# QuickPay Report 3 - Full Final Report

**Group B**

Chanrattnak Mong, Seanglong Lim, Seth Tharo Hour, Sok Sreng Chan

Fort Hays State University

2025F CSCI441 A Software Engineering

Dr. Mike Mireku Kwakye

November 23, 2025

**GitHub Repository:** https://github.com/CSCI441-QuickPay/QuickPay-MobileApp

## *Table of Contents*

*Summary of Changes*

## I. Authentication System Migration

- Change: Migrated from Firebase Authentication to Clerk Authentication
- Previous Implementation (Reports #1 & #2): System utilized Firebase Authentication for user login, registration, and session management
- Current Implementation: System now uses Clerk Authentication as the primary authentication service
- Rationale: Clerk provides more robust authentication features, superior security controls, better developer experience, and more flexible user management capabilities that better align with QuickPay's requirements for financial applications
- Impact:
    - Updated UserController and AuthController to integrate with Clerk API
    - Modified all authentication-related workflows across UC-1, UC-2, UC-3, UC-8, and UC-9
    - Changed session token validation and user credential management
    - Replaced Firebase authentication classes with Clerk integration components
    - Updated interaction diagrams and class diagrams throughout the report

## II. Database Infrastructure Overhaul

- Change: Migrated from Firebase Realtime Database to Supabase PostgreSQL
- Previous Implementation (Reports #1 & #2): User data, transactions, budgets, and persistent data stored in Firebase Realtime Database (NoSQL document-based structure)
- Current Implementation: All data storage now utilizes Supabase, a cloud-hosted PostgreSQL relational database with real-time capabilities, hosted on AWS infrastructure
- Rationale: Supabase provides:
    - Structured relational data model with ACID compliance critical for financial transactions
    - Superior query performance for complex financial analytics
    - Built-in PostgreSQL features: foreign keys, constraints, triggers, and stored procedures
    - Better integration with Clerk Authentication

    - Enhanced data integrity and consistency required for financial applications
    - Real-time subscriptions similar to Firebase but with full SQL capabilities
    - Support for complex joins, aggregations, and robust transaction management
- Impact:
    - Complete redesign of all Model classes: UserModel, BankAccountModel, TransactionModel, BudgetModel, AlertModel, GroupExpenseModel, FavoriteModel
    - Changed from document-based queries to SQL-based CRUD operations
    - Updated Domain Model to reflect relational database concepts with proper foreign key relationships
    - Modified all interaction diagrams to show SQL queries instead of Firebase document operations
    - Enhanced Data Model and Persistent Data Storage section with comprehensive SQL schema details
    - Updated class diagrams with SQL data types and PostgreSQL-specific operations

**III. Notification Service Modification**

- Change: Clarification on Cloud Messaging implementation and migration to Expo Push Notifications
- Previous Implementation (Reports #1 & #2): System referenced Firebase Cloud Messaging (FCM) as the primary notification service
- Current Implementation: System now uses Expo Push Notifications as the primary cloud messaging service for mobile app notifications
- Clarification:
    - QuickPay utilizes Expo Push Notifications, which is natively integrated with the Expo/React Native framework
    - Notifications for low balances, bill reminders, and group expense updates are delivered through Expo's push notification service
    - Expo Push Notifications provides a unified API for both iOS and Android platforms, simplifying cross-platform notification delivery
    - The system leverages Expo's notification infrastructure, eliminating the need for separate Firebase Cloud Messaging configuration

    - Backend triggers notifications via Expo's push notification API, which handles delivery to user devices
- Impact:
    - AlertController updated to integrate with Expo Push Notification API
    - Notification delivery mechanisms redesigned to use Expo's notification service
    - System architecture documentation updated to reflect Expo Push Notifications as the primary messaging provider
    - Simplified notification implementation by leveraging existing Expo framework infrastructure
    - Reduced vendor lock-in and improved compatibility with Expo-based mobile development workflow

## IV. System Architecture Refinements

- Migrated authentication from Firebase to Clerk for enhanced security
- Migrated database from Firebase Realtime Database to Supabase PostgreSQL on AWS for ACID compliance and data integrity
- Adopted Expo Push Notifications for cross-platform mobile notification delivery
- Enhanced MVC architecture documentation with detailed subsystem descriptions
- Updated hardware requirements to reflect AWS-hosted Supabase infrastructure

## V. User Case Specification Updates

- Fully elaborated all nine use cases (UC-1 through UC-9) with complete dressed descriptions
- Added comprehensive System Sequence Diagrams for each use case
- Updated authentication flows to reflect Clerk integration
- Enhanced traceability matrices mapping use cases to updated domain concepts

## VI. Interaction Diagrams Modifications

- Replaced Firebase Authentication sequences with Clerk Authentication flows
- Updated database operations from NoSQL document queries to SQL-based queries
- Modified Model interactions to show SQL CRUD operations
- Enhanced error handling for external API integrations
- Updated ExternalServiceLogModel to track Clerk and Supabase interactions

## VII. Class Diagram Revisions

- Redesigned UserModel for Supabase UID storage
- Updated AuthController for Clerk API integration
- Replaced Firebase classes with Clerk integration components
- Redesigned all Model classes with SQL data types and operations
- Added foreign key relationships and SQL transaction handling

## VIII. Data Model and Storage Updates

- Transformed from document-based to relational table structure
- Implemented foreign key relationships across all entities (Users, BankAccounts, Transactions, Budgets, Alerts, GroupExpenses, Favorites)
- Added constraints, indexes, and triggers for data integrity
- Created ExternalServiceLog table for API interaction tracking

## IX. Traceability Matrix Enhancements

- Updated all three matrices with descriptive text explanations
- Revised matrices to reflect Clerk Authentication and Supabase concepts
- Documented evolution from domain concepts to implementation classes

## X. Algorithms and Data Structures Clarifications

- Documented reliance on built-in Supabase SQL and JavaScript operations
- Clarified use of simple arithmetic for budgets and expense splitting
- Updated concurrency model for Node.js event-driven architecture

## XI. Test Design Improvements

- Developed comprehensive Jest test cases for all controllers
- Included specific inputs, expected outcomes, and pass/fail criteria
- Updated tests for Clerk and Supabase integrations
- Confirmed bottom-up integration testing strategy

## XII. External Service Integrations

- Added Clerk API and updated Supabase API references
- Created service adapters for Clerk and Supabase
- Updated ExternalServices component for multi-API coordination

## XIII. Hardware Deployment Update

- Specified Supabase PostgreSQL on AWS infrastructure
- Documented Clerk Authentication as managed service
- Enhanced server requirements for webhooks and connection pooling

## XIV. Project Objective Refinements

- Emphasized enterprise-grade security through Clerk
- Highlighted ACID compliance for financial transactions
- Added explicit transactional consistency objectives
- Emphasized scalability through optimization strategies

## XV. User Interface Updates

- Updated all pages to Version 2 design
- Integrated Clerk authentication components in UI flows
- Updated API patterns for Clerk tokens and Supabase client
- Maintained consistent React Native navigation structure

***Individual Contributions Breakdown***

| Sections | Chanrattnak Mong | Seanglong Lim | Seth Tharo Hour | Sok Sreng Chan |
|---|---|---|---|---|
| Cover Page | x | x | x | x |
| Table of Contents | | | x | |
| Summary of Changes | x | | | |
| Individual Contributions Breakdown | | | x | |
| Work Assignment | x | | x | x |
| 1a. Problem Statement | x | | | x |
| 1b. Decomposition into Sub-problems | | | | x |
| 1c. Glossary of Terms | | x | | |
| 2a. Business Goals | | x | | |
| 2b. Enumerated Functional Requirements | x | | x | |
| 2c. Enumerated Nonfunctional Requirements | x | | x | |
| 2d. User Interface Requirements | x | | | |
| 3a. Stakeholders | | | | x |
| 3b. Actors and Goals | | | | x |
| 3ci.Casual Description | | x | x | |
| 3cii. Use Case Diagram | | x | | |
| 3ciii. Traceability Matrix (1) - REQ to UC | | | x | |
| 3civ. Fully Dressed Description | | | x | |
| 3d. System Sequence Diagrams | x | | | |
| 4a. Preliminary Design | x | | | |
| 4b. User Effort Estimation | x | | | |
| 5a. Identifying Subsystems | x | x | | |
| 5b. Architecture Styles | x | x | | |
| 5c. Mapping Subsystems to Hardware | | x | | |
| 5d. Connectors and Network Protocols | | x | | x |
| 5e. Global Control Flow | | x | | |
| 5f. Hardware Requirements | | x | x | x |

| Sections | Chanrattnak Mong | Seanglong Lim | Seth Tharo Hour | Sok Sreng Chan |
|---|---|---|---|---|
| 6ai. Conceptual Model | | | | x |
| 6aii. Association definitions | | | | x |
| 6aiii. Attribute definitions | | | x | |
| 6avi. Traceability matrix (2) – UCs to DCOs | | | x | |
| 6b. System Operation Contracts | x | | x | |
| 6c. Data Model and Persistent Data Storage | | x | | |
| 7. Interaction Diagrams | x | | x | x |
| 8a. Class Diagram | x | | x | |
| 8b. Data Types and Operation Signatures | | x | | |
| 8c. Traceability Matrix (3) – DCOs to COs | | | x | |
| 9a. Algorithms | | x | | |
| 9b. Data Structures | | x | | |
| 9c. Concurrency | | x | | |
| 9d. Design Patterns | | | x | |
| 10. UI Design and Implementation | x | | | |
| 11a. Test Cases and Unit Testing | | x | x | x |
| 11b. Test Coverage | x | | | |
| 11c. Integration Testing | | x | | |
| 11d. System Testing | | | x | |
| 12a. Merging Contributions from Members | | x | | |
| 12b. Coordination and Progress Report | | | x | |
| 12c. History of Work | x | | x | x |
| 12d. Breakdown of Responsibilities | x | x | x | x |
| 12e. Current Status | | x | | |
| 12f. Future Work | x | | | |
| 13. References | | x | | |

## *Work Assignment*

**Chanrattnak Mong (Team Leader):** I am certified in full-stack development, with expertise in UI/UX design and front-end development to deliver seamless and accessible user experience. Additionally, I manage data security using Supabase and oversee CI/CD pipelines with GitLab to streamline backend development and deployment cycles. As team leader, I oversee both iOS and Android development to ensure cross-platform consistency and performance, while focusing my own development efforts on iOS. I coordinate communication, guide technical and design decisions, and ensure tasks are distributed fairly and completed on time to achieve smooth, high-quality delivery.

**Seanglong Lim:** My strengths are in backend development with Node.js and Express.js, along with database management using Supabase. I focus on designing secure and efficient database structures that support reliable system performance. By integrating Supabase with backend APIs, I help ensure seamless data flow between the server and the client side. I also troubleshoot issues related to performance and security, contributing to the overall scalability and dependability of our applications.

**Seth Tharo Hour:** I specialize in quality assurance, security testing, and deployment. My responsibilities include conducting rigorous testing to find bugs and vulnerabilities, then applying solutions to improve system reliability. I also implement security measures that help protect user data and maintain compliance with best practices. Using CI/CD pipelines, I streamline deployment, so updates and new features are rolled out smoothly without disruption. My work ensures the team's projects are both stable and secure before reaching users.

**Sok Sreng Chan:** I work primarily with React Native, the Expo Framework, and TypeScript to build responsive, cross-platform applications. My role also includes mobile optimization, ensuring the applications are intuitive and accessible on both iOS and Android devices. I contribute by developing reusable components, refining layouts, and enhancing mobile responsiveness. I also focus on performance improvements, so the user experience feels smooth and polished. By combining design and functionality, I help create mobile apps that are both engaging and reliable.

*Customer Problem Statement of Requirements*

# I. Problem Statement

## A. Fragmented Finances in Daily Life

Managing personal finances today can feel overwhelming. Most of the tools we have encountered either do not connect directly to our accounts or require manual entry for every transaction. While we seek to maintain control over our finances, we often find ourselves constantly chasing numbers.

The budgeting applications we have used fail to provide a clear view of how our funds are allocated between needs, wants, and savings. Although they offer categories, these are not always intuitive, and we cannot immediately visualize the balance of our money. Moreover, because transactions occur daily, the balances of each budget category are constantly changing, and current tools do not allow us to track these changes in real time. Consequently, we often rely on spreadsheets, spending hours entering expenses, updating balances, and reallocating funds across categories. However, life does not pause for manual updates; if we miss updating for a day or two, the figures no longer reflect our actual account balances. This discrepancy forces us to make financial decisions without full visibility, sometimes resulting in unintentional overspending.

We are not facing these challenges alone. While individual experiences may vary, a shared frustration persists: the existing tools are not designed to align with the way we live.

### 1. Emily

Emily is a thirty-four-year-old marketing coordinator living in the suburbs of a mid-sized city. She is married and has two young children, ages five and eight. Between balancing her full-time job, getting her kids ready for school, and making sure the household runs smoothly, Emily feels like she is always on the move. On the outside, her life looks stable, with a nice home in a safe neighborhood and enough money to cover her family's needs, but beneath the surface, managing her finances has become one of her greatest stressors.

Emily and her husband keep three separate bank accounts: one joint account for rent, mortgage, and utilities; a personal account for her husband's expenses; and a personal account for her own spending. In addition, they share a savings account that they try to contribute to each month for emergencies and their children's future education. While the setup was originally intended to

keep things organized, it has turned into a complicated juggling act. Emily constantly finds herself unsure which account to pull from when making purchases, and she worries about over drafting one account while another sits untouched.

Her morning routine often sets the tone for her financial stress. Before heading to work, she quickly glances at her bank apps to check balances. One account shows her deposited paycheck, but another is dangerously low because the utility bill was automatically drafted the night before. She immediately transfers money over, but she knows it will take a couple of days to show up. Until then, she carries a lingering sense of unease, wondering if her debit card might be declined for something as simple as groceries or gas.

### a. The Strain of Manual Budgeting.

Emily has tried to maintain a family budget through spreadsheets. She tracks categories like groceries, kids' extracurricular activities, and savings goals, but every transaction requires her to manually input numbers. She sets aside thirty minutes every Sunday evening to "update the budget," but with two energetic kids demanding her attention, this process rarely goes smoothly. Sometimes she forgets to log purchases for days at a time, and when she finally sits down to catch up, the numbers do not line up with what her bank statements show.

One month, she discovered that they had overspent on takeout by nearly $300 without realizing it. Because the spreadsheet was not updated in real time, Emily thought they were still within their limits, but their bank account told a different story. That mistake forced them to dip into savings just to cover rent. It left her frustrated and ashamed, as though she had failed at something she was trying so hard to manage.

She has tried a few budgeting apps, but most of them only connect partially to her accounts or fail to categorize expenses in ways that make sense to her. They also bombard her with confusing charts or ads for credit cards, which only adds to her frustration. What Emily truly wants is a simple, intuitive way to see where her money is going, without spending hours updating spreadsheets or second-guessing whether the information is current.

### b. Group Payments and Everyday Stress.

On top of her own household expenses, Emily frequently faces the headache of splitting costs with others. For example, she and a group of other parents often organize after-school activities for their children. Sometimes one parent covers the cost of snacks, tickets, or supplies, and then everyone else is supposed to pay them back. But the process rarely goes smoothly.

Last month, Emily covered the cost of admission for her daughter's entire soccer team when they went to a weekend tournament. She expected the other parents to send her their share, but it took nearly three weeks of reminders before everyone paid. In the meantime, her checking account was short by nearly $400, forcing her to put off a scheduled payment to their credit card. She hated sending reminder texts, worrying that she sounded pushy or rude, but she also could not afford to absorb the cost herself.

This is not a one-time issue; it happens almost every time she participates in group activities. Different parents prefer different payment apps, some people pay immediately while others delay, and sometimes payments get lost altogether. Each time Emily is left keeping track of who owns what, writing down names and amounts in yet another spreadsheet. What should be simple ends up feeling like unpaid part-time bookkeeping work.

### c. Transparency and Trust.

Another thing that frustrates Emily is the lack of transparency in the tools she uses. One budgeting app she tried would link to her accounts, but when she made a purchase, it did not tell her which account the money came from until after the transaction was complete. More than once, she assumed it would pull from her spending account, only to discover later that it had drawn from her savings. That kind of confusion makes her feel like she is not truly in control of her money.

She also struggles with the inability to split a single payment across multiple accounts. Sometimes she wants to balance her spending by using part of the money from her Bank of America account and the rest from her Commerce account, but current tools do not give her that flexibility. Instead, she is forced to move money around manually before making a purchase, wasting time and creating even more opportunities for mistakes. If she forgets to transfer funds, she risks over drafting one account while leaving the other untouched, which only adds to her frustration.

She has also tried apps that require her to preload money into a wallet before making payments. At first, she thought it might be helpful, but the longer she used it, the more uncomfortable she felt. She worried about security risks and what would happen if the company froze her account or was hacked. The idea of handing over her family's hard-earned money to a third party made her uneasy, and she stopped using the service after just a few weeks.

Emily needs a financial system that respects her trust. She does not want to jump through hoops or wonder where her money is stored. She wants her funds to remain safe in her own accounts, with tools that act as a guide rather than a middleman.

### d. The Emotional Burden.

For Emily, the biggest toll of all, this is not just the numbers; it is the stress. Money affects every decision she makes; from what groceries she buys to whether she signs her kids up for a summer camp. The lack of clarity around her finances weighs on her mind constantly.

Some nights, after the kids are in bed, she lies awake worrying about whether they are saving enough for emergencies, whether her accounts are balanced, or whether another surprise expense will throw everything into chaos. This constant state of worry bleeds into other parts of her life. She finds herself distracted at work, short-tempered with her kids, and anxious when talking with her husband about money. Instead of feeling empowered, she feels trapped in an endless cycle of managing and re-managing the same problems.

## II. Decomposition into Sub-problem

**Individual Customers Like Emily:**

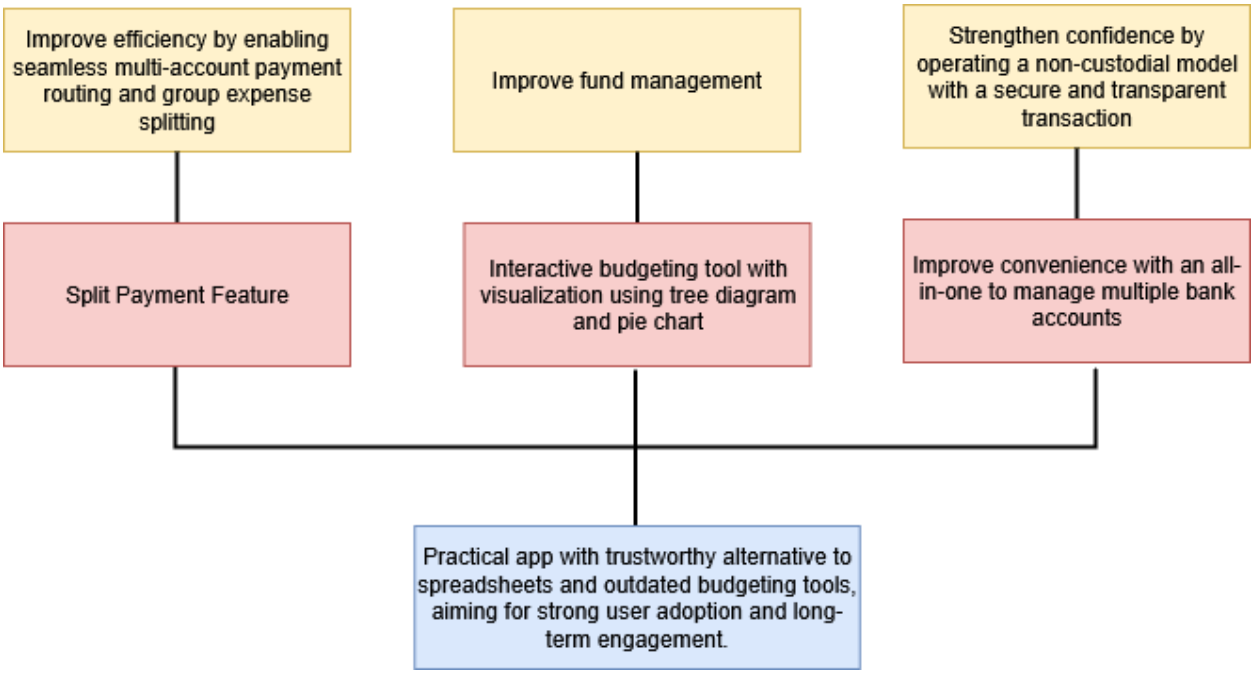
1. Customers want to see all their bank accounts in one place instead of switching between multiple apps.
2. Customers want real-time tracking of spending across categories (needs, wants, savings) to avoid overspending.
3. Customers want to reduce or eliminate the time spent manually updating spreadsheets or budgeting tools.
4. Customers want automated expense categorization so they can better understand where their money is going.
5. Customers want group payments to be easier, with clear tracking of who has paid and who still owns.
6. Customers want full transparency about which account payment is drawn from to prevent confusion.
7. Customers want the flexibility to split a single payment across multiple accounts
8. Customers want to keep control of their money without being forced to move funds into custodial wallets.
9. Customers want peace of mind and reduced financial stress through accurate, real-time insights.

## III. Glossary of Terms

- **Administrator:** An individual with the authority and responsibility to configure and manage a system.
- **API (Application Programming Interfaces):** The set of rules that enable applications to communicate with each other. In our system context, we use API keys to connect with banks and payment platforms.
- **Authentication:** The process of verifying a user's identity before granting access to the system.
- **Automatic Budgeting:** The app organizes your spending into categories (like groceries, rent, or entertainment) without you having to type everything in.
- **Bill/Expense Splitting:** A way to easily divide shared costs (like dinner with friends) so everyone pays their share without awkward reminders.
- **Budget Visualization:** A graphical representation that shows how funds are divided.
- **Dashboard:** The main interface where users can view balances, spending categories, and financial summaries briefly.
- **Expense Category:** A label used to organize spending, such as groceries, rent, dining out, or savings.
- **Instant Payment:** Sending or receiving money right away, without waiting for transfers to clear.
- **Non-Custodial:** A system design where the customer’s money always stays in their own bank accounts, never held by the app itself.
- **Transparency:** Always knowing exactly where your money is coming from and where it is going with no hidden steps or confusion.
- **Plaid API:** Securely links user bank accounts and retrieves balance and transaction.[1]
- **Google Charts API:** Generates budget graphs and spending flow visualizations. [4]

*Goals, Requirements, and Analysis - (System Requirements and Analysis)*

## I. Business Goals




With Quickpay first we want to Improve efficiency in finance management by enabling seamless multi-account payment routing and group expense splitting, improve fund management and strengthen confidence by operating a non-custodial model with a secure and transparent transaction. With these improvements we introduced to society we implemented features such as split payment features, an interactive budgeting tool and an all-in-one feature to manage multiple bank accounts. Our app QuickPay will be a trustworthy alternative to spreadsheets and outdated budgeting tools, aiming for strong user adoption and long-term engagement.

## II. Enumerated Functional Requirements

| Identifier | Priority | Requirement |
|---|---|---|
| REQ-1 | 5 | The system shall provide an interface for linking multiple bank accounts and allow users to set preferences for automatic transactions on essentials such as rent, electricity etc. |
| REQ-2 | 5 | The system shall support QR code–based payments that allow personal payments and efficiently support payment between multiple parties. |
| REQ-3 | 5 | The system shall include a visual budgeting tool that displays spending and allocations using an intuitive tree or flow-chart. |
| REQ-4 | 5 | The system shall eliminate the need for third-party fund storage by connecting directly to users' existing bank accounts, ensuring full user control over funds and providing complete transparency about which accounts are used for each transaction. |
| REQ-5 | 5 | The system shall provide a digital wallet interface that allows users to view balances across all linked accounts in one place and manage transactions from a unified dashboard. |
| REQ-6 | 4 | The system shall provide group expense management with automatic bill-splitting and payment reminders. |
| REQ-7 | 3 | The system shall support customizable alerts for low balances, unusual activity, or upcoming bills. |
| REQ-8 | 4 | The system shall let users add other users as their favorites for easy payment. |
| REQ-9 | 5 | The system shall allow users to view and update information in their profile and adjust the app setting. |

## III. Enumerated Non-functional Requirements

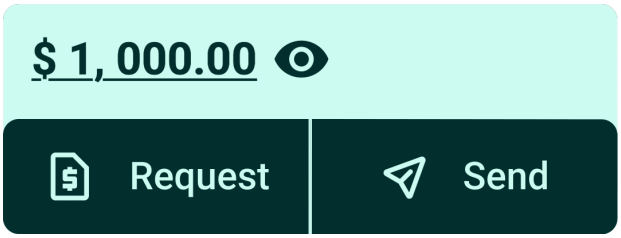
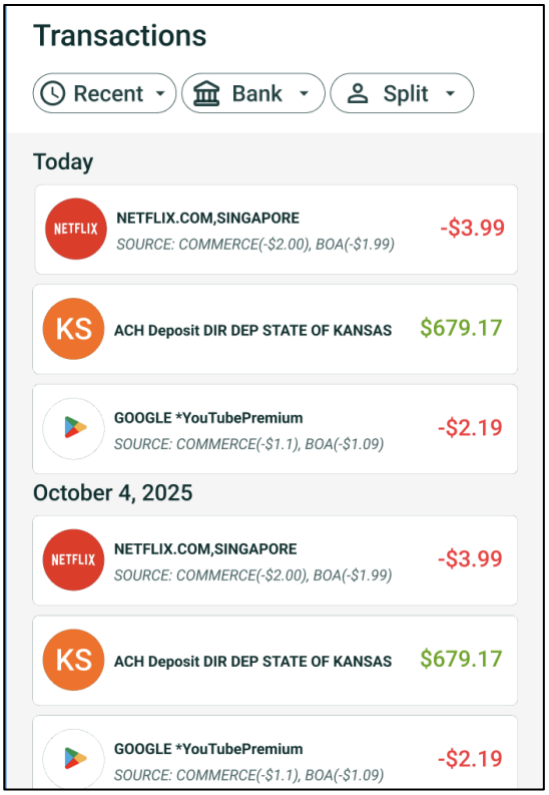
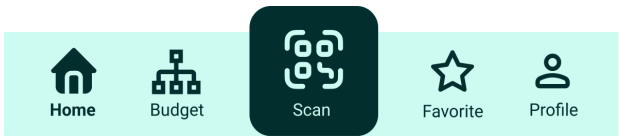
| Identifier | Priority | Requirement |
|---|---|---|
| REQ-10 | 5 | The system shall protect user data with encryption in transit (TLS 1.3) and at rest (AES-256), secure APIs (OAuth 2.0, OpenID Connect), and multi-factor authentication (MFA). |
| REQ-11 | 4 | The system shall process payments within 2–3 seconds, generate/scan QR codes in less than 1 second, and maintain a 99.95% transaction success rate. |
| REQ-12 | 5 | The system ensures 99.9% uptime with automated backup, disaster recovery (RPO < 15 min, RTO < 1 hour), and failover across multiple data centers. |
| REQ-13 | 3 | The system should provide a user-friendly interface with clear typography, readable text, and simple navigation for accessibility. |
| REQ-14 | 4 | The system shall collect only necessary transaction metadata, provide GDPR/CCPA-compliant consent, and allow users to download or delete their data. |

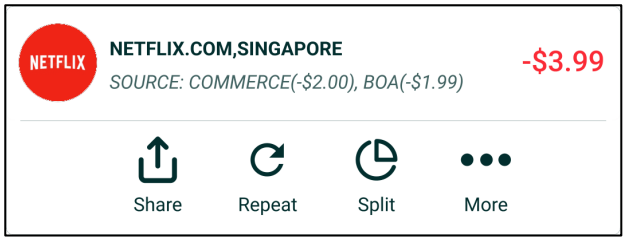
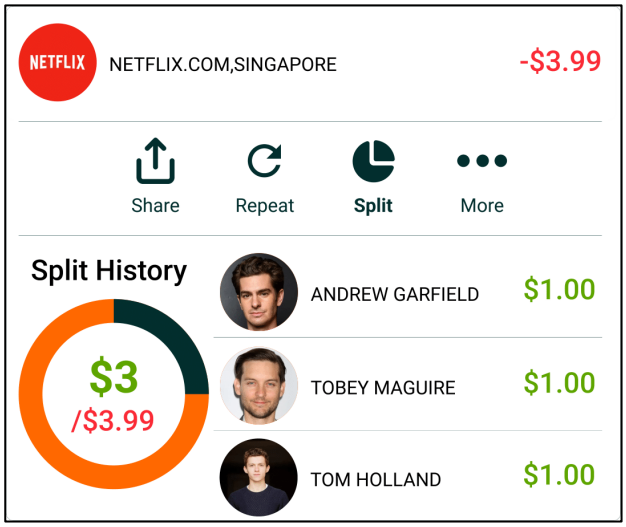
**FURPS+ Table**

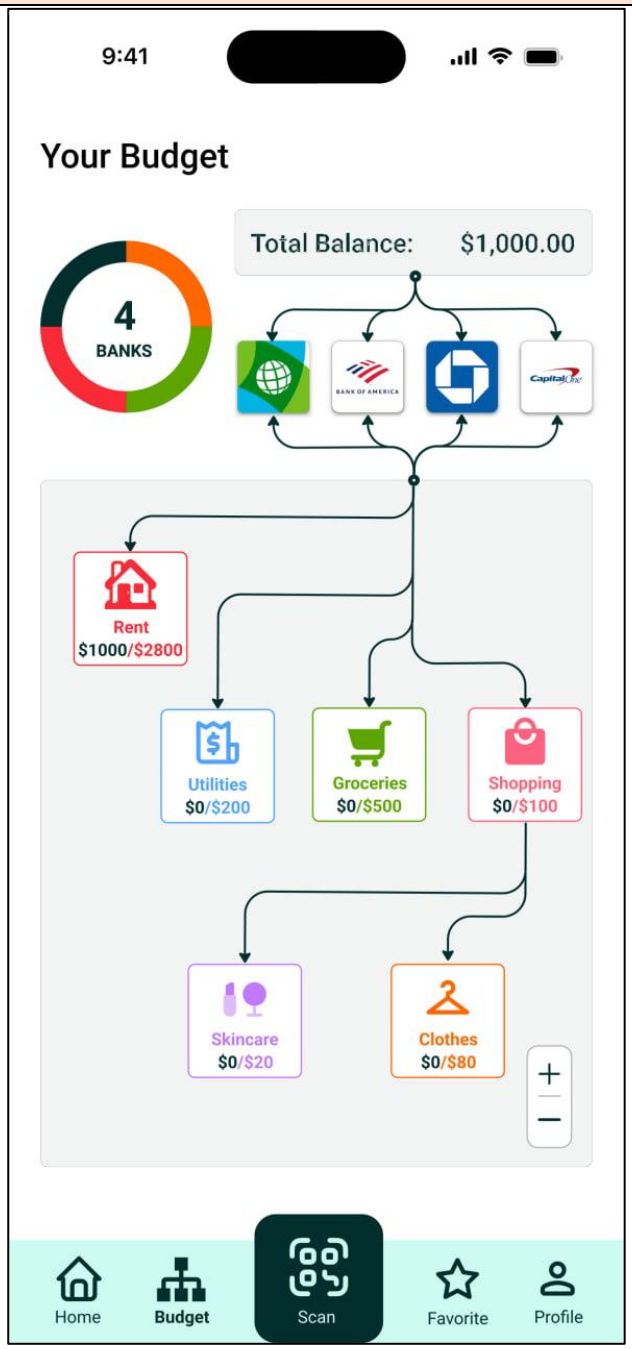
| Category | Requirement | Description |
|---|---|---|
| Functionality | REQ-1 to 9 | Core system capabilities: linking accounts, QR code payments, budgeting visualization, net worth views, non-custodial operations, bill-splitting, and alerts. |
| Usability | REQ-13 | Ensures a simple, intuitive, and easy-to-navigate interface for users. |
| Reliability | REQ-12 | High system availability with minimal downtime. |
| Performance | REQ-11 | Fast execution of payments and system actions (within seconds). |
| Security | REQ-10 | Strong authentication and protection against unauthorized access. |

| Category | Requirement | Description |
| --- | --- | --- |
| Privacy (+) | REQ-14 | Collects minimal user data, with user-controlled permissions and transparency. |
| Compliance (+) | REQ-5 | Non-custodial design ensures compliance with financial regulations. |

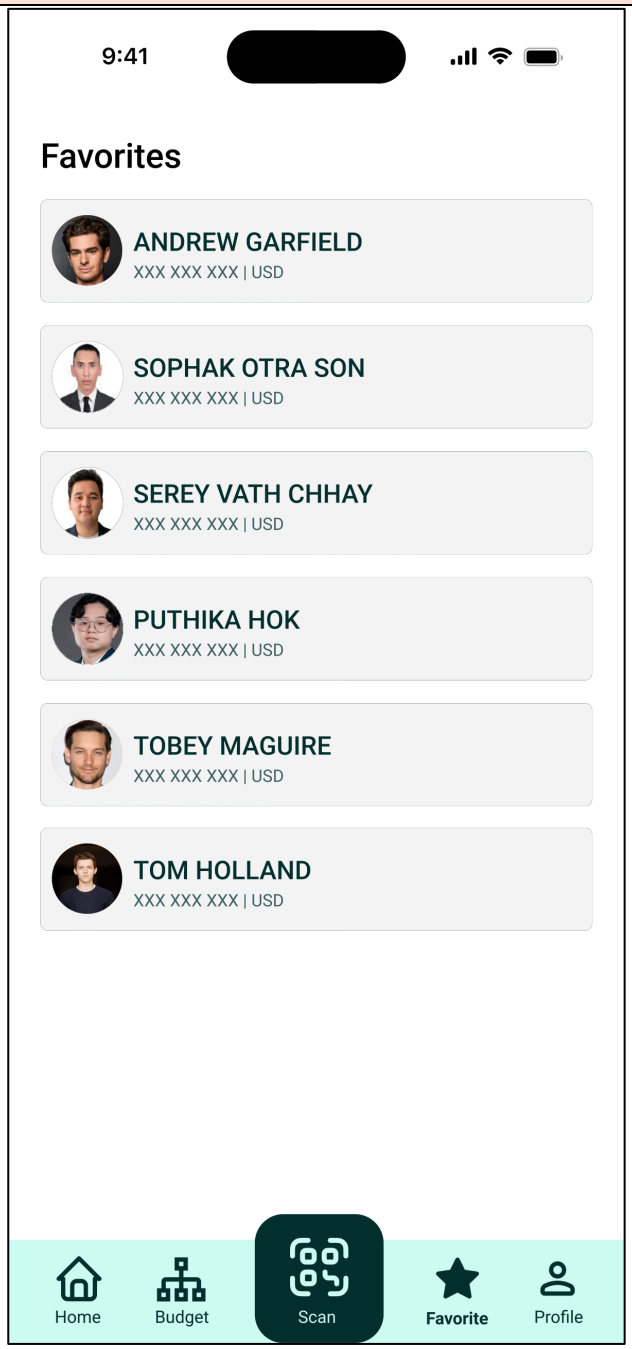
## IV. User Interface Requirements

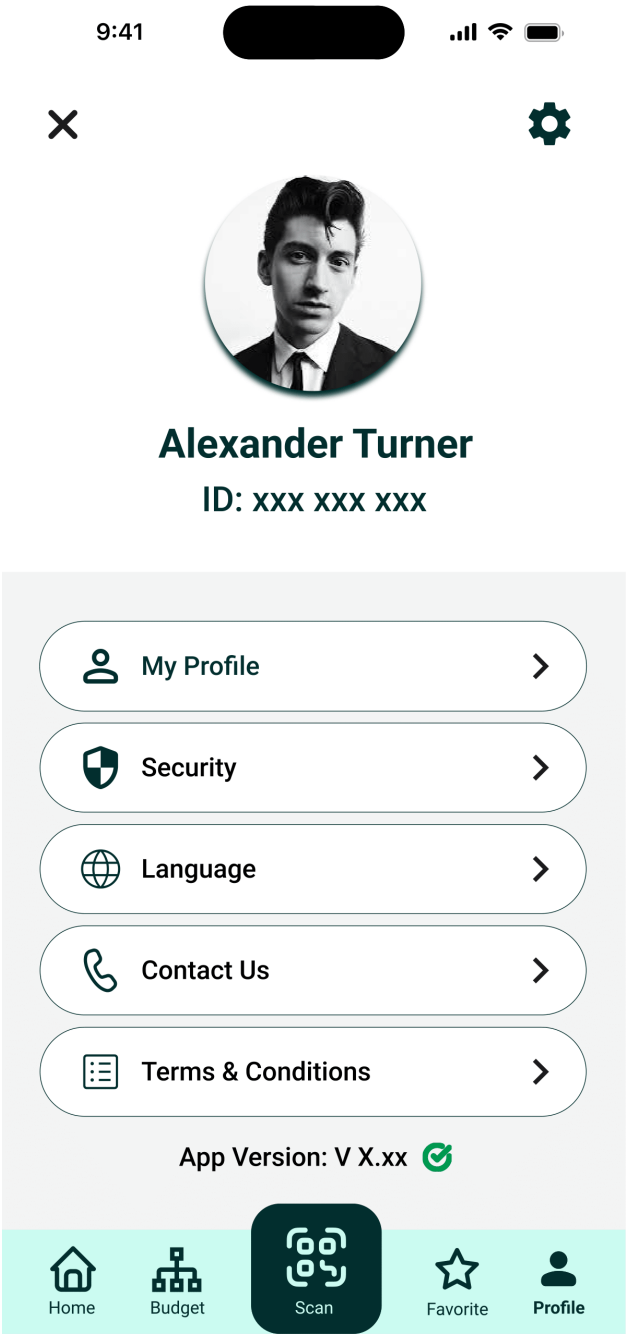
| Identifier | Priority | Requirement |
|---|---|---|
| REQ-15 | 5 | Users shall be able to see their profile and App Setting Icons   |
| REQ-16 | 5 | Users shall be able to see their bank net worth, request, and send money.   |
| REQ-17 | 5 | Users shall be able to see their transaction history and filter.   |
| REQ-18 | 5 | Users shall be able to use the navigation bar to go to different pages.   |

| Identifier | Priority | Requirement |
|---|---|---|
| REQ-19 | 5 | Users should be able to view details of their transactions with abilities to share, repeat, split, and more.<br>  |
| REQ-20 | 5 | Users shall be about to view their split payment with other people's contributions.<br>  |
| | | Users can configure budget blocks, set spending limits, and track real-time deductions from each block, with alerts when limits are exceeded. |

| Identifier | Priority | Requirement |
|---|---|---|
| REQ-21 | 5 |  |
| REQ-22 | 5 | Users can scan or upload QR code to initiate peer-to-peer payments. After scanning, they can choose the budget category and allocate deductions across customized budget blocks and select bank accounts based on predefined percentages. |

| Identifier | Priority | Requirement |
|---|---|---|
| | |  |
| REQ-23 | 5 | Users shall be able to save other users' bank account details for easier future transfers. |

| Identifier | Priority | Requirement |
| --- | --- | --- |
| | |  |

| Identifier | Priority | Requirement |
|---|---|---|
| REQ-24 | 5 | Users shall be about to see their profile, ID, check security, Language, Contact Info, and Terms & Conditions.<br>  |

*Functional Requirement Specification and Use Cases*

## I. Stakeholders

Having a convenient and secure way to manage multiple bank accounts, track budgets, and make payments can reduce financial stress and improve transparency for users. By simplifying the payment process and eliminating the need for manual calculations, QuickPay helps individuals and groups make smarter financial decisions. This not only benefits users by giving them greater control and clarity but also supports society by encouraging responsible money management and reducing errors or disputes in shared expenses. However, we have identified these primary stakeholders.

### A. Primary Stakeholders

**Individual Users:** Use the application for personal finance management. These users will benefit from linking multiple bank accounts, tracking spending through clear visualizations, and making quick QR code payments. Having transparent, real-time updates about where their money is coming from will reduce confusion and help them stay on top of their financial goals.

**Group or Shared Users:** Use the application to split expenses with roommates, friends, or family. These users will benefit by avoiding awkward manual calculations and reimbursement requests. The ability to automatically route payments from preferred accounts will make shared financial responsibilities smoother and more reliable.

**Developer:** Are dedicated to ensuring that QuickPay is reliable, secure, and user-friendly. Their focus on continuous improvement, smooth integration of new features, and regulatory compliance is critical to the success of the application. This commitment ensures that the app delivers lasting value to users and institutions alike.

### B. Secondary Stakeholders

**Financial Institutions:** While they are not directly involved in QuickPay's operations, they benefit from the app's non-custodial design, which keeps user funds within their accounts rather than requiring third-party storage. This reinforces customer trust in their services and encourages continued engagement with their financial product.

## II. Actors and Goals

| Actor | Participating/ Initiating | Role | Goal |
|---|---|---|---|
| User | **Initiating** | The user interacts with the app to conveniently manage personal finance | Link accounts, view balances, track budgets, and initiate transfers. |
| Group user | **Initiating** | The group user uses the app to split and settle share expenses | Simplify group payments and avoid manual reimbursement calculations |
| Plaid API | **Participating** | The Plaid API allows users to link their bank account | Return account, balance, and transaction data safely. |
| Supabase | **Participating** | Stores user profiles, budgeting data, and transaction history | Persist and update financial data reliably in real time. |
| Stripe Payment Service | **Participating** | Processes bank-to-bank transfers and QR-based payments. | Execute secure, compliant money transfers and deliver funds between users or merchants. |
| Expo (Framework & Deployment) | **Participating** | Provides libraries and tools for app development and deployment. | Enable fast cross-platform development and support cloud builds and updates for users. |
| Docker | **Participating** | Provides a containerized environment for development and deployment. | Ensure consistent, portable, and scalable deployment across different systems. |
| Developer | **Participating** | Design, build, and maintain the application. | Deliver a schedule, reliable, and user-friendly financial platform. |

| Actor | Participating/ Initiating | Role | Goal |
|---|---|---|---|
| Clerk API | **Participating** | Handles user authentication and identity management | Securely authenticate users via email, password and manage user sessions and access control |

## III. Use Cases

### A. Causal Description

#### 1. UC-1: Link Multiple Banks

- Description: Allows users to link multiple bank accounts
- Responds to requirements: REQ-1, REQ-4, REQ-5, REQ-7, REQ-10, REQ-12, REQ-13, REQ-16

#### 2. UC-2: Make QR code payment

- Description: Allows users to scan QR code to make payment from user to user
- Responds to requirements: REQ-2, REQ-3, REQ-4, REQ-6, REQ-11, REQ-12, REQ-13, REQ-18, REQ-20, REQ-22

#### 3. UC-3: Group Expense Splitting

- Description: Allows user to split a group expense across multiple accounts.
- Responds to requirements: REQ-2, REQ-6, REQ-11, REQ-12, REQ-13, REQ-20

#### 4. UC-4: Visualize and Manage Budget

- Description: Allow users to see spending and allocations in real time using an interactive tree/flow-chart
- Responds to requirements: REQ-3, REQ-7, REQ-11, REQ-12, REQ-13, REQ-21

#### 5. UC-5: Route Payments Non-Custodially Across Accounts

- Description: Execute payments from users' own accounts without holding funds in the app
- Responds to requirements: REQ-1, REQ-4, REQ-5, REQ-11, REQ-12, REQ-13, REQ-20

#### 6. UC-6: Receive Alerts for Low Balances or Bills (Future Work)

- Description: Provide the user with real-time alerts for important financial events
- Responds to requirements: REQ-7, REQ-12, REQ-13, REQ-24

#### 7. UC-7: View Balances & Manage via Unified Dashboard

- Description: Provide users with a single, unified dashboard where they can view balances across all linked accounts, check recent activity, and perform quick actions without switching between multiple apps.
- Responds to requirements: REQ-1, REQ-5, REQ-12, REQ-13, REQ-15, REQ-16, REQ-21

#### 8. UC-8: Add users as favorites (Future Work)

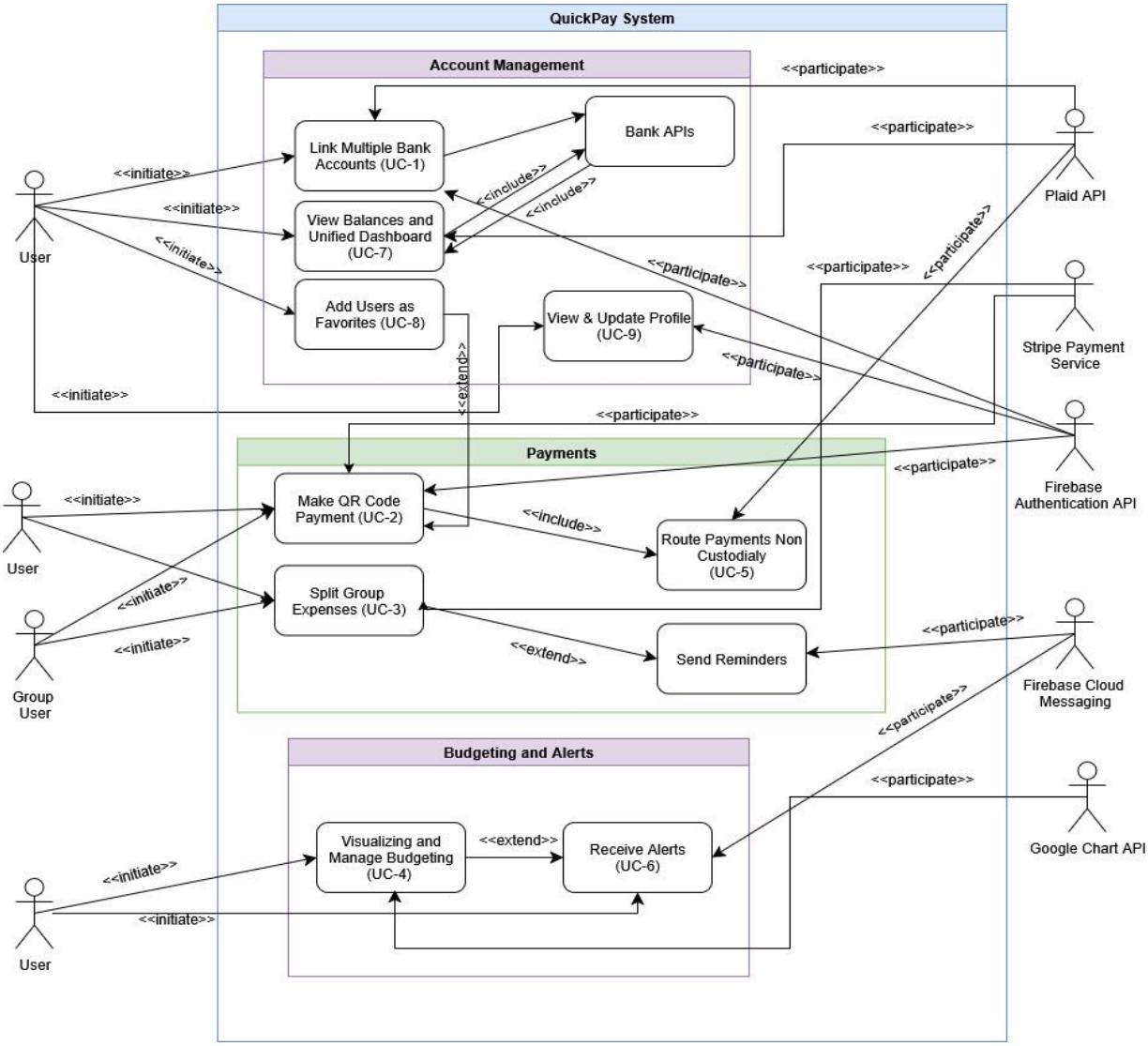
- Description: Let users add other users as their favorites for easy payment.

- Responds to requirements: REQ-8, REQ-12, REQ-13, REQ-23

### 9. UC-9: Allow users to view and update information in their profile

- Description: Allow users to view and update information in their profile and adjust the app settings.
- Responds to requirements: REQ-9, REQ-12, REQ-13, REQ-24

## B. Use Case Diagram



### C. Traceability Matrix (1) - System Requirements to Use Cases

| REQ | Power | UC-1 | UC-2 | UC-3 | UC-4 | UC-5 | UC-6 | UC-7 | UC-8 | UC-9 |
|---|---|---|---|---|---|---|---|---|---|---|
| REQ-1 | **5** | x | | | | x | | x | | |
| REQ-2 | **5** | | x | x | | | | | | |
| REQ-3 | **5** | | x | | x | | | | | |
| REQ-4 | **5** | x | x | | | x | | | | |
| REQ-5 | **5** | x | | | | x | | x | | |
| REQ-6 | **4** | | x | x | | | | | | |
| REQ-7 | **3** | x | | | x | | x | | | |
| REQ-8 | **5** | | | | | | | | x | |
| REQ-9 | **4** | | | | | | | | | x |
| REQ-10 | **5** | x | | | | | | | | |
| REQ-11 | **3** | | x | x | x | x | | | | |
| REQ-12 | **4** | x | x | x | x | x | x | x | x | x |
| REQ-13 | **5** | x | x | x | x | x | x | x | x | x |
| REQ-14 | **5** | | | | | | | | | |
| REQ-15 | **5** | | | | | | | x | | |
| REQ-16 | **5** | x | | | | | | x | | |
| REQ-17 | **5** | x | | | | | | | | |
| REQ-18 | **5** | | x | | | | | | | |
| REQ-19 | **5** | x | | x | | | | | | |
| REQ-20 | **5** | | x | x | | x | | | | |
| REQ-21 | **5** | | | | x | | | x | | |
| REQ-22 | **5** | | x | | | | | | | |
| REQ-23 | **5** | | | | | | | | x | |
| REQ-24 | **5** | | | | | | x | | | x |
| **Max Power** | | **5** | **5** | **5** | **5** | **5** | **5** | **5** | **5** | **5** |
| **Total Power** | | **47** | **46** | **31** | **25** | **32** | **17** | **34** | **19** | **18** |

Use Case 1 (Multiple Banks Linking) provides the foundation for account management, allowing users to connect all their bank accounts in one place and set up automatic payments. This use case enables the core functionality that users need to manage their finances without switching between different banking apps.

Use Cases 2 and 5 work together to handle payments, with UC2 enabling QR code scanning for quick transactions and UC5 ensuring payments come directly from user accounts without storing money in the app. These use cases address the main payment needs while keeping user funds secure in their own banks.

Use Cases 3 and 4 focus on collaboration and visualization features. UC3 allows friends and family to easily split shared expenses like restaurant bills or group activities, while UC4 provides interactive budget charts that help users see their spending patterns in real time through visual displays.

Use Cases 6 through 9 enhance the daily user experience by providing helpful notifications for low balances, a unified dashboard to view all account information, favorite contacts for quick payments, and profile management for personal settings. Together, these use cases create a complete financial management system that simplifies money handling for individuals and groups.

### D. Fully Dressed Description

**Use Case UC-1: Link Multiple Banks**

- **Related Requirements:** REQ-1, REQ-4, REQ-5, REQ-7, REQ-10, REQ-12, REQ-13, REQ-16
- **Initiating Actor:** User
- **Participating Actors:** Plaid API, Supabase, Developer
- **Preconditions:** User is authenticated via Clerk and has a stable internet connection; Plaid link flow is available and configured; consent screen and data privacy disclosures are presented
- **Postconditions:** Selected bank accounts are linked to the user profile; account metadata and balances are synced and persisted; dashboard reflects consolidated balances
- **Flow of Events of Main Success Scenario:**
  1. User opens the unified dashboard and selects Link Bank Accounts

**Use Case UC-1: Link Multiple Banks**

2. System launches Plaid link flow to fetch bank list and consent
3. User selects a financial institution and authenticates with the bank
4. System receives tokens and retrieves account, balance, and transaction metadata
5. System stores linked to account references and minimal metadata in Supabase and refreshes the dashboard

- **Flow of Events for Extensions (Alternate Scenarios):**

1a. Invalid bank credentials: System explains the mismatch and prompts retry or change

1b. User denies consent: System cancels linking and returns to dashboard without changes

2a. Plaid API unavailable: System queues a retry and notifies with a non-blocking alert

3a. Partial account selection: System links only selected accounts and records user preferences

**Use Case UC-2: Make QR code payment**

- **Related Requirements:** REQ-2, REQ-3, REQ-4, REQ-6, REQ-11, REQ-12, REQ-13, REQ-18, REQ-20, REQ-22
- **Initiating Actor:** User
- **Participating Actors:** Stripe Payment Service, Supabase, Plaid
- **Preconditions:** Both sender and recipient have QuickPay accounts; sender has at least one linked funding account; camera or QR image upload is permitted
- **Postconditions:** Payment is authorized and executed; sender transaction history updated; recipient balance state updated; receipt recorded and optionally shareable via link or QR
- **Flow of Events of Main Success Scenario:**
  1. User taps the center QR button to initiate payment
  2. System opens camera scanner and decodes recipient QR payload or accepts uploaded QR image
  3. User enters amount and selects source account or allocation preset if configured
  4. System requests payment execution through Stripe with non-custodial routing data
  5. System confirms success, records the transaction in Supabase, and shows receipt with share options

**Use Case UC-2: Make QR code payment**

- **Flow of Events for Extensions (Alternate Scenarios):**

  1a. Invalid QR payload: System shows error and allows rescanning or manual recipient selection

  2a. Insufficient funds: System suggests alternate source accounts or split across accounts if enabled

  3a. Network failure: System retrieves gracefully and informs the user of pending status

  3b. Payment timeouts: System cancels authorization and logs a non-finalized attempt

**Use Case UC-3: Group Expense Splitting**

- **Related Requirements:** REQ-2, REQ-6, REQ-11, REQ-12, REQ-13, REQ-20
- **Initiating Actor:** User Group
- **Participating Actors:** Stripe Payment Service, Supabase
- **Preconditions:** An original transaction exists, or a new expense is being created. Group participants are identified or shareable via QR link; notifications are enabled for reminders
- **Postconditions:** Split request is created with per-person amounts; contribution status is tracked; payer receives funds as participants complete payments
- **Flow of Events of Main Success Scenario:**
  1. Group user opens a past transaction or creates a new shared expense and selects Split Payment
  2. System calculates equal or custom shares and generates a QR or share link for participants
  3. Participants scan the QR or open the link and complete their share via supported payment method
  4. System updates contribution status and notifies the originator upon each payment
- **Flow of Events for Extensions (Alternate Scenarios):**

  1a. Custom share ratios: System allows manual adjustments and locks totals to the bill amount

  2a. Late or missing contributors: System schedules reminders and provides a status list to the originator

**Use Case UC-3: Group Expense Splitting**

3a. Partial cancellations: System recalculates remaining shares and updates invitations

**Use Case UC-4: Visualize and Manage Budget**

- **Related Requirements:** REQ-3, REQ-7, REQ-11, REQ-12, REQ-13, REQ-21
- **Initiating Actor:** User
- **Participating Actors:** Supabase , Google Charts API
- **Preconditions:** At least one budget block exists, or the user can create new blocks; transactions are synced for categorization; alerts are configured optionally
- **Postconditions:** Budget blocks and limits are saved; visualizations update; alerts for threshold and limit breaches
- **Flow of Events of Main Success Scenario:**
  1. User opens the interactive budgeting page from the navigation bar
  2. System displays total balance, account contributions, and existing budget blocks with limits
  3. User creates or edits blocks, sets limits, and enables alerts for low-balance or overspend events
  4. System stores the configuration, and updates live charts and deduction logic for future transactions
- **Flow of Events for Extensions (Alternate Scenarios):**

  1a. Reallocate funds: User drags amounts between blocks; system confirms and updates histories

  2a. Category misclassification: User corrects category, and system learns or applies rules for future items

  3a. Alert tuning: User adjusts thresholds or snoozes notifications for a defined period

**Use Case UC-5: Route Payments Non-Custodially Across Accounts**

- **Related Requirements:** REQ-1, REQ-4, REQ-5, REQ-11, REQ-12, REQ-13, REQ-20
- **Initiating Actor:** User
- **Participating Actors:** Plaid API, Stripe Payment Service, Supabase Database

**Use Case UC-5: Route Payments Non-Custodially Across Accounts**

- **Preconditions:** At least one funding account is linked; allocation presets or percentages may be configured; compliance and consent are satisfied
- **Postconditions:** Payment is executed directly from user accounts without app custody; routing metadata and receipts are recorded; dashboard reflects updated balances
- **Flow of Events of Main Success Scenario:**
  1. User initiates payment and selects Multi-account Allocation
  2. System displays available accounts and saves percentage presets for routing
  3. User confirms percentages or overrides them, then submits for execution
  4. System process payment requests per allocation, receives confirmations, and composes a unified receipt
- **Flow of Events for Extensions (Alternate Scenarios):**

  1a. One account fails: System retries or redistributes remaining amount per user approval before finalizing

  2a. Allocation exceeds limit: System suggests a nearest valid allocation within balance and policy constraints

  3a. Compliance hold: System blocks the payment and provides guidance on remediation steps

**Use Case UC-6: Receive Alerts for Low Balances or Bills**

- **Related Requirements:** REQ-1, REQ-4, REQ-5, REQ-11, REQ-12, REQ-13, REQ-20
- **Initiating Actor:** User
- **Participating Actors:** Supabase Database, Plaid API
- **Preconditions:** Alerts are enabled with thresholds or schedules; devices have notification permissions
- **Postconditions:** User receives push notifications for configured triggers; alert history is retained; recommended actions may be presented
- **Flow of Events of Main Success Scenario:**
  1. User configures alerts for low balance, unusual activity, and upcoming bills in settings
  2. System monitors linked accounts and budget blocks for threshold crossings and due dates

**Use Case UC-6: Receive Alerts for Low Balances or Bills**

3. System sends push notifications with context and quick actions such as transfer or snooze

- **Flow of Events for Extensions (Alternate Scenarios):**

  1a. False positive alerts: User marks as not useful and the system tunes sensitivity where applicable

  2a. Device offline: System queues for notifications and delivers when connectivity is restored

**Use Case UC-7: View Balances & Manage via Unified Dashboard**

- **Related Requirements:** REQ-1, REQ-5, REQ-12, REQ-13, REQ-15, REQ-16, REQ-21
- **Initiating Actor:** User
- **Participating Actors:** Supabase Database, Plaid API
- **Preconditions:** At least one account is linked; the user is authenticated; recent data sync is available
- **Postconditions:** The dashboard shows real-time balances, recent transactions, and quick actions
- **Flow of Events of Main Success Scenario:**
  1. User opens the home page to view current consolidated balance and recent transactions with filters
  2. System presents quick actions for Send, Request, and transaction sharing or splitting
  3. User drills into a transaction to view details, repeat a payment, or generate a share link
- **Flow of Events for Extensions (Alternate Scenarios):**

  1a. Hidden balance mode: System masks amount until toggled visible

  2a. Bank-specific filter: System shows only selected institution transactions and updates totals

**Use Case UC-8: Add users as favorites**

- **Related Requirements:** REQ-8, REQ-12, REQ-13, REQ-23
- **Initiating Actor:** User
- **Participating Actors: Supabase** Database

**Use Case UC-8: Add users as favorites**

- **Preconditions:** The recipient has a QuickPay account or shareable identifier; user has permission to store contact metadata
- **Postconditions:** Favorite is created with saved identifiers and optional profile image; future payments can target favorites quickly
- **Flow of Events of Main Success Scenario:**
  1. User opens Favorites and selects Add Favorite
  2. System prompts for recipient details or imports from a completed transaction
  3. User saves the favorite and optionally pins it for quick access on the Send flow
- **Flow of Events for Extensions (Alternate Scenarios):**

  1a. Duplicate favorite: System alerts and offers to merge or keep separate with a label

  2a. Recipient updates account: System notifies on next payment attempt and requests confirmation

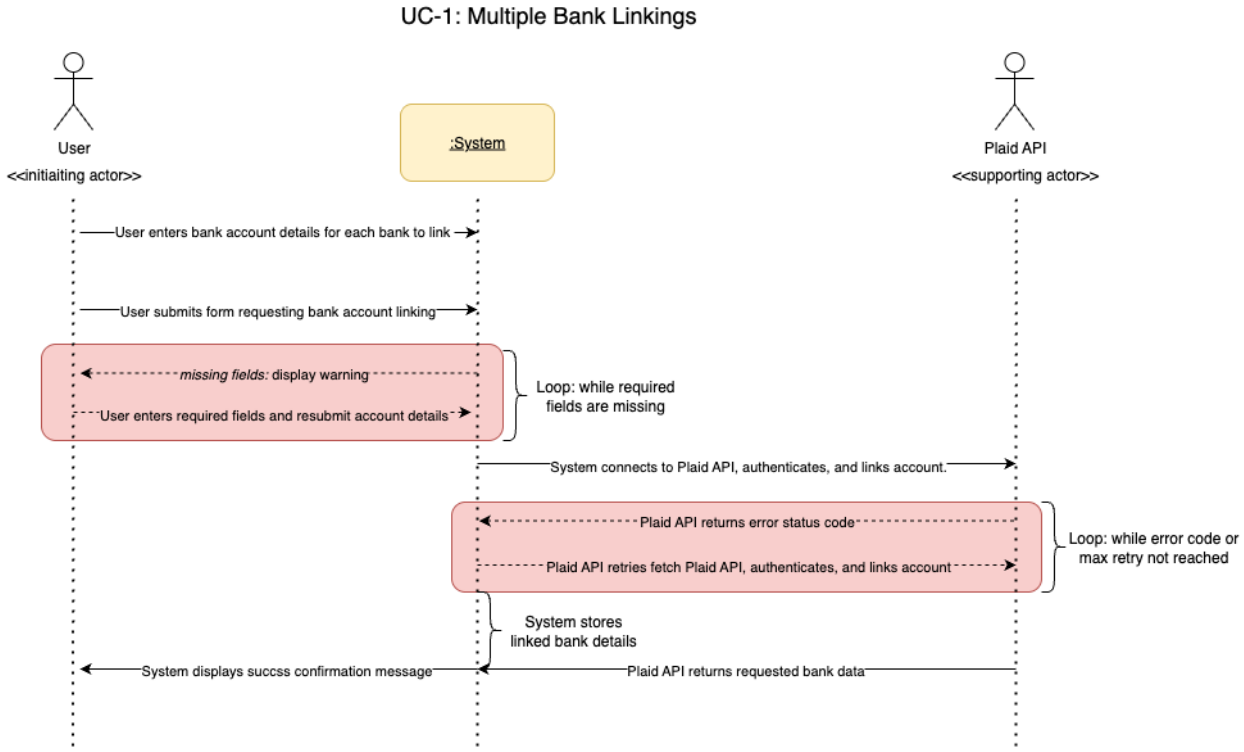
**Use Case UC-9: Allow users to view and update information in their profile**

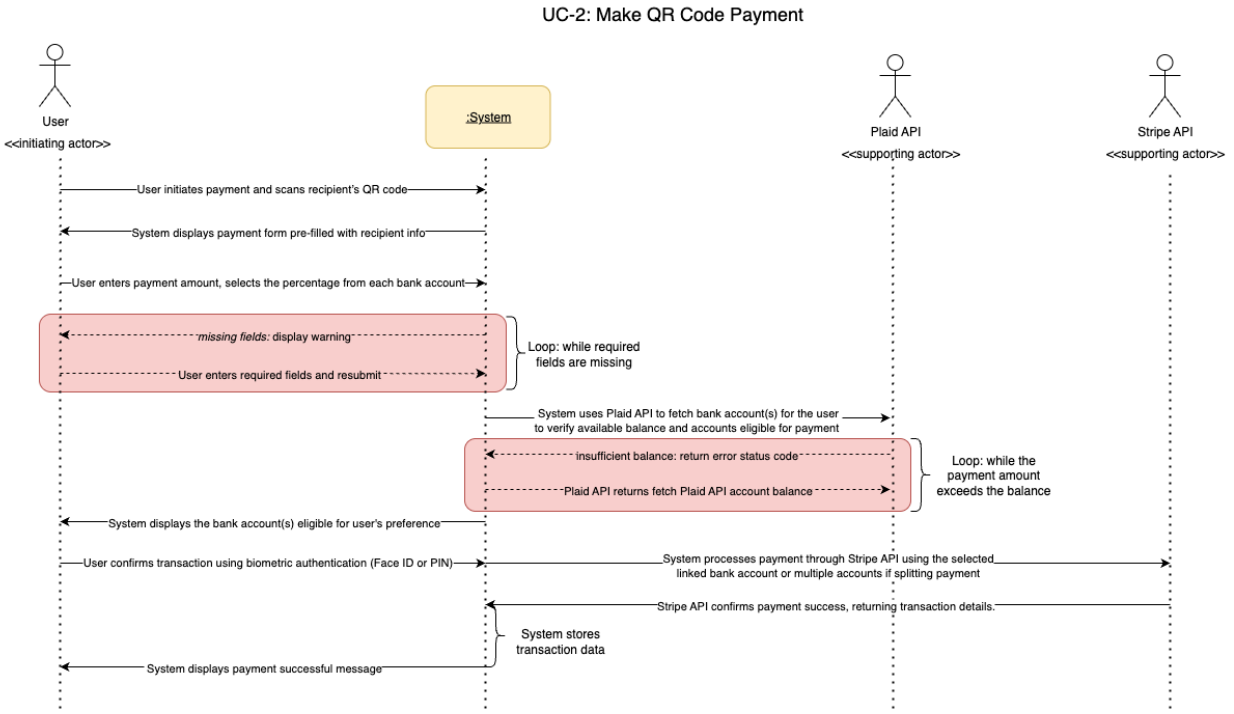
- **Related Requirements:** REQ-9, REQ-12, REQ-13, REQ-24
- **Initiating Actor:** User
- **Participating Actors:** Clerk, Supabase
- **Preconditions:** User is authenticated; profile page is accessible; permissions for MFA and language settings are available
- **Postconditions:** Profile details, security settings, language, and terms acceptance state are saved; changes propagate to other features
- **Flow of Events of Main Success Scenario:**
  1. User opens Profile and edits contact information, security options, and preferences
  2. System validates input, updates records, and refreshes the UI state accordingly
  3. User reviews the updated profile and returns to the dashboard
- **Flow of Events for Extensions (Alternate Scenarios):**

  1a. Weak password or missing MFA: System prompts to strengthen and enable MFA before saving

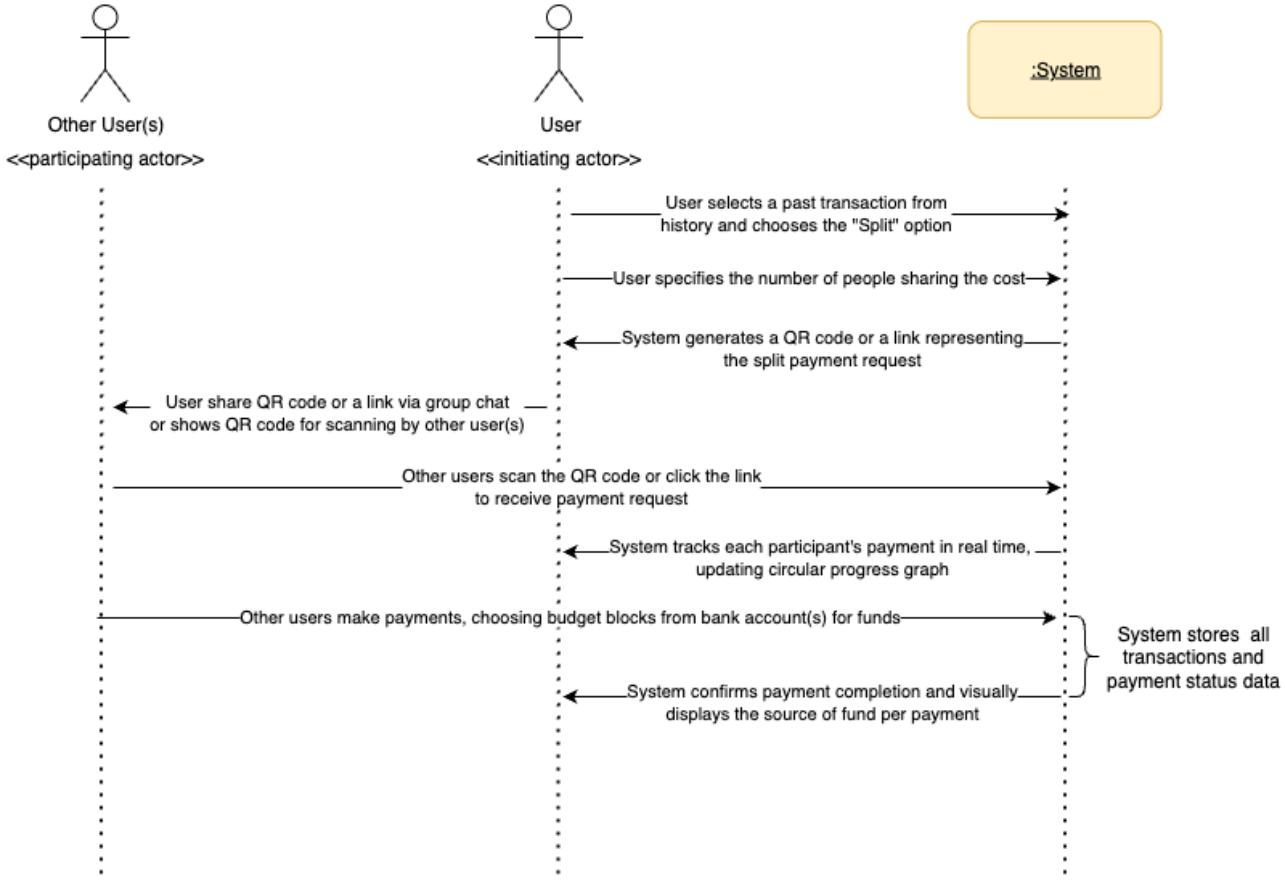
| **Use Case UC-9: Allow users to view and update information in their profile** |
| --- |
| 2a. Data download or delete request: System initiates GDPR or CCPA flow per privacy requirements |

# IV. System Sequence Diagrams

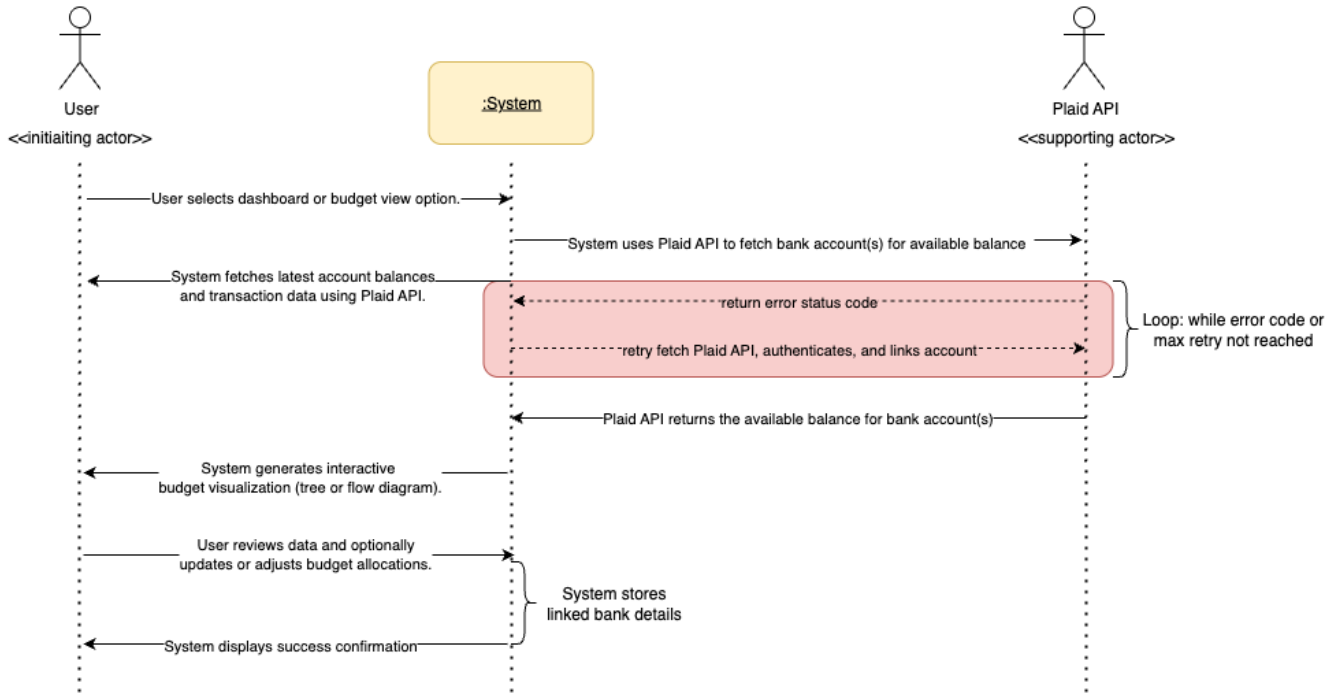


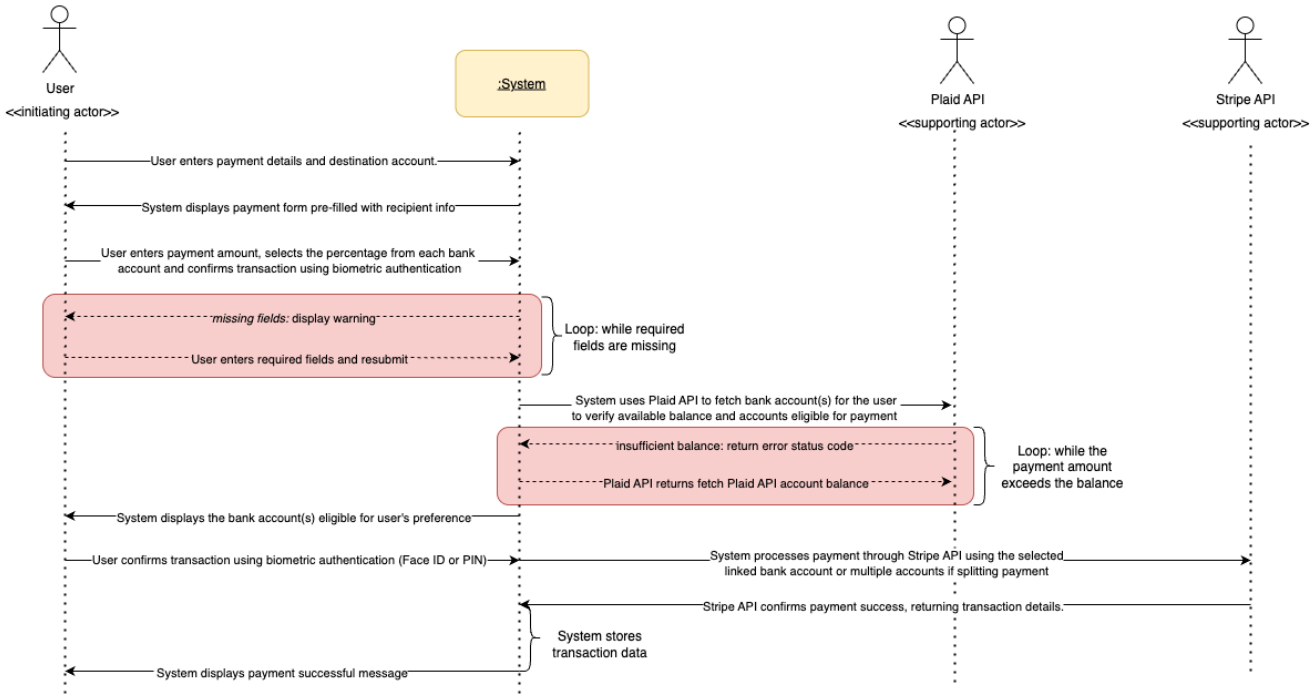
# UC-3: Group Expense Splitting



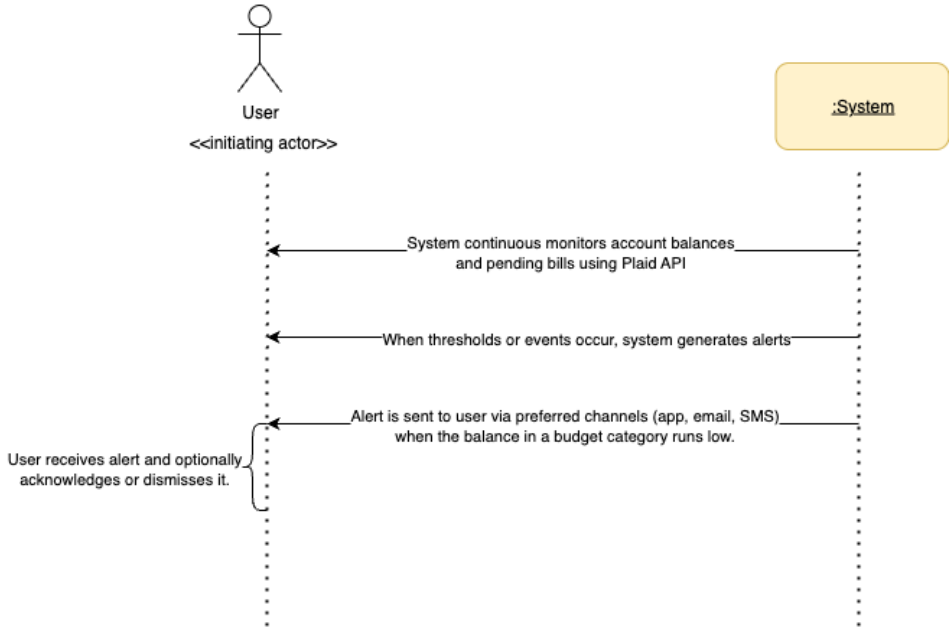
## UC-4: Visualize and Manage Budget



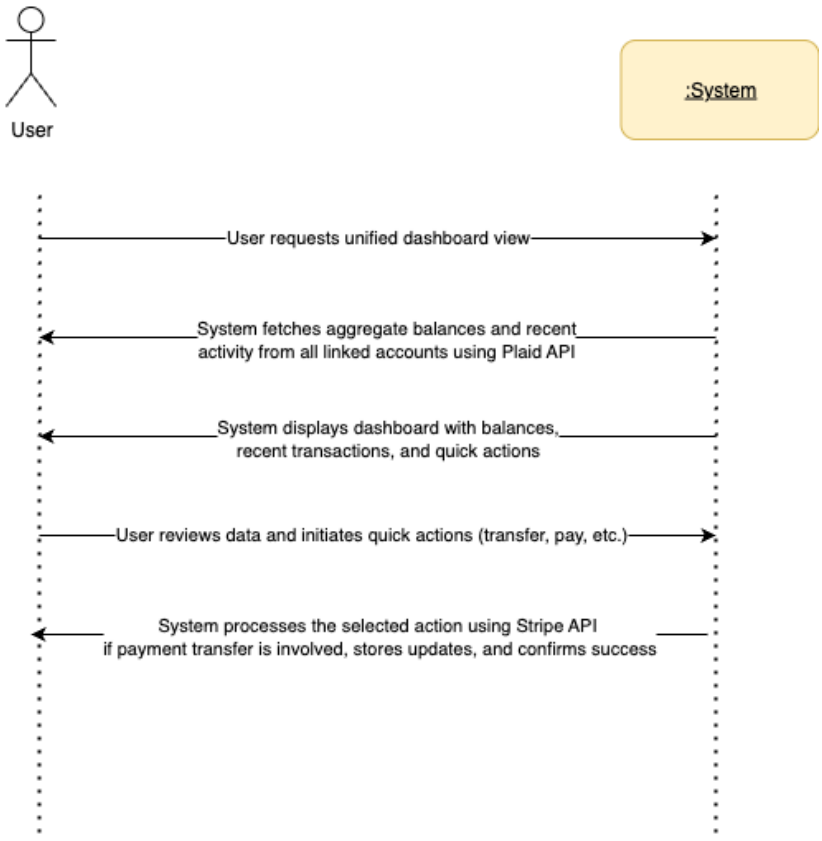

## UC-5: Route Payments Non-Custodially Across Accounts

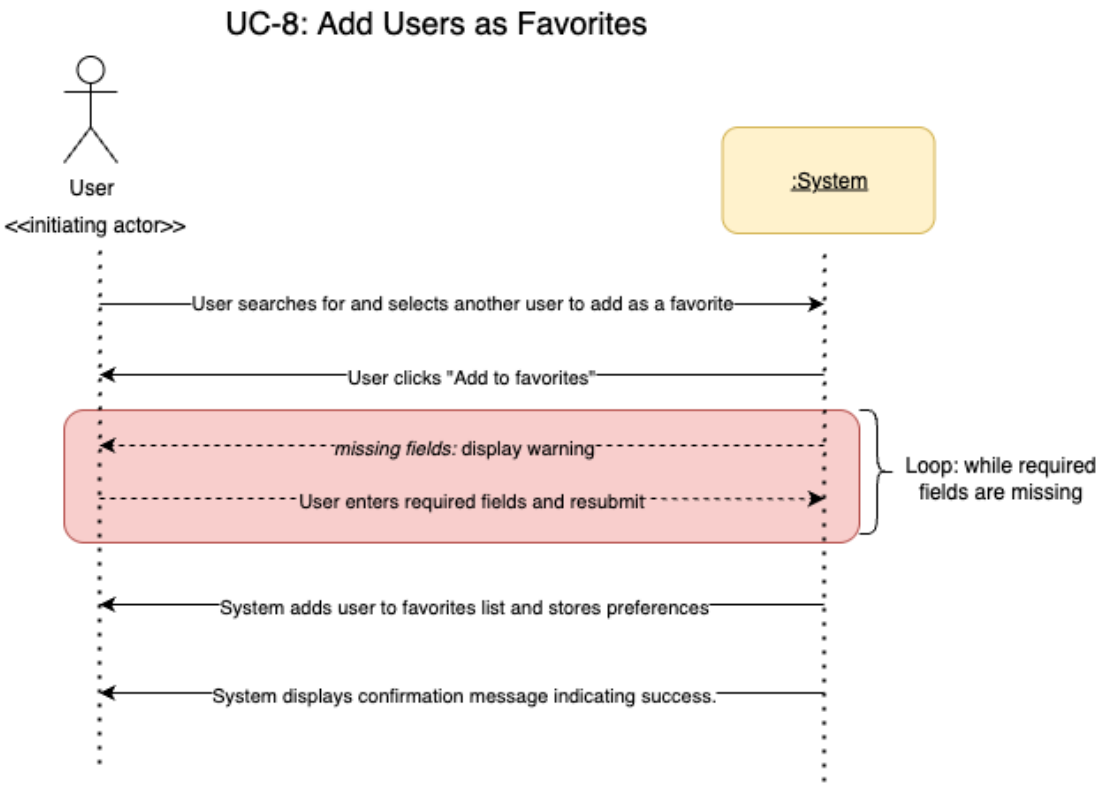


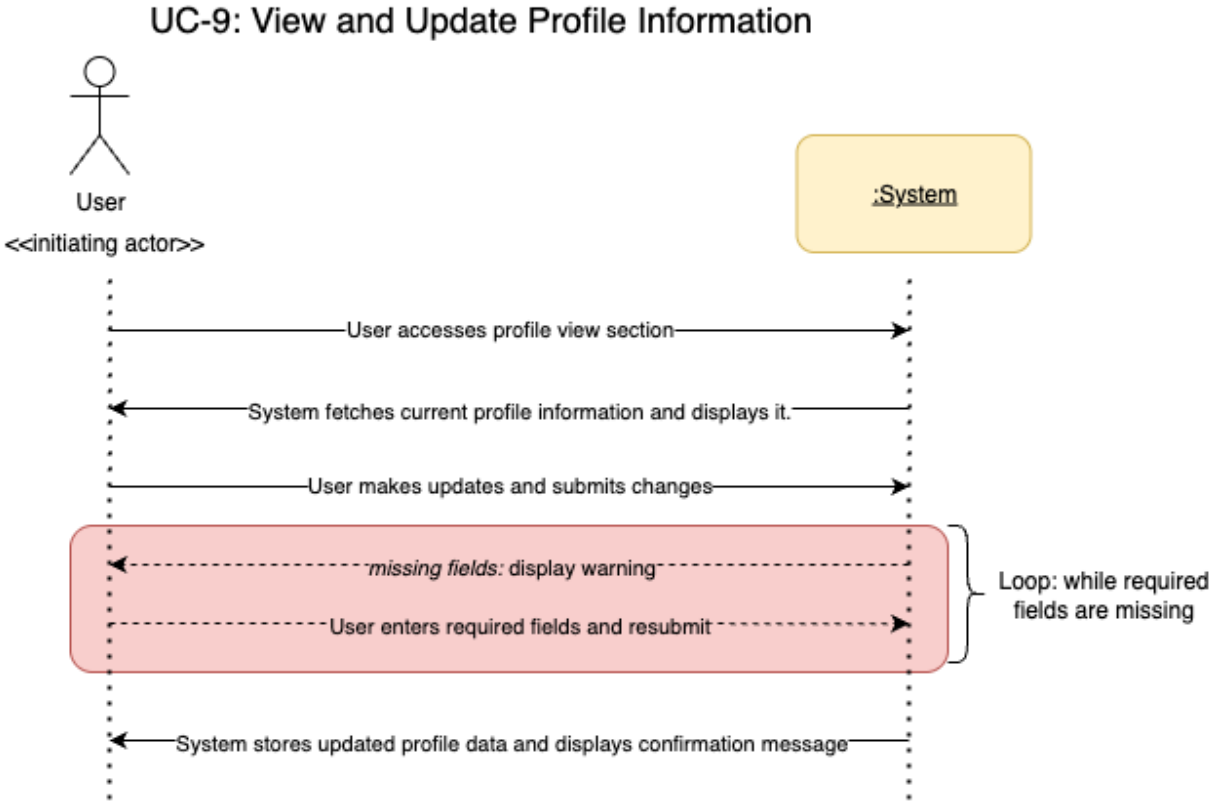
## UC-6: Receive Alerts for Low Balances or Bills




## UC-7: View Balances & Manage via Unified Dashboard




UC-8: Add Users as Favorites
User
<<initiating actor>>
:System
User searches for and selects another user to add as a favorite
User clicks "Add to favorites"
missing fields: display warning
User enters required fields and resubmit
Loop: while required fields are missing
System adds user to favorites list and stores preferences
System displays confirmation message indicating success.


UC-9: View and Update Profile Information
User
<<initiating actor>>
:System
User accesses profile view section
System fetches current profile information and displays it.
User makes updates and submits changes
missing fields: display warning
User enters required fields and resubmit
Loop: while required fields are missing
System stores updated profile data and displays confirmation message

# I. Preliminary Design

## A. Navigation Bar

1.1. The **first button in the navigation bar**, represented by a home icon, serves as the default page where users can view their current balance and recent transactions.

1.2. The **second button**, **with a** Flowchart icon, opens the interactive budget view, allowing users to see all linked bank accounts and their real-time financial standing.

1.3. The **center dark teal QR code button** in the navigation bar provides a quick shortcut for instant payment scanning.

1.4. The **fourth button**, with a **Star icon**, displays the user's saved favorite accounts for faster transactions.

1.5. The **fifth button**, with a **Person icon**, opens the user profile and general settings.

## B. Home Page

1.6. Users can view their verified profile, which they may choose to share with others when transferring money. Unverified users will not be able to view other users' profiles during transactions.

1.7. The profile and settings icons at the top-right corner allow users to update their profile, configure security settings, change the language, or review the terms & conditions.

1.8. Users can see their current balance across all connected bank accounts in QuickPay, with the option to show or hide the balance.

1.9. The "Request" button lets users receive money by sharing either a QR code or a payment link.

1.10. The "Send" button allows users to transfer money to another QuickPay user by entering their account number manually or scanning a QR code.

1.11. Transactions Section:

1.12. Users can filter recent transactions across all linked bank accounts, view transactions from a specific bank, or isolate shared (split) payments and personal transactions.

1.13. Incoming transactions are displayed as green positive values, while outgoing transactions are shown as red negative values on the right side.

1.14. Each transaction can be expanded to access a Share button, allowing users to generate a link or QR code to request a split payment from multiple users or share the transaction receipt.

1.15. Users can **repeat a transaction** to send a payment to the same recipient.

1.16. The pie chart icon provides access to the split payment history for all payments received from any account where a split payment was requested.

1.17. The three-dot icon displays detailed information about a transaction, including the exact date and other relevant details.

### C. Interactive Budgeting Page

1.18. Users can visualize their financial overview through a pie chart displaying contributions from each connected bank account, view their total balance across all accounts, and see which banks are linked to their QuickPay balance.

1.19. Users can access an interactive "playground" interface to manage their budget blocks, which branch from their main balance and are customized with names reflecting their spending categories (wants, needs, general expenses, etc.).

1.20. Users can monitor fund allocation across budget blocks to track their financial position and transfer money between blocks, with real-time deduction of payments from the appropriate budget categories.

1.21. Users can expand individual budget blocks for detailed views, including allocation breakdowns, spending limits, and transaction histories specific to each budget category.

### D. QR Code Scanning Page

1.22. Users can activate their mobile camera and position the QR code within the designated scanning frame.

1.23. Users can turn on the flash function to improve QR code capture in low-light conditions under the scanning frame.

1.24. Users can upload a QR code image from their device using the upload button located next to the flashlight control.

### E. Favorites Page

1.25. Users can view all their saved accounts for faster transactions, with profile images displayed for contacts who have chosen to share their photos.

### F. Profile Page

1.26. Users can view their profile picture along with a list of options for general and security settings. Contact and Terms and Conditions buttons are also available.

## II. User Effort Estimation

### A. User Scenario 1: Effortless Multi-Account Payments with Smart Allocation

#### 1. Task A: Setting Up Automatic Rent Payment with Custom Allocation

**Steps:**

1. Navigate to home page (tap: 1)
2. Tap "Send" button (tap: 1)
3. Enter recipient details or scan QR (keystrokes: ~15 or taps: 3)
4. Enter payment amount (keystrokes: ~6)
5. Access multi-account allocation settings (tap: 2)
6. Set percentage for Account 1 (keystrokes: ~3)
7. Set percentage for Account 2 (keystrokes: ~3)
8. Set percentage for Account 3 (keystrokes: ~3)
9. Enable recurring payment (tap: 1)
10. Set payment schedule/date (taps: 3)

Confirm and save (tap: 1)

**Total Efforts:** 12 taps + 30 keystrokes = 42 actions

**Navigation:** 12 actions (29%)

**Data Entry:** 30 actions (71%)

#### 2. Task B: Viewing Payment Status and History

**Steps:**

1. Open app to home page (tap: 1)
2. Scroll to transactions section (swipe: 1)
3. Tap on specific transaction (tap: 1)
4. View allocation breakdown (tap: 1)

**Total Efforts:** 3 taps + 1 swipe = 4 actions

**Navigation:** 3 taps + 1 swipe = 4 actions (100%)

**Data Entry:** 0 action (0%)

### B. User Scenario 2: Interactive Visual Budgeting for Smart Spending

#### 3. Task A: Setting Up Budget Category and Limits (Single Category)

**Steps:**

1. Tap flowchart icon in navigation (tap: 1)
2. Create "Shopping" budget block (tap: 2)
3. Name the category (keystrokes: ~8)
4. Set budget limit (keystrokes: ~6)
5. Save configuration (tap: 1)
6. View allocation breakdown (tap: 1)

**Total Efforts:** 4 taps + 14 keystrokes = 18 actions

**Navigation:** 4 actions (22%)

**Data Entry:** 14 actions (78%)

#### 4. Task B: Monitoring and Reallocating Funds

**Steps:**

1. Tap flowchart icon (tap: 1)
2. Tap on budget block to expand (tap: 1)
3. View current allocation and spending (no action)
4. Drag funds between blocks (drag gestures: 3)
5. Confirm reallocation (tap: 1)

**Total Effort:** 3 taps + 3 drags = 6 actions

**Navigation:** 6 actions (100%)

**Data Entry:** 0 actions (0%)

### C. User Scenario 3: Instant Bill Splitting with QR Payments

#### 5. Task A: Creating a Split Payment from Transaction History

**Steps:**

1. Open app to home page (tap: 1)
2. Scroll to transactions section (swipe: 1)
3. Tap on grocery transaction (tap: 1)
4. Tap "Share" button (tap: 1)
5. Select "Split Payment" option (tap: 1)
6. Enter number of people sharing (keystrokes: ~1)
7. Generate QR code (tap: 1)
8. Share QR code via group chat (tap: 2)

**Total Effort:** 7 taps + 1 swipe + 1 keystroke = 9 actions

**Navigation:** 8 actions (89%)

**Data Entry:** 1 action (11%)

### 6. Task B: Making Direct Peer-to-Peer Payment

**Steps:**

1. Tap center QR button or "Send" (tap: 1)
2. Scan friend's QR code or select from favorites (tap: 1)
3. Enter amount (keystrokes: ~5)
4. Select source account(s) (tap: 2)
5. Confirm payment (tap: 1)

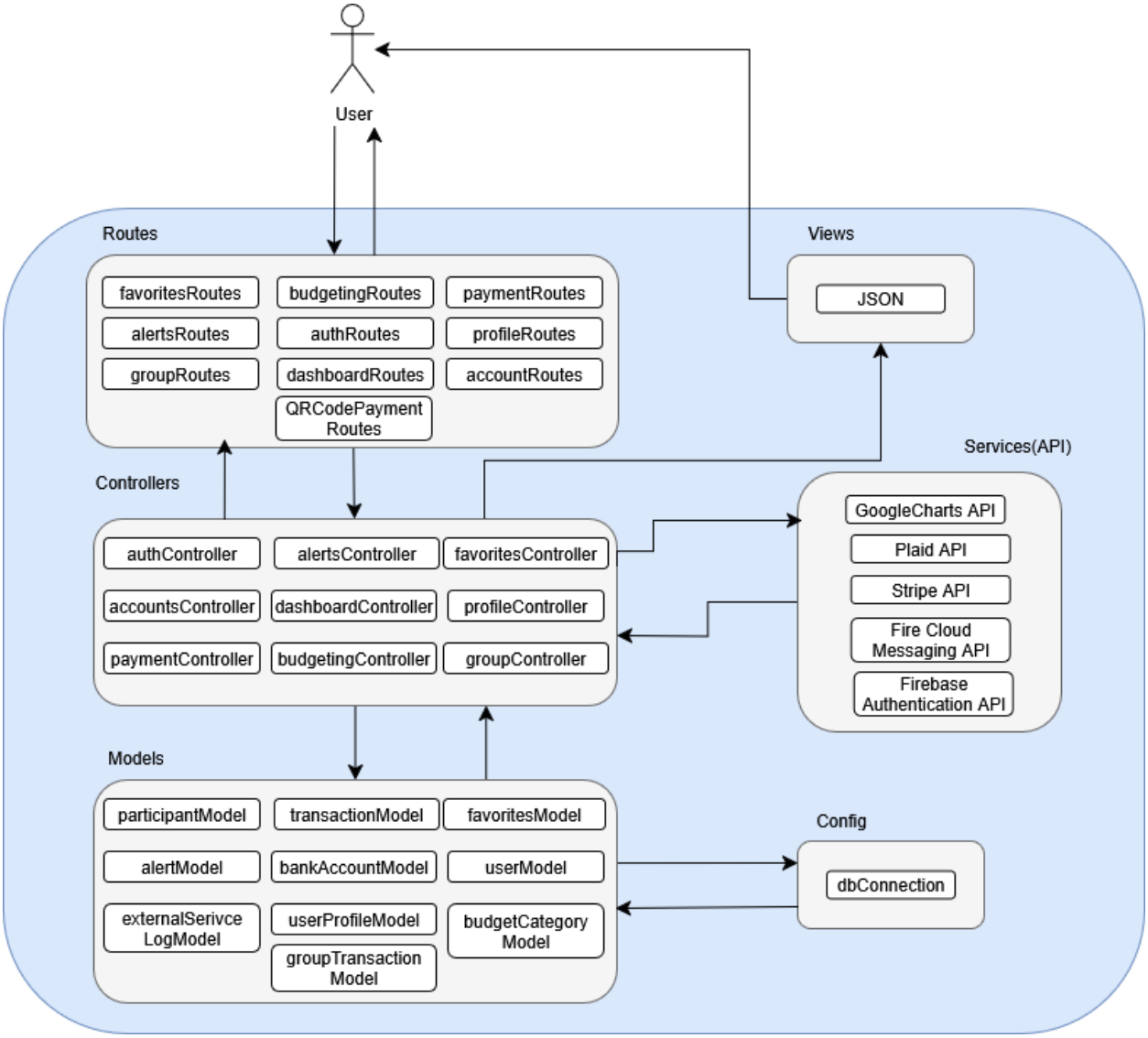
**Total Effort:** 5 taps + 5 keystrokes = 10 actions

**Navigation:** 5 actions (50%)

**Data Entry:** 5 actions (50%)

*System Architecture and System Design*

## I. Identifying Subsystems




The system follows MVC design pattern and consists of the following subsystems: models, controllers, routes, views, config and services.

### A. Routes

The subsystem Routes serves as entry points for User request such as making payments, accessing favorites or dashboard and forwarding them to the appropriate Controllers for processing. Basically, Routes provides a structure for request handling without containing business logic themselves.

### B. Controllers

The controllers serve as the core of the QuickPay backend system. It orchestrates application logic and user interactions. The controllers receive API requests through Routes, and it validates and authorizes inputs and invokes Models for database operations while also interacting with services to handle external APIs (Plaid, Stripe, Clerk, Supabase, Google Charts). Overall, the controller assembles the results and generates JSON views for the User to interact with.

### C. Models

This subsystem is the data access layer of the application. It follows the CRUD operation (Create, Read, Update, Delete) against the SQL database through the shared database connection, and interacts with Supabase services. The models represent core entities such as Users, Bank Accounts, Transaction, Budget, Alerts, Favorites and Group Transaction. The models are invoked by Controllers to perform data retrieval and manipulation which ensures persistence, consistency and easy integration with both relational data and Fire-based managed database and notifications.

### D. Services

The QuickPay Services subsystem contains adapters that connect the backend with external systems. Basically, it integrates API such as Plaid API (bank linking and balances), Stripe API (payment processing), Clerk Authentication (identity) and Google Charts (analytics/visualization support).

### E. Config

This subsystem manages configuration and infrastructure settings such as database connection, API keys and Supabase setup. Config ensures all setups are consistent and initialized correctly and can work smoothly.

### F. Views

In QuickPay, Views represent the JSON responses returned to the mobile User After Controllers process requests and gather results from Models or Services, they format the output into structured JSON payloads. These responses are then processed by the React Native application, which handles rendering and user interaction on the User side.

## II. Architecture Styles

We chose MVC(Model-View-Controller) architecture style for our application. MVC is a good choice for mobile applications that are also API based since it offers flexibility and scalability.

In QuickPay, the MVC architecture style each and subsystems have its own responsibility. Models handle CRUD operations on the SQL database while also integrating with Supabase services for authentication and notifications. Controllers orchestrate application logic calling Models for data access and Services for APIs such as Plaid API, Stripe API, Supabase and Google Charts API. Views are in JSON format as it is processed by React Native Client App.

QuickPay also follows a clint-service architecture, since it communicates with Node.JS/Express backend over secure HTTPS. It processes requests, applies business logic, queries the SQL database, and interacts with external providers. By decoupling the frontend UI from backend logic, the system ensures modularity and makes it possible to expand features without redesigning the entire application.

Furthermore, QuickPay uses an event-driven design for alerts. Domain events set off alerts (such low balances, bill reminders, or group expense changes). To alert users in real time, controllers bundle these events and send them using Supabase Cloud Messaging. This makes it possible to process data asynchronously and send important financial data on time.

QuickPay creates a reliable and expandable architecture by fusing client-server, MVC, and event-driven concepts. This method supports long-term maintainability and scalability by enabling the addition of new features (such as more payment sources, new budgeting visualizations, or extended alert types) with little alteration to the current code.

## III. Mapping Subsystems to Hardware

### A. Database Subsystem

This subsystem manages persistent storage of user data, linked bank accounts, transactions, budgets, alerts, favorites, and group expenses. We host our database on a cloud server Supabase for additional profile and real-tune data handling.

### B. Mobile Client Subsystem

This subsystem runs on the user's smartphone (IOS) by the react native applications. It displays JSON response from the backend as the user interface handles QR code scanning and manages the User's budget.

### C. Backend Application Subsystem

This subsystem runs on a cloud server. It hosts Routes, Controllers. Models, and Config to process requests from mobile users, validate **users** and execute business logic.

### D. Payment and Banking API Interfaces

This subsystem connects API to operate financial transactions. It uses Plaid API to link and retrieve account balances, and with Stripe API to handle payments.

### E. Authentication and Notification Services

This subsystem leverages Supabase Authentication for secure user login and identity authentication while Cloud Messaging is used for sending notifications to Users about alerts, low balances reminder, etc.

### F. Analytics & Visualization Subsystem

This subsystem uses Google Charts API to generate budget visualizations and spending flow diagrams. The backend fetches and formats the data, which is returned to the mobile app as JSON for rendering.

## IV. Connectors and Network Protocols

### G. HTTPS

All communication between the QuickPay mobile client and the backend uses the HTTPS protocol to ensure secure, encrypted data transfer.

### H. API Connectors

QuickPay integrates with several external services through API connectors. These include Plaid API, Stripe API, Clerk Authentication, and Google Charts. These connectors handle API authentication, requests, and responses, all transmitted securely over HTTPS.

### I. Notification Center

QuickPay uses Cloud Messaging (CM) as its notification connector. This enables the system to deliver real-time push notifications to users, such as low-balance alerts, bill reminders, or group expense updates

## V. Global Control Flow

### A. Execution Orderliness

Our system uses event-driven architecture, the backend waits for User's events such as logging in, linking accounts or making payments and System's events such as over budgeting reminders and upcoming bills alert. The events trigger the controller that processes the events by calling the models or services to return to the User as JSON responses.

### B. Time Dependency

Our system has important timing requirements such as reminders, low balance notifications, etc. Scheduled checks run in the background to detect when our user's bank account balances are running low or delayed group expense payments. When these conditions are met, the system triggers alert events, which are processed and dispatched to users through Cloud Messaging (CM).

## VI. Hardware Requirements

### A. Mobile Screen Display

Our system is for mobile applications only and is built on React Native. It's designed to run on both IOS and Android, so it requires support for various screen sizes between each smartphone.

### B. Communication Network

QuickPay requires a stable and secure internet connection to maintain real-time synchronization between the mobile client, backend server, SQL database, and Supabase services.

### C. Application Server

The backend subsystem (Node.js/Express) will be deployed on a cloud-hosted server (e.g., AWS, Azure, or Google Cloud).

### D. Database Server

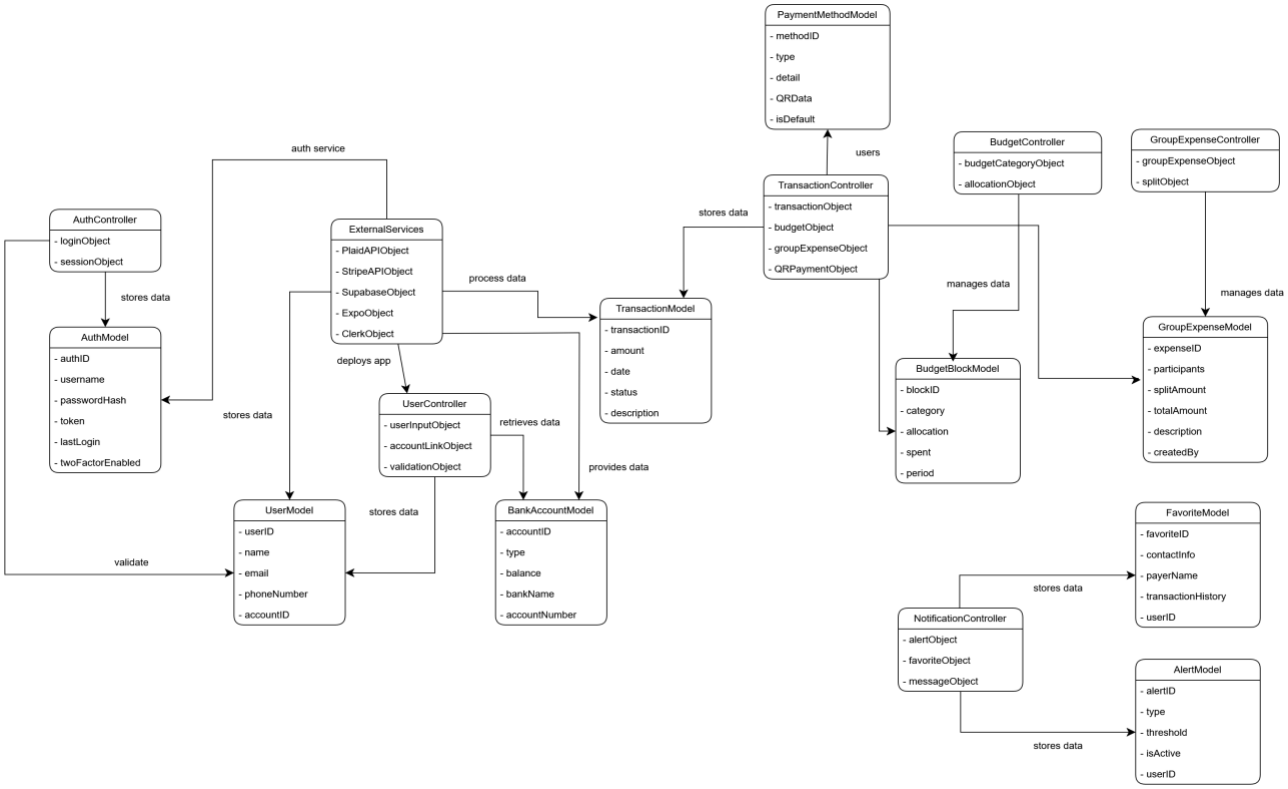
The database subsystem uses Supabase , a cloud-hosted SQL database managed on Google Cloud infrastructure.

### E. Supabase Service

QuickPay leverages Clerk Authentication, Supabase, and Cloud Messaging, all hosted on Google Cloud infrastructure.

*Analysis and Domain Modeling*

# I. Conceptual Model



### A. Concept Definitions

| Responsibility Description | Type | Concept Name |
|---|---|---|
| Handle user input and account linking. | D | UserController |
| Manages transactions, budgets, and group expenses. | D | TransactionController |
| Manages budget categories and allocations. | D | BudgetController |
| Manages shared expenses and splits. | D | GroupExpenseController |
| Manages alerts, favorites, and message. | D | NotificationController |
| Manages login objects and session objects for authentication. | D | AuthController |
| Stores user information. | K | UserModel |
| Stores bank account details | K | BankAccountModel |
| Stores transaction data. | K | TransactionModel |
| Stores budget block data. | K | BudgetBlockModel |
| Stores group expense data. | K | GroupExpenseModel |
| Stores alert details. | K | AlertModel |
| Store favorite payee/contact information. | K | FavoriteModel |
| Stores authentication data. | K | AuthModel |
| Stores payment method data. | K | PaymentMethodModel |
| Provides external services. | D | ExternalServices |

### A. Association Definitions

| Concept Pair | Association Description | Association Name |
|---|---|---|
| UserController ↔ UserModel | The UserController stores and retrieves user data through the UserModel to manage user information. | stores data |
| UserController ↔ BankAccountModel | The UserController retrieves account data from the BankAccountModel to handle account linking and management. | retrieves data |
| TransactionController ↔ TransactionModel | The TransactionController stores transaction objects in the TransactionModel to maintain transaction history. | stores data |
| TransactionController ↔ BudgetBlockModel | The TransactionController updates BudgetBlockModel to reflect spending against budget categories. | updates data |
| TransactionController ↔ GroupExpenseModel | The TransactionController updates GroupExpenseModel to manage shared expenses and splits. | updates data |
| TransactionController PaymentMethodModel | TransactionController uses payment method data, including QR payment data, to process payments. | uses |
| BudgetController ↔ BudgetBlockModel | The BudgetController manages data in the BudgetBlockModel to maintain category allocations and budgets. | manages data |
| GroupExpenseController ↔ BudgetBlockModel | The GroupExpenseController manages group expense and split data stored in GroupExpenseModel. | manages data |
| NotificationController ↔ AlertModel | The NotificationController stores and retrieves alert data from the AlertModel to notify users. | stores data |

| Concept Pair | Association Description | Association Name |
|---|---|---|
| NotificationController ↔ FavoriteModel | The NotificationController stores and retrieves favorite data from the FavoriteModel for frequent payees. | stores data |
| AuthController ↔ AuthModel | AuthController stores and retrieves authentication data. | stores data |
| AuthController ↔ UserModel | AuthController validates user identity against UserModel. | validates |
| ExternalServices ↔ BankAccountModel | External services such as Plaid provide bank account data that is stored in the BankAccountModel. | provides data |
| ExternalServices ↔ TransactionModel | External services such as Stripe process transaction data that is stored in the TransactionModel. | stores data |
| ExternalServices ↔ UserModel | External services such as Supabase store user data in UserModel for authentication and profile management. | stores data |
| ExternalServices ↔ AuthModel | External services provide authentication support through Supabase. | auth service |
| ExternalServices ↔ UserController | External services such as Expo deploy the application used by the UserController. | deploys app |

### B. Attribute Definitions

| Concept | Attributes | Attribute Description |
|---|---|---|
| UserController | userInputObject | Object containing all user interface input data and form submissions |
| | accountLinkObject | Object managing the linking process between user accounts and external bank services |
| | validationObject | Object containing validation rules and results for user input verification |
| TransactionController | transactionObject | Object containing transaction data being processed or executed |
| | budgetObject | Object managing budget-related transaction processing and category allocation |
| | groupExpenseObject | Object handling shared expense transactions and participant management |
| | qrPaymentObject | Object containing QR code payment data and processing information |
| BudgetController | budgetCategoryObject | Object defining budget categories with spending limits and rules |
| | allocationObject | Object managing the distribution of funds across different budget categories |
| GroupExpenseController | groupExpenseObject | Object containing shared expense details and participant information |
| | splitObject | Object calculating and managing expense splits among group members |
| NotificationController | alertObject | Object containing alert configuration and trigger conditions |
| | favoriteObject | Object managing frequently used payees and preferred transaction settings |

| Concept | Attributes | Attribute Description |
|---|---|---|
| | messageObject | Object containing notification message content and delivery information |
| AuthController | loginObject | Object containing user login credentials and authentication state |
| | sessionObject | Object managing active user sessions and token validation |
| UserModel | userID | Unique identifier for each user in the system |
| | name | User's full name for display and identification purposes |
| | email | User's email address for communication and account verification |
| | phoneNumber | User's phone number for two-factor authentication and notifications |
| | accountID | Primary account identifier linking to bank account information |
| BankAccountModel | accountID | Unique identifier for each linked bank account |
| | type | Account type (checking, savings, credit, etc.) |
| | balance | Current account balance retrieved from banking APIs |
| | bankName | Name of the financial institution holding the account |
| | accountNumber | Masked account number for identification (last 4 digits) |
| TransactionModel | transactionID | Unique identifier for each transaction record |
| | amount | Transaction amount in the specified currency |

| Concept | Attributes | Attribute Description |
|---|---|---|
| | date | Date and time when the transaction was executed |
| | status | Current transaction status (pending, completed, failed, cancelled) |
| | description | Transaction description or merchant information |
| | category | Budget category associated with the transaction |
| BudgetBlockModel | blockID | Unique identifier for each budget category block |
| | category | Name of the budget category (food, transportation, entertainment, etc.) |
| | allocation | Amount of money allocated to this budget category |
| | spent | Current amount spent from this budget category |
| | period | Budget period (monthly, weekly, yearly) |
| GroupExpenseModel | expenseID | Unique identifier for each shared expense |
| | participants | List of user IDs participating in the shared expense |
| | totalAmount | Total amount of the shared expense |
| | splitAmount | Individual amount each participant owes |
| | description | Description of the shared expense |
| | createdBy | User ID of the person who created the expense |
| AlertModel | alertID | Unique identifier for each alert configuration |
| | type | Type of alert (budget limit, low balance, payment due, etc.) |

| Concept | Attributes | Attribute Description |
|---|---|---|
| | threshold | Trigger value for the alert (amount, percentage, date) |
| | isActive | Boolean indicating if the alert is currently active |
| | userID | User ID associated with this alert |
| FavoriteModel | favoriteID | Unique identifier for each favorite entry |
| | payeeName | Name of the frequently contacted payee |
| | contactinfo | Contact information (phone, email, account details) |
| | transactionHistory | Record of previous transactions with this payee |
| | userID | User ID who marked this as a favorite |
| AuthModel | authID | Unique identifier for authentication record |
| | username | User's login username or email |
| | passwordHash | Encrypted password hash for secure storage |
| | token | Current authentication token for session management |
| | lastLogin | Timestamp of the user's last successful login |
| | twoFactorEnabled | Boolean indicating if two-factor authentication is enabled |
| PaymentMethodModel | methodID | Unique identifier for each payment method |
| | type | Payment method type (bank transfer, QR code, digital wallet) |
| | detail | Specific details about the payment method (account info, wallet ID) |
| | QRData | QR code data for quick payment processing |
| | isDefault | Boolean indicating if this is the user's default payment method |

| Concept | Attributes | Attribute Description |
|---|---|---|
| ExternalServices | PlaidAPIObject | Object containing Plaid API configuration and connection details |
| | StripeAPIObject | Object containing Stripe payment processing configuration |
| | ClerkObject | Object containing Clerk authentication and database configuration |
| | ExpoObject | Object containing Expo mobile app deployment and notification settings |

### C. Traceability Matrix (2) - Use Cases to Domain Concept Objects

| Domains Concepts | Use Case | UC-1 | UC-2 | UC-3 | UC-4 | UC-5 | UC-6 | UC-7 | UC-8 | UC-9 |
|---|---|---|---|---|---|---|---|---|---|---|
| | **Power** | **47** | **46** | **31** | **25** | **32** | **17** | **34** | **19** | **18** |
| UserController | | x | x | x | x | | | x | x | x |
| TransactionController | | | x | x | x | x | | | | |
| BudgetController | | | | | x | | | | | |
| GroupExpenseController | | | | x | | | | | | |
| NotificationController | | | | | | | x | | x | |
| AuthController | | x | x | x | | | | | | x |
| UserModel | | x | x | x | x | x | x | x | x | x |
| BankAccountModel | | x | x | | | x | x | x | | |
| TransactionModel | | | x | x | x | x | | | | |
| BudgetBlockModel | | | | | x | | | | | |
| GroupExpenseModel | | | | x | | | | | | |
| AlertModel | | | | | | | x | | | |
| FavoriteModel | | | | | | | | | x | |
| AuthModel | | x | x | x | | | | | | x |
| PaymentMethodModel | | x | x | | | x | | | | |
| ExternalServices | | x | x | x | x | x | x | x | x | x |

#### 1. Use Case 1 - Link Multiple Banks:

- UserController: manages user interface for account linking and handles user input validation
- BankAccountModel: stores linked bank accounts information and account details
- AuthController: manages user authentication for secure account linking
- UserModel: maintains user profile information for account association

- AuthModel: handles authentication tokens and security credentials
- PaymentMethodModel: registers new payment methods from linked accounts
- ExternalServices: integrates with Plaid API for secure bank account verification and linking

### 2. Use Case 2 - Make QR Code Payment:

- UserController: manages QR code scanning interface and payment confirmation
- TransactionController: processes payment transactions and manages payment flow
- TransactionModel: stores payment transaction details and status
- BankAccountModel: retrieves account balance and updates after payment
- AuthController: verifies user authentication for payment authorization
- UserModel: maintains user information for payment processing
- AuthModel: validates user session and payment authorization
- PaymentMethodModel: manages QR payment data and payment method selection
- ExternalServices: processes payments through Stripe API and manages QR code validation

### 3. Use Case 3 - Group Expense Splitting:

- UserController: manages group expense creation interface
- TransactionController: handles individual transaction creation for each participant
- GroupExpenseController: manages expense splitting logic and participant management
- TransactionModel: stores individual payment records for each group member
- GroupExpenseModel: maintains group expense details and split calculations
- AuthController: authenticates expense creator and validates participant access
- UserModel: stores participant information and expense relationships
- AuthModel: manages authentication for all group participants
- ExternalServices: sends notifications and processes group payment transactions

### 4. Use Case 4 - Create Budget Category:

- UserController: manages budget category creation interface

- TransactionController: links budget categories to transaction processing
- BudgetController: manages budget creation and allocation
- BudgetBlockModel: stores budget category details
- UserModel: associates budget categories with user accounts
- AuthController: authenticates user for budget operations
- ExternalServices: integrate with visualization services

### 5. Use Case 5 - Route Payments Non-Custodially Across Accounts:

- TransactionController: manages payment routing logic and cross-account transactions
- BankAccountModel: provides account balance information for routing decisions
- TransactionModel: records routing transactions and maintains transaction history
- PaymentMethodModel: manages multiple payment methods for routing options
- UserModel: maintains user preferences for payment routing rules
- ExternalServices: coordinates with banking APIs for non-custodial payment execution

### 6. Use Case 6 - Receive Alerts for Low Balances or Bills:

- NotificationController: manages alert creation, monitoring, and notification delivery
- AlertModel: stores alert configurations, thresholds, and trigger conditions
- BankAccountModel: provides real-time balance data for alert monitoring
- UserModel: maintains user alert preferences and notification settings
- ExternalServices: delivers push notifications through Supabase Cloud Messaging

### 7. Use Case 7 - View Balances & Manage via Unified Dashboard:

- UserController: manages dashboard interface and user interactions
- BankAccountModel: provides consolidated balance information across all linked accounts
- UserModel: maintains user dashboard preferences and account display settings
- ExternalServices: retrieves real-time balance data from banking APIs and manages dashboard visualization

### 8. Use Case 8 - Add Users as Favorites:

- UserController: manages favorite user selection interface
- NotificationController: handles favorite user notifications and updates
- FavoriteModel: stores favorite user information and relationship data
- UserModel: maintains user social connections and favorite relationships
- ExternalServices: manages user discovery and social features

### 9. Use Case 9 - Allow Users to View and Update Information in their Profile:

- UserController: manages profile viewing and editing interface
- UserModel: stores and updates user profile information
- AuthController: handles profile security and authentication for sensitive changes
- AuthModel: manages authentication tokens and profile update authorization
- ExternalServices: handles email verification and profile synchronization across services

## II. System Operation Contracts

| Operation | UC-1 - Link Multiple Banks |
|---|---|
| **Preconditions** | • User is authenticated and logged into the QuickPay system<br>• User has valid banking credentials for the account to be linked<br>• Bank supports Plaid API integration<br>• User has not exceeded maximum account limit (5 accounts) |
| **Postconditions** | • New BankAccountModel instance is created with unique accountID<br>• Bank account details are securely stored and encrypted<br>• Account balance is retrieved and stored from banking API<br>• PaymentMethodModel is updated with new payment option<br>• User receives confirmation notification of successful account linking |

| Operation | UC-2 - Make QR Code Payment |
|---|---|
| **Preconditions** | • User is authenticated and logged in<br>• QR code is valid and not expired<br>• User has at least one linked bank account with sufficient balance<br>• Camera permissions are granted for QR scanning |
| **Postconditions** | • QR payment data is decoded and validated<br>• New TransactionModel instance is created with payment details<br>• BankAccountModel balance is updated (decreased by payment amount)<br>• Payment is processed through Stripe API<br>• Real-time payment confirmation is sent to both parties |

| Operation | UC-3 - Group Expense Splitting |
|---|---|
| **Preconditions** | • User is authenticated as expense creator<br>• Total expense amount is greater than $0.01<br>• All participants have valid QuickPay accounts<br>• Minimum 2 participants are selected for expense splitting |

| Operation | UC-3 - Group Expense Splitting |
|---|---|
| Postconditions | • New GroupExpenseModel instance is created with unique expenseID<br>• Split amounts are calculated and assigned to each participant<br>• Individual TransactionModel records are created for each participant<br>• Push notifications are sent to all participants about the shared expense<br>• Expense tracking is activated for payment collection |

| Operation | UC-4 - Visualize and Manage Budget |
|---|---|
| Preconditions | • User is authenticated<br>• Budget category name is unique for the user<br>• Allocated amount is positive and within reasonable limits<br>• Budget period is valid (weekly, monthly, yearly) |
| Postconditions | • New BudgetBlockModel instance is created with unique blockID<br>• Budget allocation is set and spent amount initialized to $0.00<br>• Budget visualization data is updated for user dashboard<br>• Alert thresholds are optionally configured for budget monitoring |

| Operation | UC-5 - Route Payments Non-Custodially Across Accounts |
|---|---|
| Preconditions | • User is authenticated<br>• User has multiple linked bank accounts<br>• Payment routing rules are configured<br>• Source and destination accounts have sufficient balance for routing<br>• All accounts are active and accessible via APIs |
| Postconditions | • Payment routing algorithm selects optimal account based on rules<br>• TransactionModel instances are created for both source and destination<br>• BankAccountModel balances are updated for affected accounts<br>• Non-custodial routing is executed through external banking APIs<br>• Transaction confirmations are recorded with routing details |

| Operation | UC-6 - Receive Alerts for Low Balances or Bills |
|---|---|
| **Preconditions** | • User is authenticated<br>• Alert preferences are configured in user profile<br>• AlertModel instances exist for low balance or bill due alerts<br>• User device has push notification permissions enabled |
| **Postconditions** | • Alert conditions are monitored against current account balances<br>• Push notification is sent when alert threshold is met<br>• AlertModel status is updated to reflect notification sent<br>• Notification delivery confirmation is logged<br>• Users can acknowledge or dismiss alerts from notification |

| Operation | UC-7 - View Balances & Manage via Unified Dashboard |
|---|---|
| **Preconditions** | • User is authenticated<br>• User has at least one linked bank account<br>• Banking APIs are accessible for real-time data<br>• Dashboard permissions are properly configured |
| **Postconditions** | • All BankAccountModel balances are retrieved and displayed<br>• Account information is aggregated into unified dashboard view<br>• Real-time balance updates are synchronized across all accounts<br>• Dashboard management options are made available to users<br>• Account activity summary is generated for display |

| Operation | UC-8 - Add Users as Favorites |
|---|---|
| **Preconditions** | • User is authenticated<br>• Target users exist in QuickPay system<br>• Target user is not already in favorites list<br>• Users do not exceed favorite users limit (50 favorites) |
| **Postconditions** | • New FavoriteModel instance is created with target user information<br>• Target user is added to quick-access favorites list |

| Operation | UC-8 - Add Users as Favorites |
|---|---|
| | • Favorite relationship is established in the system<br>• User interface is updated to reflect new favorite<br>• Transaction history tracking is initialized for this favorite user |

| Operation | UC-9 - Allow Users to View and Update Information in their Profile |
|---|---|
| **Preconditions** | • User is authenticated<br>• User has valid session token<br>• Profile data fields meet validation requirements<br>• New email is not already registered (if updating email) |
| **Postconditions** | • UserModel instance is retrieved and displayed to user<br>• Updated profile information is validated and stored<br>• Email verification process is initiated if email changed<br>• Profile update confirmation is sent to user<br>• Authentication session is refreshed with updated information |

## III. Data Model and Persistent Data Storage

This is an example of what our database would look like:


The QuickPay application uses a cloud-hosted SQL-based backend (Supabase) to store user profiles, transactions, linked bank accounts, budget categories, alerts, favorites, and group expense details. We chose Supabase because it provides a scalable, fully managed PostgreSQL database with real-time synchronization, authentication, and storage services, all accessible through a unified API.

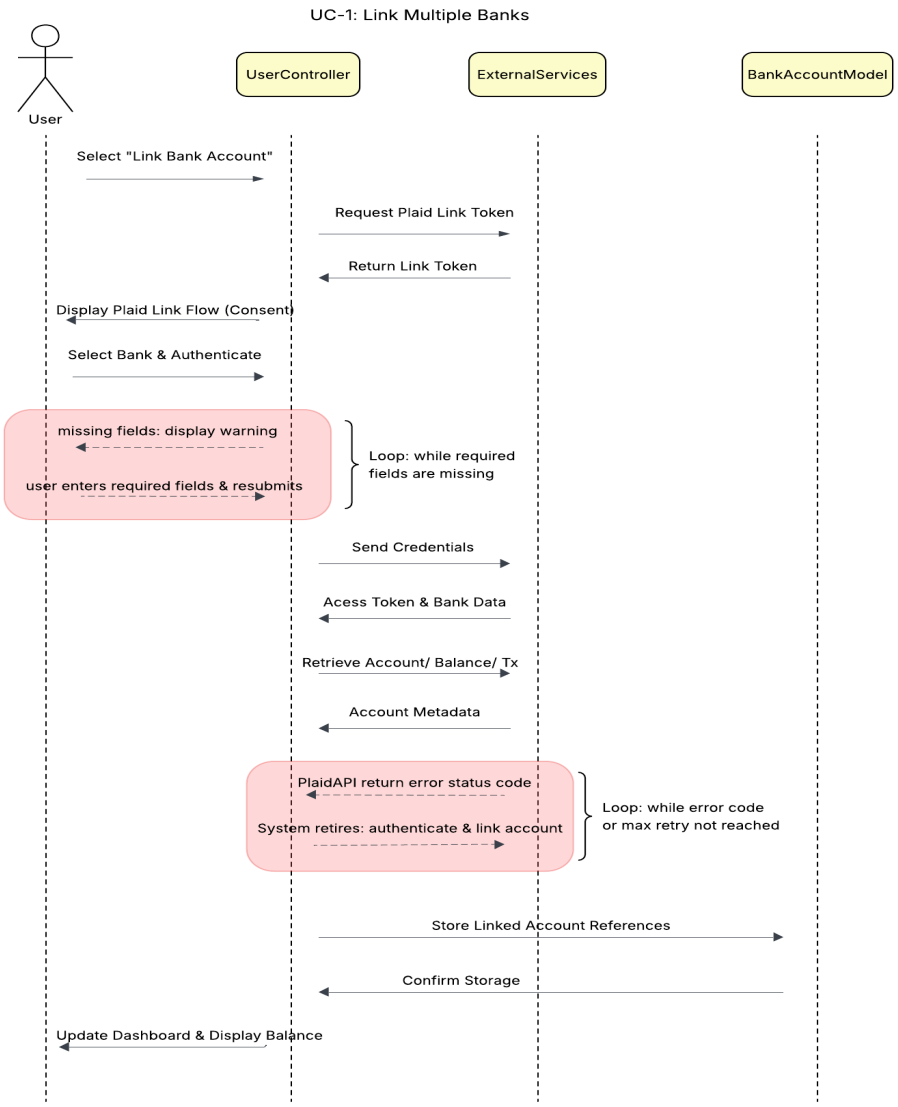
We use Supabase's relational data model, where data is organized into tables and relationships. Each table represents a system entity such as **Users**, **Transactions**, or **Budgets**, and supports constraints, foreign keys, and row-level security for consistent and secure access.

Supabase also integrates seamlessly with other core services used in QuickPay:
- **Supabase Auth** handles secure user authentication and session management.
- **Supabase Realtime** enables live updates for transactions, balance tracking, and notifications.
- **Supabase Storage** manages user-uploaded files such as receipts or profile images.

QuickPay also maintains an **External Service Log Table** to store records of interactions with third-party APIs such as **Plaid API** and **Stripe API**. This supports auditability, debugging, and compliance, ensuring that all financial events remain transparent and traceable.

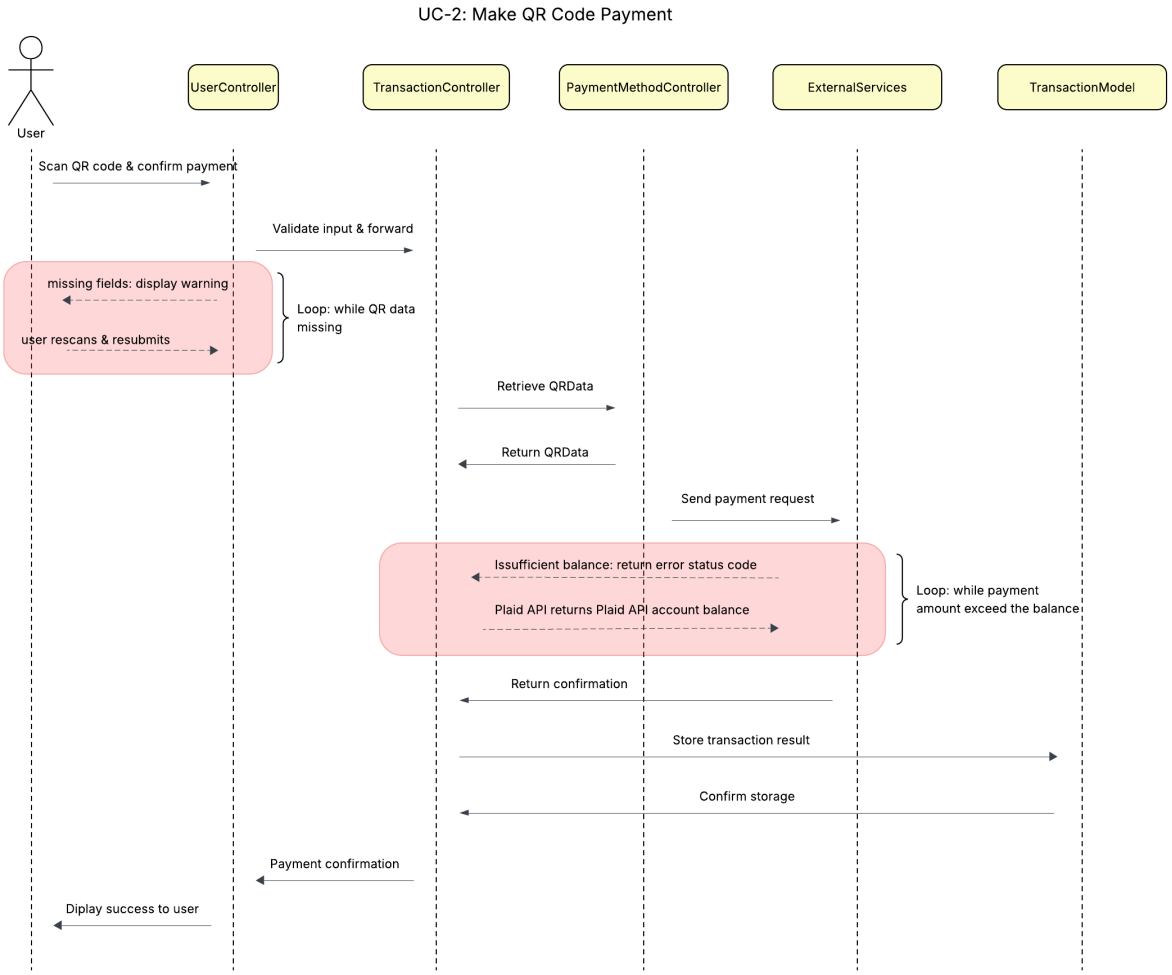
## *Interaction Diagrams*

## I. UC-1 Link Multiple Banks



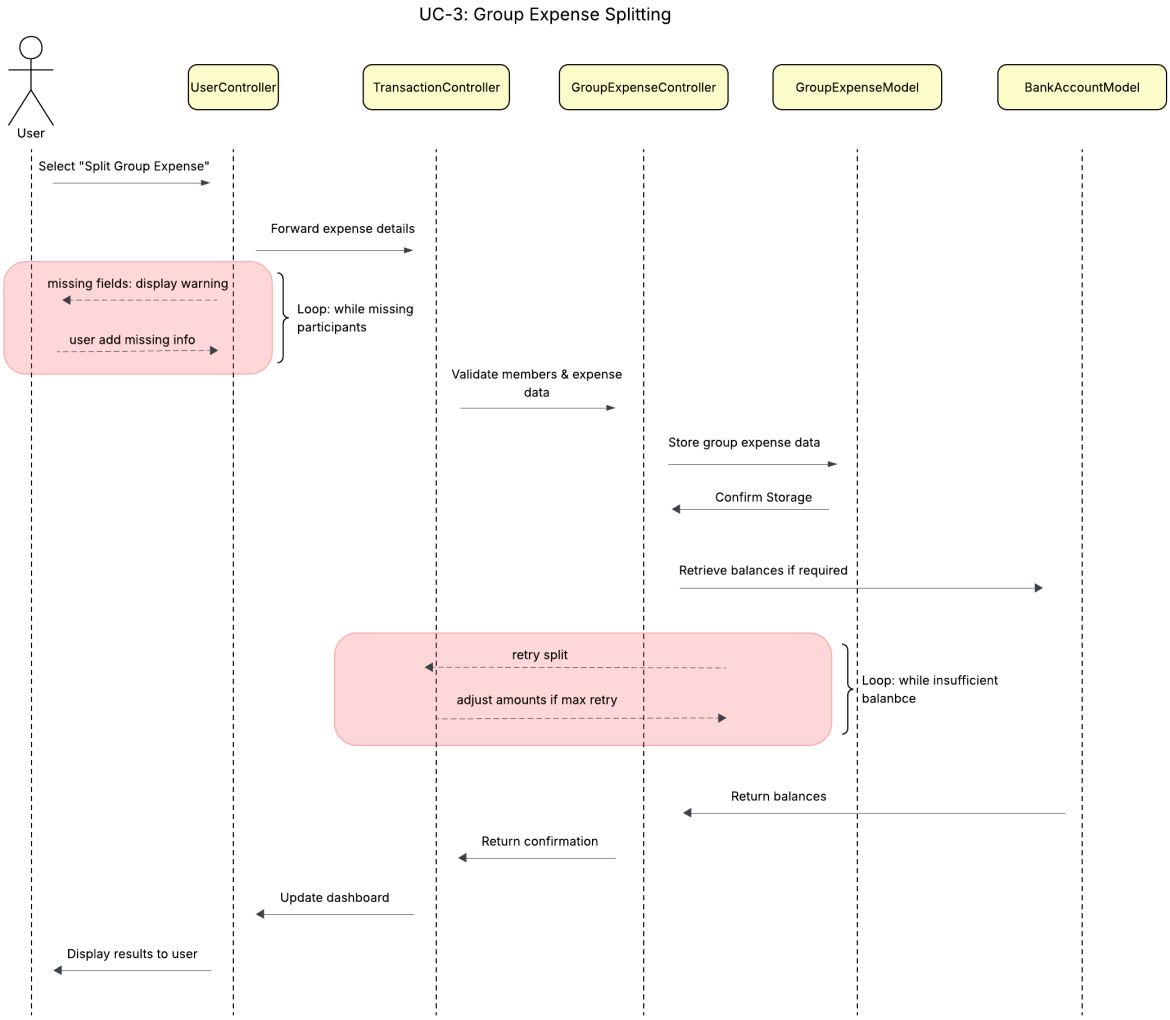

In this case, we followed the Creator principle by assigning the UserController responsibility for initiating account linking. The BankAccountModel stores the retrieved account details, while external services like Plaid handle authentication and return account data securely. By keeping the account retrieval externalized but storing results internally, we ensured low coupling between the system and external services. The UserController acts as the entry point for user requests and validates input before passing control. This approach maintains high cohesion within the BankAccountModel while reducing dependency across unrelated controllers.

## II. UC-2 - Make QR Code Payment



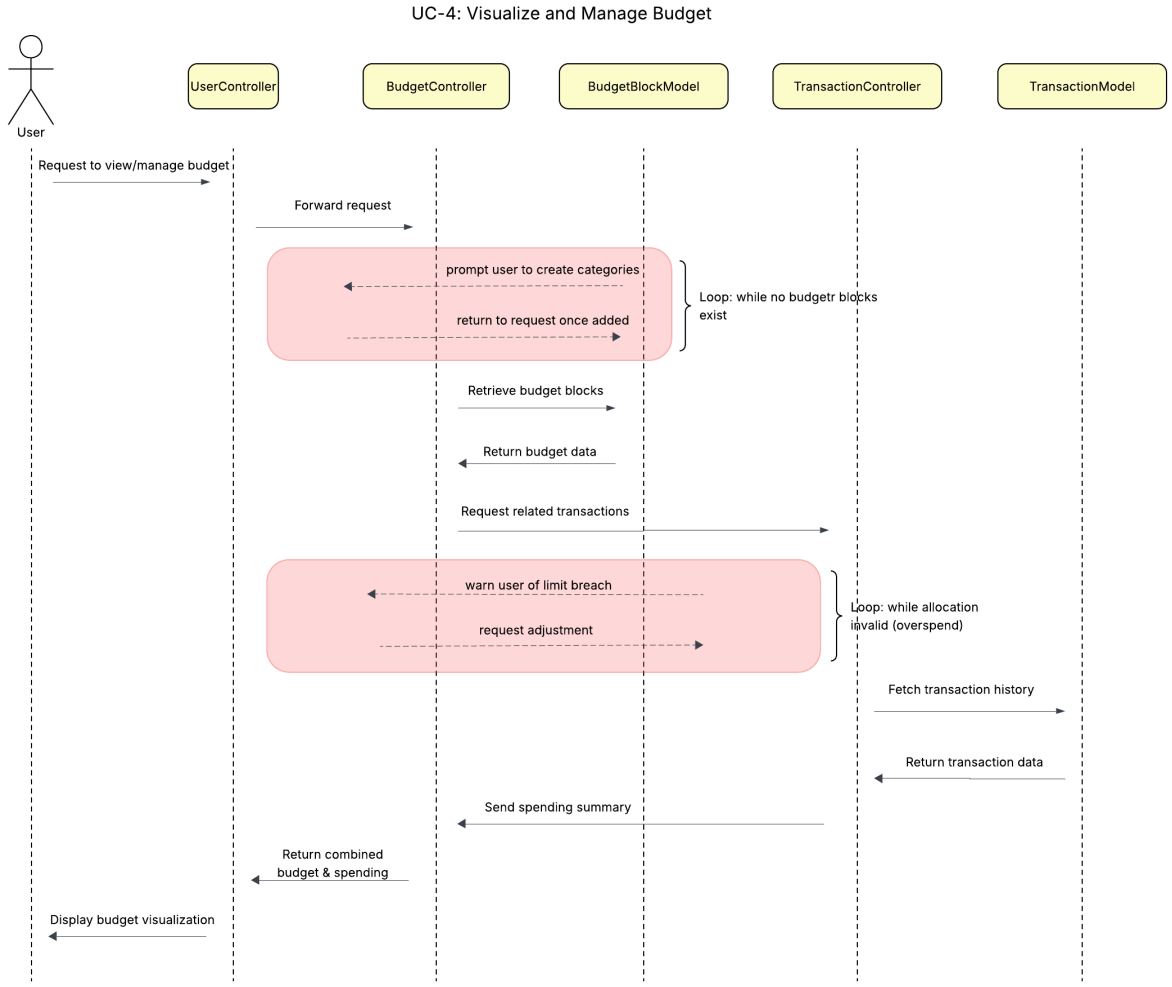

For QR-based transactions, the TransactionController manages the end-to-end payment process. Following the Expert Doer principle, the controller directly interacts with the PaymentMethodModel to retrieve QR data and processes the transaction through external services like Stripe. The TransactionModel is updated to reflect completed payments, ensuring transaction history is consistently recorded. By centralizing this responsibility, we avoided spreading payment logic across multiple controllers. The error haSndling loop for invalid QR codes is also contained within the TransactionController, reinforcing high cohesion while isolating failures from the rest of the system.

## III. UC-3 - Group Expense Splitting



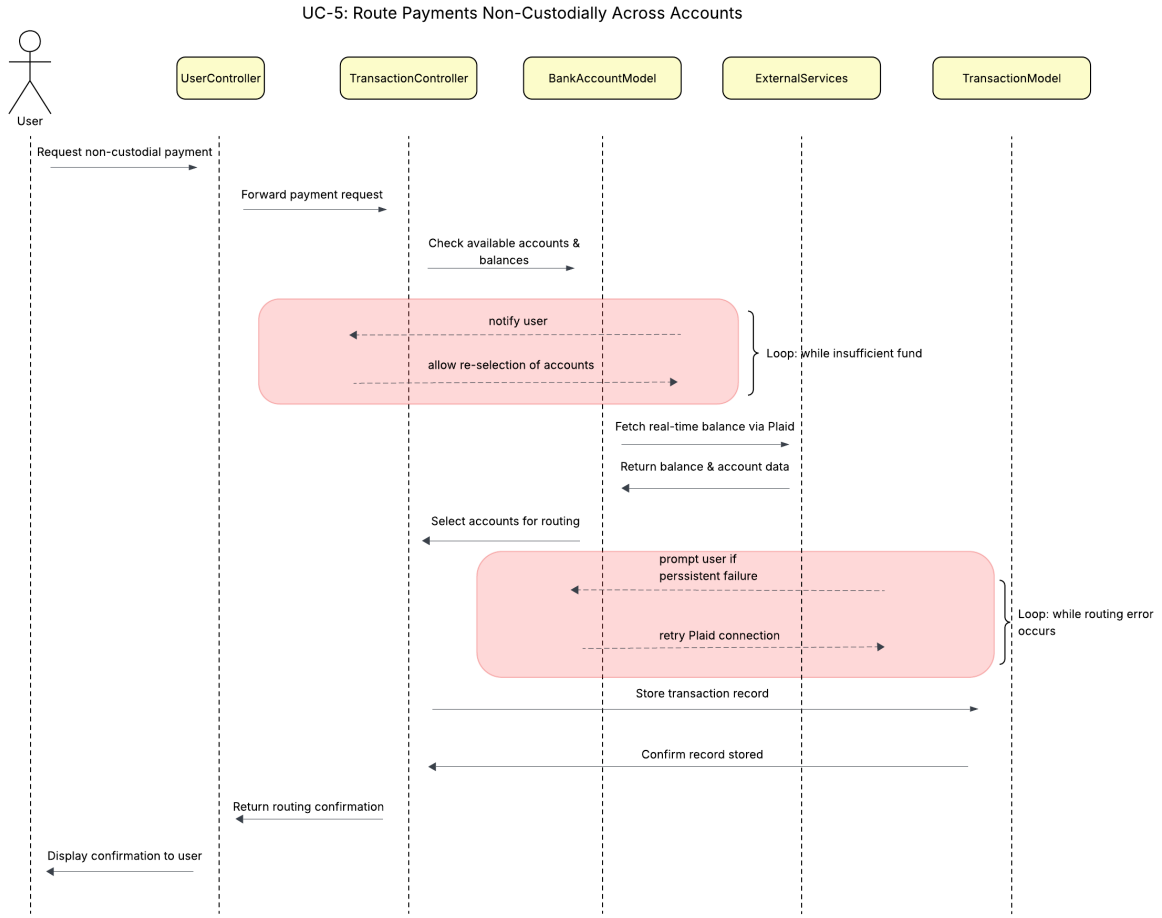

Here, the GroupExpenseController is responsible for coordinating shared payments and splits. It interacts with the GroupExpenseModel to store expense details and participant splits, while the BudgetBlockModel is updated to reflect group allocations. This aligns with the Expert Doer principle since the controller is specialized in managing shared expenses. The design ensures low coupling, as only the GroupExpenseController interacts with both models, keeping other controllers from being overburdened with group logic. The error handling loop ensures invalid splits or participant mismatches are corrected at the controller level without affecting unrelated processes.

## IV. UC-4 - Visualize and Manage Budget



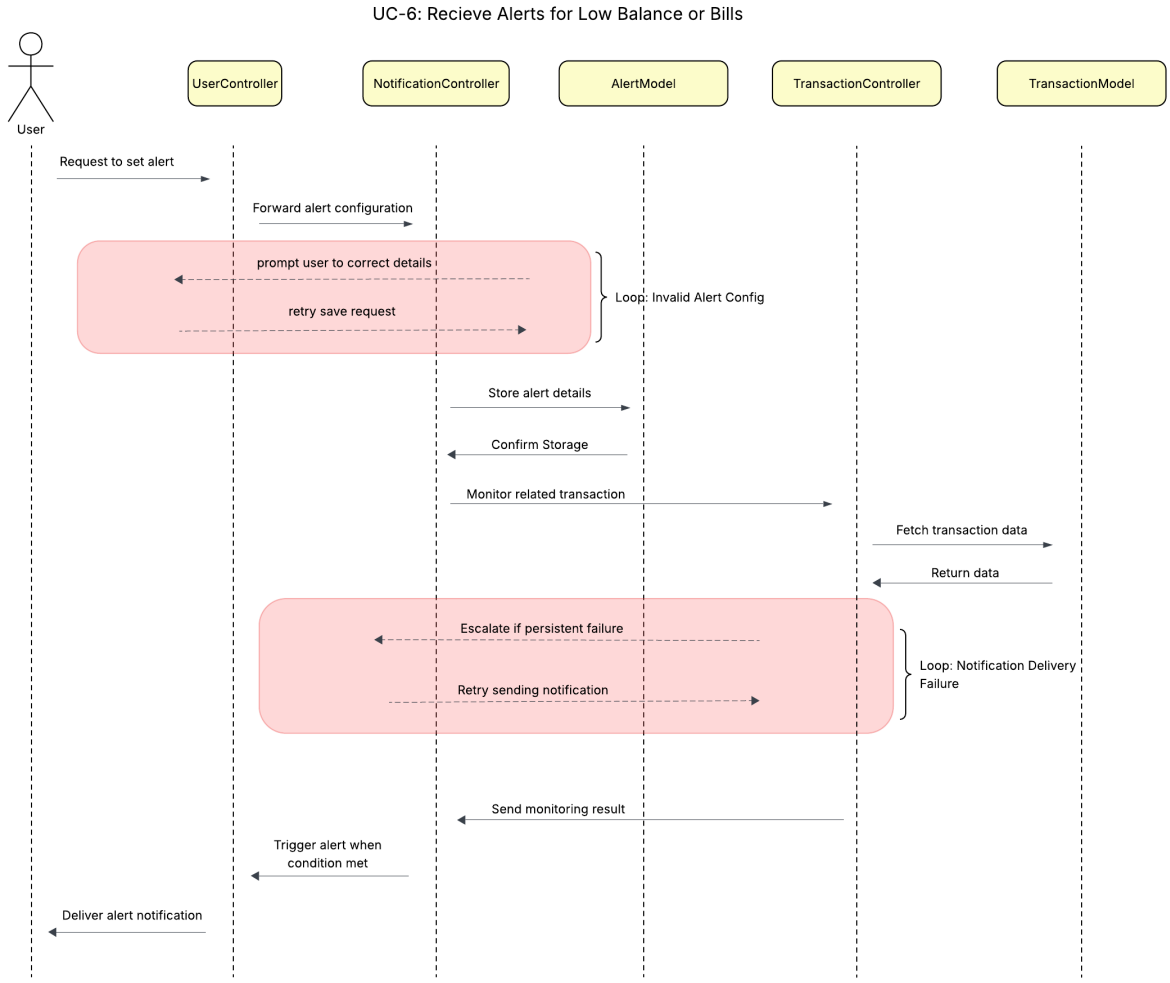

In this case, the BudgetController maintains direct control over the BudgetBlockModel. Following the Creator principle, the controller is responsible for creating and updating budget blocks, which store category allocations, spending, and limits. The visualization is driven by data retrieved from the BudgetBlockModel, ensuring the model remains the single source of truth. By restricting budget-related logic to the BudgetController, we maintained high cohesion and reduced dependency on other controllers. Error handling is also localized here, ensuring invalid inputs or allocation mismatches are managed without propagating errors elsewhere in the system.

## V. UC-5 - Route Payments Non-Custodially Across Accounts



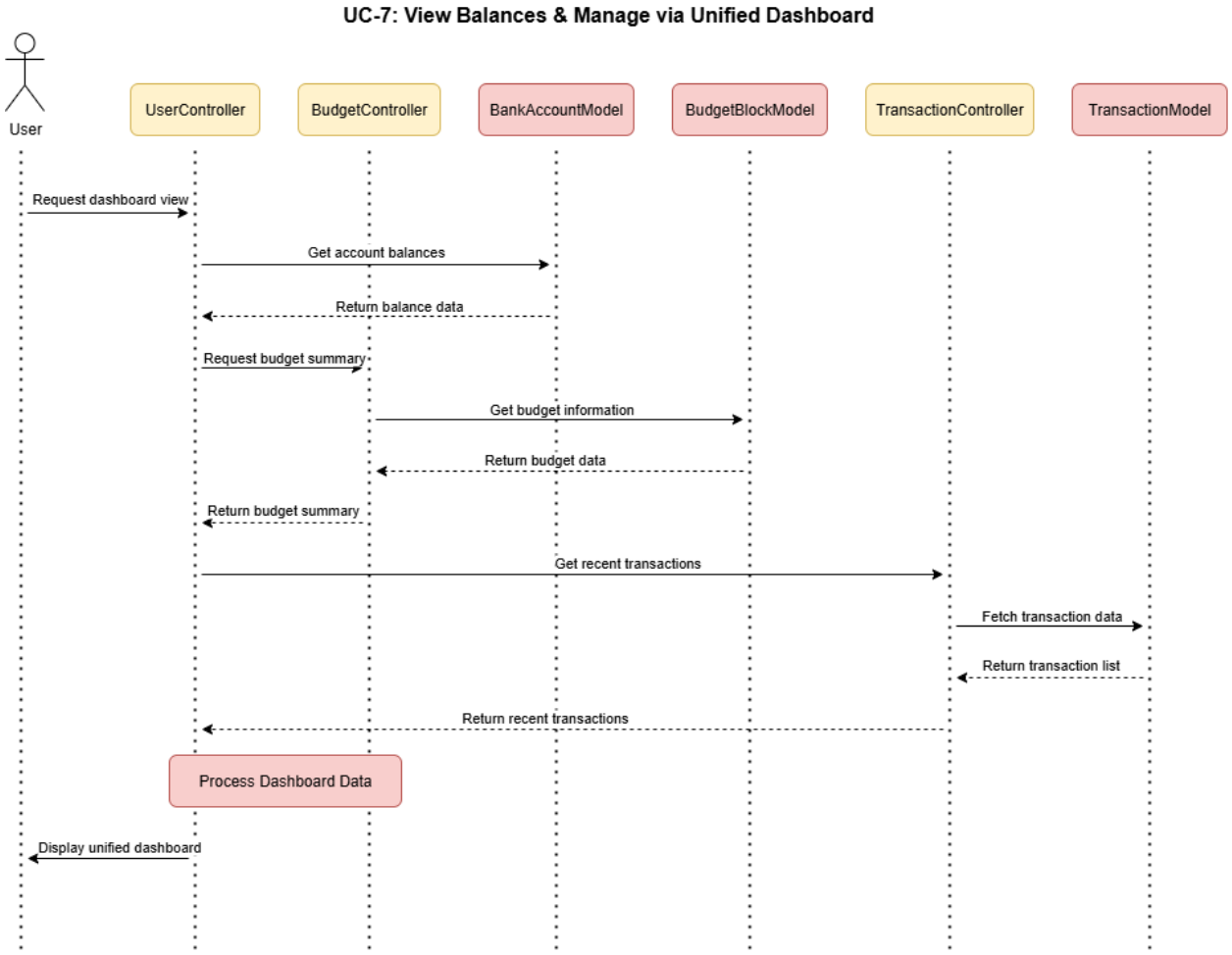

The TransactionController coordinates payments across multiple linked accounts. In line with the Expert Doer principle, it uses the BankAccountModel to validate balances and the TransactionModel to log transaction details. External services such as Plaid and Stripe are used for account access and transfer execution, but the system itself never stores funds, maintaining the non-custodial model. This separation ensures low coupling by preventing direct dependencies between controllers and external services. Failures, such as insufficient balances or routing issues, are contained within the TransactionController, improving system reliability and transparency.

## VI. UC-6 - Receive Alerts for Low Balances or Bills



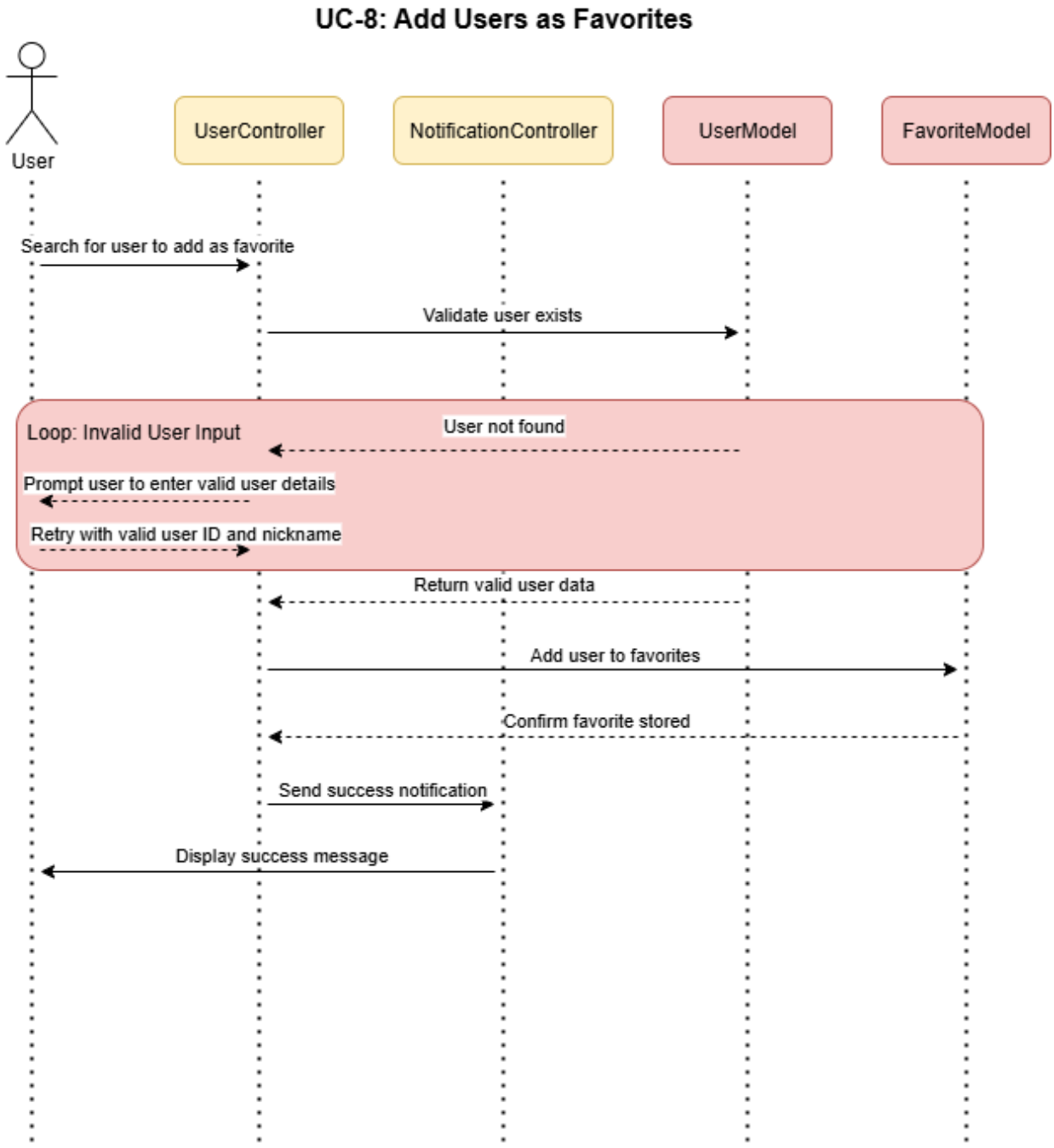

The NotificationController is tasked with managing alerts, following the Creator principle by instantiating AlertModel objects. It also monitors related activity by communicating with the TransactionController and TransactionModel to detect low balances or bill due dates. This design centralizes alert responsibility within the NotificationController, ensuring high cohesion while keeping alert logic isolated from unrelated features. The error loops for invalid alert configurations and failed notifications are handled within the NotificationController, preventing system-wide disruption. This approach aligns with the Expert Doer principle by letting each controller focus on its domain while reducing unnecessary coupling.

## VII. UC-7 - View Balances & Manage via Unified Dashboard




The UserController acts as the main organizer for the dashboard feature. It follows good design by letting each part do its own job. The UserController asks the BankAccountModel for account balances, talks to the BudgetController to get budget information, and requests recent transactions from the TransactionController. The BudgetController only handles budget tasks and gets data from the BudgetBlockModel. The TransactionController only deals with transactions and uses the TransactionModel to fetch data. This setup keeps each controller focused on one thing, making the system easier to maintain. The UserController combines all the data and shows it to the user in one simple dashboard view.

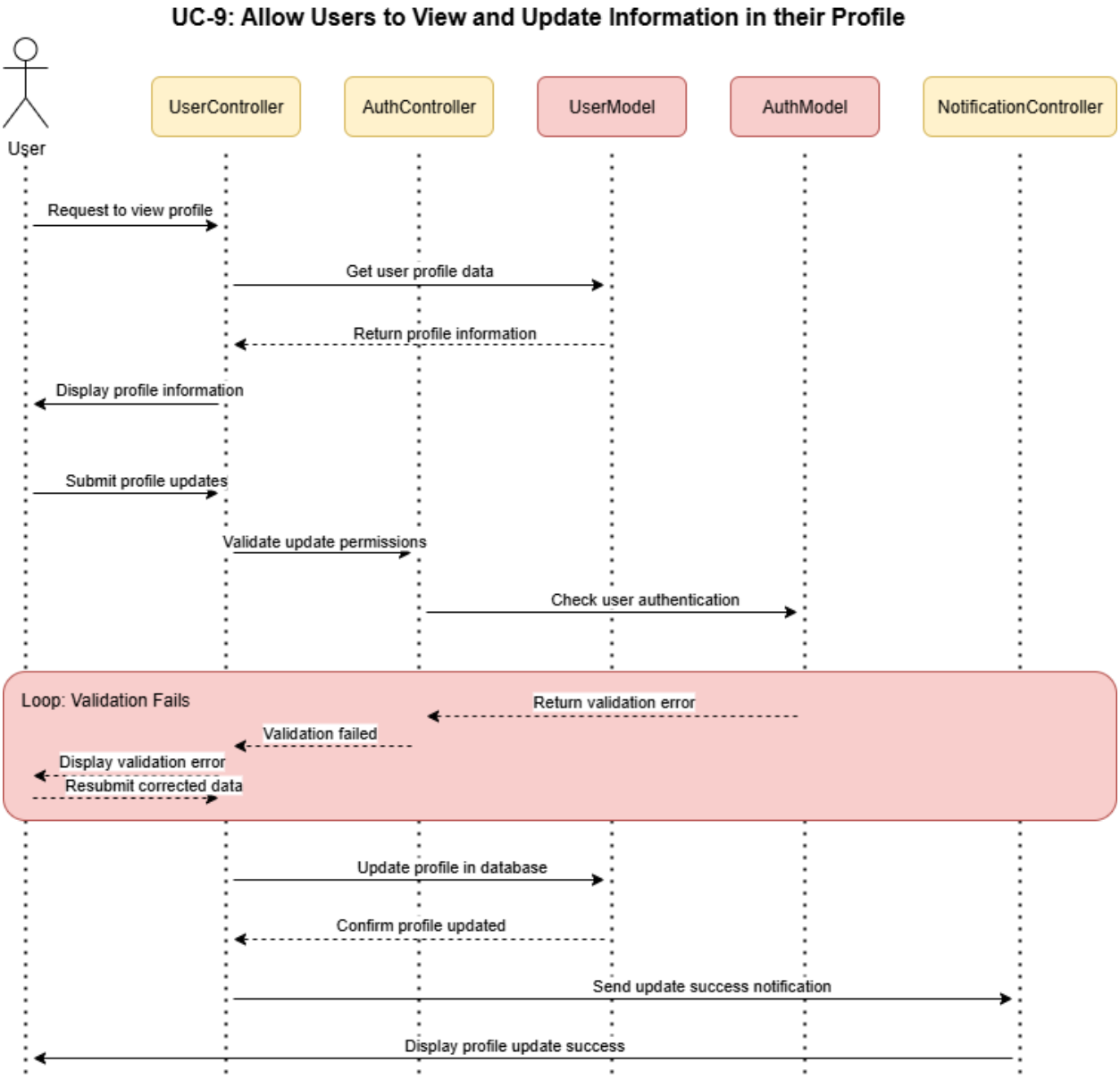
## VIII. UC-8 - Add Users as Favorites




The UserController manages adding users to favorites by working with other system parts. When a user searches for someone to add, the UserController checks with the UserModel to make sure that person exists. If the user doesn't exist or the search is wrong, the system shows an error and lets the user try again. This error loop keeps running until the user enters valid information. Once a valid user is found, the UserController tells the FavoriteModel to save the new favorite.

Finally, the NotificationController sends a success message to the user. This design keeps each part doing its own job - UserModel handles user data, FavoriteModel handles favorites, and NotificationController handles messages.

## IX. UC-9 - Allow Users to View and Update Information in their Profile



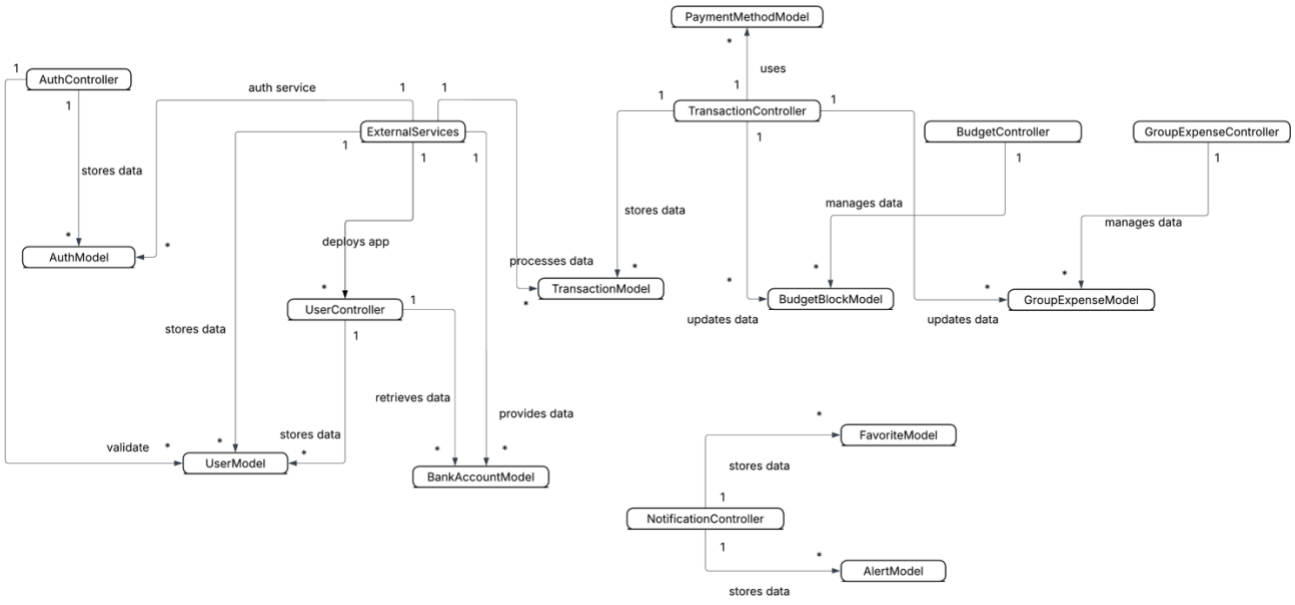

The UserController handles both viewing and updating user profiles in two separate steps. First, when a user wants to see their profile, the UserController gets the information from the UserModel and shows it to the user. Second, when a user wants to update their profile, the UserController checks with the AuthController to make sure the user has permission to make changes. If the update data has errors, the system shows error messages and lets the user fix their information. This error loop continues until the data is correct. Once everything is valid, the UserController saves the changes through the UserModel and the NotificationController tells the

user the update was successful. This approach keeps security checks separate from data storage and makes sure users get clear feedback about their actions.

*Class Diagram and Interface Specification*

# I. Class Diagram

## A. Class Diagram – Overview




### 1. Class Diagram Overview Description

Class Diagram Explanation: This section clarifies the architecture and design decisions that were incorporated when creating the classes in our QuickPay financial management system.

### 2. MVC Architecture

The MVC (Model-View-Controller) pattern enables our team to work on the user interface, business logic, and data management independently, increasing productivity through parallel development and decreasing coupling across our classes. When a change is made to a class, it typically impacts only part of the system, and it is easy to visualize how the system works as a whole since the components are organized into distinct layers.

### 3. Key Class Responsibilities and Interactions

AuthController: This class serves as the system's authentication entry point, handling user login/registration, processing security credentials, and managing authentication state. It validates user credentials through the AuthModel and coordinates with external authentication services to ensure secure access to the QuickPay platform.

#### a. UserController

Acting as the primary user management hub, this controller handles user profile operations, account settings, and user data persistence. It serves as the bridge between user interface interactions and the underlying UserModel, ensuring proper data validation and business rule enforcement.

**b. TransactionController**

This class assumes the critical responsibility of processing all financial transactions within the system. It orchestrates payment flows, validates transaction data, and coordinates between multiple payment methods and external services. The controller ensures transaction integrity while managing the complex workflows of QR payments, bank transfers, and budget tracking.

**c. BudgetController & GroupExpenseController**

Following the single responsibility principle, we allocated budget management and group expense tracking to dedicated controllers. The BudgetController manages individual spending limits and financial goals, while the GroupExpenseController handles shared expenses and bill splitting functionality, significantly enhancing the system's modularity.

**d. NotificationController**

This class manages the multi-channel notification delivery system, handling alerts for transaction confirmations, budget warnings, and payment reminders. It coordinates with various notification channels to ensure users receive timely updates about their financial activities.

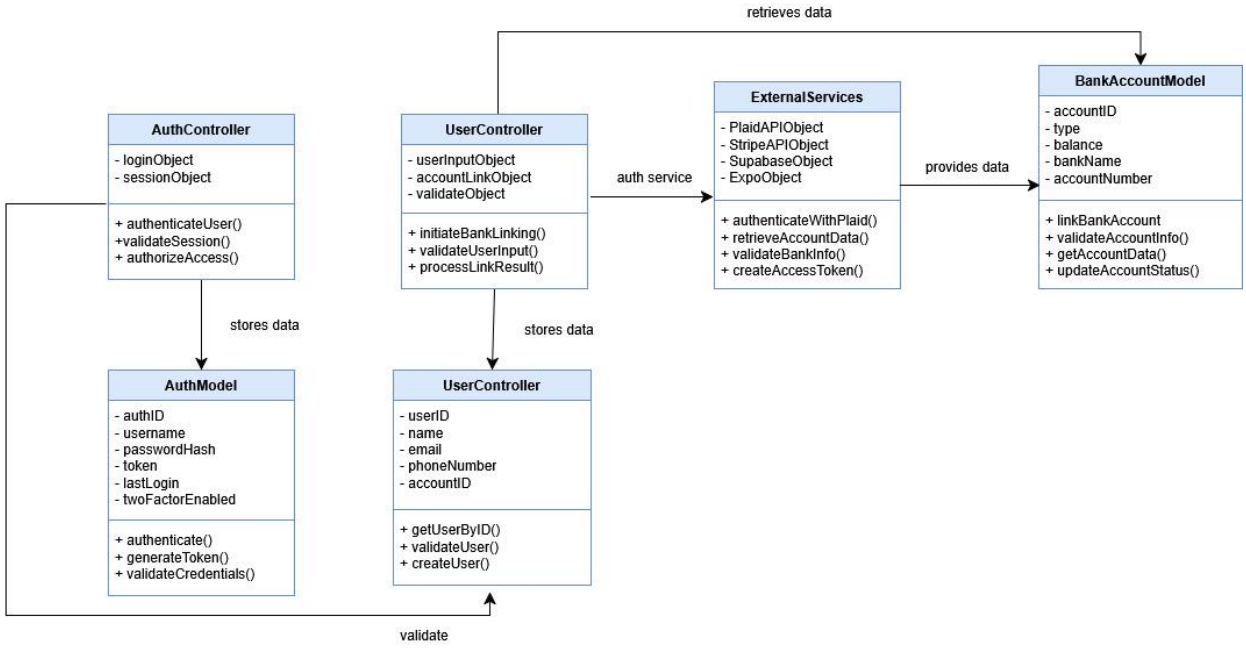
**e. PaymentMethodModel, BankAccountModel, & TransactionModel**

These core models were assigned to the primary responsibility of data management, handling the storage, retrieval, and validation of financial data. They serve as the data access layer, ensuring proper persistence and maintaining referential integrity across the financial ecosystem.

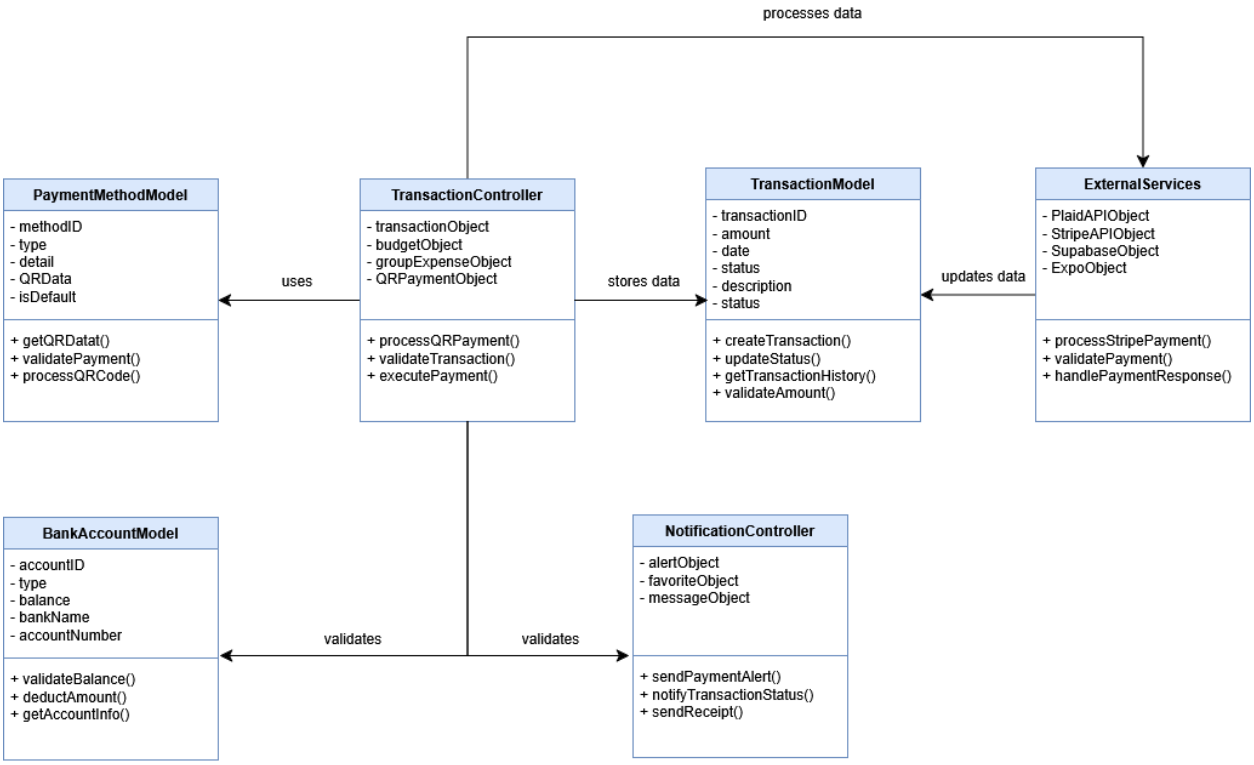
**f. External Services Integration**

The ExternalServices component manages connections to third-party financial institutions and payment processors, preprocessing API responses and transforming raw financial data into structured formats suitable for internal consumption. This design choice cleanly separates external service communication from internal business logic.

## B. Partial Class Diagram - UC-1 Link Multiple Banks


retrieves data
AuthController
- loginObject
- sessionObject
+ authenticateUser()
+validateSession()
+ authorizeAccess()
UserController
- userInputObject
- accountLinkObject
- validateObject
+ initiateBankLinking()
+ validateUserInput()
+ processLinkResult()
auth service
ExternalServices
- PlaidAPIObject
- StripeAPIObject
- SupabaseObject
- ExpoObject
+ authenticateWithPlaid()
+ retrieveAccountData()
+ validateBankInfo()
+ createAccessToken()
provides data
BankAccountModel
- accountID
- type
- balance
- bankName
- accountNumber
+ linkBankAccount
+ validateAccountInfo()
+ getAccountData()
+ updateAccountStatus()
stores data
stores data
AuthModel
- authID
- username
- passwordHash
- token
- lastLogin
- twoFactorEnabled
+ authenticate()
+ generateToken()
+ validateCredentials()
UserController
- userID
- name
- email
- phoneNumber
- accountID
+ getUserByID()
+ validateUser()
+ createUser()
validate

## C. Partial Class Diagram - UC-2 Make QR code payment



## II. Data Types and Operation Signatures

### D. Class: UserModel

#### 1. Attributes:

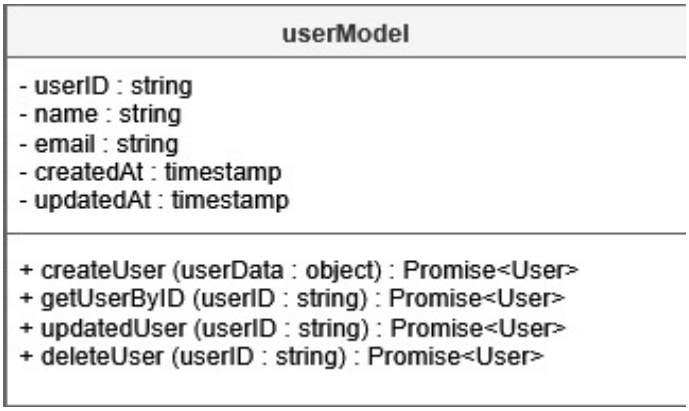
- userID : string - Supabase UID for the user.
- name : string - Full name of the user.
- email : string - Unique email address.
- createdAt : timestamp - Account creation time.
- updatedAt : timestamp - Last update time.

#### 2. Operations:

- createUser(userData: Object): Promise<User> - Creates a new user record in Clerk Authentication and Supabase with the provided details.
- getUserByID(userID: string): Promise<User> - Retrieves a specific user's profile data using their unique Supabase user ID.
- updateUser(userID: string, updates: Object): Promise<User> - Updates an existing user's information (e.g., name, email, settings) in Supabase.
- deleteUser(userID: string): Promise<void> - Removes the user's account and associated profile data from the system.

#### 3. Meaning:

This class represents the user entity and provides CRUD operations for user data stored in Clerk Authentication and Supabase.




### E. Class: BankAccountModel

#### 4. Attributes:

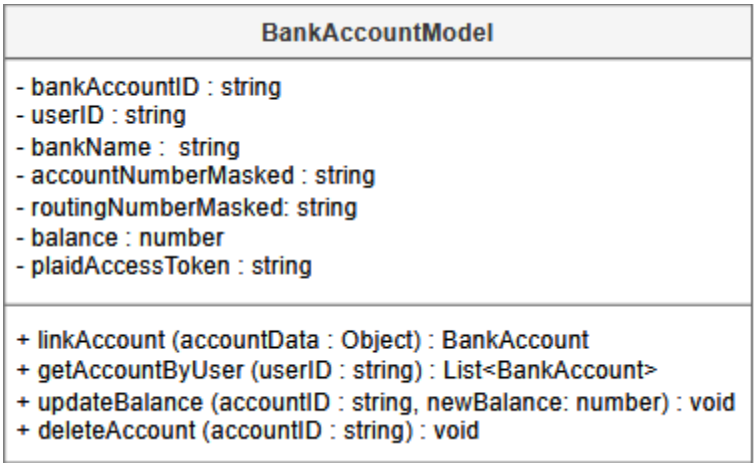
- bankAccountID : string - Unique identifier for the linked account.
- userID : string - Reference to the owning user.
- bankName : string - Name of the bank.
- accountNumberMasked : string - Masked account number for display.
- routingNumberMasked : string - Masked routing number.
- balance : number - Current balance.
- plaidAccessToken : string - Secure token reference from Plaid.

**5. Operations:**

- linkAccount(accountData: Object): Promise<BankAccount> - Adds a new bank account for a user by linking details retrieved from Plaid.
- getAccountsByUser(userID: string): Promise<List<BankAccount>> - Retrieves all bank accounts associated with a specific user.
- updateBalance(accountID: string, newBalance: number): Promise<void> - Updates the stored balance for a bank account after syncing with the bank.
- deleteAccount(accountID: string): Promise<void> - Removes a linked bank account from the user's profile.

**6. Meaning:**

This class represents a user's bank account(s) and manages integration with Plaid for balance retrieval and updates.




### F. Class: TransactionModel

**7. Attributes:**

- transactionID : string - Unique transaction identifier.

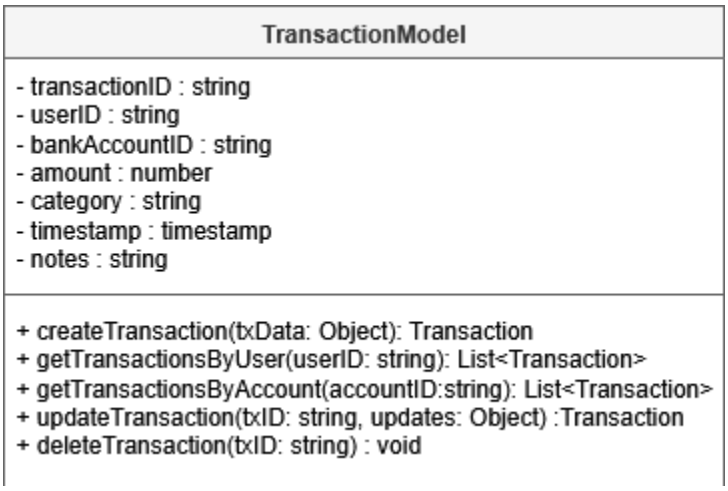
- userID : string - Reference to Users.
- bankAccountID : string - Reference to BankAccounts.
- amount : number - Transaction amount.
- category : string - Budget category.
- timestamp : timestamp - Date and time of transaction.
- notes : string - Optional description.

**8. Operations:**

- createTransaction(txData: Object): Promise<Transaction> - Records a new transaction linked to a user and one of their bank accounts.
- getTransactionsByUser(userID: string): Promise<List<Transaction>> - Retrieves all transactions made by a specific user.
- getTransactionsByAccount(accountID: string): Promise<List<Transaction>> - Retrieves all transactions associated with a specific bank account.
- updateTransaction(txID: string, updates: Object): Promise<Transaction> - Updates details of an existing transaction (e.g., category, notes).
- deleteTransaction(txID: string): Promise<void> - Removes a transaction from the system.

**9. Meaning:**

This class represents financial transactions and links them to accounts, budgets, and alerts.




## G. Class: BudgetModel

**10. Attributes:**

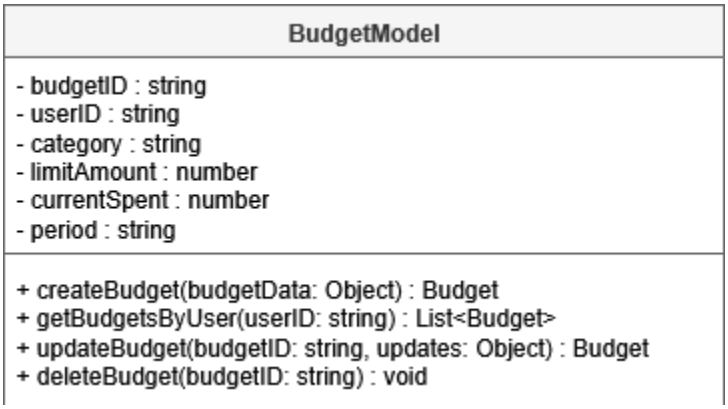
- budgetID : string - Unique budget identifier.
- userID : string - Reference to Users.
- category : string - e.g., “needs”, “wants”, “savings”.
- limitAmount : number - Budget cap.
- currentSpent : number - Amount already spent.
- period : string - “monthly” or “weekly”.

**11. Operations:**

- createBudget(budgetData: Object): Promise<Budget> - Creates a new budget category with spending limits for a user.
- getBudgetsByUser(userID: string): Promise<List<Budget>> - Retrieves all budgets defined by a specific user.
- updateBudget(budgetID: string, updates: Object): Promise<Budget> - Updates details of an existing budget (e.g., limit amount, period).
- deleteBudget(budgetID: string): Promise<void> - Deletes a budget and removes its tracking information from the system

**12. Meaning:**

This class models budget categories and enforces spending caps by tracking totals over time.




**H. Class: AlertModel**

**13. Attributes:**

- alertID : string - Unique alert identifier.
- userID : string - Reference to Users.

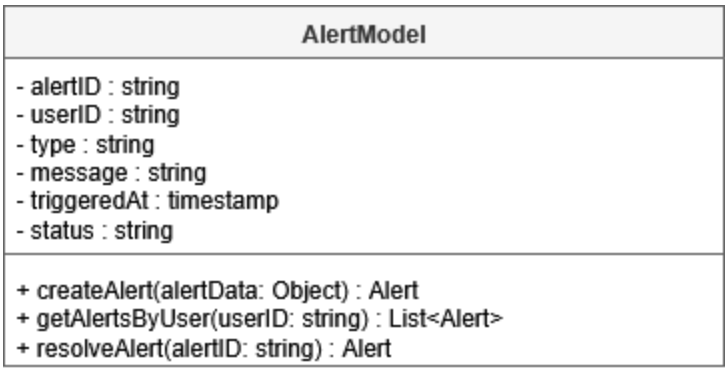
- type : string - e.g., lowBalance, billReminder, budgetExceeded.
- message : string - Notification text.
- triggeredAt : timestamp - When the alert fired.
- status : string - active | resolved.

### 14. Operations:

- createAlert(alertData: Object): Promise<Alert> - Creates a new alert for a user based on conditions such as low balance or bill reminders.
- getAlertsByUser(userID: string): Promise<List<Alert>> - Retrieves all alerts associated with a specific user.
- resolveAlert(alertID: string): Promise<Alert> - Marks an existing alert as resolved after the issue has been addressed.

### 15. Meaning:

This class manages financial alerts and integrates with Alerts/Notifications to push notifications to users.




## I. Class: GroupExpenseModel

### 16. Attributes:

- groupExpenseID : string - Unique ID for the group expense.
- createdBy : string - UserID of the creator.
- title : string - Description of the group expense.
- totalAmount : number - Total expense amount.
- createdAt : timestamp - Date created.
- Subcollection: Participants

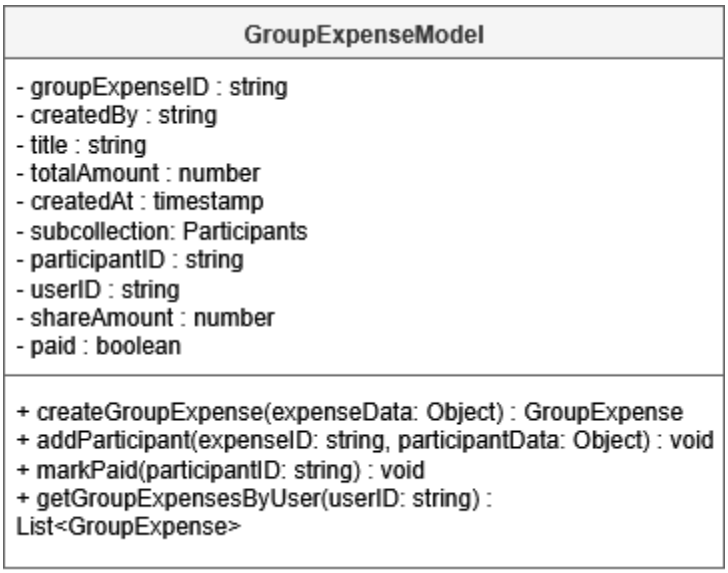
- participantID : string
- userID : string - Reference to Users.
- shareAmount : number
- paid : boolean

**17. Operations:**

- createGroupExpense(expenseData: Object): Promise<GroupExpense> - Creates a new group expense entry with details such as title and total amount.
- addParticipant(expenseID: string, participantData: Object): Promise<void> - Adds a participant to a group expense and assigns their share of the cost.
- markPaid(participantID: string): Promise<void> - Updates a participant's status to indicate their share has been paid.
- getGroupExpensesByUser(userID: string): Promise<List<GroupExpense>> - Retrieves all group expenses created by or involving a specific user.

**18. Meaning:**

This class manages group expenses and tracks participants' shares, integrating with reminders and notifications.




## J. Class: FavoriteModel

1. Attributes:

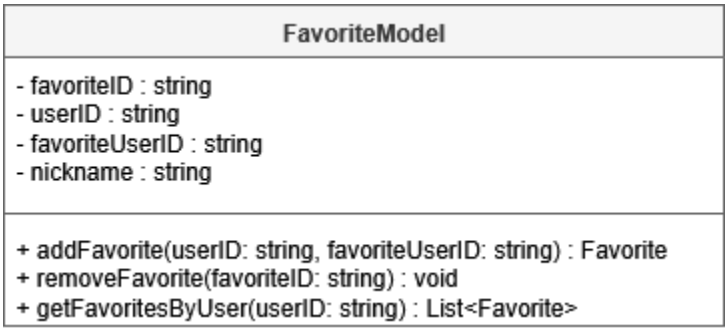
- favoriteID : string - Unique identifier.
- userID : string - Reference to Users.
- favoriteUserID : string - Reference to Users marked as favorites.
- nickname : string - Optional nickname for quick reference.

**19. Operations:**

- addFavorite(userID: string, favoriteUserID: string): Promise<Favorite> - Adds another user to the current user's favorites list for quicker payments.
- removeFavorite(favoriteID: string): Promise<void> - Deletes an existing favorite entry from the user's list
- getFavoritesByUser(userID: string): Promise<List<Favorite>> - Retrieves all favorite users saved by a specific user.

**20. Meaning:**

This class manages a user's favorites list to simplify repeated payments or transfers.




### K. Class: ExternalServiceLogModel

**21. Attributes:**

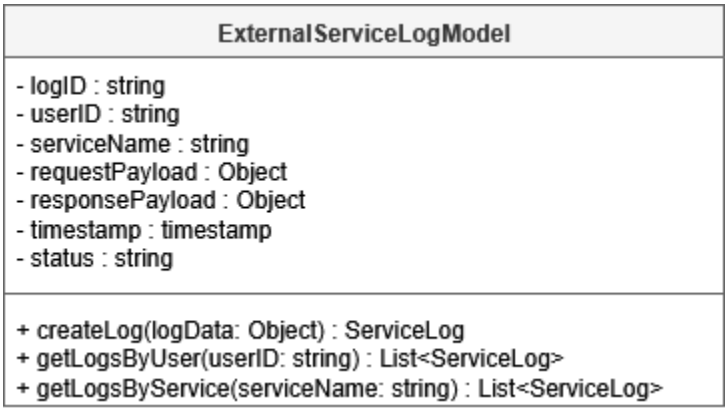
- logID : string - Unique identifier.
- userID : string - Reference to Users.
- serviceName : string - Name of the external service (Plaid, Stripe, Supabase, etc.).
- requestPayload : Object - Data sent to the service.
- responsePayload : Object - Data returned by the service.
- timestamp : timestamp - When the interaction occurred.
- status : string - success | error.

**22. Operations:**

- createLog(logData: Object): Promise<ServiceLog> - Records a new log entry for an interaction with an external service (e.g., Plaid, Stripe, Supabase).
- getLogsByUser(userID: string): Promise<List<ServiceLog>> - Retrieves all service logs associated with a specific user.
- getLogsByService(serviceName: string): Promise<List<ServiceLog>> - Retrieves all logs related to a specific external service.

**23. Meaning:**

This class tracks external service interactions for auditing, debugging, and compliance.




## L. Class: PaymentController

**24. Attributes:**

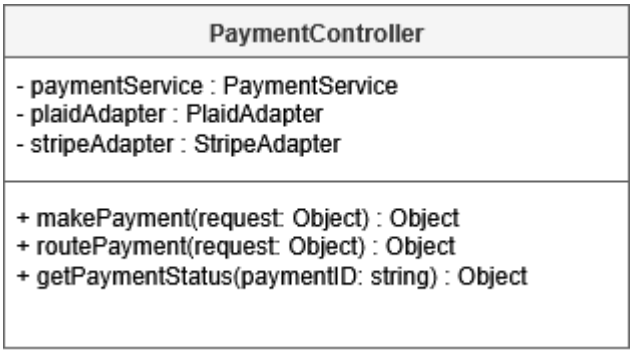
- paymentService : PaymentService - service handling payment logic.
- plaidAdapter : PlaidAdapter - adapter for account linking and balance checks.
- stripeAdapter : StripeAdapter - adapter for routing payments securely.

**25. Operations:**

- makePayment(request: Object): Promise<Object> - initiates a QR code or direct payment.
- routePayment(request: Object): Promise<Object> - routes payments non-custodially across linked accounts.
- getPaymentStatus(paymentID: string): Promise<Object> - checks the status of a payment.

**26. Meaning:**

This controller manages all user payment interactions. It validates requests, communicates with Plaid and Stripe through adapters, and ensures that payments are executed securely without holding funds in the QuickPay system.




### M. Class: AlertController

#### 27. Attributes:

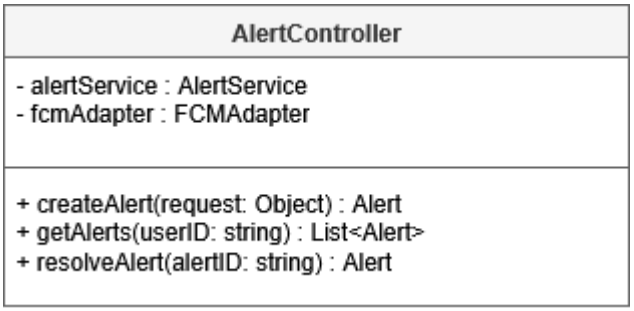
- alertService : AlertService - service for monitoring alert conditions.
- fcmAdapter : FCMAdapter - adapter for sending push notifications.

#### 28. Operations:

- createAlert(request: Object): Promise<Alert> - creates a new financial alert.
- getAlerts(userID: string): Promise<List<Alert>> - retrieves all alerts for a user.
- resolveAlert(alertID: string): Promise<Alert> - marks an alert as resolved.

#### 29. Meaning:

This controller oversees financial alerts such as low balances, budget overruns, and bill reminders. It triggers alerts based on conditions in the models and dispatches push notifications to users through Firebase Cloud Messaging.



### N. Class: GroupExpenseController

#### 30. Attributes:

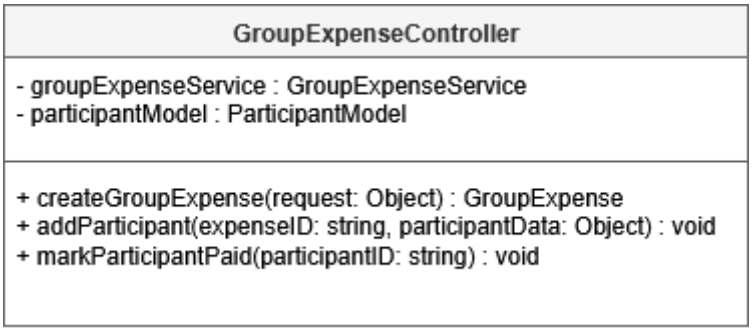
- groupExpenseService : GroupExpenseService - service for handling group splits.
- participantModel : ParticipantModel - manages participants in group expenses.

#### 31. Operations:

- createGroupExpense(request: Object): Promise<GroupExpense> - creates a new group expense.
- addParticipant(expenseID: string, participantData: Object): Promise<void> - adds a participant to a group expense.
- markParticipantPaid(participantID: string): Promise<void> - updates payment status for a participant.

#### 32. Meaning:

This controller manages group expense workflows. It coordinates between users, updates participant statuses, and ensures reminders or notifications are sent if payments are delayed.




### O. Class: ProfileController

#### 33. Attributes:

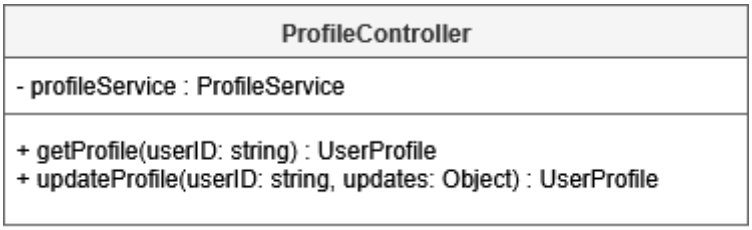
- profileService : ProfileService - service for managing user profiles.

#### 34. Operations:

- getProfile(userID: string): Promise<UserProfile> - retrieves a user profile.
- updateProfile(userID: string, updates: Object): Promise<UserProfile> - updates profile details such as preferences and settings.

**35. Meaning:**

This controller handles user profile management, allowing users to view and update their personal information, preferences, and settings.




**P. Class: FavoriteController**

**36. Attributes:**

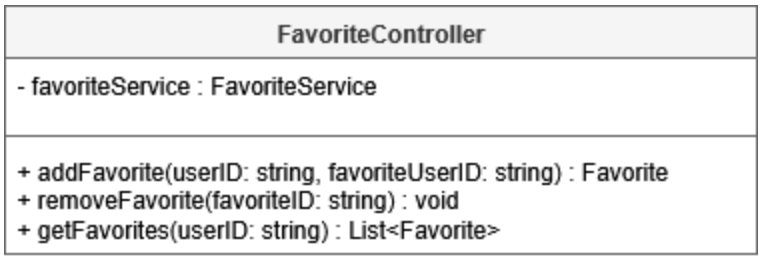
- favoriteService : FavoriteService - service for managing favorites.

**37. Operations:**

- addFavorite(userID: string, favoriteUserID: string): Promise<Favorite> - adds a user to the favorites list.
- removeFavorite(favoriteID: string): Promise<void> - removes a favorite entry.
- getFavorites(userID: string): Promise<List<Favorite>> - retrieves a user's favorites.

**38. Meaning:**

This controller manages the favorites feature, simplifying frequent transactions by letting users quickly select preferred recipients.




**Q. Class: DashboardController**

**39. Attributes:**

- dashboardService : DashboardService - service aggregating multiple data sources.

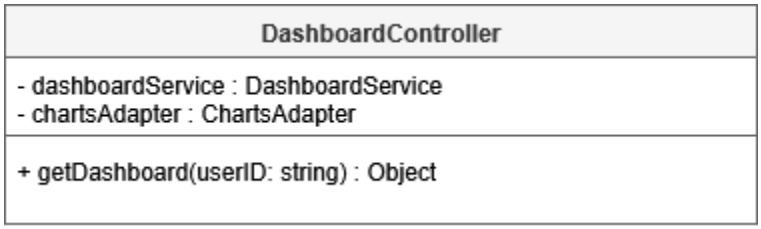
- chartsAdapter : ChartsAdapter - adapter for creating budget and spending visualizations.

**40. Operations:**

- getDashboard(userID: string): Promise<Object> - retrieves consolidated data including balances, budgets, alerts, and recent transactions.

**41. Meaning:**

This controller provides a unified entry point for the QuickPay app's home screen. It aggregates data from models and services into a single JSON response tailored to the user's needs.



## III. Traceability Matrix (3) – Domain Concept Objects to Class Objects

| | Domain Concepts | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Software Classes** | UserController | TransactionController | BudgetController | GroupExpenseController | NotificationController | AuthController | UserModel | BankAccountModel | TransactionModel | BudgetBlockModel | GroupExpenseModel | AlertModel | FavoriteModel | AuthModel | PaymentMethodModel | ExternalServices |
| UserModel | x | | | | | | x | | | | | | | | | |
| BankAccountModel | | | | | | | | x | | | | | | | | |
| TransactionModel | | x | | | | | | | x | | | | | | | |
| BudgetModel | | | x | | | | | | | x | | | | | | |
| AlertModel | | | | | x | | | | | | | x | | | | |
| GroupExpenseModel | | | | x | | | | | | | x | | | | | |
| FavoriteModel | | | | | | | | | | | | | x | | | |
| ExternalServiceLogModel | | | | | | | | | | | | | | | | x |
| PaymentController | | x | | | | | | | | | | | | | x | |
| AlertController | | | | | x | | | | | | | x | | | | |
| GroupExpenseController | | | | x | | | | | | | x | | | | | |
| ProfileController | x | | | | | x | x | | | | | | | x | | |
| FavoriteController | | | | | | | | | | | | | x | | | |
| DashboardController | x | x | x | x | | | | | | | | | | | | |

This traceability matrix establishes the direct correspondence between the Domain Concept Objects (DCOs) identified in the conceptual model and the Class Objects (COs) implemented in the system's design. It ensures that every conceptual element defined during the analysis phase has been accurately transformed into a tangible class within the object-oriented structure of the system.

The purpose of this matrix is to verify the continuity and completeness of design by tracing each domain concept such as controllers, models, and external services to its implemented class counterpart in the software architecture. For example, conceptual objects like UserController, TransactionController, and BankAccountModel are mapped to their respective implemented classes within the MVC architecture.

By maintaining these mappings, the development team ensures that:

- All conceptual requirements are fully realized in the design phase.
- No critical domain concept is omitted or redundantly represented.
- The class structure accurately reflects the logical entities and relationships defined in the conceptual model.

This traceability also supports effective validation and verification, helping confirm that system functionalities such as linking multiple banks, processing QR code payments, managing budgets, and handling group expenses are rooted in well-defined and consistently implemented class structures. Overall, the matrix reinforces design alignment, improves maintainability, and provides a transparent audit trail from conceptual modeling to class implementation.

## *Algorithms and Data Structures*

### I. Algorithms

Our application does not utilize any complex algorithms such as sorting, graph traversal or machine learning algorithms. We rely on built-in Supabase and JavaScript operating for data retrieval and usage. For example, our application algorithm is only used for:

- To calculate the total spending or per budget category
- Split group expenses evenly among participants
- Trigger alerts when conditions such as low balances

The algorithms mentioned above are not complex but straightforward arithmetic and comparison operations.

### II. Data Structures

Within QuickPay's architecture, several data structures are employed to manage user data efficiently:

Objects and Maps: Used by representing structured data like users, transactions, budgets, and alerts. An object is used to keep key-value pairs that map to Firestore documents.

Arrays (Lists): They are utilized to keep lists of items such as connected bank accounts, transaction listings, and group expense members. Their arrays are processed to increase spending sums or produce reports.

Queue (Event Handling): An Event-driven queue-like structure is employed internally by Firebase Cloud Messaging and background schedulers. This will ensure that notifications and alarms (e.g., low balances or bill reminders)

### III. Concurrency

QuickPay accommodates parallel work with the event-driven, non-blocking design of its Node.js. More than one user request-e.g., payment, transaction updates, and notification triggers-can be processed concurrently without side effects.

This is done by:

- Asynchronous Promises and Callbacks: Enabling multiple Supabase and API requests (Plaid, Stripe, etc.) running in parallel.
- Concurrent Database Access: Supabase takes care of sequential reads and writes concurrently, making data always consistent.

- Parallel Notifications: Firebase Cloud Messaging supports parallel handling of multiple push notifications, providing simultaneous notification for all users.
- It enables QuickPay to process multiple operations concurrently efficiently with data integrity and real-time response.

## IV. Design Patterns

For the QuickPay mobile banking application, we carefully evaluated several design patterns to ensure the architecture would support secure financial transactions, real-time data synchronization, and a responsive user interface. We selected the Model-Service-Controller pattern as our primary architectural approach, with several supporting patterns integrated throughout the system.

The Model-Service-Controller pattern emerged as the optimal choice for QuickPay's architecture because it naturally aligns with React Native's component-based structure while providing clear separation between data, business logic, and user interface. Unlike the traditional Model-View-Controller pattern commonly used in web applications, MSC introduces an intermediate service layer that handles all external API communications and complex business operations. The Model layer represents the data structures that define user profiles, bank accounts, transactions, and favorite contacts. The Service layer acts as the intermediary between our application and external systems, managing all communication with the banking API, user authentication service, and database. The Controller layer, implemented through React components, manages user interactions and coordinates between the UI and services, handling state management and rendering logic without directly communicating with external APIs. This separation proved essential for security and maintainability. By isolating all external communications within the service layer, we created a single point of control for sensitive operations like API token management and encrypted data transmission.

The Proxy pattern was implemented extensively in our service layer to control access to external APIs and provide simplified interfaces to complex systems. A proxy acts as an intermediary that controls access to another object, adding additional functionality like caching, validation, or access control. Our banking service implementation serves as a proxy to the external banking. API, intercepting all requests and adding security measures, error handling, and data transformation. This pattern proved critical because it allowed us to abstract away the complexity of the banking API's authentication requirements, token refresh logic, and rate limiting. Components simply request transaction data or account information without needing to understand the intricacies of OAuth token management or API pagination. The proxy pattern

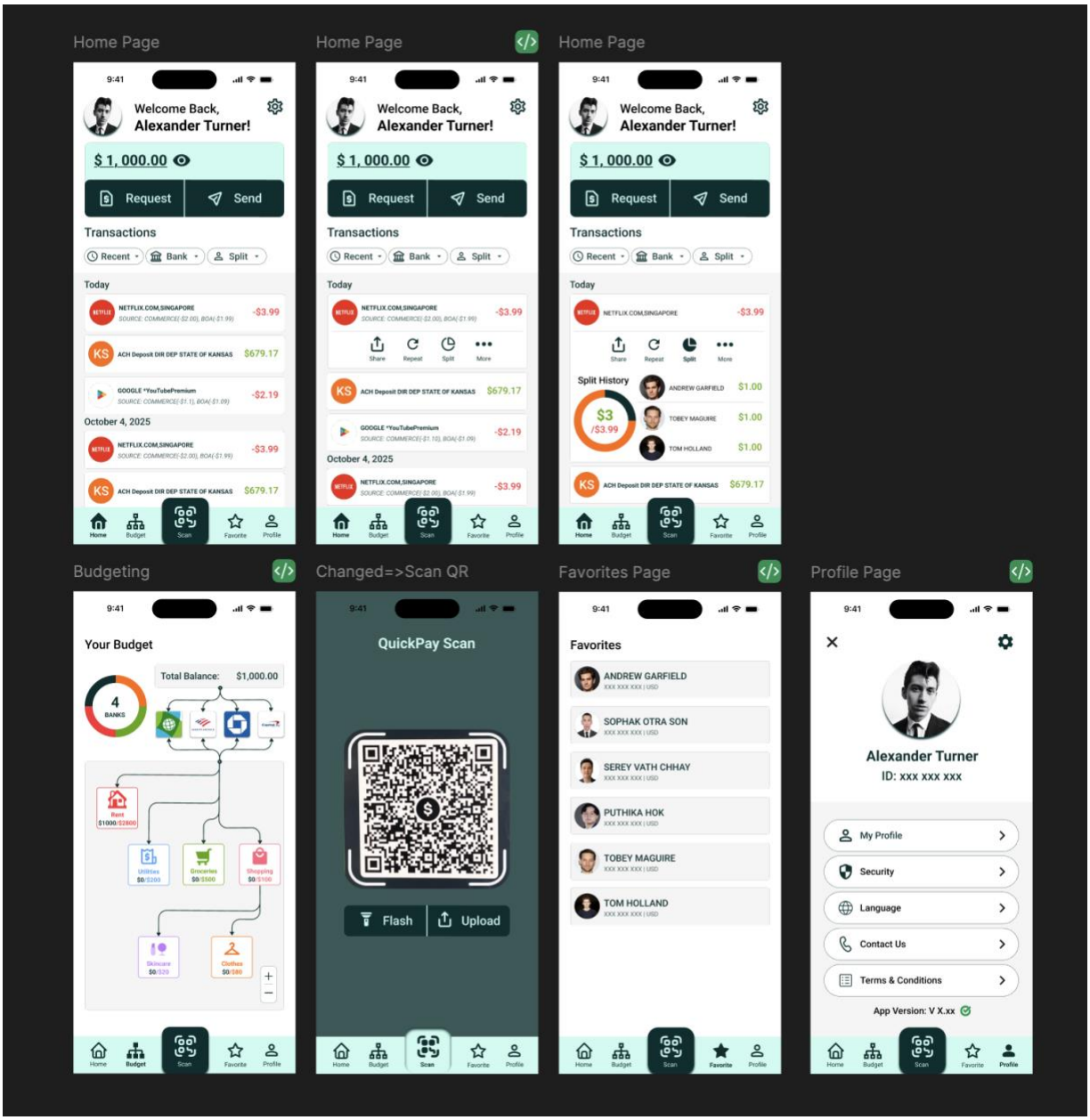
also enhanced security by ensuring all API communications pass through a controlled access point where we enforce encryption, validate request parameters, and sanitize responses.

The Repository pattern provides an abstraction layer between the application's business logic and data storage mechanisms, presenting a collection-like interface for accessing data regardless of its source. We implemented this pattern to manage interactions with our database, providing a consistent API for creating, reading, updating, and deleting user profiles, favorites, and application settings. The repository abstraction allowed us to change database implementations or add additional data sources without modifying the rest of the application. User profile data comes from both the authentication service and our database, but the repository presents a unified interface that merges data from both sources transparently. By centralizing data access logic, the repository pattern simplified testing and improved code maintainability.

The Observer pattern establishes a one-to-many dependency between objects where state changes in one object automatically notify all dependent objects. This pattern is fundamental to React's architecture and is implemented through state management hooks. When application state changes, all components observing that state automatically receive updates and re-render as needed. We leveraged this built-in pattern extensively for managing real-time updates to account balances, transaction lists, and user profiles. When new transaction data arrives from the banking API, the state update triggers automatic re-rendering of all screens displaying transaction information, eliminating the need to manually manage observer registration and notification.

The design patterns implemented in QuickPay reflect a pragmatic approach to mobile banking application architecture. Rather than forcing patterns into the codebase for academic purposes, we selected patterns that solved genuine architectural challenges while working harmoniously with React Native's component model. The Model-Service-Controller pattern provided overall structure, while Proxy and Repository patterns addressed specific requirements around API integration and data access. This combination created a maintainable, secure, and scalable architecture that supports QuickPay's current features while remaining flexible for future enhancements.

## *User Interface Design and Implementation*




We do not have updates on user interface design at this time. However, we are confident in our ability to complete all frontend development within the scheduled timeframe.

*Test Designs*

## I. Test Cases and Unit Testing

### A. UserController

| **Test Case Identifier**: TC-1<br>**Use Case Tested:** UC-1<br>**Pass/Fail Criteria:** Test passes if a Plaid link token is generated and returned successfully.<br>**Input Data:** Link token request | |
|---|---|
| **Test Procedure** | **Expected Results** |
| User initiates "Link Bank Accounts" request. | System requests a Plaid link token and receives a valid response. |
| Network connection lost during token request. | System retries requests or displays connection error message. |

| **Test Case Identifier**: TC-2<br>**Use Case Tested:** UC-1<br>**Pass/Fail Criteria:** Test passes if bank authentication succeeds and returns valid account metadata.<br>**Input Data:** Bank credentials (username, password) | |
|---|---|
| **Test Procedure** | **Expected Results** |
| User enters valid credentials. | Plaid authenticates and returns account list |
| User enters invalid credentials. | Error message "Authentication failed" displayed; user prompted to retry. |
| Bank API returns error code | System retries up to 3 times or aborts gracefully with a failure notice. |

| **Test Case Identifier**: TC-3<br>**Use Case Tested:** UC-1<br>**Pass/Fail Criteria:** Test passes if retrieved account data is saved correctly in the database.<br>**Input Data:** Plaid account metadata | |
|---|---|
| **Test Procedure** | **Expected Results** |

| **Test Case Identifier**: TC-3<br>**Use Case Tested:** UC-1<br>**Pass/Fail Criteria:** Test passes if retrieved account data is saved correctly in the database.<br>**Input Data:** Plaid account metadata | |
|---|---|
| Account data retrieved successfully. | Data is stored in BankAccountModel and displayed in user dashboard |
| Database write fails. | Error log created; user notified of failure to save account. |

### B. TransactionController

| **Test Case Identifier**: TC-4<br>**Use Case Tested:** UC-2<br>**Pass/Fail Criteria:** Test passes if valid QR code is scanned and decoded properly.<br>**Input Data:** QR payment data | |
|---|---|
| **Test Procedure** | **Expected Results** |
| User initiates “Link Bank Accounts” request. | Payment info decoded and displayed. |
| User scans expired or tampered QR code. | “Invalid QR Code” warning displayed. |

| **Test Case Identifier**: TC-5<br>**Use Case Tested:** UC-2<br>**Pass/Fail Criteria:** Test passes if payment is processed successfully.<br>**Input Data:** Confirmed QR payment request | |
|---|---|
| **Test Procedure** | **Expected Results** |
| User confirms valid payment. | Transaction completed and stored in TransactionModel. |
| User cancels payment. | Transaction aborted with “Payment cancelled” status. |
| API timeout or error occurs. | Retry initiated or “Payment failed, please try again” shown. |

| **Test Case Identifier**: TC-6<br>**Use Case Tested:** UC-2<br>**Pass/Fail Criteria:** Test passes if post-payment confirmation is accurate.<br>**Input Data:** Transaction receipt | |
|---|---|
| **Test Procedure** | **Expected Results** |
| Successful transaction completed. | App displays success screen and updates dashboard balance. |
| Failed transaction logged. | App shows failure message and records event in log. |

### C. GroupExpenseController

| **Test Case Identifier**: TC-7<br>**Use Case Tested:** UC-3<br>**Pass/Fail Criteria:** Test passes if group expense form accepts valid participants and amount.<br>**Input Data:** Expense details, participant list | |
|---|---|
| **Test Procedure** | **Expected Results** |
| User enters valid expense and all participants. | Expense created successfully. |
| Participant field left blank. | System highlights missing input and blocks submission. |

| **Test Case Identifier**: TC-8<br>**Use Case Tested:** UC-3<br>**Pass/Fail Criteria:** Test passes if the system correctly calculates split amounts.<br>**Input Data:** Total amount, participant shares | |
|---|---|
| **Test Procedure** | **Expected Results** |
| Equal split selected. | Each participant assigned correct share. |
| Custom ratio provided. | Custom amounts calculated correctly. |
| Division error (zero or invalid input). | System displays input error message. |

| **Test Case Identifier**: TC-9<br>**Use Case Tested:** UC-3<br>**Pass/Fail Criteria:** Test passes if updated expense data is stored correctly.<br>**Input Data:** Updated expense info | |
|---|---|
| **Test Procedure** | **Expected Results** |
| User edits expense details. | Database updates GroupExpenseModel and confirms change. |
| Invalid update attempted. | "Invalid entry" message displayed, and no change is made. |

### D. BudgetController

| **Test Case Identifier**: TC-10<br>**Use Case Tested:** UC-4<br>**Pass/Fail Criteria:** Test passes if budget data is retrieved and displayed correctly in the visualization view.<br>**Input Data:** UserID, exisitng budget categories | |
|---|---|
| **Test Procedure** | **Expected Results** |
| User opens "Budget Visualization" screen | App retrieves budgets from BudgetModel and displays tree/flow view |
| Budget data missing or empty | "No budget data available" message displayed |

| **Test Case Identifier**: TC-11<br>**Use Case Tested:** UC-4<br>**Pass/Fail Criteria:** Test passes if new budgets are created with valid limits.<br>**Input Data:** Category = "Needs", Limit = 500 | |
|---|---|
| **Test Procedure** | **Expected Results** |
| User enters valid budget and submits form | Budget created and saved in BudgetModel |
| User enters negative or invalid limit | System rejects entry and prompts for valid input |

| **Test Case Identifier**: TC-12<br>**Use Case Tested:** UC-4<br>**Pass/Fail Criteria:** Test passes if budget updates are correctly reflected in the visualization.<br>**Input Data:** Updated limit or category name | |
|---|---|
| **Test Procedure** | **Expected Results** |
| User updates an existing budget | Visualization refreshes to show new limit and totals |
| Database update fails | Error message displayed and logged in ExternalServiceLog |

### E. PaymentController

| **Test Case Identifier**: TC-13<br>**Use Case Tested:** UC-5<br>**Pass/Fail Criteria:** Test passes if payment routing successfully completes without storing user funds.<br>**Input Data:** Payment request with multiple linked accounts | |
|---|---|
| **Test Procedure** | **Expected Results** |
| User initiates a payment with multi-account routing | System routes payment using Stripe API without holding funds |
| One account lacks sufficient balance | Transaction declined with “Insufficient funds” message |

| **Test Case Identifier**: TC-14<br>**Use Case Tested:** UC-5<br>**Pass/Fail Criteria:** Test passes if routing errors are handled gracefully.<br>**Input Data:** Payment data with API timeout or connection loss | |
|---|---|
| **Test Procedure** | **Expected Results** |
| Stripe or Plaid API call fails | System retries or displays “Connection error, please try again |

| **Test Case Identifier**: TC-14<br>**Use Case Tested:** UC-5<br>**Pass/Fail Criteria:** Test passes if routing errors are handled gracefully.<br>**Input Data:** Payment data with API timeout or connection loss | |
|---|---|
| Multiple retries fail | System logs failure in ExternalServiceLog and notifies user |

| **Test Case Identifier**: TC-15<br>**Use Case Tested:** UC-6<br>**Pass/Fail Criteria:** Test passes if alert conditions correctly trigger notifications.<br>**Input Data:** Account balance < threshold | |
|---|---|
| **Test Procedure** | **Expected Results** |
| Balance drops below set limit | AlertModel creates low-balance alert and sends notification via FCM |
| Threshold not reached | No alert triggered |

### F. AlertController

| **Test Case Identifier**: TC-16<br>**Use Case Tested:** UC-6<br>**Pass/Fail Criteria:** Test passes if user can view and clear received alerts.<br>**Input Data:** Active alerts in AlertModel | |
|---|---|
| **Test Procedure** | **Expected Results** |
| User opens “Alerts” section | System retrieves and displays all active alerts |
| User marks an alert as resolved | Alert status updated to “resolved” |

### G. DashboardController

| **Test Case Identifier**: TC-17<br>**Use Case Tested:** UC-7<br>**Pass/Fail Criteria:** Test passes if transaction history is retrieved and displayed correctly for the specified time period.<br>**Input Data: u**serID, date range filter | |
|---|---|
| **Test Procedure** | **Expected Results** |
| User opens "Transaction History" screen | System retrieves all transactions from TransactionModel and displays chronologically |
| User applies date filter (e.g., last 30 days) | System filters transactions within specified range and updates display |
| No transactions found for selected period | "No transactions found" message displayed |

| **Test Case Identifier**: TC-18<br>**Use Case Tested:** UC-7<br>**Pass/Fail Criteria:** Test passes if transaction details are displayed accurately when selected.<br>**Input Data: t**ransactionID | |
|---|---|
| **Test Procedure** | **Expected Results** |
| User selects a specific transaction from list | System displays complete transaction details including date, amount, category, and recipient |
| Transaction contains group expense data | System shows split details and participant information |
| Database query fails | Error message displayed and logged in ExternalServiceLog |

| **Test Case Identifier**: TC-19<br>**Use Case Tested:** UC-7<br>**Pass/Fail Criteria:** Test passes if transaction search and filter functionality works correctly.<br>**Input Data:** Search keyword, filter criteria (category, amount range) | |
|---|---|
| **Test Procedure** | **Expected Results** |
| User enters search term (e.g., merchant name) | System filters transactions matching search criteria and displays results |
| User applies multiple filters simultaneously | System combines filters and displays only matching transactions |
| Invalid filter parameters entered | System prompts user to enter valid criteria |

### H. FavoriteController

| **Test Case Identifier**: TC-20<br>**Use Case Tested:** UC-8<br>**Pass/Fail Criteria:** Test passes if user can successfully add another user to favorites list.<br>**Input Data:** userID or username to add as favorite | |
|---|---|
| **Test Procedure** | **Expected Results** |
| User searches for another user by username | System displays matching users from UserModel |
| User selects "Add to Favorites" for a user | System saves favorite to FavoriteModel and displays confirmation message |
| User attempts to add duplicate favorite | System displays "User already in favorites" message |

| **Test Case Identifier**: TC-21<br>**Use Case Tested:** UC-8<br>**Pass/Fail Criteria:** Test passes if user can successfully remove a user from favorites list.<br>**Input Data:** favoriteID | |
|---|---|
| **Test Procedure** | **Expected Results** |

| **Test Case Identifier**: TC-21<br>**Use Case Tested:** UC-8<br>**Pass/Fail Criteria:** Test passes if user can successfully remove a user from favorites list.<br>**Input Data:** favoriteID | |
|---|---|
| User selects "Remove from Favorites" for a user | System removes favorite from FavoriteModel and updates display |
| User confirms removal action | Favorite successfully deleted and confirmation message shown |
| Removal failed due to database error | Error message displayed and logged in ExternalServiceLog |

| **Test Case Identifier**: TC-22<br>**Use Case Tested:** UC-8<br>**Pass/Fail Criteria:** Test passes if favorites list is displayed correctly and can be accessed.<br>**Input Data:** userID | |
|---|---|
| **Test Procedure** | **Expected Results** |
| User opens "Favorites" page | System retrieves all favorites from FavoriteModel and displays list with user details |
| User has no favorites | "No favorites added yet" message displayed |
| Database query fails | Error message displayed and logged in ExternalServiceLog |

### I. DashboardController

| **Test Case Identifier**: TC-23<br>**Use Case Tested:** UC-9<br>**Pass/Fail Criteria:** Test passes if spending report is generated correctly with accurate calculations.<br>**Input Data:** userID, report period (weekly/monthly/yearly) | |
|---|---|
| **Test Procedure** | **Expected Results** |

| **Test Case Identifier**: TC-23<br>**Use Case Tested:** UC-9<br>**Pass/Fail Criteria:** Test passes if spending report is generated correctly with accurate calculations.<br>**Input Data:** userID, report period (weekly/monthly/yearly) | |
|---|---|
| User requests monthly spending report | System aggregates transactions by category and generates visual reports with charts |
| User selects custom date range for report | System calculates spending for specified period and displays breakdown |
| No transactions exist for selected period | System displays "No data available for this period" message |

| **Test Case Identifier**: TC-24<br>**Use Case Tested:** UC-9<br>**Pass/Fail Criteria:** Test passes if spending categories are correctly aggregated and visualized.<br>**Input Data:** Transaction data with categories | |
|---|---|
| **Test Procedure** | **Expected Results** |
| Report generated with multiple categories | System displays pie charts or bar graphs showing spending distribution across categories |
| User drills down into specific category | System shows detailed transaction list for that category |
| Visualization rendering fails | System falls back to table view and displays error message |

### J. ProfileController

**Test Case Identifier**: TC-25

**Use Case Tested:** UC-9

**Pass/Fail Criteria:** Test passes if profile information updates are saved and displayed correctly.

**Input Data:** Updated username, phoneNumber, email

| **Test Procedure** | **Expected Results** |
|---|---|
| User updates username in profile settings | System saves new username to UserModel and displays updated name throughout app |
| User updates phone number with valid format | System validates format, saved to database, and displays new phone number in profile |
| User updates email with invalid format | System rejects input and displays "Please enter valid email address" |
| User updates multiple fields simultaneously | All valid changes saved to UserModel and reflected immediately in profile view |

## II. Test Coverage

The test cases developed by our team comprehensively validate the functionality of all critical controllers and core workflows within our financial management system. Our test suite demonstrates strong coverage of essential user operations including bank account linking, QR payment processing, and group expense management, the three pillars of our application value proposition. These tests are designed with future extensibility in mind, and we remain committed to expanding our test coverage as we implement additional features and integrate stretch goals into our product roadmap.

Our testing philosophy emphasizes functional verification and user-centric scenarios. We have strategically designed test cases that mirror real-world usage patterns, ensuring that each controller correctly handles both successful operations and predictable failure modes. The test cases validate critical integration points with external services like Plaid, confirm proper data persistence across our database models, and verify that user-facing error messages provide clear, actionable feedback. By testing complete user journeys from initiation through completion, we ensure that our system delivers a cohesive, reliable experience across all primary use cases.

Each test case incorporates clear pass/fail criteria with well-defined input data and expected outcomes, making it straightforward for any team member to execute tests and interpret results. This clarity is particularly valuable as we continue to iterate on our codebase. When changes are made to core logic, our test suite provides immediate feedback about whether modifications have introduced regressions. Our tests cover essential error recovery mechanisms, including network failure handling, invalid input validation, and graceful degradation when external APIs experience issues, which are critical for building user trust in a financial application.

We have adopted a systematic integration testing approach that validates interactions between controllers, models, and external services. By testing not just individual functions in isolation but also their coordinated behavior within complete workflows, we can identify integration issues early in the development cycle. This methodology reduces the risk of cascading failures and ensures that components work harmoniously together. Our test cases validate database transactions, API responses, and state management across the full stack, providing confidence that our system maintains data consistency and integrity throughout all operations.

The structure of our test suite supports continuous improvement and rapid debugging. When issues arise, our granular test cases help pinpoint exactly which component or interaction is failing,

dramatically reducing troubleshooting time. As we move forward, we will expand coverage to include additional edge cases, performance benchmarks, and security validations, building upon the solid foundation we have established. Our current test suite strikes an excellent balance between comprehensive coverage of core functionality and practical maintainability, positioning us well for sustainable growth as our application evolves.

This testing framework demonstrates our commitment to delivering a robust, production-ready financial platform that users can depend on for managing their banking, payments, and shared expenses. By validating critical paths, error handling, and data integrity from the outset, we are building quality into every layer of our system rather than treating it as an afterthought. Our test-driven approach minimizes the likelihood of defects reaching end users and provides a safety net that enables our team to develop new features with confidence. As we continue refining our application, these tests will serve as both guardrails and documentation, ensuring that our system remains reliable, maintainable, and ready to scale.

## III. Integration Testing

Our test integration will be a bottom-up integration style. We will start by testing the lower-level units like Models and Services (e.g., UserModel, TransactionModel, BudgetModel, and PlaidAdapter) before progressing towards the top, namely the Controllers and then the mobile app level.

This method is best suited for QuickPay because the development team can ensure that data access and service integrations (like Supabase, Plaid, Stripe, and Firebase Cloud Messaging) work properly before integrating them into deeper workflows. Testing units on the lowest level first facilitates the discovery of bugs in low-level components that are vital in aiding upper-level operations like payments, budgeting, and notifications earlier.

After the low-level units have been tested, we will incrementally incorporate and test the Controllers (e.g., PaymentController, AlertController, DashboardController). They control activities on multiple services and models, thus making them reliant on the favorable run of the low-level units.

Last is the integration testing stage, which will incorporate the React Native client, ensuring end-to-end whole functionality - from authenticating users and retrieving data through making payments and handling notifications. By applying the bottom-up method, we reduce errors that cumulatively occur, make debugging easy, and ensure that all the system's layers work correctly before integration on the upper levels.

This structured approach supports QuickPay’s modular design and reduces the risk of cascading failures during testing, allowing for efficient and reliable verification of system behavior across all tiers.

## IV. System Testing

After our individual units are thoroughly tested through unit testing and bottom-up integration testing, any modifications necessary to fully integrate the components will be completed. Once all components pass unit and integration tests, the focus will move to system testing. These types of tests will include end-to-end testing, installation testing, deployment testing, and API testing. Our overall goal is to ensure the reliability, security, and smooth functioning of the QuickPay mobile application.

For installation testing, we will test our React Native application using the iOS emulator during development. We will verify that the user interface renders correctly across different screen sizes and that all features, including the dashboard, budget visualization, QR scanner, and transaction history, function smoothly on iOS devices.

For API testing, we will utilize Postman to test all backend endpoints and external service integrations. We will verify that our Supabase Cloud Functions, Plaid API integration, Stripe payment processing, and Firebase Cloud Messaging all respond correctly to various request scenarios. Using Postman, we will test authentication flows, transaction creation, budget updates, and alert triggering to ensure proper request handling, validation, and error responses. Any issues with API endpoints, authentication tokens, or data formatting will be identified and resolved through iterative testing in Postman before deployment.

For deployment testing, we will deploy our application locally for initial testing. We will ensure that all dependencies including Supabase SDK, Plaid SDK, Stripe SDK, and React Native libraries are properly configured and bundled for iOS. We will test our backend deployment through Supabase Cloud Functions, monitoring logs to confirm that all serverless functions deploy successfully and respond correctly to API requests from the mobile application.

Finally, we will confirm that our application is functional and operates reliably from end-to-end across all major workflows on iOS devices. This will include testing the complete user onboarding process from registration through bank account linking, the payment flow from QR code scanning through transaction processing, the group expense workflow from creation through settlement, and budget management from creation through alert triggering.

*Project Management and Plan of Work*

## I. Project Development Progress

### K. Merging Contributions from Individual Team Members

To ensure smooth project progress, our group holds in-person meetings daily from 7:00 to 9:30 PM, Sunday through Thursday, and from 3:30 to 6:00 PM on Saturdays. This meeting enabled us to consistently be in sync with each other on which path the project should take. During the first week we were able to set up our shared documents such as Microsoft word share point, GitHub repo and Gitlab repo. Other than shared documents we were able to come to agreement on our project's framework, API, database system and functionality.

We were able to make projects flow smoothly with our shared documents that we created with Microsoft word share point. We prefer to use Microsoft word share point instead of googling docs because Microsoft word share point has better formatting features, flexibility and better table formatting.

With Microsoft Words share point we were able to work collaboratively while also being able to review each other's work. At the end of each meeting each member will review each section and note down suggested changes to talk about at the next meeting. Even with the group working well with each other and being able to come to an agreement with the concepts, logic and design behind our system and report sections, we met some hiccups such as the formatting of our documents such as headings, and formatting inconsistencies with our writing style. The final step of our project should be reviewing the documents to make sure that the headings, tables and each section fit Dr Michael's standards for a project report.

## II. Project Coordination and Progress Report

### L. Report Progress Report

Currently, we are planning the implementation of individual components that will contribute to each use case. The following components are planned for development:

| Models | Controllers | UI Components (Completed) |
|---|---|---|
| • UserModel<br>• BankAccountModel<br>• TransactionModel<br>• BudgetModel<br>• AlertModel<br>• GroupExpenseModel<br>• FavoriteModel<br>• ExternalServiceLogModel | • PaymentController<br>• AlertController<br>• GroupExpenseController<br>• ProfileController<br>• FavoriteController<br>• DashboardController | • Home Page<br>• Budget Visualization Page<br>• QR Code Scanner Page<br>• Favorites Page<br>• Profile Page<br>• Profile Page |

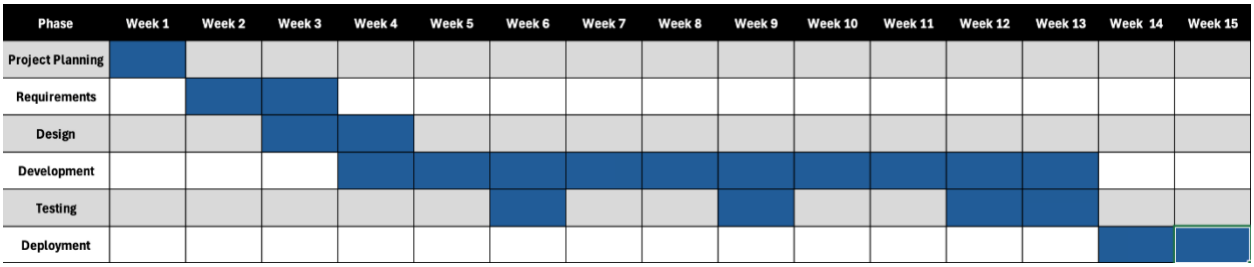
We have established our UI design on Figma and completed the frontend development with our Expo React Native framework. We are currently versioning our work via GitHub and setting up our development environment with weekly meetings as usual. We have researched and identified all the external APIs we will be using, including Plaid for bank account linking, Stripe for payment processing, and Firebase Cloud Messaging for push notifications. We are now preparing to test connectivity to these APIs and establish our Supabase database structure.

## III. History of Work

**Key accomplishments:**

- Migrated authentication from Firebase Authentication to Clerk, enabling secure MFA, session management, and modern user identity handling.
- Migrated database from Firebase Realtime Database (NoSQL) to Supabase PostgreSQL.
- Developed core services:
    - Supabase Service for SQL operations and real-time sync
    - Clerk Auth Service for authentication and user sessions
- Integrated external APIs (Plaid, Stripe, Supabase, Clerk) for secure linking, transactions, and data flow.
- Implemented ExternalServiceLog model to track and monitor all API and service interactions.
- Delivered all project milestones with updated UML diagrams, sequence diagrams, and traceability matrices.

### A. Project Timeline




### B. Actual Work Timeline

| Task Name | Plan Duration | Actual Start | Actual End | Actual Duration |
|---|---|---|---|---|
| **Requirements and Documentation** | | | | |
| Project Proposal | 7 | 08/25/25 | 08/29/25 | 5 |
| Report #1 Part 1 | 7 | 09/03/25 | 09/06/25 | 4 |
| Report #1 Part 2 | 7 | 09/08/25 | 09/14/25 | 7 |
| Report #1 Part 3 - Full Report | 7 | 09/15/25 | 09/21/25 | 7 |

| Task Name | Plan Duration | Actual Start | Actual End | Actual Duration |
|---|---|---|---|---|
| Report #2 Part 1 | 7 | 09/22/25 | 09/28/25 | 7 |
| Report #2 Part 2 | 7 | 09/29/25 | 10/05/25 | 7 |
| Report #2 Part 3 – Full Report | 7 | 10/06/25 | 10/12/25 | 7 |
| Report #3 – Full Final Report | 7 | 11/17/25 | 11/23/25 | 7 |
| **Design** | | | | |
| UI/UX Design on Figma | 7 | 09/03/25 | 09/06/25 | 4 |
| Database Design | 15 | 10/13/25 | 11/03/25 | 21 |
| **Development** | | | | |
| Frontend Development on Expo | 30 | 09/10/25 | 10/10/25 | 30 |
| Backend Development | 40 | 10/13/25 | 11/30/25 | 48 |
| **Testing** | | | | |
| Unit Testing | 35 | 10/20/25 | 11/29/25 | 40 |
| Integration Testing | 30 | 10/22/25 | 11/30/25 | 40 |
| System Testing | 5 | 11/26/25 | 12/01/25 | 5 |
| **Deployment** | | | | |
| Local Deployment Setup | 3 | 10/20/25 | 11/19/25 | 30 |

## IV. Project Report Progress

In the second week, our group worked on the project proposal. This included writing the problem statement, project goals, and system ideas. The proposal gave us a shared understanding and helped set a clear direction for the project.

In the third week, we started Report 1, Part 1. This part included the cover page, table of contents, work assignment, and customer problem statement. Each member worked on their assigned section, and we later reviewed everything together to maintain consistent formatting and style.

In the fourth week, we completed Report 1, Part 2, which covered the functional requirement specification with use cases, actors and goals, use case diagrams, traceability matrix, fully dressed descriptions, system sequence diagrams, and user interface specifications. We continued our collaborative approach, with each member contributing to different sections and conducting thorough group reviews to ensure consistency.

In the fifth week, we completed Report 1, Part 3, focusing on system architecture, subsystem identification, connectors and network protocols, hardware requirements, and project management planning. As before, we divided responsibilities among members and conducted collective reviews to ensure accuracy, consistency, and alignment with our overall project direction.

In the sixth week, we successfully completed Report 1 in its entirety and began working on Report 2, Part 1. This phase focused on advancing our analysis and domain modeling components, including detailed conceptual model development, comprehensive system operation contracts, refined traceability matrices, data model specifications, and the establishment of mathematical frameworks guiding our system's core algorithms. Each team member contributed specialized expertise while maintaining our collaborative review process to ensure technical precision and consistency.

In the seventh week, we completed Report 2, Part 2, which marked another significant milestone in our project progress. This section focused on refining the system design, integrating updated

models, and ensuring alignment between our analytical framework and the forthcoming implementation phase. Our team maintained its systematic approach of dividing tasks and conducting group reviews, which helped ensure the quality, completeness, and coherence of our deliverables.

In the eighth week, we completed Report 2, Part 3, which is the full Report 2. This comprehensive report covered algorithms, data structures, concurrency, test designs including unit testing, integration testing, and system testing, as well as the responsibilities breakdown for the implementation phase. With Report 2 finalized, our focus has now shifted to active development.

In the fourteenth week, we completed Report 3, which serves as the final report. We reviewed the entire document to update any sections necessary to reflect our current project status. Additionally, we added a Summary of Changes, expanded the Design Patterns section, and made changes to History of Work, Current Status, and Future Work.

So far, we have successfully completed the project proposal, Report 1 (Parts 1, 2, and 3), Report 2 (Parts 1, 2, and 3), and Final Report 3. With all these stages finalized, our group is now fully engaged in the implementation phase, focusing primarily on backend development including API integrations, database setup, controller implementation, and testing. We are also making improvements to the frontend as needed to ensure seamless integration with the backend services.

## V. Breakdown of Responsibilities

The following table outlines the distribution of tasks and responsibilities among team members for the QuickPay project. Each team member will mark an "x" under their name to indicate which components they will be responsible for developing, implementing, and testing.

| Responsibilities | Chanrattnak Mong | Seanglong Lim | Seth Tharo Hour | Sok Sreng Chan |
|---|---|---|---|---|
| **Models** | | | | |
| UserModel | x | | | |
| BankAccountModel | | | x | |
| TransactionModel | x | | | |
| BudgetModel | x | | | |
| AlertModel | | | | x |
| GroupExpenseModel | | | | x |
| FavoriteModel | | x | | |
| ExternalServiceLogModel | | | x | |
| **Controllers** | | | | |
| PaymentController | x | | | |
| AlertController | | | x | |
| GroupExpenseController | | | x | |
| ProfileController | | x | | |
| FavoriteController | | x | | |
| DashboardController | x | | | |
| **APIs Integration** | | | | |
| Plaid API Integration | | x | | |
| Stripe API Integration | | | x | |
| Supabase Database | x | | | |
| Firebase Cloud Messaging | | x | | x |
| Clerk Authentication | x | | x | x |
| **Testing** | | | | |
| Unit Testing | x | x | x | x |

| Responsibilities | Chanrattnak Mong | Seanglong Lim | Seth Tharo Hour | Sok Sreng Chan |
|---|---|---|---|---|
| Integration Testing | x | | x | |
| System Testing | x | | | |
| **Deployment** | | | | |
| Local Development Setup | x | | | x |
| iOS Build Configuration | x | | | x |

This responsibility breakdown will be reviewed and updated regularly during our team meetings to ensure balanced workload distribution and to address any overlapping or dependent tasks. Team members are encouraged to collaborate and assist each other when needed, particularly on components that require integration between frontend and backend systems. Clear communication and coordination will be maintained throughout the development process to ensure all components work together seamlessly.

## VI. Current Status

Our QuickPay app is now able to securely link multiple banks through Plaid, processing real-time transactions via Stripe, and storing user data through Supabase. As for Authentication and session management are handled seamlessly by Clerk, ensuring both usability and security. The app provides an intuitive user interface with smooth navigation, visual budgeting tools, and fast payment processing.

Our team is proud to have successfully implemented all primary use cases defined for QuickPay. Although a few secondary requirements were deferred due to project timeline constraints (such as advanced analytics and recurring payment automation), the core system functionalities, authentication, multi-account linking, budgeting visualization, QR-based payments, and group expense management, were all fully implemented and tested by the time of our final submission.

We are also satisfied with the technical and design progress achieved in a short timeframe and confident that QuickPay demonstrates both strong engineering depth and user-centered design. Future improvements, such as automated recurring payments, machine learning–based spending insights, and expanded analytics dashboards, are outlined in the next section.

## VII. Future Work

The following features represent logical next steps for QuickPay's continued development. These enhancements build upon the existing architecture and would further improve user experience, security, and financial management capabilities. Priority should be given to UC-6 (Real-Time Alerts) and UC-8 (Enhanced Favorites), as this use cases were designed during the initial planning phase but deferred for post-launch implementation.

### C. Real-Time Alert System (UC-6)

- Description: Implement automated alerts for low balances, upcoming bills, and budget thresholds
- Requirements Addressed: REQ-7, REQ-12, REQ-13, REQ-24
- Technical Implementation: Integrate Expo Push Notifications with scheduled background tasks to monitor account balances and trigger alerts based on user-defined thresholds
- Benefits: Proactive financial management, reduced overdraft risk, improved user engagement

### D. Add Users as Favorites (UC-8)

- Description: Allow users to add other QuickPay users to their favorites list for quick and easy payment access
- Requirements Addressed: REQ-8, REQ-12, REQ-13, REQ-23
- Technical Implementation: Enhance FavoriteController and FavoriteModel to support adding, removing, and retrieving favorite contacts with optional nicknames and custom labels for personalized organization
- Benefits: Streamlined payment experience for frequent transactions, reduced input errors, faster recipient selection for regular payments, improved user efficiency

### E. Recurring Payment Management

- Description: Allow users to set up and manage recurring payments with automatic scheduling for rent, utilities, subscriptions, and other regular expenses
- Requirements Addressed: REQ-2, REQ-4, REQ-11, REQ-12, REQ-13, REQ-20
- Features:
    - Schedule recurring transfers between accounts
    - Automatic bill payments

    - Subscription tracking and management
    - Payment reminders and confirmations
- Technical Implementation: Extend TransactionController and BudgetModel to support scheduled transactions with cron-based background jobs, integrate with Plaid API for automatic bill detection, and implement payment scheduling logic with retry mechanisms.
- Benefits: Eliminates manual payment tasks, prevents late payment fees, ensures timely bill payments, reduces cognitive load for users managing multiple recurring expenses

**F. Enhanced Group Expense Features**

- Description: Expand group expense capabilities beyond simple splitting to support complex expense scenarios and improve collaboration
- Requirements Addressed: REQ-2, REQ-6, REQ-11, REQ-12, REQ-13, REQ-20
- Features:
    - Unequal split options (percentage-based or custom amounts)
    - Expense categories within group expenses
    - Group expense history and analytics
    - Settlement suggestions (who owes whom)
- Technical Implementation: Enhance GroupExpenseController and GroupExpenseModel to support flexible split algorithms, add settlement calculation logic, integrate expense categorization, and implement real-time expense tracking with Supabase real-time subscriptions
- Benefits: Handles real-world expense scenarios more accurately, simplifies complex group financial management, reduces confusion about who owes what, improves transparency in shared expenses

**G. Advanced Security Features**

- Description: Implement additional security layers for financial protection including biometric authentication, transaction verification, activity monitoring, and device management
- Requirements Addressed: REQ-12, REQ-13, REQ-16, REQ-17, REQ-19
- Features:
    - Biometric authentication (fingerprint, Face ID)
    - Transaction verification for large payments

    - Suspicious activity detection
    - Temporary account freeze capability
    - Device authorization management
- Technical Implementation: Integrate React Native biometric libraries (expo-local-authentication) for fingerprint and Face ID support, implement transaction amount thresholds in PaymentController with additional verification steps, add anomaly detection algorithms in TransactionModel to flag unusual activity patterns, extend AuthController to track authorized devices with session management, create account freeze functionality in UserController with immediate session revocation, and integrate Expo Push Notifications for security alerts
- Benefits: Enhanced account security, reduced fraud risk, user control over account access during emergencies, protection against unauthorized large transactions, device-level security management, quick response to compromised accounts, peace of mind for users handling sensitive financial data

## *References*


1. Clerk: https://clerk.com/docs
2. Firebase Cloud Messaging: https://firebase.google.com/docs/cloud-messaging
3. PLAID: https://plaid.com/docs/
4. Stripe: https://docs.stripe.com/
5. Supabase: https://supabase.com/docs