WalletWise Team

WalletWise
Software Architecture Document

Version 1.1

Revision History

Date	Version	Description	Author
16/May/24	1.0	First version of the Software Architecture Document	WalletWise Team
25/May/24	1.1	Add content for sections 5 and 6, and edit subsections 4.4 and 4.5	WalletWise Team

Table of Contents

Software Architecture Document

1. Introduction

The WalletWise application will follow the MVVM architectural pattern, which promotes separation of concerns and testability. The View layer will be responsible for the UI, the ViewModel layer for the presentation logic, and the Model layer for the business logic and data access. The application will utilize Jetpack Compose for building the UI, Room for local data persistence, Retrofit for API interactions, and Hilt for dependency injection.

2. Architectural Goals and Constraints

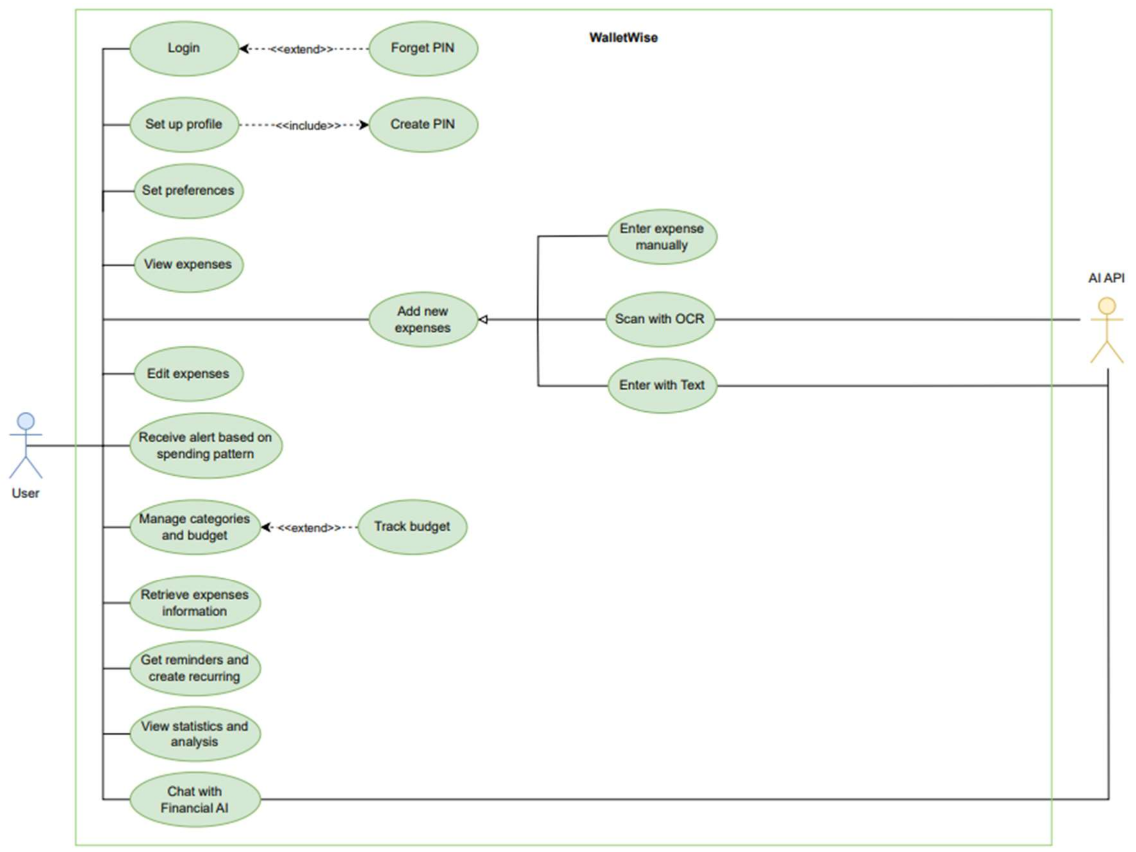
- Applicable Standards, Hardware, or Platform Requirements:
  - The application must be compatible with Android operating systems version 9.0 and above.
  - It should support various screen sizes and resolutions to ensure usability across different Android devices.
- Performance Requirements:
  - The application should have fast response times for navigating between different screens and performing actions such as adding expenses or generating reports.
  - Loading times for retrieving financial data should be minimal, ensuring a smooth user experience.
  - The application should consume minimal system resources (CPU, memory, battery) to prevent draining the device's battery or causing performance issues.
- Environmental Requirements:
  - The application should be resilient to fluctuations in network connectivity, ensuring functionality even in areas with poor or intermittent internet access.
- Quality Ranges for Performance, Robustness, Fault Tolerance, and Usability:
  - Performance: Response time for common tasks should typically be under 2 seconds.
  - Robustness: The application should handle errors gracefully, providing informative error messages and maintaining data integrity in case of unexpected events.
  - Fault Tolerance: The app should have mechanisms in place to recover from failures, such as data backup and recovery options.
  - Usability: The user interface should be intuitive and easy to navigate, with clear instructions and minimal cognitive load for users.
- Design Constraints, External Constraints, or Dependencies:
  - The application may rely on third-party APIs or services for features such as currency conversion or data synchronization, AI model.
  - Compliance with data privacy regulations, ensuring that user financial data is securely stored and protected.
- Documentation Requirements:
  - User manuals should be provided within the application, offering guidance on how to use its features effectively.
  - Online help resources should be available, including FAQs or tutorials accessible from within the app.
  - Installation instructions should be provided to guide users through the setup process.
- Priority of Other Product Requirements:
  - Stability: High priority to ensure the application functions reliably without frequent crashes or errors.
  - Benefit: High priority to deliver value to users by effectively managing their personal expenses.
  - Effort: Medium priority to ensure that development efforts are feasible within reasonable timeframes and resources.
  - Risk: Low priority, with measures in place to mitigate potential risks such as data breaches or technical issues.
- **Spend pattern prediction model**:
  - Performance: The model training and forecasting should be efficient using optimized algorithms and hardware to avoid long wait times for users.
  - Easy to Use: WalletWise should automate data collection and model training. Users should be able

to access forecasts and insights easily through visualizations and clear explanations.

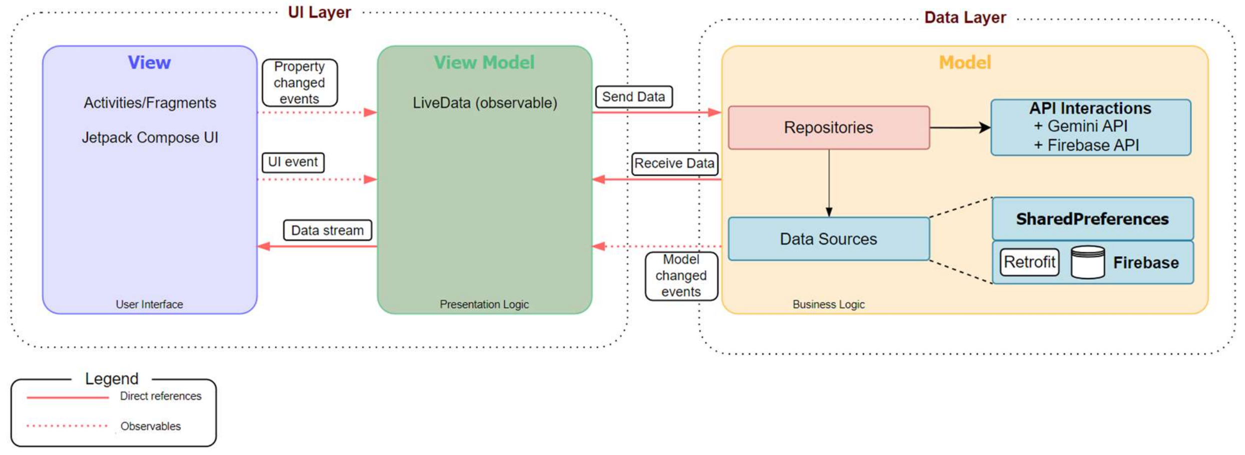
- Accuracy: The system should strive for the highest possible accuracy.

- **Chatbot using AI API:**
  - Performance: The chatbot should respond to user queries quickly and efficiently.
  - Easy to Use: The chatbot interface should be user-friendly and intuitive, allowing users to ask questions in natural language.
  - Accuracy: The chatbot should provide accurate and up-to-date financial information and recommendations using Gemini's knowledge and reasoning capabilities

3. Use-Case Model

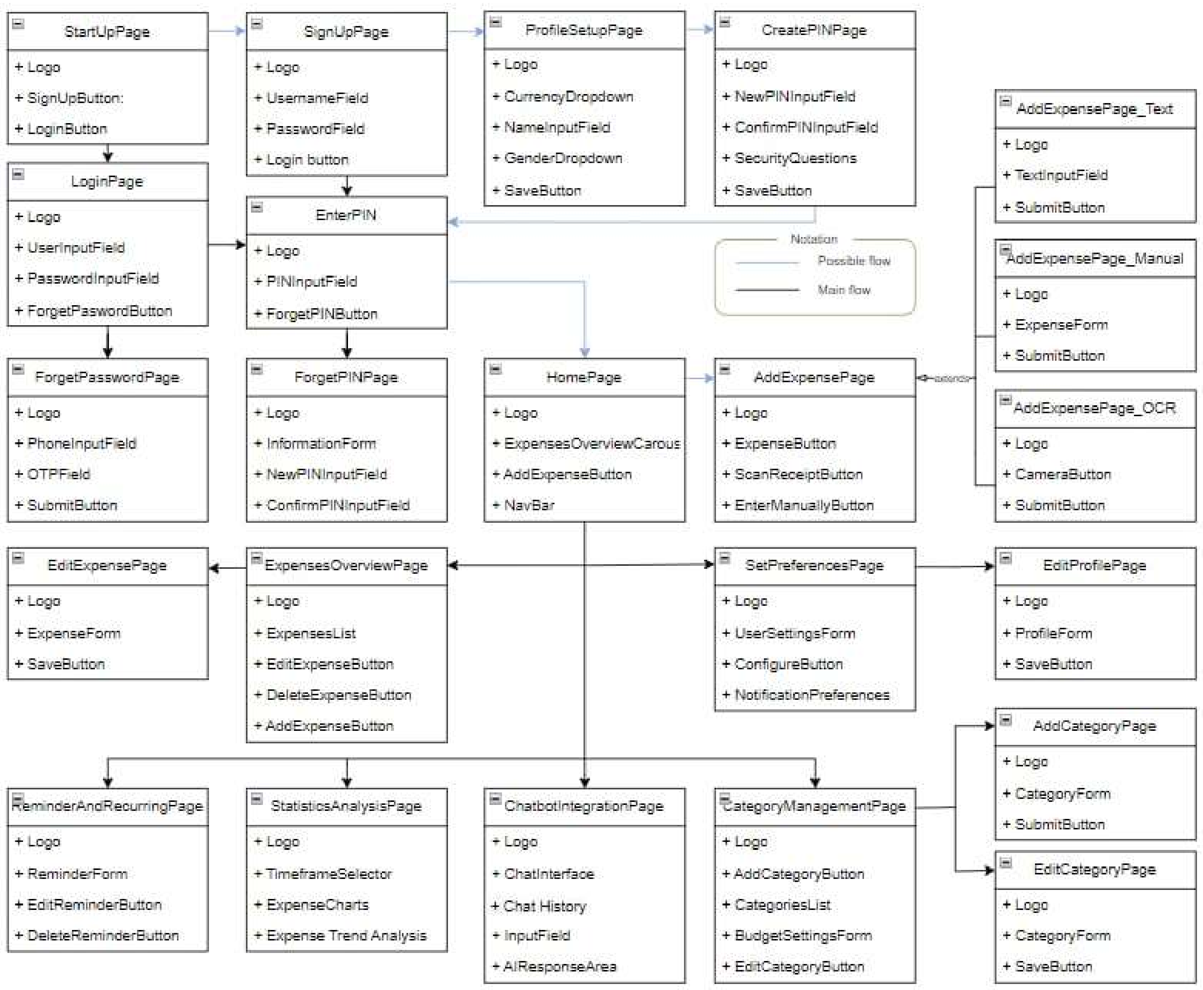


4. Logical View



4.1 Component: View

4.1.1 Class diagram



4.1.2 Description

- **HomePage:** Is the most crucial part in the Views component, it linked to EnterPin page and the following sub-components:
  - Logo: represents the application's image or emblem, displayed at the top of each page for.
  - ExpensesOverviewCarousel: Carousel displaying an overview of expenses.
  - AddExpenseButton: Button for adding new expenses.
  - NavBar: Navigation bar for accessing different sections of the application.
- **StartUpPage:** contains the given sub-components:
  - Logo: Application logo.
  - SignUpButton: Button to navigate to the signup page.
  - LoginButton: Button to navigate to the login page.
- **SignUpPage:** Is linked to StartUpPage and contains the given sub-components:
  - Logo: Application logo.
  - UsernameField: Field for entering the username during signup.
  - PasswordField: Field for entering the password during signup.
  - Login button: Button to submit the signup form and create an account.
- **ProfileSetupPage**: Is linked to SignUpPage and contains the given sub-components:
  - Logo: represents the application's image or emblem, displayed at the top of each page for.
  - CurrencyDropdown: Dropdown for selecting currency.
  - NameInputField: Input field for entering the user's name.
  - GenderDropdown: Dropdown for selecting gender.
  - SaveButton: Button to save profile settings.
- **CreatePINPage:** Is linked to ProfieSetupPage and contains the given sub-components:
  - Logo: represents the application's image or emblem.
  - NewPINInputField: Input field for creating a new PIN.
  - ConfirmPINInputField: Input field for confirming the new PIN.
  - SecurityQuestions: Possibly a section for setting security questions.
  - SaveButton: Button to save the new PIN and security questions.
- **LoginPage:** linked to StartUpPage and contains the following sub-components:
  - Logo: represents the application's image or emblem.
  - UserInputField: Input field for entering the user.
  - PasswordInputField: Input field for entering the password.
  - ForgetPasswordButton: Button to navigate to the ForgetPasswordPage.
- **ForgetPasswordPage:** Is linked to LoginPage and contains the given sub-components:
  - Logo: represents the application's image or emblem.

    - PhoneInputField: Input field for entering the phone of account.
    - OTPField: Input field for confirming the OTP.
    - SubmitButton: Button to submit the OTP.
- **EnterPin:** Is linked to LoginPage and CreatePINPage, contains the given sub-components:
    - Logo: represents the application's image or emblem.
    - PINInputField: Input field for entering the PIN.
    - ForgetPinbutton: Button to navigate to the ForgetPINPage.
- **ForgetPINPage:** Is linked to EnterPin page and contains the given sub-components:
    - Logo: represents the application's image or emblem.
    - InformationForm: A form to collect necessary information for PIN recovery.
    - NewPINInputField: Input field for entering a new PIN.
    - ConfirmPINInputField: Input field for confirming the new PIN.
- **SetPreferencesPage:** Is linked to HomePage and contains the following sub-components:
    - Logo: Application logo.
    - UserSettingsForm: Form for configuring user settings.
    - ConfigureButton: Button to save and configure preferences.
    - NotificationPreferences: Section for setting notification preferences.
- **EditProfilePage:** Is linked to SetPreferencesPage and contains the following sub-components:
    - Logo: Application logo.
    - ProfileForm: Form for editing user profile details.
    - ExpenseForm + SaveButton: Form and button to save changes to the user profile.
- **ExpensesOverviewPage:** Is linked to HomePage and contains the following sub-components:
    - Logo Application logo.
    - ExpensesList: List displaying the user's expenses.
    - EditExpenseButton: Button for editing expenses.
    - DeleteExpenseButton: Button for deleting expenses.
    - AddExpenseButton: Button for adding new expenses.
- **EditExpensePage:** Is linked to ExpensesOverviewPage and contains the following sub-components:
    - Logo: Application logo.
    - ExpenseForm: Form for editing expense details.
    - SaveButton: Button to save the changes made to the expense form.
- **StatisticsAnalysisPage:** Is linked to HomePage and contains the following sub-components:
    - Logo: Application logo.
    - TimeframeSelector: Selector for choosing the timeframe for expense analysis.
    - ExpenseCharts: Charts displaying expense statistics.

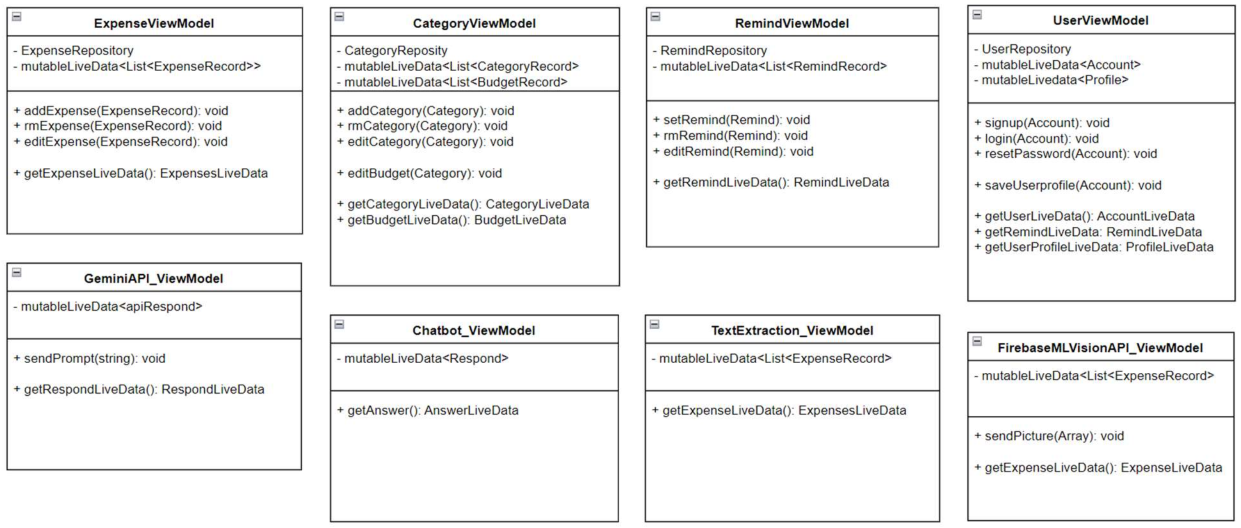
  - Expense Trend Analysis: Analysis of expense trends.
  - Cost-Saving Recommendations: Recommendations for cost-saving.
- **AddExpensePage:** Is linked to HomePage and contains the following sub-components:
  - Logo: Application logo.
  - ExpenseButton: Form for adding a new expense.
  - ScanReceiptButton: Button for scanning receipts.
  - EnterManuallyButton: Button for manually entering expense details.
- **AddExpensePage_Text:** Is linked to AddExpensePage and contains the following sub-components:
  - Logo: Application logo.
  - TextInputField: Field for entering text details of the expense.
  - SubmitButton: Button to submit the text-based expense details.
- **AddExpensePage_Manual:** Is linked to AddExpensePage and contains the following sub-components:
  - Logo: Application logo.
  - ExpenseForm: Form for adding a new expense manually.
  - SubmitButton: Button to submit the expense manually.
- **AddExpensePage_OCR:** Is linked to AddExpensePage and contains the following sub-components:
  - Logo: Application logo.
  - CameraButton: Button for scanning receipts.
  - SubmitButton: Button to submit the expense after OCR scanning.
- **CategoryManagementPage**: Is linked to HomePage and contains the following sub-components:
  - Logo: represents the application's image or emblem, displayed at the top of each page for.
  - AddCategoryButton: Button for adding new expense categories.
  - CategoriesList: List of expense categories.
  - BudgetSettingsForm: Form for setting budget limits.
  - EditCategoryButton: Button for editing expense categories.
- **AddCategoryPage**: Is linked to CategoryManagementPage and contains the following sub-components:
  - Logo: Application logo.
  - ExpenseForm: Form for editing expense details.
  - SubmitButton: Button to submit and save the new category.
- **EditCategoryPage**: Is linked to CategoryManagementPage and contains the following sub-components:
  - Logo: represents the application's image or emblem, displayed at the top of each page for.
  - CategoryForm: Form for editing an existing expense category.
  - SaveButton: Button to save the changes made to the category.
- **ReminderAndRecurringPage:** Is linked to HomePage and contains the following sub-components:
  - Logo: represents the application's image or emblem, displayed at the top of each page for.

- ReminderForm: Form for setting reminders.
- EditReminderButton: Button for editing reminders.
- DeleteReminderButton: Button for deleting reminders.

- **ChatbotIntegrationPage:**
  - Logo: represents the application's image or emblem, displayed at the top of each page for.
  - ChatInterface: Interface for interacting with the chatbot.
  - Chat History: History of chatbot conversations.
  - InputField: Field for entering queries.
  - AIResponseArea: Area displaying responses from the chatbot.

4.2 Component: View model

4.2.1 Class diagram


ExpenseViewModel
- ExpenseRepository
- mutableLiveData<List<ExpenseRecord>>
+ addExpense(ExpenseRecord): void
+ rmExpense(ExpenseRecord): void
+ editExpense(ExpenseRecord): void
+ getExpenseLiveData(): ExpensesLiveData
CategoryViewModel
- CategoryReposity
- mutableLiveData<List<CategoryRecord>
- mutableLiveData<List<BudgetRecord>
+ addCategory(Category): void
+ rmCategory(Category): void
+ editCategory(Category): void
+ editBudget(Category): void
+ getCategoryLiveData(): CategoryLiveData
+ getBudgetLiveData(): BudgetLiveData
RemindViewModel
- RemindRepository
- mutableLiveData<List<RemindRecord>
+ setRemind(Remind): void
+ rmRemind(Remind): void
+ editRemind(Remind): void
+ getRemindLiveData(): RemindLiveData
UserViewModel
- UserRepository
- mutableLiveData<Account>
- mutableLivedata<Profile>
+ signup(Account): void
+ login(Account): void
+ resetPassword(Account): void
+ saveUserprofile(Account): void
+ getUserLiveData(): AccountLiveData
+ getRemindLiveData: RemindLiveData
+ getUserProfileLiveData: ProfileLiveData
GeminiAPI_ViewModel
- mutableLiveData<apiRespond>
+ sendPrompt(string): void
+ getRespondLiveData(): RespondLiveData
Chatbot_ViewModel
- mutableLiveData<Respond>
+ getAnswer(): AnswerLiveData
TextExtraction_ViewModel
- mutableLiveData<List<ExpenseRecord>
+ getExpenseLiveData(): ExpensesLiveData
FirebaseMLVisionAPI_ViewModel
- mutableLiveData<List<ExpenseRecord>
+ sendPicture(Array): void
+ getExpenseLiveData(): ExpenseLiveData

4.2.2 Description

The ViewModel layer serves as the bridge between the View and Model components in the WalletWise application. It is responsible for preparing and managing the data displayed in the user interface (UI), handling user interactions, and orchestrating the necessary business logic. This layer ensures a separation of concerns, making the codebase more maintainable and testable.

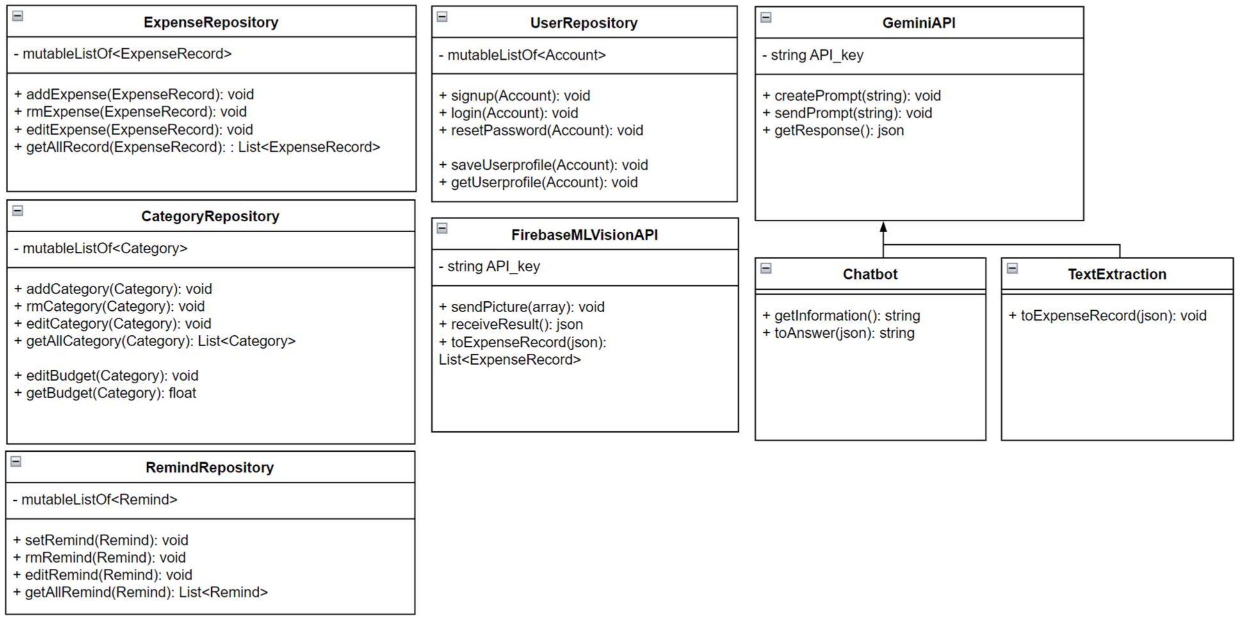
- **ViewModels**: Each ViewModel corresponds to a specific screen or feature in the app and holds the UI state for that screen.
  - ExpenseViewModel: Manages the data and operations related to expense records, such as adding, removing, editing, and displaying expenses.
  - CategoryViewModel: Handles the data and operations related to expense categories, including adding, removing, editing, and displaying categories.
  - RemindViewModel: Manages reminders for recurring expenses, allowing users to set, remove, edit, and view reminders.
  - UserViewModel: Handles user account-related tasks like signup, login, password reset, saving, and retrieving user profiles.
  - GeminiApiViewModel: Facilitates communication with the Gemini API, sending prompts and receiving responses.
  - ChatbotViewModel: Manages interactions with the AI chatbot, sending user messages and

displaying chatbot responses.
  - TextExtractionViewModel: Processes images using the Firebase ML Vision API to extract text (e.g., from receipts) and convert it into expense records.
  - FirebaseMLVisionAPIViewModel: Sends images to the Firebase ML Vision API for text extraction.
- **LiveData:** A lifecycle-aware observable data holder class. ViewModels use LiveData to expose data to the View, and the View observes LiveData for changes. This ensures that the UI is automatically updated whenever the underlying data changes.

The ViewModel layer interacts with both the View and Model layers. The View triggers actions in the ViewModel through user interactions, and observes LiveData objects exposed by the ViewModel to receive updates and refresh the UI. The ViewModel interacts with Repositories in the Model layer to fetch, update, or delete data, and may trigger actions in other ViewModels to interact with external APIs. It also updates LiveData objects to trigger UI updates in the View, and can use LiveData to notify the View of events. This separation of concerns enhances code maintainability, testability, and scalability.

4.3 Component: Model

4.3.1 Class diagram



ExpenseRepository
- mutableListOf<ExpenseRecord>
+ addExpense(ExpenseRecord): void
+ rmExpense(ExpenseRecord): void
+ editExpense(ExpenseRecord): void
+ getAllRecord(ExpenseRecord): : List<ExpenseRecord>
CategoryRepository
- mutableListOf<Category>
+ addCategory(Category): void
+ rmCategory(Category): void
+ editCategory(Category): void
+ getAllCategory(Category): List<Category>
+ editBudget(Category): void
+ getBudget(Category): float
RemindRepository
- mutableListOf<Remind>
+ setRemind(Remind): void
+ rmRemind(Remind): void
+ editRemind(Remind): void
+ getAllRemind(Remind): List<Remind>
UserRepository
- mutableListOf<Account>
+ signup(Account): void
+ login(Account): void
+ resetPassword(Account): void
+ saveUserprofile(Account): void
+ getUserprofile(Account): void
FirebaseMLVisionAPI
- string API_key
+ sendPicture(array): void
+ receiveResult(): json
+ toExpenseRecord(json): List<ExpenseRecord>
GeminiAPI
- string API_key
+ createPrompt(string): void
+ sendPrompt(string): void
+ getResponse(): json
Chatbot
+ getInformation(): string
+ toAnswer(json): string
TextExtraction
+ toExpenseRecord(json): void

4.3.2 Description

The Model layer is the foundation of the WalletWise application, responsible for managing and accessing data, encapsulating business logic, and interacting with external services. It consists of the following key components:

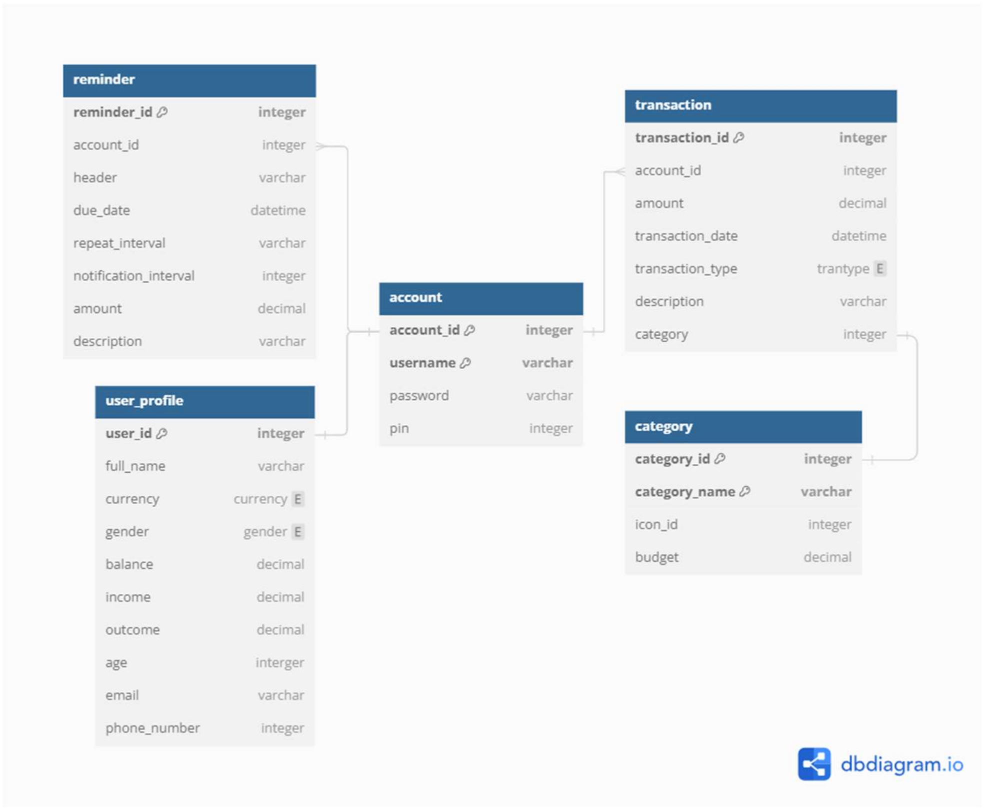
- Repositories serve as intermediaries between the ViewModel and the underlying data sources. They provide a clean and consistent interface for the ViewModel to access and manipulate data, abstracting away the complexities of the data sources. In WalletWise, there are several repositories, each responsible for a specific domain of data:
  - ExpenseRepository: Manages expense records, including adding, removing, editing, and retrieving expenses. It interacts the GeminiApi for potential AI-powered expense analysis.
  - CategoryRepository: Handles categories and their associated budgets. It allows adding, removing, editing categories, and updating their budgets. It also interacts with the GeminiApi for potential AI-based budget recommendations.
  - RemindRepository: Manages reminders for recurring expenses. It enables setting, removing,

editing, and retrieving reminders.
    - UserRepository: Handles user accounts and profiles. It provides functionalities for user signup, login, password reset, and profile management.
- Data sources are the actual locations where data is stored or retrieved. WalletWise utilizes the following data sources:
    - API (Retrofit): A networking library for communicating with external APIs. WalletWise uses Retrofit to interact with Firebase Cloud Database, with the Gemini API for AI-powered features (chatbot, text extractions) and the Firebase ML Vision API for text extraction from receipts (OCR).
    - SharedPreferences: A lightweight mechanism for storing key-value pairs, typically used for user preferences and settings. WalletWise stores user preferences like default currency, notification settings, and themes in SharedPreferences.
- Additional classes:
    - Chatbot: This class encapsulates the logic for interacting with the Gemini API's chatbot functionality. It handles sending user queries, receiving responses, and potentially processing the responses for presentation in the UI.
    - TextExtraction: This class is responsible for extracting relevant information from images (receipts) using the Firebase ML Vision API. It processes the API response and converts the extracted text into a structured format (ExpenseRecord) that can be used within the app.

The Model layer interacts with the ViewModel layer through the repositories. The ViewModels request data from the repositories, which in turn fetch the data from the appropriate data sources (database, API, or SharedPreferences). The repositories may also perform data transformations or caching before returning the data to ViewModels.

4.4 Component: Database

4.4.1 Entity relationship diagrams



4.4.2 Description

- **“account” table:**
  - **Description**: This table stores the usename and password of each user.
  - **Key Fields**:
    - account_id (Primary Key): Unique identifier for each account.
    - user_name (Primary Key): Username of the account.
    - password: Password of the account.
    - pin: Pin of the account.
- **“transaction” table:**
  - **Description**: This table records all the transactions (income, outcome) made by users.
  - **Key Fields**:
    - transaction_id (Primary Key): Unique identifier for each transaction.
    - account_id (Foreign Key): References the account table to indicate which account the transaction is associated with.
    - amount: Amount of money involved in the transaction.
    - transaction_date: Date and time when the transaction occurred.
    - transaction_type: Type of transaction included income and outcome.
    - description: Additional details about the transaction.
    - category (Foreign Key): References the category table to classify the transaction type.
- **“category” table**
  - **Description**: This table categorizes the transactions for better tracking and reporting, such as groceries, utilities, entertainment, etc.
  - **Key Fields**:
    - category_id (Primary Key): Unique identifier for each category.
    - category_name: Unique name of the category (e.g., Groceries, Utilities).
    - icon_id: Identifier for the icon representing the category.
    - budget: Budget limit set for the category, if applicable.
- **“user_profile” table**
  - **Description**: This table contains personal details of users.
  - **Key Fields**:
    - user_id (Primary Key): Unique identifier for each user.
    - full_name: Full name of the user.
    - currency: Default currency for the user's financial transactions and balances.
    - gender: Gender of the user (e.g., male, female, non-binary).
    - balance: Current balance across all accounts of the user.
    - income: Total income recorded for the user.
    - outcome: Total expenses recorded for the user.age: age of the user.

- age: Age of the user.
- email: Email address of the user.
- phone_number: Contact phone number of the user.

- **"reminder" table**
  - **Description**: This table manages reminders set by users for various financial activities, such as bill payments or financial goals.
  - **Key Fields**:
    - reminder_id (Primary Key): Unique identifier for each reminder.
    - account_id (Foreign Key, optional): References the account table if the reminder is related to a specific account.
    - header: Header of the reminder.
    - expiry_date: Date and time when the reminder expires.
    - repeat_interval: Interval at which the reminder repeats (e.g., daily, weekly, monthly).
    - notification_interval: Interval before the reminder date to notify the user.
    - amount: Amount associated with the reminder, if applicable (e.g., bill amount).
    - description: Note about the reminder.

4.5 Machine Learning model design and analysis

4.5.1 Gemini API

Name: Gemini API.
How it works:

- We create prompts tailored to each functionality.
  - AI Chat bot: Prompts are generated based on data from the database to enhance understanding of user spending habits. This allows the AI to provide personalized advice.
  - Text extracting: Prompts are designed to assist the AI in extracting relevant information from user messages. For example, prompts may guide the AI in identifying expenditure details such as amount spent, category, date, and any additional notes.
- Within the MVVM architecture, the API call component manages the interaction with the Gemini API, sending prompts and handling the responses to integrate the AI chatbot seamlessly into our application.
- Gemini API models are deployed dynamically. This means that the models do not require retraining or redeployment from scratch. Instead, they utilize predefined prompts to tailor responses based on the data they receive in real-time. This approach allows for flexibility and quick adaptation to new data or requirements without needing extensive downtime for retraining.
- **Dynamic Deployment:** Models are fetched from Gemini servers at runtime based on the prompts provided by the application. Here is a detailed operation flow when the software is deployed:
  - **Data Input:** Users provide input data, such as text or images.
  - **Preprocessing:** Optional preprocessing steps may be performed to prepare the data.
  - **Prompt Generation:** The application generates prompts based on the input data to send to the Gemini API.
  - **Model Execution:** The Gemini API processes the prompts using the dynamically fetched models to produce the desired output.
  - **Output Handling:** The processed data is returned to the application and used accordingly, such as displaying recognized text or analyzed data to the user.
- **Revision Process:**
  - **Prompt Adjustments:** As user interactions and data evolve, the prompts used by the Gemini API can be adjusted dynamically. This allows for continuous improvement of the AI's accuracy and

relevance without needing a complete redeployment of the application.
    - **Model Updates:** Gemini periodically updates the underlying models to improve performance. These updates are automatically available to applications using the API, ensuring they benefit from the latest enhancements.
- **Revision and Redeployment**: Prompts can be dynamically adjusted, and model updates are automatically integrated, ensuring continuous improvement and minimal disruption.
- Redeployment:
    - **Automatic Updates:** Since the models are hosted and managed by Gemini, any updates to the models are automatically utilized by the application. This eliminates the need for manual updates or redeployment by the developers.
    - **Configuration Management:** Developers can use configuration tools provided by Gemini to manage how and when prompts are updated, ensuring a smooth and controlled rollout of updates.
- **Integration within MVVM Architecture:** The ViewModel manages API interactions, repositories handle data, and LiveData ensures the UI updates dynamically based on processed results.
    - ViewModel Layer:
        - GeminiApiViewModel: This ViewModel is responsible for interacting with the Gemini API. It sends prompts based on user inputs and processes the responses received from the API.
        - LiveData: Exposes ViewModel exposes the results of the API processing via LiveData, which the View observes to update the UI dynamically.
    - Model Layer: The repository handles the download process and caches the model. Once the model is ready, the image is processed, and text is extracted.
        - DataProcessing Class: Manages the creation of prompts and handles the interaction with the Gemini API. Ensures the latest model is used for processing user inputs.
        - Repositories: Act as intermediaries between the ViewModel and the data sources, handling model versions and ensuring efficient data access and management.
    - Database Interactions:
        - Entities and Repositories: The application's database schema includes tables for storing processed data and other relevant information. Repositories manage the storage and retrieval of this data, ensuring consistency and integrity.
        - Retrofit: Used for making network requests, including fetching models and sending prompts to the Gemini API.

Why it is chosen:
- The Gemini API's text processing capabilities make it suitable for extracting structured data from unstructured text messages. By utilizing prompts tailored to each data field, we can efficiently parse user messages and extract pertinent information related to expenditures.

Advantages:
- **Ease of** Implementation**:** Integrating the Gemini API into our MVVM architecture is straightforward, requiring minimal configuration within the API call component.
- **No Training Required:** Unlike custom machine learning models, the Gemini API does not require training, saving time and resources.
- **High Accuracy:** The Gemini API offers high accuracy in understanding user queries and generating relevant responses, ensuring quality interactions with users.

Disadvantages/Limitations:
- **Cost Considerations:** Utilizing the Gemini API may incur costs, depending on usage volume and pricing plans.
- **Dependency on Pretrained Data:** The pretrained data provided by the Gemini API may not fully encompass all financial scenarios, potentially limiting its effectiveness in certain contexts.
- **Not Fully Customizable:** While the Gemini API offers robust functionality, it may not provide the level of customization available with custom-built models tailored to specific use cases.

4.5.2 Firebase ML Vision

Name: Firebase ML Vision
How it works:
- Firebase ML Vision is a part of Firebase ML Kit, Google's machine learning toolkit for mobile app

developers. This technology utilizes machine learning models to recognize text from images captured by the device's camera or uploaded from the device. The API provides the capability to recognize text from images on mobile devices without requiring a network connection. The processing can be performed both on-device (locally) and in the cloud, allowing the app to operate smoothly and flexibly under various network conditions.

- Firebase ML Vision Models are deployed dynamically.
- Dynamic Deployment: Firebase ML Vision models are not bundled with the application. Instead, they are fetched from Firebase servers at runtime, ensuring the application always uses the latest models.
- Here is a detailed operation flow when the software is deployed:
    - **Image Input:** Users capture or select an image using their device's camera or storage.
    - **Preprocessing** (Optional): The image may be preprocessed for better text recognition.
    - **Model Download:** The application checks if the latest model is available locally. If not, it downloads the model from Firebase servers. This is handled by the Firebase ML Kit, which ensures that the most up-to-date model is used.
    - **Text Recognition:** The downloaded model is used to process the image and recognize text. The model runs on the device, leveraging its computational power while ensuring that it's the latest version.
    - **Character Segmentation:** Recognized text is segmented into individual characters.
    - **Text Output:** The extracted text is provided as raw text or structured data.
- **Revision Process:**
    - **Model Updates:** Firebase periodically updates its machine learning models to improve their performance. These updates are made available on Firebase servers.
    - **Testing:** Before making these updates available to all users, Firebase performs thorough testing to ensure the models work correctly and provide the expected accuracy and performance.
- **Redeployment:**
    - **Automatic Fetching of Updates:** When the application starts or when it is about to use the model, it checks Firebase for the latest version of the model. If a new version is available, it is downloaded and cached locally. This ensures that the application uses the latest model without requiring a full app update.
    - **Configuration Management:** Developers can use Firebase Remote Config to manage how and when the models are updated. This allows for controlled rollouts and feature management, ensuring that updates do not disrupt the user experience.
- **Integration within MVVM Architecture:**
    - View Layer: The user captures an image of a receipt. Observes LiveData and updates the UI to display the recognized text.
    - ViewModel Layer: it receives the image and initiates the text recognition process. It checks if the latest model is available locally. If not, it triggers the model download from Firebase. It also processes the recognized text and updates LiveData.
        - FirebaseMLVisionAPIViewModel: This ViewModel handles the interaction with the Firebase ML Vision API. It is responsible for initiating the model download if the latest model is not available locally and for processing images using the dynamically downloaded model.
        - LiveData: Exposes the results of text recognition to the View. The View observes LiveData, ensuring the UI updates automatically when new text is recognized.
    - Model/Repository Layer: The repository handles the download process and caches the model. Once the model is ready, the image is processed, and text is extracted.
        - TextExtraction Class: Manages model downloading and text extraction.
        - Repositories: Handle model versions and updates, ensuring efficient access and data integrity.
    - Database Interactions:
        - Entities and Repositories: Manage storage and access of recognized text and related data.
        - Retrofit: Used for network requests, including downloading models from Firebase.

Why it is chosen:

- Firebase ML Vision is chosen for its dynamic model updates, seamless integration, and high accuracy. It offers pre-trained models that are automatically updated, ensuring the application always uses the latest versions without redeployment. Firebase provides a comprehensive SDK for easy integration, supports both on-device and cloud processing for real-time performance, and scales efficiently. Its infrastructure is cost-effective, offers offline capabilities, and includes robust analytics and A/B testing tools for continuous improvement.

Advantages:

- **Ease of integration:** Firebase ML Vision provides easy-to-use APIs and comprehensive documentation, making it easy for developers to integrate and deploy OCR features into Android apps.
- **High performance:** Firebase ML Vision is built on top of Google's updated machine learning models, ensuring high performance and accuracy in text recognition from images.

Disadvantages/Limitations:

- **Cloud usage cost:** While Firebase Vision OCR can operate offline, using cloud-based processing features may incur costs depending on usage volume and Firebase service pricing.
- **Dependency on Pretrained Data:** it relies on pretrained models from Google, so it may not fully reflect all financial scenarios and specific languages.

4.5.3 Spending Pattern Prediction Model

Name: Spending Pattern Prediction Model

How it works:

- The Spending Pattern Prediction Model utilizes a Seasonal Autoregressive Integrated Moving Average (SARIMA) model to forecast future expenses based on historical spending data.
- SARIMA analyzes past spending patterns, taking into account trends, seasonality, and other factors, to predict future spending amounts for different categories.

Why it is chosen:

- SARIMA is chosen for its effectiveness in handling time series data with seasonality, which is a common characteristic of personal finance data.
- It can capture complex patterns in spending behavior, such as recurring monthly expenses or annual fluctuations, and provide accurate short-term forecasts.

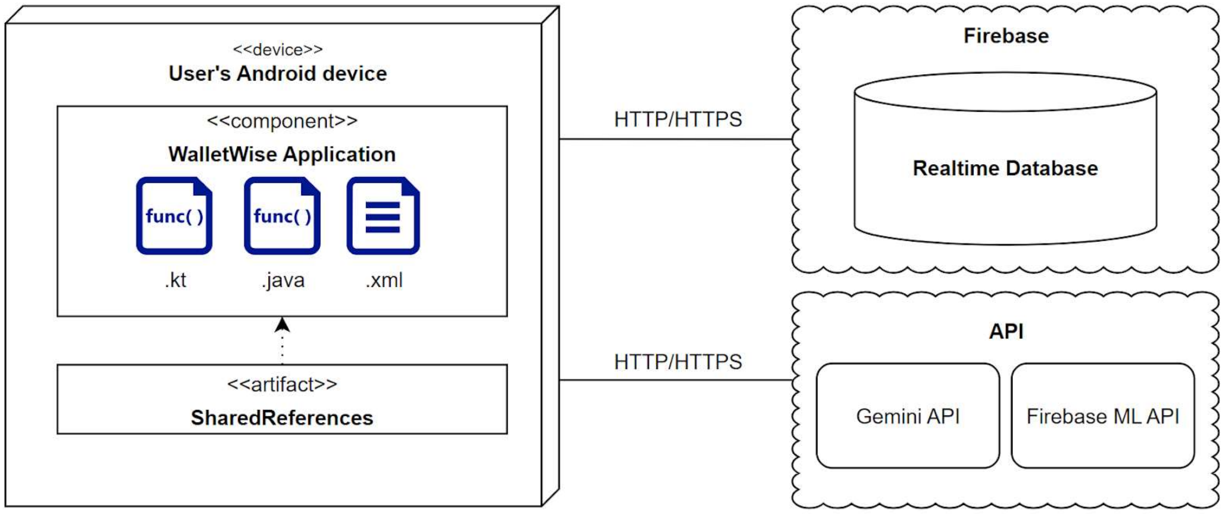
Advantages:

- Accurate Short-Term Forecasts: SARIMA excels at predicting near-future expenses, which is valuable for budgeting and financial planning.
- Handles Seasonality: It effectively models seasonal variations in spending, providing insights into recurring patterns.
- Interpretable: The model's parameters and outputs can be interpreted to understand the factors influencing spending predictions.

Disadvantages/Limitations:

- Requires Sufficient Data: SARIMA requires a substantial amount of historical data to produce reliable forecasts.
- Sensitive to Outliers: Extreme values or outliers in the data can affect the model's accuracy.
- Limited to Short-Term: While accurate for short-term predictions, SARIMA may not be suitable for long-term forecasting due to the dynamic nature of personal finances.

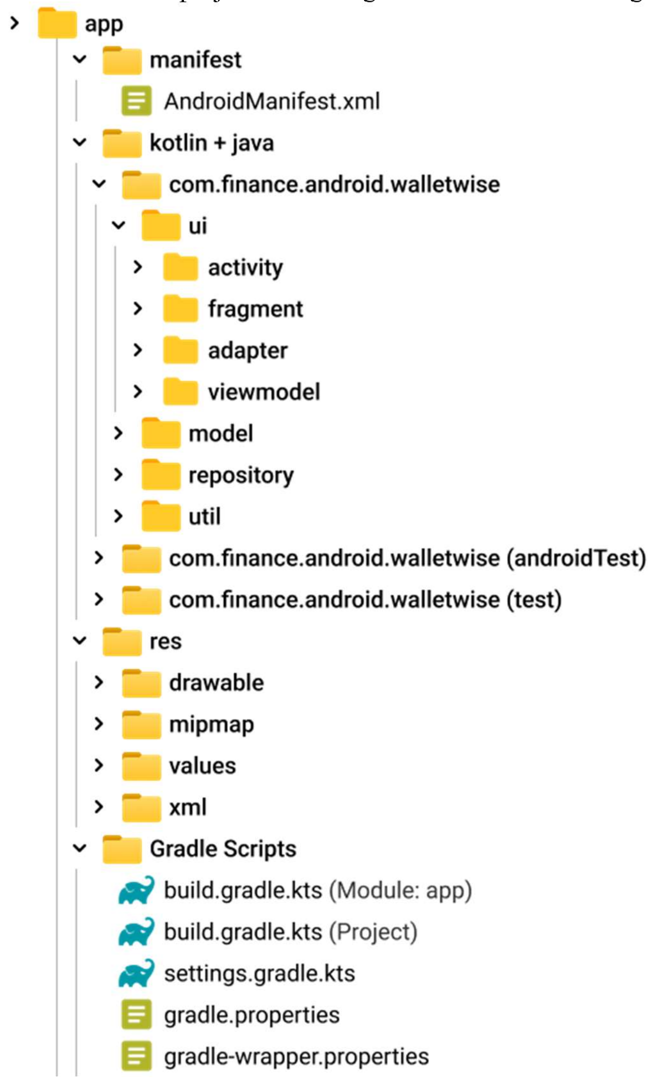
5. Deployment



- The application resides on the User's Android Device, utilizing Kotlin (.kt), Java (.java), and XML (.xml) files for its implementation. It leverages SharedPreferences to store user settings and preferences locally.
- The app communicates with Firebase, a cloud-based platform, through the HTTPS protocol. Firebase provides a Realtime Database for storing and synchronizing financial and user data.
- Additionally, the app interacts with two external APIs, also via HTTPS:
  - Gemini API: This API likely provides AI-powered features like natural language processing or financial analysis.
  - Firebase ML API: This API is probably used for tasks like text recognition or image processing, potentially for extracting data from receipts.
- The WalletWise application will primarily be deployed as an Android Package (APK) file, available for download and installation on Android devices (smartphones and tablets) running Android 9.0 (Pie) and above. This deployment strategy targets the vast majority of Android users and ensures compatibility with a wide range of devices.

6. Implementation View

- The WalletWise project will be organized into the following folder structure:



```
> app
  v manifest
      AndroidManifest.xml
  v kotlin + java
    v com.finance.android.walletwise
      v ui
        > activity
        > fragment
        > adapter
        > viewmodel
      > model
      > repository
      > util
    > com.finance.android.walletwise (androidTest)
    > com.finance.android.walletwise (test)
  v res
    > drawable
    > mipmap
    > values
    > xml
  v Gradle Scripts
      build.gradle.kts (Module: app)
      build.gradle.kts (Project)
      settings.gradle.kts
      gradle.properties
      gradle-wrapper.properties
```

- User application source code view:
  - **manifest** folder: contain AndroidManifest.xml file, it is essential file for any Android app. It declares the components of the application (activities, services, receivers) and application's permissions.
  - **kotlin+java** folder:
    - **com.finance.android.walletwise**:
      - **ui**: contains UI-related components.
        - activity: activity classes, which represent the screens of the app.
        - fragment: fragment classes, which are reusable UI components.
        - adapter: help connect data to views.
        - viewmodel: viewModel classes, responsible for preparing and managing data for app's UI.
      - **model:** data classes that represent the entities in the application.

            - **repository:** repository classes, which act as a bridge betweenViewModels and data sources.
            - **util:** contains utility classes for various purposes
        - **com.finance.android.walletwise** (AndroidTest): instrumented tests (tests that run on an Android device or emulator)
        - **com.finance.android.walletwise** (Test): local unit tests (tests that run on development machine's JVM).
    - **res** folder:
        - **drawable**: image and vector files (PNG, XML, ...) used in app's UI.
        - **mipmap**: launcher icons for different screen densities.
        - **values**: XML files defining various resources like strings, colors, styles, …
        - **xml**: other XML files.
    - **Gradle Scripts**:
        - **build.gradle.kts** (Module: app): This file is specific to the app module and defines its build configurations, dependencies, etc.
        - **build.gradle.kts** (Project): This file is for the entire project and includes configurations that apply to all modules.
        - **settings.gradle.kts**: Defines the modules included in the project.
        - **gradle.properties**: Project-wide Gradle properties.
        - **gradle-wrapper.properties**: Manages the Gradle wrapper, ensuring consistent builds across different environments.