The Islamia University of Bahawalpur

Department of Software Engineering

Faculty of Computing

SOFTWARE DESIGN DOCUMENT (SDD)

for

Financemate

Version 1.0.0

By

Student Name: Shoaib Ahmad

Roll No: F21BSEEN1E02039

Section: 7th-1E

Session: Fall 2021 – 2025

Supervisor: Sir Umair Zafar Khan

Bachelor of Science in Software Engineering

Revision History

Date	Description	Author	Comments
10/11/2024	Selected the name KhataApp	Shoaib Ahmad	First Name Selection
01/01/2025	Changed name to Financemate	Shoaib Ahmad	Changed and Finalized Name

Document Approval

The following Software Design Document (SDD) has been accepted and approved by the following:

Signature	Printed Name	Title	Date
		Supervisor,	

Table of Contents

1. INTRODUCTION

The "Financemate" System Sequence Diagram (SSD) provides a detailed representation of the interactions between users and the system's core functionalities. This document is structured to ensure a clear understanding of how the "Financemate" system processes user inputs and delivers desired outcomes, facilitating personal and small business financial management.

The SSD visualizes the flow of operations for key scenarios, offering insights into the system's behaviour during user interactions. These scenarios include authentication, financial data management, and report generation. By modelling these sequences, the SSD highlights how the system handles user requests step by step, ensuring transparency and alignment with the project's requirements.

Key Objectives:

1. **Illustrate Core Processes**: Show the sequential interaction between users and the system.
2. **Simplify Understanding**: Provide stakeholders with a clear visualization of system behaviour.
3. **Support Development**: Aid developers in implementing processes aligned with user expectations.

The subsequent sections detail specific use cases, including user authentication, dashboard interactions, transaction management, and financial report generation. Each SSD ensures that all system functionalities operate cohesively to deliver a seamless user experience.

1.1. Purpose

This System Sequence Diagram (SSD) document describes the architecture and design of the "Financemate" system. It aims to provide a comprehensive view of the sequential interactions between the system and its users for various use cases. The primary purpose is to guide developers, stakeholders, and testers in understanding and implementing the system's design efficiently and effectively.

Intended Audience:

- **Developers**: To gain clarity on the sequence of interactions and the system's expected behaviour.
- **Stakeholders**: To ensure the system's design aligns with user needs and business goals.
- **Testers**: To validate that the system functions as intended based on the documented sequences.

1.2. Scope

The "Financemate" software is a comprehensive personal financial management application designed to assist users in tracking, organizing, and analyzing their financial activities. It

provides an intuitive and user-friendly interface to streamline financial decision-making processes for both individuals and small businesses.

Goals and Objectives:

1. **Simplify Financial Management**: Provide tools for effortless tracking of income, expenses, and savings.
2. **Enhance Decision-Making**: Offer personalized insights and real-time analytics to empower users in making informed financial choices.
3. **Ensure Accessibility and Security**: Create a scalable platform that prioritizes data security and is accessible across devices.
4. **Promote Financial Literacy**: Deliver features that educate and guide users toward achieving financial stability and goals.

Scope:

The system encompasses the following core functionalities:

1. **User Authentication and Authorization**: Secure login processes with multi-factor authentication to ensure data protection.
2. **Dashboard and Reporting**: Interactive dashboards to visualize spending patterns and generate detailed financial reports.
3. **Transaction Management**: Tools to record, categorize, and manage financial transactions with options for automation.
4. **Goal Setting and Budgeting**: Features to set financial goals and track budgets with real-time updates.
5. **Integration with Financial Systems**: Seamless integration with banking APIs for automated data synchronization.
6. **Alerts and Notifications**: Customizable reminders for bill payments, budget limits, and savings progress.

Benefits:

- **Streamlined User Experience**: A unified platform for all financial activities reduces the complexity of personal financial management.
- **Personalized Insights**: Advanced analytics powered by machine learning offer tailored advice for improved financial outcomes.
- **Time-Saving Features**: Automation of transaction tracking and report generation minimizes manual effort.
- **Security Assurance**: Robust encryption and multi-factor authentication safeguard user data.
- **Scalability**: Designed to accommodate diverse user needs, from individuals to small businesses.

1.3. Overview

This document is organized to provide a comprehensive understanding of the "Financemate" System Sequence Diagram (SSD). It serves as a structured guide for stakeholders, including developers, testers, and business managers, to ensure alignment with the project's objectives and requirements.

Organization:

1. **Introduction**: Sets the context for the SSD and outlines its purpose and objectives.
2. **Purpose**: Explains the rationale behind the SSD and identifies its intended audience.
3. **Description and Scope**: Details the software's functionalities, goals, and benefits.
4. **System Sequence Diagrams**: Provides detailed graphical representations of user-system interactions for key use cases.
5. **Supporting Information**: Includes appendices, references, and other supplementary materials.

1.4. Reference Material

• Financemate Project Proposal Document

- Provides detailed information about the objectives, scope, methodology, and proposed features of the Financemate project.

• Financemate Software Requirements Specification (SRS) Document

- Version: 1.0
- Author: Shoaib Ahmad
- Key Sections Referenced:
  - Functional Requirements (Section 3.2)
  - Use Cases (Section 3.3)
  - Analysis Models: Sequence Diagrams, Data Flow Diagrams (Section 4)

• IEEE Standard for Software Requirements Specifications (IEEE 830-1998)

- Source: IEEE Xplore
- Provides guidelines for structuring the SRS and test plans.

• Flutter Documentation

- URL: Flutter Official Website
- Framework documentation for creating cross-platform user interfaces.

• Dart Programming Language Documentation

- URL: Dart Programming Language
- Language reference and best practices.

• MySQL Documentation

- Version: 8.0
- URL: MySQL Documentation
- Details about database design, queries, and optimization techniques.

• Plaid API Documentation

- URL: Plaid API
- Used for integrating banking data and managing financial transactions.

• **Machine Learning Yearning by Andrew Ng**

- Author: Andrew Ng
- Concepts for implementing predictive analytics and transaction categorization.

• **General Data Protection Regulation (GDPR)**

- Source: Official EU GDPR Documentation
- URL: GDPR Compliance
- Reference for security and data privacy compliance.

• **Material Design Guidelines**

- Source: Google Design
- URL: Material Design Guidelines
- Best practices for UI/UX design.

1.5. Definitions and Acronyms

This section provides definitions of terms, acronyms, and abbreviations used in the System Sequence Diagram (SSD) for Financemate to ensure clarity and proper interpretation.

Definitions

1. **Financemate**: A personal financial management application designed to help users track their spending, manage budgets, and make informed financial decisions.
2. **System Sequence Diagram (SSD)**: A UML diagram that illustrates the sequence of interactions between actors (users or external systems) and the system under development for specific scenarios.
3. **Actor**: An entity (e.g., user, admin, external system) that interacts with the Financemate system.
4. **Transaction Categorization**: The process of assigning financial transactions to predefined categories (e.g., groceries, utilities) for analysis.
5. **Machine Learning (ML)**: A subset of artificial intelligence used for analyzing user behavior and providing personalized insights.
6. **Real-Time Analytics**: Processing and displaying financial data dynamically as users interact with the system.
7. **Localization**: Adapting the system to different languages, regional financial formats, and cultural norms.
8. **Multi-Factor Authentication (MFA)**: A security method requiring two or more verification factors to authenticate a user's identity.
9. **RESTful API**: An architectural style for designing networked applications, enabling communication between the frontend and backend systems.

Acronyms

1. **API**: Application Programming Interface – A set of rules that allow software applications to communicate with each other.
2. **UI**: User Interface – The graphical part of the application where users interact with the system.
3. **UX**: User Experience – The overall experience of a user when interacting with the system.
4. **ML**: Machine Learning – Algorithms used for data analysis and predictions.
5. **CRUD**: Create, Read, Update, Delete – Basic operations performed on database entities.
6. **JSON**: JavaScript Object Notation – A lightweight data-interchange format used in APIs.
7. **CSV**: Comma-Separated Values – A file format used for importing/exporting data.
8. **GDPR**: General Data Protection Regulation – A European regulation for data protection and privacy.
9. **MFA**: Multi-Factor Authentication – A method of securing access through multiple verification steps.
10. **HTTP**: HyperText Transfer Protocol – A protocol used for data communication on the web.
11. **TLS**: Transport Layer Security – A protocol for encrypting data in transit.
12. **DBMS**: Database Management System – Software used to manage and query databases.
13. **RDBMS**: Relational Database Management System – A type of DBMS that organizes data into tables.
14. **DART**: The programming language used for developing Financemate.
15. **SRS**: Software Requirements Specification – A document detailing system requirements.
16. **SSD**: System Sequence Diagram – A UML representation of system interactions.
17. **DFD**: Data Flow Diagram – A visual representation of the flow of data in a system.

2. SYSTEM OVERVIEW

The **Financemate** project is a personal financial management application designed to empower users by simplifying financial tracking and decision-making. It provides a centralized platform where users can monitor their income, expenses, and savings while leveraging real-time analytics and machine learning-based insights for better financial planning.

Functionality

The key functionalities of Financemate include:

1. **User Authentication**: Secure access to the system through multi-factor authentication (MFA) and third-party login integrations (e.g., Google).
2. **Transaction Management**:
    - Adding, editing, and categorizing transactions.
    - Automatic categorization using machine learning algorithms.

    - Viewing and filtering transaction history.
3. **Budgeting Tools**:
    - Setting and managing budgets for specific timeframes (e.g., monthly, yearly).
    - Notifications for nearing budget limits.
4. **Personalized Insights**:
    - Financial advice and predictions based on user behavior and spending patterns.
    - Visualized analytics, including spending trends and saving opportunities.
5. **Reports and Alerts**:
    - Generating downloadable reports in formats like PDF and Excel.
    - Alerts for upcoming bills and payment reminders.
6. **Data Import/Export**:
    - Seamless integration with banking systems through APIs like Plaid.
    - Support for importing/exporting data in formats such as CSV and JSON.

Context

Financemate operates within the domain of personal and small business financial management. It bridges the gap between overly complex tools used by professionals and the simplistic apps available for casual users.

- **Target Users**:
    - Individuals: Seeking simple tools to track daily expenses and savings.
    - Small Business Owners: Managing budgets, cash flows, and financial records.
- **Integration**:
    - Connects to banking APIs for real-time transaction synchronization.
    - Supports cross-platform accessibility via responsive web and mobile applications.

Design

The architecture of Financemate emphasizes modularity, scalability, and security.

1. **Frontend**:
    - Built with Flutter for cross-platform compatibility on web, Android, and iOS.
    - Intuitive and responsive UI following Material Design guidelines.
2. **Backend**:
    - RESTful API for communication between the user interface and the server.
    - Developed using Dart and integrated with a PostgreSQL database.
3. **Machine Learning Module**:
    - Uses TensorFlow or Scikit-learn for financial behavior prediction and transaction categorization.
4. **Security**:
    - AES-256 encryption for sensitive data storage and transmission.
    - OAuth 2.0 for secure authentication.

Background Information

The system was conceptualized to address common pain points faced by users in managing their finances. Existing solutions like Mint and YNAB often fail to strike a balance between usability and functionality, leading to complexity for non-technical users. Financemate differentiates itself by:

- **Simplifying financial tracking** through automation and machine learning.
- **Focusing on accessibility**, with localization options for diverse user needs.
- **Providing real-time insights** to help users make informed decisions.

3. SYSTEM ARCHITECTURE

The architecture of **Financemate** is designed to ensure modularity, scalability, security, and seamless integration across multiple platforms. It follows a multi-layered architecture that includes the presentation layer, business logic layer, and data layer, supported by modern frameworks and technologies.

High-Level Architecture

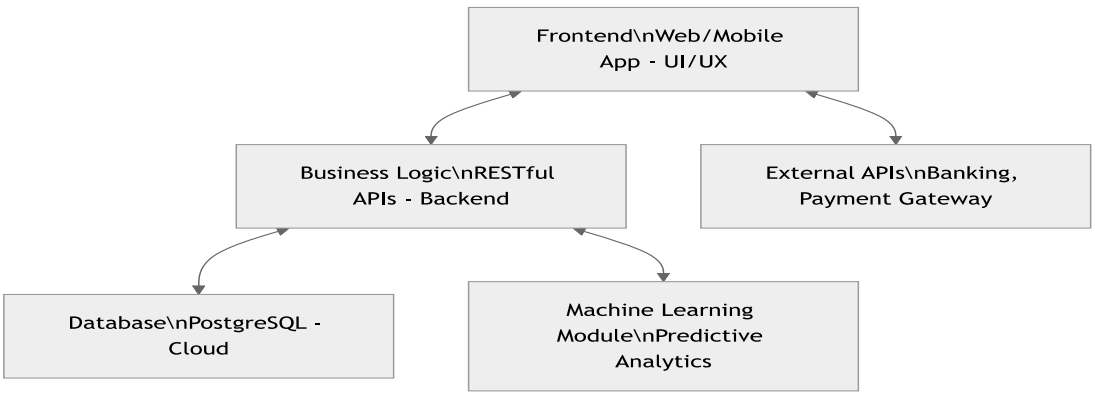
1. **Presentation Layer**:
    - **Purpose**: Provides the user interface (UI) and user experience (UX) for web and mobile applications.
    - **Technologies**: Flutter framework for developing cross-platform applications (web, iOS, Android).
    - **Features**:
        - Responsive design ensuring a seamless experience across devices.
        - Localization support for diverse languages and regional financial formats.
2. **Business Logic Layer**:
    - **Purpose**: Handles the core functionalities, processes, and system logic.
    - **Technologies**:
        - Dart programming language.
        - RESTful APIs for communication between the frontend and backend.
    - **Key Components**:
        - **Authentication Service**: Manages user login, registration, and role-based access control.
        - **Transaction Processor**: Handles data validation, transaction categorization, and real-time analytics.
        - **Notification Manager**: Sends alerts and reminders via email, SMS, and push notifications.
        - **Machine Learning Module**: Implements predictive analytics and personalized insights.
3. **Data Layer**:
    - **Purpose**: Stores and manages all system data, ensuring integrity, security, and availability.
    - **Technologies**: PostgreSQL as the primary relational database.
    - **Key Features**:

- AES-256 encryption for secure data storage.
- Support for structured (transaction data) and semi-structured (user preferences) data.

System Components and Interactions

1. **Frontend (Flutter)**:
   - Manages the user interface and captures user inputs.
   - Sends requests to the backend via RESTful APIs.
   - Displays real-time data and visual analytics using prebuilt widgets and charts.
2. **Backend (RESTful APIs)**:
   - Processes incoming requests from the frontend.
   - Implements business rules and logic for features like transaction categorization, budget tracking, and report generation.
   - Communicates with the database and external services (e.g., banking APIs).
3. **Database (PostgreSQL)**:
   - Stores user information, transactions, budget details, and analytics data.
   - Ensures data consistency through relational schema design.
   - Facilitates fast queries for generating real-time insights.
4. **Machine Learning Module**:
   - Utilizes TensorFlow or Scikit-learn for:
     - Predicting user spending patterns.
     - Automating transaction categorization.
     - Providing actionable financial insights.
   - Operates as a microservice, communicating with the backend for model training and inference.
5. **External Services Integration**:
   - **Banking APIs** (e.g., Plaid): Retrieves real-time transaction data.
   - **Payment Gateways** (if applicable): Facilitates secure online payments.
   - **Cloud Messaging Services** (e.g., Firebase): Sends alerts and push notifications.
6. **Security Features**:
   - Encryption protocols (TLS for data in transit and AES for data at rest).
   - OAuth 2.0 for secure authentication and session management.
   - Regular security audits to address vulnerabilities.

System-Level Block Diagram



Design Considerations

1. **Scalability**:
    - Horizontal scaling supported by RESTful API architecture.
    - PostgreSQL database optimized for handling growing data volumes.
2. **Security**:
    - Encryption of sensitive data (e.g., transactions, user details).
    - Regular security audits to maintain compliance with GDPR and CCPA.
3. **Performance**:
    - Asynchronous processing for faster API responses.
    - Machine learning models optimized for quick predictions and low latency.
4. **Usability**:
    - Intuitive UI for users with varying technical expertise.
    - Accessibility features for inclusivity (e.g., screen reader compatibility).
5. **Maintainability**:
    - Modular design to support updates without affecting unrelated components.
    - Code documentation following standard guidelines.

3.1. Architectural Design

The **Financemate** system is structured into high-level subsystems, each responsible for distinct functionality. These subsystems work collaboratively to ensure a modular, scalable, and maintainable design. This modularization enables efficient development, testing, and future scalability.

Modular Program Structure

The system is decomposed into the following high-level subsystems:

1. User Interface (UI) Subsystem

- **Responsibilities**:
    - Captures user inputs and displays processed outputs.
    - Provides interactive dashboards, forms, and reports.
    - Adapts to various devices through responsive design.
- **Technologies**: Built using Flutter for cross-platform compatibility (web, iOS, Android).

2. Authentication and Authorization Subsystem

- **Responsibilities**:
    - Manages user registration, login, and session handling.
    - Implements multi-factor authentication (MFA) and role-based access control.
- **Collaboration**: Communicates with the Database Subsystem to validate credentials and store tokens.

3. Transaction Management Subsystem

- **Responsibilities**:
  - Allows users to add, edit, categorize, and view transactions.
  - Integrates with banking APIs for automatic transaction synchronization.
- **Collaboration**: Utilizes the Machine Learning Subsystem for categorization and the Database Subsystem for storage.

4. Budgeting and Analytics Subsystem

- **Responsibilities**:
  - Enables users to set budgets, track spending, and view financial analytics.
  - Generates visualizations like charts and graphs.
- **Collaboration**: Fetches and processes data from the Transaction Management and Machine Learning subsystems.

5. Machine Learning Subsystem

- **Responsibilities**:
  - Implements predictive analytics for spending patterns and transaction categorization.
  - Provides personalized financial insights to users.
- **Technologies**: Uses TensorFlow or Scikit-learn for model training and inference.
- **Collaboration**: Communicates with the Database Subsystem for training data and shares predictions with the Transaction Management Subsystem.

6. Notification and Alert Subsystem

- **Responsibilities**:
  - Sends reminders for bill payments, budget limits, and other user-configured alerts.
  - Supports email, SMS, and push notifications.
- **Collaboration**: Integrates with external messaging services like Firebase for push notifications and uses the Database Subsystem for user preferences.

7. Database Subsystem

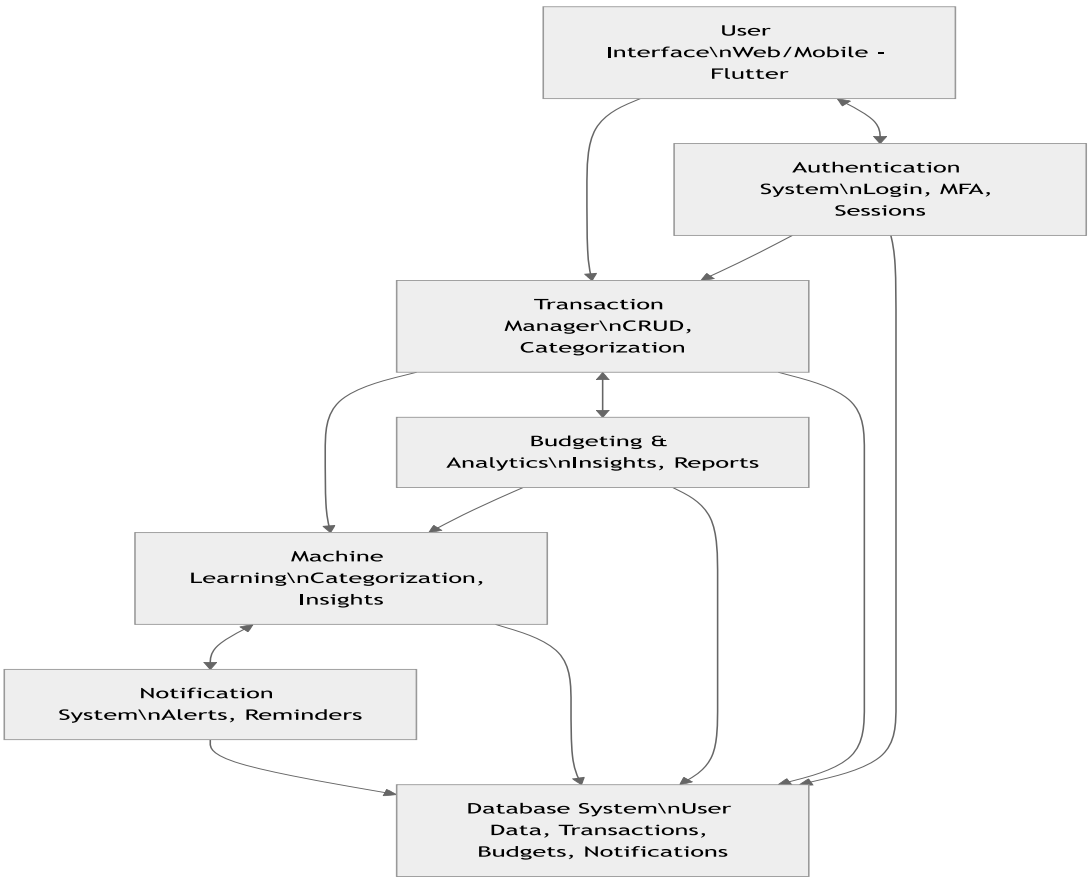
- **Responsibilities**:
  - Stores and manages user data, transactions, budgets, and system logs.
  - Ensures data security through encryption and access controls.
- **Technologies**: PostgreSQL for relational data management.
- **Collaboration**: Serves as the central repository for all subsystems.

Subsystem Interactions

The subsystems interact through well-defined APIs and shared data repositories, ensuring modularity and separation of concerns:

1. **User Interactions**:
   - The **UI Subsystem** captures user actions and communicates with backend APIs in the Business Logic Layer.
2. **Data Flow**:
   - The **Authentication Subsystem** validates credentials and shares session tokens with other subsystems.
   - The **Transaction Management Subsystem** retrieves and updates data in the **Database Subsystem** and invokes the **Machine Learning Subsystem** for categorization.
3. **Analysis and Insights**:
   - The **Budgeting and Analytics Subsystem** processes data fetched from the **Database Subsystem** and generates insights using models from the **Machine Learning Subsystem**.
4. **Notification Delivery**:
   - The **Notification and Alert Subsystem** fetches alert configurations from the **Database Subsystem** and sends messages through external services.

Diagram of Major Subsystems and Data Repositories



Description of the Diagram

1. **User Interface (UI Subsystem)**:
    - Interacts with the **Authentication System** for login and user verification.
    - Sends user requests to the **Transaction Manager** and **Budgeting System** via APIs.
2. **Authentication System**:
    - Communicates with the **Database System** to validate credentials and manage sessions.
3. **Transaction Manager**:
    - Retrieves and updates transaction data in the **Database System**.
    - Collaborates with the **Machine Learning Subsystem** for automated categorization.
4. **Budgeting and Analytics**:
    - Fetches financial data from the **Database System**.
    - Utilizes insights from the **Machine Learning Subsystem** to provide visual analytics.
5. **Machine Learning Subsystem**:
    - Uses historical data from the **Database System** to train and infer predictions.
    - Shares results with the **Transaction Manager** and **Budgeting System**.
6. **Notification System**:
    - Reads user preferences and configurations from the **Database System**.
    - Sends reminders and alerts via external messaging services.
7. **Database System**:
    - Acts as the central repository for all system data.
    - Ensures secure, consistent, and efficient data storage and retrieval.

3.2. Decomposition Description

The decomposition of the **Financemate** system is provided using an **Object-Oriented (OO)** description. This approach leverages subsystem models, object diagrams, generalization hierarchies, and sequence diagrams to demonstrate how the system's subsystems interact and are structured.

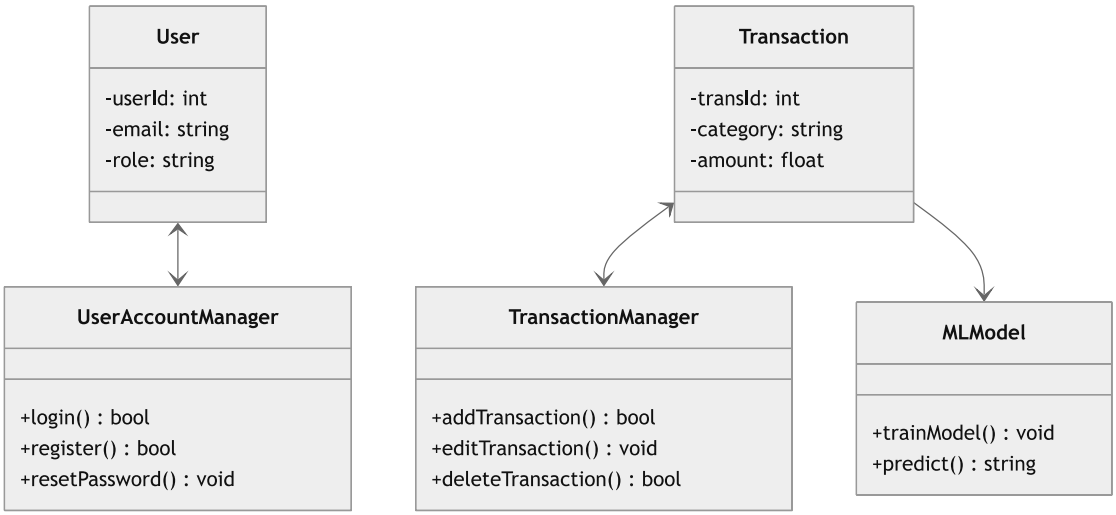
Subsystem Model

The **Financemate** system consists of the following core subsystems:

1. **User Management Subsystem**: Handles user registration, login, and role-based access control.
2. **Transaction Management Subsystem**: Manages CRUD operations for financial transactions and applies ML categorization.
3. **Budgeting Subsystem**: Allows users to create budgets, track expenditures, and set goals.
4. **Analytics and Insights Subsystem**: Processes data to generate visualizations and personalized insights.
5. **Notification Subsystem**: Sends user-configured alerts for reminders and budget limits.

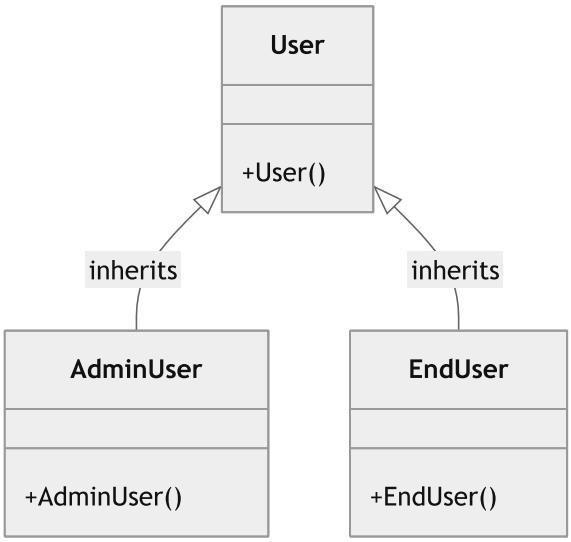
6. **Integration Subsystem**: Facilitates integration with banking APIs and third-party tools for importing/exporting data.
7. **Database Subsystem**: Manages data storage and retrieval securely.
8. **Security Subsystem**: Provides encryption, authentication, and secure data access.

Object Diagram



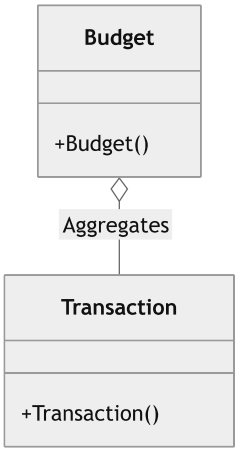
Generalization Hierarchy Diagram

The generalization hierarchy showcases the inheritance structure for the **User** and **Transaction** classes:



Aggregation Hierarchy Diagram

The aggregation hierarchy highlights the relationship between the **Budget** and **Transaction** classes:



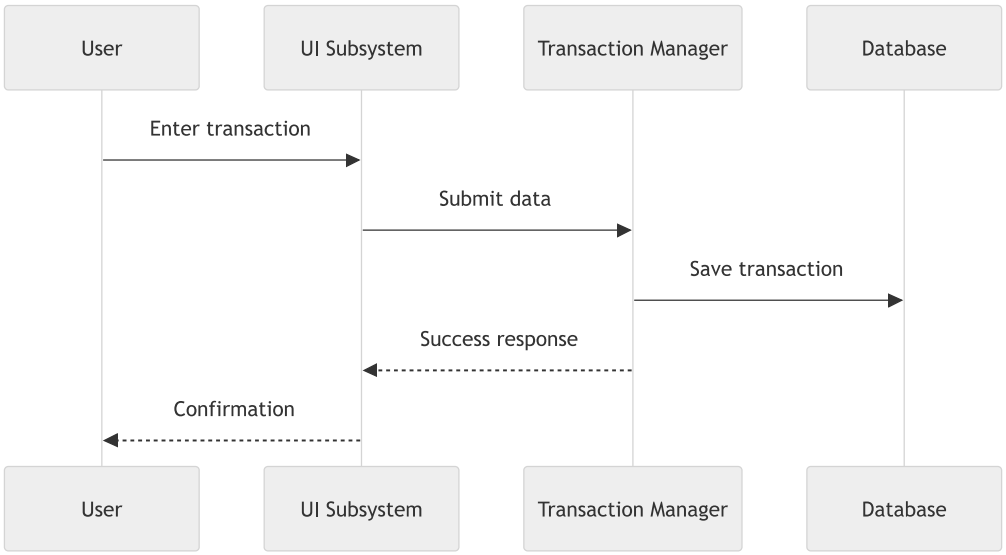
Interface Specifications

Each subsystem provides a well-defined interface for communication. Example specifications:

1. **User Management Subsystem**:
   - login(email: string, password: string) -> bool: Authenticates a user.
   - register(details: dict) -> bool: Registers a new user.
2. **Transaction Management Subsystem**:
   - addTransaction(transaction: dict) -> bool: Adds a new transaction.
   - categorizeTransaction(transactionId: int) -> string: Applies ML categorization.
3. **Budgeting Subsystem**:
   - createBudget(budgetDetails: dict) -> bool: Creates a new budget.
   - checkBudgetStatus(budgetId: int) -> string: Checks if spending is within limits.

Sequence Diagram

Use Case: Add Transaction



Subsystem Model Summary

1. **User Management**
   - Maintains user roles and credentials.
   - Interfaces with authentication and database subsystems.
2. **Transaction Management**
   - Centralizes CRUD operations for transactions.
   - Communicates with ML models for categorization and insights.
3. **Budgeting**
   - Tracks user budgets and alerts for overspending.
4. **Analytics and Insights**
   - Processes data for visualizations and ML-based predictions.
5. **Notifications**
   - Handles alerts for due dates, budgets, and financial goals.

3.3. Design Rationale

The architecture selected for **Financemate** was chosen to meet the project's goals of scalability, security, modularity, and ease of use. The chosen design follows a multi-layered architecture combining a **RESTful API-driven backend**, **cross-platform frontend**, and **centralized database** to achieve these objectives.

Rationale for Selected Architecture

1. **Scalability**:
   - The multi-layered architecture allows horizontal scaling of the backend and database as user demand grows.

   - By decoupling the frontend and backend, independent scaling is achievable.
2. **Modularity**:
   - Subsystems such as user management, transaction management, and budgeting are modularized to isolate responsibilities, making the system easier to develop, test, and maintain.
   - The machine learning module operates as an independent service, ensuring that ML workloads do not affect core system performance.
3. **Cross-Platform Compatibility**:
   - Flutter was chosen for the frontend to ensure a single codebase can deliver consistent user experiences on web, Android, and iOS platforms.
4. **Security**:
   - The architecture incorporates security best practices such as AES-256 encryption for data at rest and TLS for data in transit.
   - OAuth 2.0 ensures secure authentication, and role-based access control prevents unauthorized actions.
5. **Integration Capabilities**:
   - The use of RESTful APIs allows seamless integration with external systems like banking APIs (e.g., Plaid) and payment gateways.
6. **Ease of Maintenance**:
   - The modular structure allows updates or replacements of individual components without disrupting the entire system.
   - The use of widely supported technologies like PostgreSQL ensures long-term maintainability.

Critical Issues and Trade-offs

1. **Monolithic vs. Modular Design**:
   - **Chosen**: Modular design with clearly defined subsystems.
   - **Rationale**: While monolithic architectures can simplify development in smaller projects, they limit scalability and hinder maintenance. A modular approach ensures each subsystem can evolve independently.
2. **Database Choice (SQL vs. NoSQL)**:
   - **Chosen**: PostgreSQL (Relational SQL database).
   - **Rationale**: Relational databases provide strong consistency and integrity, which is essential for financial data. NoSQL databases were considered for their scalability but were deemed unsuitable due to the structured nature of the data.
3. **Frontend Framework**:
   - **Chosen**: Flutter.
   - **Rationale**:
     - Flutter allows the development of a single codebase for web, Android, and iOS, reducing time and costs.
     - Alternative frameworks like React Native and Swift were considered but lacked Flutter's flexibility for cross-platform support.
     - React Native was close in consideration but required additional customization for web compatibility.
4. **Machine Learning Integration**:
   - **Chosen**: A separate microservice for ML workloads.

- **Rationale**: Integrating ML directly into the backend would have introduced performance bottlenecks. Running ML as a microservice allows independent scaling and updates to ML models without affecting core services.

5. **REST vs. GraphQL APIs**:
   - **Chosen**: RESTful APIs.
   - **Rationale**:
     - RESTful APIs are widely supported and simpler to implement for the project's requirements.
     - GraphQL was considered for its flexibility but was deemed unnecessary given the project's relatively straightforward data requirements.

Alternative Architectures Considered

1. **Monolithic Architecture**
   - **Description**: A single codebase for the frontend, backend, and database interactions.
   - **Reason for Rejection**:
     - Limited scalability as the system grows.
     - Any failure or update in one part of the system could affect the entire application.
2. **Microservices Architecture**
   - **Description**: Independent services for user management, transactions, notifications, and more.
   - **Reason for Rejection**:
     - Overhead in managing and deploying multiple services, especially for a project of this scope.
     - Required expertise in containerization and service orchestration tools like Kubernetes, increasing complexity and development time.
3. **Serverless Architecture**
   - **Description**: Using cloud functions for each subsystem, with no dedicated backend.
   - **Reason for Rejection**:
     - High costs for services with frequent read/write operations, as is common in financial applications.
     - Limited control over the backend environment and potential challenges in debugging and optimizing performance.

Trade-offs in Chosen Architecture

1. **Performance vs. Modularity**:
   - A modular approach may introduce slight overhead in communication between subsystems but improves maintainability and scalability.
2. **Learning Curve vs. Future Proofing**:
   - Flutter has a smaller developer community compared to React Native, but its features and performance outweigh this limitation for cross-platform development.
3. **Flexibility vs. Complexity**:

- RESTful APIs provide simplicity and broad compatibility but are less flexible than GraphQL. However, REST was sufficient for the project's data flow needs.

4. DATA DESIGN

Overview

The data design for Financemate ensures that data is structured, managed, and stored effectively to support all system operations, including financial tracking, reporting, and analytics. This section details the database schema, relationships, and data flow within the system.

2. Logical Database Requirements

1. **User Data**
   - **Entities**: User accounts, roles, and preferences.
   - **Attributes**: User ID, username, email, hashed password, role, and preferences.
   - **Relationships**: Users have roles and may own multiple financial accounts.
2. **Financial Transactions**
   - **Entities**: Transactions, categories, and budgets.
   - **Attributes**: Transaction ID, user ID (foreign key), amount, type (income/expense), category ID, timestamp, and description.
   - **Relationships**: Each transaction belongs to a user and a category.
3. **Budget Management**
   - **Entities**: Budget categories and budget goals.
   - **Attributes**: Budget ID, category ID, user ID, target amount, and start/end dates.
   - **Relationships**: Budgets are linked to categories and users.
4. **Analytics and Insights**
   - **Entities**: Aggregated user data for insights and trends.
   - **Attributes**: User ID, summary reports, predicted spending patterns, and goal achievement status.

3. **Data Entities and Relationships**

Entity	Attributes	Relationships
User	user_id, username, email, password, role	Owns transactions, manages budgets.
Transaction	transaction_id,user_id,category_id,amount,type,description,timestamp	Belongs to a user and a category.
Category	category_id, name, type (income/expense), user_id	Associated with transactions and budgets.
Budget	budget_id,category_id,target_amount,start_date,end_date	Linked to a user and category.
Insights	insight_id,user_id,summary,predicted_trends	Generated based on user transaction history.

4. Database Schema

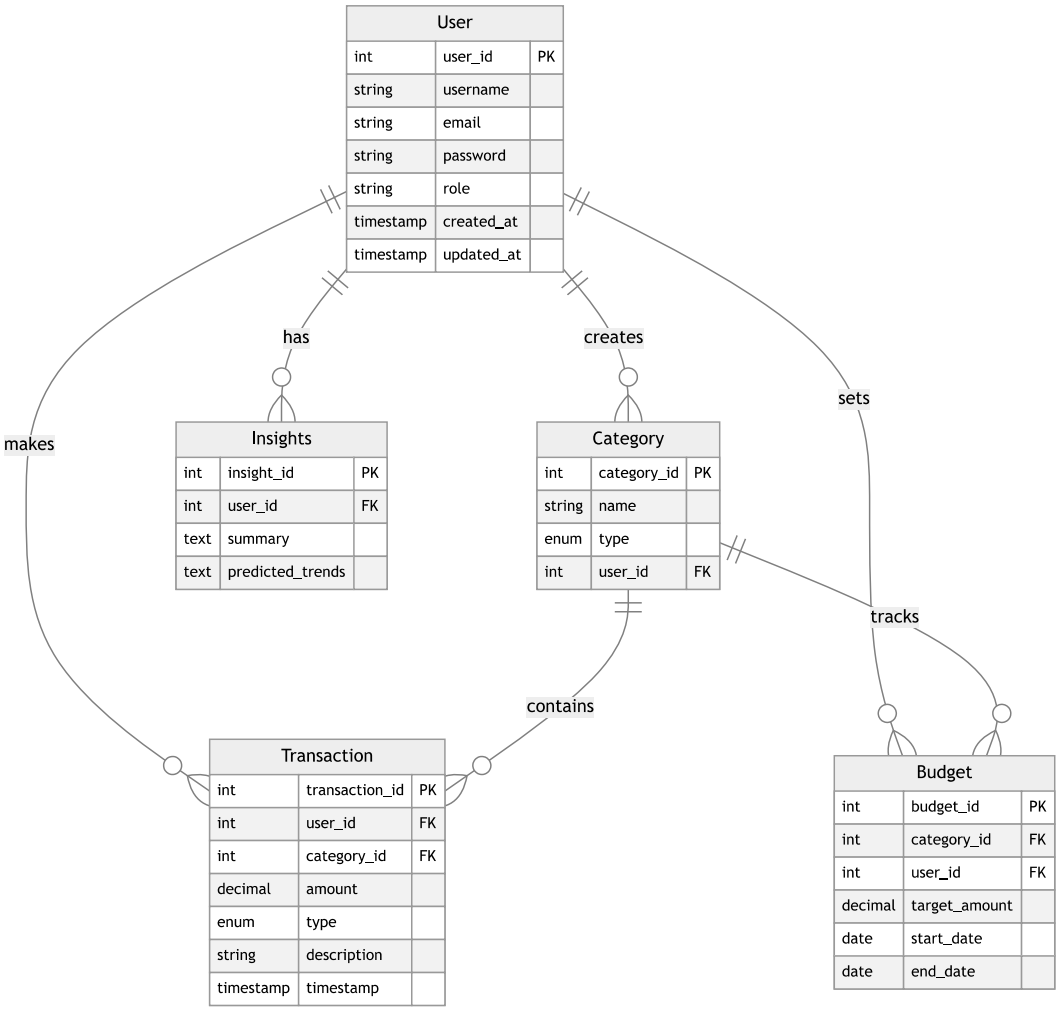
1. **User Table**
   - **Fields**: user_id (PK), username, email, password, role, created_at, updated_at.
2. **Transaction Table**
   - **Fields**: transaction_id (PK), user_id (FK), category_id (FK), amount, type, description, timestamp.
3. **Category Table**

- **Fields**: category_id (PK), name, type (income/expense), user_id (FK).

4. **Budget Table**
   - **Fields**: budget_id (PK), category_id (FK), user_id (FK), target_amount, start_date, end_date.
5. **Insight Table**
   - **Fields**: insight_id (PK), user_id (FK), summary, predicted_trends.

ER Diagram

The ER Diagram visually represents these entities and their relationships.



6. Data Flow

Data flows logically through the following steps:

1. **Input**: User enters financial data (e.g., transactions, budget goals).

2. **Processing**:
    - Transactions are categorized automatically using ML models.
    - Budgets are updated in real-time.
    - Insights are generated periodically.
3. **Storage**: Data is securely stored in relational tables.
4. **Output**: System provides analytics, reports, and reminders.

4.1. Data Description

1. Information Domain Transformation

The information domain of Financemate is transformed into data structures by:

- **Mapping system requirements** into logical data models such as entities, attributes, and relationships.
- Organizing data into normalized relational structures to ensure minimal redundancy and efficient processing.
- Using hierarchical relationships and constraints to maintain data integrity.

2. Major Data/System Entities

1. **User Entity**
    - **Description**: Represents the individuals using the system, including their roles and credentials.
    - **Storage**: Stored as records in the User table, with primary keys ensuring uniqueness.
    - **Processing**: Used for authentication, authorization, and user-specific data filtering.
2. **Category Entity**
    - **Description**: Represents categories for transactions, such as "Groceries," "Utilities," or "Salary."
    - **Storage**: Stored in the Category table, linked to a user via foreign keys.
    - **Processing**: Used for transaction classification, budget tracking, and reporting.
3. **Transaction Entity**
    - **Description**: Tracks financial inflows and outflows with details like amount, type, and category.
    - **Storage**: Stored in the Transaction table, referencing the User and Category tables.
    - **Processing**: Used for generating insights, visualizations, and reports.
4. **Budget Entity**
    - **Description**: Represents user-defined budgets for categories over specific periods.
    - **Storage**: Stored in the Budget table, linked to User and Category.
    - **Processing**: Used to monitor spending and provide alerts for budget thresholds.
5. **Insights Entity**

- **Description**: Stores analytics and machine learning predictions for user financial behavior.
- **Storage**: Stored in the Insights table, referencing the User table.
- **Processing**: Used for generating financial recommendations and summaries.

3. Data Storage Organization

- **Relational Database**: A MySQL database is used for storing data in structured tables.
- **Data Tables**:
  - User: Stores user credentials and metadata.
  - Category: Stores transaction categories.
  - Transaction: Logs all financial transactions.
  - Budget: Tracks user budgets.
  - Insights: Stores system-generated insights.
- **Normalization**:
  - Data is normalized to **3rd Normal Form (3NF)** to reduce redundancy and improve query performance.

4. Data Processing

1. **CRUD Operations**:
   - Users can create, read, update, and delete transactions, budgets, and categories.
   - Example: Adding a new transaction involves inserting a record into the Transaction table.
2. **Analytics**:
   - Machine learning algorithms process transaction data to generate insights, which are stored in the Insights table.
3. **Real-Time Updates**:
   - Real-time syncing with APIs ensures up-to-date transaction data.
4. **Reports**:
   - SQL queries aggregate and transform raw data into summaries and visualizations for users.

5. Data Storage Items

1. **Database Type**:
   - **Type**: Relational Database (MySQL or PostgreSQL).
   - **Justification**: Ensures data consistency, integrity, and efficient querying.
2. **Storage Techniques**:
   - **Encryption**: Sensitive user data (e.g., passwords) is encrypted using AES-256.
   - **Indexing**: Indexes on primary and foreign keys optimize data retrieval.
   - **Backups**: Daily automated backups stored securely in cloud storage.
3. **Data Volume**:
   - **Estimates**: Designed to handle up to 1 million transactions and 10,000 users initially.
4. **Data Security**:

- **Protocols**: TLS for secure communication, and role-based access control (RBAC) for system permissions.

4.2. Data Dictionary

Below is an alphabetical list of major system entities, their attributes, and descriptions for Financemate, including their types and methods where applicable.

1. Entities and Attributes

Entity Name	Attribute Name	Type	Description
Budget	budget_id	INT (PK)	Unique identifier for a budget.
	category_id	INT (FK)	Reference to the associated category.
	user_id	INT (FK)	Reference to the user who created the budget.
	target_amount	DECIMAL(10,2)	The target amount allocated for the budget.
	start_date	DATE	Start date for the budget period.
	end_date	DATE	End date for the budget period.
Category	category_id	INT (PK)	Unique identifier for a category.
	Name	VARCHAR(50)	Name of the category (e.g., "Groceries").
	Type	ENUM('income', 'expense')	Indicates whether the category is for income or expenses.
	user_id	INT (FK)	Reference to the user who owns the category.
Insights	insight_id	INT (PK)	Unique identifier for an insight.
	user_id	INT (FK)	Reference to the user for whom the insight is generated.

	Summary	TEXT	Summary of insights for the user.
	predicted_trends	TEXT	Predictions about user financial trends.
Transaction	transaction_id	INT (PK)	Unique identifier for a transaction.
	user_id	INT (FK)	Reference to the user who logged the transaction.
	category_id	INT (FK)	Reference to the associated category.
	Amount	DECIMAL(10,2)	The monetary value of the transaction.
	Type	ENUM('income', 'expense')	Type of transaction (income or expense).
	Description	TEXT	Details or notes about the transaction.
	Timestamp	TIMESTAMP	Date and time of the transaction.
User	user_id	INT (PK)	Unique identifier for a user.
	Username	VARCHAR(50)	The name chosen by the user.
	Email	VARCHAR(100)	Email address of the user.
	Password	VARCHAR(256)	Hashed password of the user.
	Role	VARCHAR(20)	Role of the user (e.g., admin, standard user).

2. Functions and Function Parameters

Function Name	Parameters	Return Type	Description

login()	email: VARCHAR, password: VARCHAR	BOOLEAN	Validates user credentials and starts a session.
logout()	session_id: INT	VOID	Ends the user session.
addTransaction()	user_id: INT, category_id: INT, amount: DECIMAL, type: ENUM, description: TEXT	BOOLEAN	Adds a new transaction to the system.
generateInsights()	user_id: INT	TEXT	Creates predictions and insights for the user.
createBudget()	user_id: INT, category_id: INT, target_amount: DECIMAL, start_date: DATE, end_date: DATE	BOOLEAN	Creates a budget for the user.
getReport()	user_id: INT, date_range: DATE[]	JSON	Fetches financial reports for the given date range.

updateProfile()	user_id: INT, username: VARCHAR, email: VARCHAR, password: VARCHAR	BOOLEAN	Updates the user's profile information.

Object-Oriented (OO) Description

User Class

- **Attributes**:
  - user_id: INT
  - username: VARCHAR
  - email: VARCHAR
  - password: VARCHAR
  - role: VARCHAR
- **Methods**:
  - login()
  - logout()
  - updateProfile()

Transaction Class

- **Attributes**:
  - transaction_id: INT
  - user_id: INT
  - category_id: INT
  - amount: DECIMAL
  - type: ENUM
  - description: TEXT
  - timestamp: TIMESTAMP
- **Methods**:
  - addTransaction()
  - deleteTransaction()
  - editTransaction()

Budget Class

- **Attributes**:

    - budget_id: INT
    - category_id: INT
    - user_id: INT
    - target_amount: DECIMAL
    - start_date: DATE
    - end_date: DATE
- **Methods**:
    - createBudget()
    - updateBudget()
    - deleteBudget()

5. COMPONENT DESIGN

In this section, we provide a systematic breakdown of the components and their member functions using pseudo-code. Each function corresponds to the functionalities listed in Section 3.2 of the Software Design Document (SSD).

1. User Class

Purpose: Manages user authentication, profile updates, and roles.

Attributes:

- user_id: INT
- username: VARCHAR
- email: VARCHAR
- password: VARCHAR
- role: VARCHAR

Methods:

1. **login()**
    - **Description**: Authenticates a user with their email and password.
    - **Pseudo-code**:

```
FUNCTION login(email, password):
    hashedPassword = HASH(password)
    user = QUERY "SELECT * FROM User WHERE email = email AND password =
hashedPassword"
    IF user EXISTS:
        RETURN TRUE  // Authentication successful
    ELSE:
        RETURN FALSE  // Authentication failed
```

2. **logout()**
    - **Description**: Ends a user's session.
    - **Pseudo-code**:

```
FUNCTION logout(session_id):
    DELETE session_id FROM ActiveSessions
```

```
    RETURN "Session terminated"
```

3. **updateProfile()**
    - **Description**: Updates user details like username or email.
    - **Pseudo-code**:

```
FUNCTION updateProfile(user_id, new_username, new_email):
    UPDATE User
    SET username = new_username, email = new_email
    WHERE user_id = user_id
    RETURN "Profile updated"
```

2. Transaction Class

Purpose: Handles financial transactions like income and expenses.

Attributes:

- transaction_id: INT
- user_id: INT
- category_id: INT
- amount: DECIMAL
- type: ENUM
- description: TEXT
- timestamp: TIMESTAMP

Methods:

1. **addTransaction()**
    - **Description**: Adds a new financial transaction.
    - **Pseudo-code**:

```
FUNCTION addTransaction(user_id, category_id, amount, type, description):
    INSERT INTO Transaction (user_id, category_id, amount, type, description,
timestamp)
    VALUES (user_id, category_id, amount, type, description,
CURRENT_TIMESTAMP)
    RETURN "Transaction added"
```

2. **editTransaction()**
    - **Description**: Edits an existing transaction.
    - **Pseudo-code**:

```
FUNCTION editTransaction(transaction_id, new_amount, new_description):
    UPDATE Transaction
    SET amount = new_amount, description = new_description
    WHERE transaction_id = transaction_id
    RETURN "Transaction updated"
```

3. **deleteTransaction()**
    - **Description**: Deletes a transaction.
    - **Pseudo-code**:

```
FUNCTION deleteTransaction(transaction_id):
    DELETE FROM Transaction WHERE transaction_id = transaction_id
    RETURN "Transaction deleted"
```

3. Budget Class

Purpose: Manages budget allocations and tracking.

Attributes:

- budget_id: INT
- category_id: INT
- user_id: INT
- target_amount: DECIMAL
- start_date: DATE
- end_date: DATE

Methods:

1. **createBudget()**
    - **Description**: Creates a new budget for a category.
    - **Pseudo-code**:

```
FUNCTION createBudget(user_id, category_id, target_amount, start_date, end_date):
    INSERT INTO Budget (user_id, category_id, target_amount, start_date, end_date)
    VALUES (user_id, category_id, target_amount, start_date, end_date)
    RETURN "Budget created"
```

2. **updateBudget()**
    - **Description**: Updates an existing budget.
    - **Pseudo-code**:

```
FUNCTION updateBudget(budget_id, new_target_amount, new_end_date):
    UPDATE Budget
    SET target_amount = new_target_amount, end_date = new_end_date
    WHERE budget_id = budget_id
    RETURN "Budget updated"
```

3. **deleteBudget()**
    - **Description**: Deletes a budget.
    - **Pseudo-code**:

```
FUNCTION deleteBudget(budget_id):
    DELETE FROM Budget WHERE budget_id = budget_id
    RETURN "Budget deleted"
```

4. Insights Class

Purpose: Generates and stores analytics and predictions for users.

Attributes:

- insight_id: INT
- user_id: INT
- summary: TEXT
- predicted_trends: TEXT

Methods:

1. **generateInsights()**
   - **Description**: Creates insights and trends for a user.
   - **Pseudo-code**:

```
FUNCTION generateInsights(user_id):
   transactions = QUERY "SELECT * FROM Transaction WHERE user_id =
user_id"
   patterns = ANALYZE transactions USING ML_MODEL
   summary = SUMMARIZE patterns
   INSERT INTO Insights (user_id, summary, predicted_trends)
   VALUES (user_id, summary, patterns)
   RETURN "Insights generated"
```

5. Reporting Function

Purpose: Generates downloadable financial reports.

Pseudo-code:

```
FUNCTION getReport(user_id, date_range):
   transactions = QUERY "SELECT * FROM Transaction WHERE user_id = user_id AND
timestamp BETWEEN date_range[0] AND date_range[1]"
   summary = SUMMARIZE transactions BY category, type
   report = FORMAT summary AS JSON
   RETURN report
```

6. Human Interface Design

User Interface Overview

The user interface (UI) for Financemate is designed to provide an intuitive, responsive, and user-friendly experience across both web and mobile platforms. It ensures that users can effortlessly manage financial data while maintaining clarity and accessibility.

Design Principles

1. **Simplicity:**
   - Minimalist layout with easy navigation.
   - Clear, concise labels and tooltips for all interactive elements.
2. **Consistency:**
   - Uniform design across web and mobile platforms using Flutter.
   - Adherence to Material Design guidelines for structure and styling.

3. **Accessibility:**
    - Support for high-contrast modes and screen readers.
    - Customizable font sizes and color themes to cater to users with visual impairments.

Key Features

1. **Interactive Dashboard:**
    - Graphical representation of financial metrics (e.g., bar charts, pie charts).
    - Quick access to budgeting, transaction history, and financial goals.
2. **Dynamic Navigation:**
    - Sidebars for web and bottom tabs for mobile navigation.
    - Dedicated shortcuts for frequent tasks like adding transactions.
3. **User Customization:**
    - Toggle between light and dark themes.
    - Reorder widgets on the dashboard and adjust notification preferences.

Special Interface Requirements

- **Error Feedback:** Real-time validation of user inputs with clear error messages.
- **Localization:** Multi-language support and regional financial formats.
- **Onboarding Flow:** Step-by-step tutorials for first-time users.

6.1. Overview of User Interface

The user interface of **Financemate** is designed to simplify financial management through an intuitive and responsive platform. It ensures users can efficiently track, analyze, and organize their financial activities with minimal effort.

Functionality from the User's Perspective

1. **Dashboard:**
    - **Purpose:** Provides an overview of financial health.
    - **Features:**
        - Visualizations such as charts for spending patterns and savings progress.
        - Key metrics like current balance, monthly expenses, and savings goals.
        - Quick action buttons to add transactions, set budgets, or manage accounts.
2. **Budget Management:**
    - **Purpose:** Helps users create and track budgets.
    - **Features:**
        - Allows users to define budgets by category (e.g., groceries, rent).
        - Displays progress bars for spending limits.
        - Sends alerts when nearing or exceeding budgets.
3. **Transaction Management:**
    - **Purpose:** Facilitates seamless tracking of income and expenses.
    - **Features:**

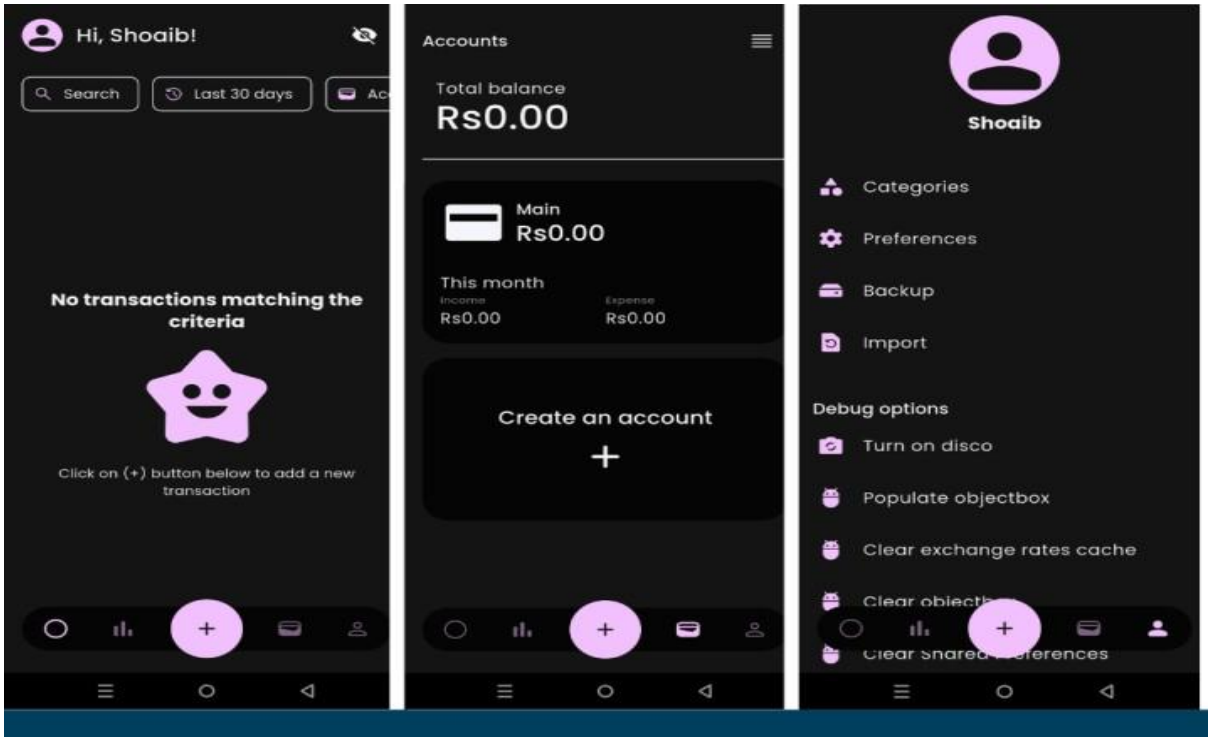
- Manual entry or automatic syncing with bank accounts.
- Categorization of transactions with suggestions using AI.
- Search and filter options for transaction history.

4. **Financial Insights:**
   - **Purpose:** Provides personalized recommendations.
   - **Features:**
     - Trends analysis (e.g., overspending warnings).
     - Predicted savings milestones based on current behavior.
5. **Notifications and Alerts:**
   - **Purpose:** Keeps users informed.
   - **Features:**
     - Alerts for bill payments, budget limits, and savings goals.
     - Real-time push notifications for critical updates.

Feedback Displayed to the User

1. **Real-Time Updates:**
   - Immediate changes to dashboards and reports upon adding, editing, or deleting transactions.
2. **Error Messages:**
   - Example: “Invalid input – please check the date format” for incorrect data entries.
3. **Confirmation Messages:**
   - Example: “Transaction successfully added” or “Budget updated.”
4. **Visual Feedback:**
   - Dynamic graphs and charts update instantly with any user action.
   - Notifications appear as badges or pop-ups to highlight key updates.

6.2. Screen Images



6.3. Screen Objects and Actions

This section outlines the primary screen objects in *Financemate* and the actions users can perform with them, ensuring an intuitive and seamless interaction experience. Below, the key screen objects are identified, and the corresponding actions users can take are depicted for better understanding.

1: Dashboard

The dashboard is the main screen after the user logs into the application, showing an overview of their financial status, recent transactions, and key features. This screen serves as the control center for all further actions.

Objects:

- **Spending Graph:** A bar or pie chart visualizing spending patterns by category.
- **Savings Progress Bar:** Tracks progress toward defined savings goals.
- **Quick Action Buttons:** Icons for adding transactions, creating budgets, and accessing reports.

Actions:

- Hover/Click on chart sections to view detailed spending data for a category.
- Click "Add Transaction" to open the transaction input form.
- Tap "View Report" to generate a detailed financial summary.

2. Accounts Management Screen

Objects:

- **Transaction List:** Displays logged transactions with columns for date, category, and amount.
- **Filter/Sort Dropdowns:** Options to filter by date, category, or amount.
- **Add Transaction Button:** Opens a form for inputting a new transaction.

Actions:

- Click "Filter" to narrow down transactions based on user preferences.
- Select a transaction to edit or delete.
- Use the search bar to locate specific transactions by keywords.

3. User Settings

Objects:

- **Toggle Options:** Switches for enabling/disabling notifications, dark mode, or syncing accounts.
- **Profile Information Fields:** Editable text fields for username, email, and password.

Actions:

- Tap "Change Theme" to toggle between light and dark modes.
- Edit profile details and click "Save" to update information.

7. REQUIREMENTS MATRIX

THE TABLE BELOW PROVIDES A CROSS-REFERENCE BETWEEN THE **FUNCTIONAL REQUIREMENTS** FROM THE SRS DOCUMENT AND THE CORRESPONDING **SYSTEM COMPONENTS** AND **DATA STRUCTURES** THAT SATISFY THEM.

SRS Functional Requirement ID	Requirement Description	System Component	Associated Data Structures
3.2.1	User Authentication(login, logout, and MFA)	User Authentication Module	User table (fields:userId,email,password)
3.2.2	Display interactive dashboard with financial metrics	Dashboard UI	Transaction, Budget tables
3.2.3	Add, edit, delete financial transactions	Transaction Management Module	Transaction table

3.2.4	Create and manage budgets	Budget Management Module	Budget table (fields: budgetId, category, limit)
3.2.5	Generate reports in PDF and Excel formats	Reporting Module	Transaction, Budget tables
3.2.6	Send alerts for overspending or nearing budget limits	Notifications Module	Notifications table (fields: alertId, type, userId)
3.2.7	Categorize transactions automatically using AI	Machine Learning Categorization Module	Transaction table (fields: category, amount)
3.2.8	Provide personalized financial insights	Insights and Analytics Module	Transaction, Budget tables
3.2.9	Sync transactions with bank accounts using APIs	API Integration Module	External API Data
3.2.10	Allow import/export of financial data in CSV and JSON formats	Data Import/Export Module	FileUpload class, Transaction table

8. APPENDICES

Appendix A: Glossary

- **SRS:** Software Requirements Specification
- **MFA:** Multi-Factor Authentication
- **UI:** User Interface
- **CRUD:** Create, Read, Update, Delete

Appendix B: References

1. IEEE Standard for Software Requirements Specification (IEEE 830-1998).
2. Flutter Documentation: https://flutter.dev/docs
3. PostgreSQL Documentation: https://www.postgresql.org/docs