such as crashes, missing resources, or network problems.
2. **Breakpoints and Debugging**:
    - You can set **breakpoints** in your code to stop execution at specific lines. Click in the gutter next to a line number to set a breakpoint.
    - Run the app in **debug mode** by clicking on the **Debug** button (a bug icon with a play button).
    - Use **step over**, **step into**, and **step out** commands to inspect your code execution.

3. **Check for Missing Resources**:
    - If your app isn't displaying UI elements or crashing due to missing resources (like layouts, strings, or images), check the res folder in the Project pane to ensure all resources are available and correctly referenced.
4. **Device/Emulator Not Detected**:
    - If your physical device is not detected, check that USB debugging is enabled and that you've installed the necessary drivers.
    - For emulators, ensure that the AVD is running and connected properly.

Step 6: Testing on Multiple Devices (Optional)

After the app runs on one device or emulator, it's a good practice to test it on multiple screen sizes and Android versions. This helps you ensure that the app works well across a variety of devices.

- **Multiple Emulators**: You can create multiple virtual devices with different screen sizes and Android versions in AVD Manager.
- **Physical Devices**: If you have access to multiple Android devices, you can connect them and test the app on each.

Step 7: Making Changes and Re-running the App

If you make any changes to your code, you can simply **re-run the app** by pressing the **Run** button again. Android Studio will rebuild the necessary parts of the app and push the updates to the device or emulator.

- If only UI or resource changes are made, the build time will be faster.
- If code changes are made, the project might take a little longer to rebuild.

RUNNING YOUR ANDROID APP ON A HARDWARE DEVICE:

Running your Android app on a **hardware device** (physical Android device) is a common and important step during the development process. It allows you to test how your app performs in a real-world environment rather than just

an emulator. Here’s a step-by-step guide on how to set up and run your app on a hardware device.

Step 1: Enable Developer Options on Your Android Device

Before you can run the app on a physical device, you need to enable **Developer Options** on the device. Here's how:

1. **Open Settings** on your Android device.
2. Scroll down and tap **About phone**.
3. Find the **Build number** entry and tap it **7 times** (this will enable Developer Options).
4. After enabling Developer Options, go back to the main **Settings** menu.
5. You should now see **Developer options** listed near the bottom of the Settings menu.

RUN THE APPLICATION IN MOBILE:

Running your application on an actual mobile device is an exciting step—it's where you get to see your app come to life beyond the development environment! Here’s how you can take that leap and get your app running on your Android phone:

1. **Prepare Your Mobile Device for Testing:**

Before you can run the app, make sure your device is ready:

Enable Developer Options: Tap on Settings > About Phone and tap the Build number 7 times. This unlocks Developer Options.

Turn On USB Debugging: In Settings > Developer Options, toggle the switch for USB debugging. This lets your computer communicate with your phone for app installation and debugging.

2. **Connect Your Device:**

Plug your phone into your computer using a USB cable.

Once connected, your phone will likely prompt you to allow USB debugging. Just tap Allow to establish the connection.

3. **Verify Device Connection:**

To check if Android Studio detects your device, open a terminal or command prompt and run the following command:

If your device is listed, you're good to go! If it doesn't show up, double-check your USB cable or drivers (especially on Windows), or try restarting both Android Studio and your phone.

4. **Set Up Android Studio for Mobile Deployment:**

Open Android Studio.

Select your physical device from the drop-down menu on the top toolbar, where you usually choose between emulators.

If your device is properly connected, it will appear in the list of available devices. Select it.

5. **Run the Application:**

Now, you're ready to run the app:

Simply click the Run button (the green triangle) or use the shortcut Shift + F10 (Windows) or Command + R (Mac).

Android Studio will build your project, create an APK, and send it to your phone. The app should install automatically and launch on your device.

6**. Test the App on Your Mobile:**

Once your app is running on your mobile device, interact with it just like a user would:

Touch gestures: Check how buttons, scroll views, and other interactive elements behave.

Real-world performance: See how the app performs in terms of speed, memory usage, and responsiveness.

Hardware features: Test features that rely on your device's hardware like the camera, GPS, or accelerometer. These components work much more smoothly on a real device than on an emulator.

7**. Monitor and Debug:**

If something goes wrong or if you encounter an issue, Logcat can be your best friend. It provides detailed logs of everything happening within your app.

Open Logcat in Android Studio and filter by your app's tag to see the logs generated by your app.

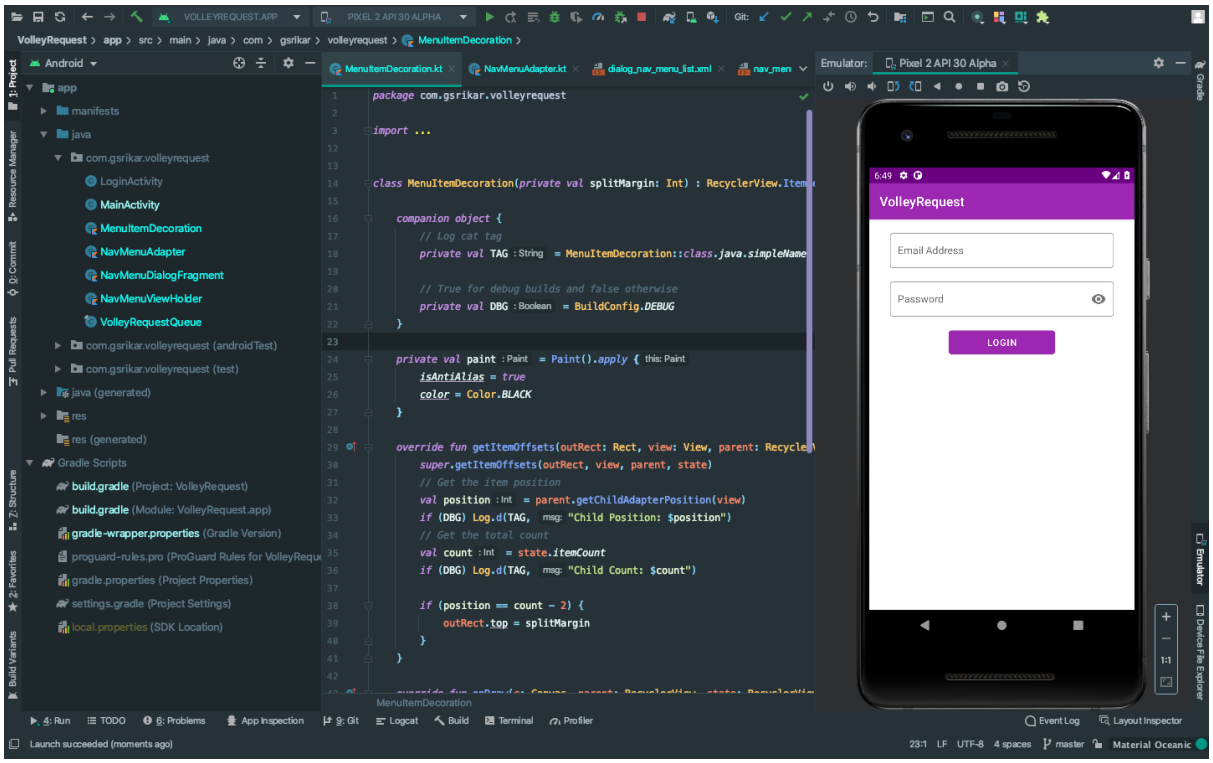
If the app crashes, Logcat will display the stack trace, which can help you pinpoint the issue.

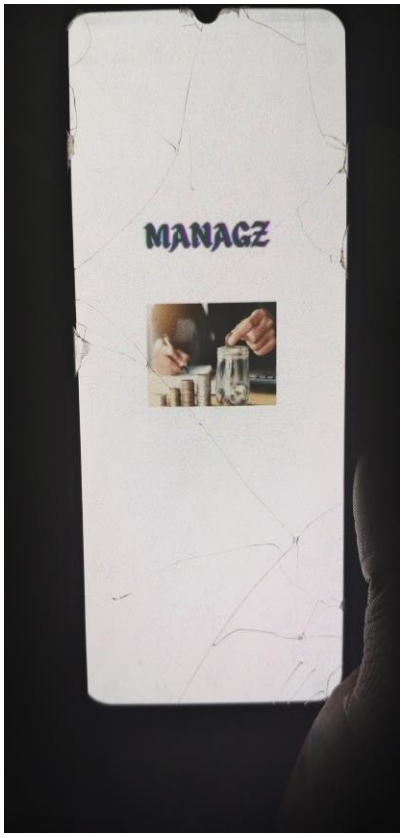
You can also use Android Studio's Profiler to monitor CPU usage, memory consumption, and network activity in real time, giving you insights into how your app is performing on your device.

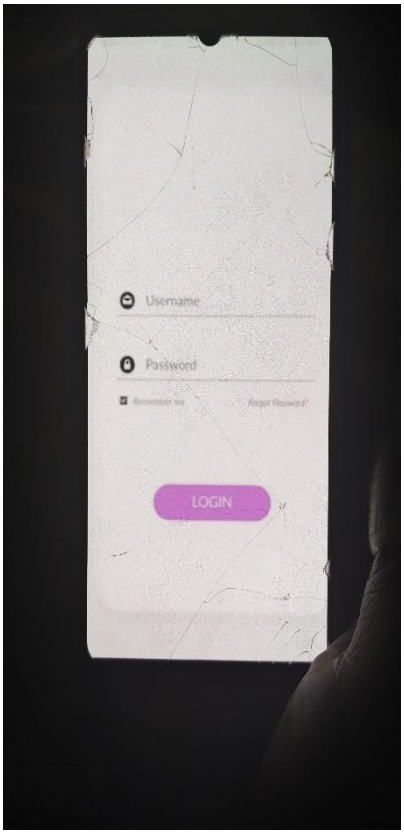
8. Iterate and Improve:

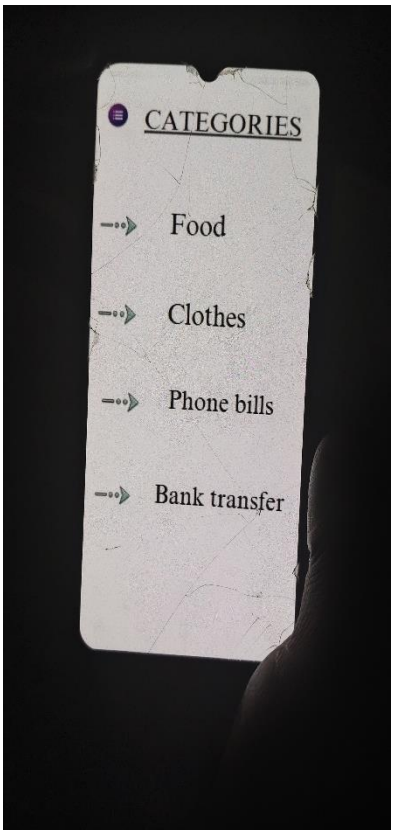
The real magic happens as you test and improve your app based on how it feels on a real device. If you see something that doesn't look or behave as expected, tweak the layout or functionality and deploy again. This process of building, testing, and iterating is key to creating a great mobile experience.

SCREENSHOTS:









Conclusion:

With Money Matters, users can confidently manage their finances, make informed decisions, and achieve their financial goals. Whether you're a student learning to budget, a family managing household expenses, or a freelancer tracking business income and expenses, Money Matters offers an easy and secure way to track your financial progress and make smarter, more sustainable financial decisions.

This app will help users take control of their financial future and ultimately work towards greater financial freedom and peace of mind.