Software Design Document (SDD)

Software Architecture Design

Introduction

The System Design Document (SDD) is created to explain the general design of the system and how all the main parts are arranged together. This document is based on the SRS we already finished, and it helps us move from understanding the requirements to actually planning how the system will work in a clear and organized way. The idea of the SDD is not to go into too much detail, but to give a complete picture of the structure of the system before starting any development.

In this document, we describe how the system is divided into different components and how these components will interact with each other during the system's operation. This includes showing the architecture model, describing the main diagrams, and explaining how the components fit together to support the features we need in the project. The SDD helps the team have the same understanding about the system and reduces confusion later during design or implementation.

Since our project uses the Agile (Scrum) method, the design is not fixed from the beginning. It can be improved or changed while we work on the sprints. This makes the design more flexible, because sometimes new things appear while working, or we find better ways to design some parts of the system. So, the SDD is not a final version, but more like a guide that we can update whenever the sprint reviews show something important.

The SDD contains several main sections. One of them is the system architecture model, which explains the layers of the system and how the application will be divided (for example, the user interface layer, the logic layer, and the storage layer). It also includes the main UML diagrams that show how the user interacts with the system and how the system handles different processes, such as adding income or recording expenses. Another important part of the SDD is the class diagram, which shows the main classes in our system, their attributes, methods, and the relationships between them.

The main purpose of this document is to provide a clear understanding of how the system is structured, so that the team can move smoothly into the design and implementation stages. It also helps make sure the system design matches the requirements we wrote earlier and supports the functions we want to include in our budget and expense management application.

Software Architecture Design

Requirement 1: Add Monthly Income

Description:
The system shall allow users to record and store their monthly income entries with details such as amount, source, and date.

Input:
· Income amount (decimal)
· Source description (optional)
· Date received
· Frequency (monthly, weekly, one-time)

Output:
· Updated total income calculation
· Updated dashboard display
· Stored income record in database

Pre-conditions:
· User must be logged into the system
· Valid positive amount must be entered

Post-conditions:
· Income record is saved to local storage
· Dashboard reflects new total income
· Available budget is recalculated

Operational Considerations:
Income entries are validated for positive values and reasonable amounts to prevent data entry errors.

Requirement 2: Record and Classify Expenses

Description:
The system shall allow users to record expenses with categorization and detailed information.

Input:
· Expense amount
· Category selection
· Date of expense
· Optional notes
· Payment method

Output:
· Updated expense total
· Category-wise expense tracking
· Budget balance update

Pre-conditions:
· User must be logged in
· Valid category must be selected
· Amount must be positive

Post-conditions:
· Expense record is stored
· Remaining budget is updated
· Category totals are recalculated

Side Effects:
· May trigger budget warning if expense exceeds remaining budget
· May affect savings goal progress calculations

Requirement 3: Display Financial Dashboard

Description:
The system shall present a comprehensive dashboard showing total income, total expenses, and remaining budget.

Input:
· Aggregated income data
· Aggregated expense data
· Time period selection

Output:
· Visual summary of finances
· Charts/graphs representation
· Quick financial status overview

Pre-conditions:
· User must be logged in
· System must have financial data to display

Post-conditions:
· User can view current financial status
· Dashboard is updated with latest calculations

Operational Considerations:
Dashboard calculations must be efficient to provide instant feedback without noticeable delay.

Requirement 5: Create and Manage Savings Goals

Description:
The system shall allow users to set, track, and manage savings goals with target amounts and deadlines.

Input:
· Goal name and description
· Target amount
· Deadline date
· Current saved amount

Output:
· Goal progress visualization
· Time-based tracking
· Achievement notifications

Pre-conditions:
· Target amount must be positive
· Deadline must be in the future

Post-conditions:
· Goal is saved and tracked
· Progress is calculated automatically

Operational Considerations:
Goal tracking must update automatically with each income/expense transaction.

Requirement 6: Transaction History Management

Description:
The system shall maintain and display a complete history of all financial transactions with filtering and search capabilities.

Input:
· Date range filters
· Category filters
· Search keywords

Output:
· Filtered transaction list
· Detailed transaction view
· Edit/delete options

Pre-conditions:
· User must have transaction history
· Valid filter criteria

Post-conditions:
· User can review and manage past transactions
· Changes are reflected in financial totals

Requirement 7: Edit/Delete Past Transactions

Description:
The system shall allow users to modify or remove previously recorded transactions.

Input:
· Selected transaction for editing
· Updated transaction details
· Delete confirmation

Output:
· Updated transaction record
· Recalculated financial totals
· Confirmation message

Pre-conditions:
· Transaction must exist
· User must have permission to modify

Post-conditions:
· Financial totals are updated
· All related calculations are refreshed

Side Effects:
· May affect historical reports and trend analysis

Requirement 8: Non-Functional Requirements Implementation

Description:
The system shall meet all specified non-functional requirements including usability, performance, security, and aesthetics.

Implementation Strategy:
· Usability: Intuitive interface with minimal learning curve
· Performance: Response time under 3 seconds for all operations
· Security: Local data storage with encryption where applicable
· Aesthetics: Clean, uncluttered interface with consistent design

Operational Considerations:
All non-functional requirements are treated as critical success factors and are measured throughout development.

Process Model And Activities

Selected Software Process Activity Method and Rationale

Selected Method: Agile (Scrum Methodology)
The Agile Methodology, through (Scrum) is selected as the process framework for developing the personal budget and expense calculator. The method is selected because of the project need for frequent refinements of the interface, as well as the evolving nature of the project.

Rationale for Selecting the Method:

1- Frequent User Input and Feedback: The project requires frequent input from users to help ensure complex features (like budget management and saving goals) are useful and responsive to actual user needs; the Scrum framework supports frequent interaction each sprint that enables changes immediately.

2-Flexibility and Ability to Adapt When Requirements Change: In an ongoing project, user requirements are subject to change (for example, the addition of new categories or features, and refinement of the budgeting logic). The Scrum methodology supports changing requirements without a disruption to the core development activity.

3-Incremental/Incremental Delivery: Scrum organizes project activities to allow delivery of working portions of the system in short delivery cycles (sprints). This supports early delivery of key features, like expense tracking, though more complex feature will be inclementally delivered, in lo sectinos to avoid risk.

4-Transparency and Ownership: The Scrum process - framework idea and transparency is alive through Sprint Review and Retrospective events.

Scrum Implementation and Documentation Information In order to illustrate the use of this model, the System Design Document (SDD) will address the implementation of Agile (Scrum):

1- Software Process Activities:

Development will occur as described around the following scrum ceremonies:

- Sprint Planning: Determine which features (e.g., expense tracking and savings goals) will get prioritized and then broken into smaller tasks for development within that sprint.
- Daily Stand-Ups: Daily short meetings to continually communicate about progress being made as well as quickly solve any issues.
- Sprint Review: Once part of the work is completed, the work will be demonstrated to elicit feedback for improvements in the future.
- Sprint Retrospective: The team will reflect on workflow and process to enhance efficiency as well as strategy in the next sprint.

2- SDD Documentation Considerations:

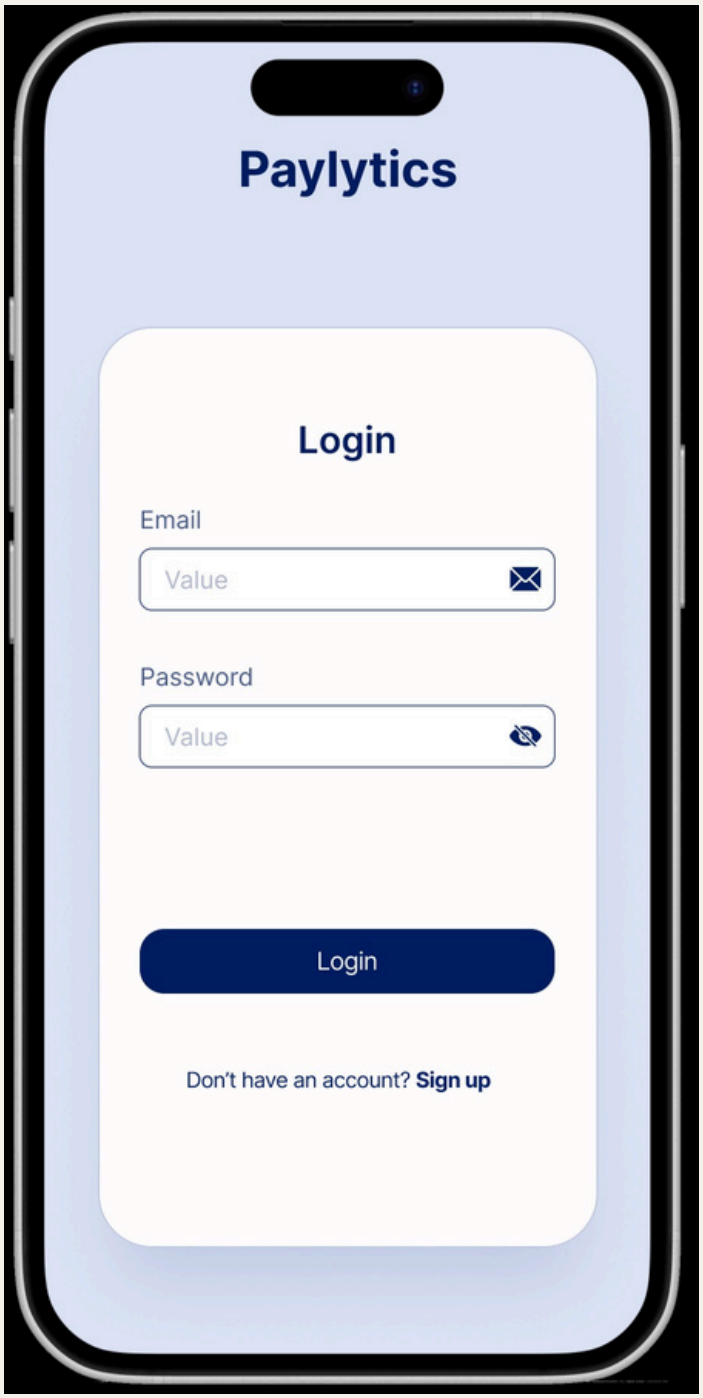
The SDD will focus on describing the Scrum practice more specifically:

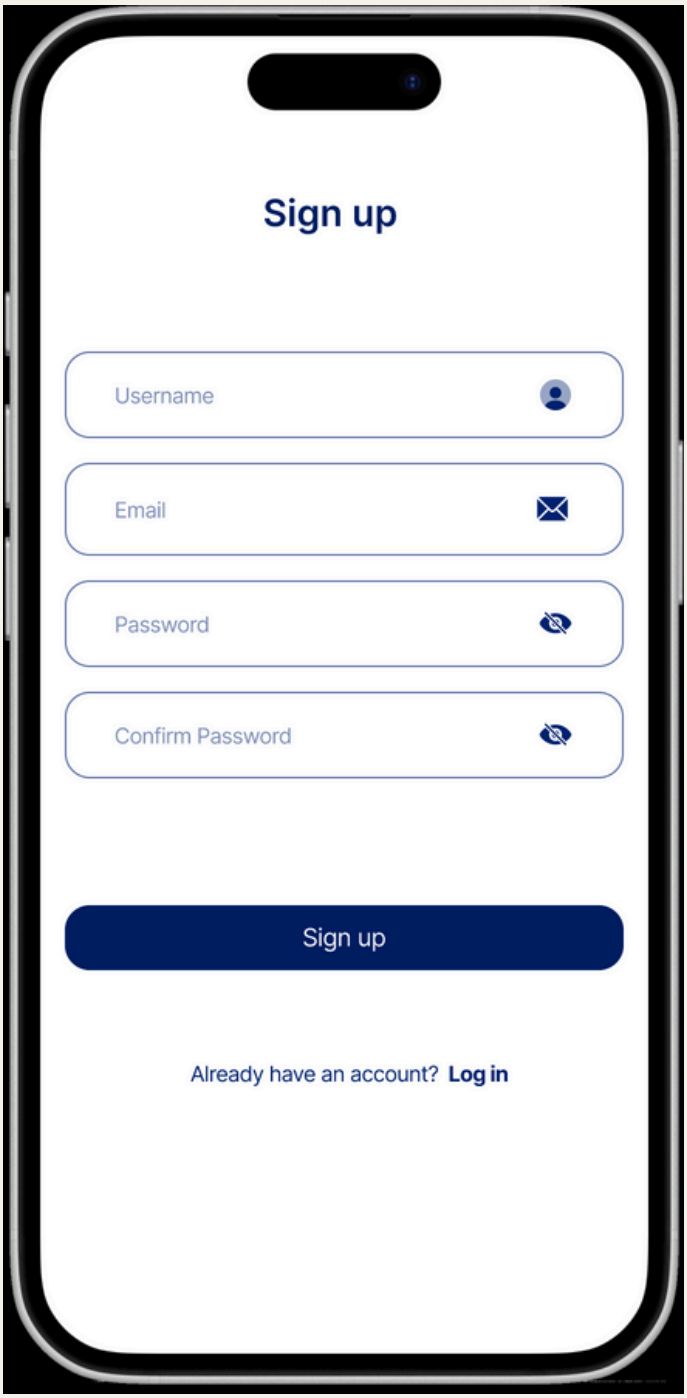
- Breakdown of Sprint: How the project is organized into sprints, each part addressing specific and deliverable feature sets.
- Task Priorization: Criteria for prioritizing features and tasks for each sprint iterations.
- User Input: The way user input will be formally collected and addressed from the previous iteration into the Agile type for the next iteration reflecting the fact that it is an iterative process, as well as continuous improvement.

1- Software Process Activities:

SPRINT	CONTRIBUTION
SPRINT 1	*Project setup, requirements gathering, basic Ul design*
SPRINT 2	*Core architecture, database design, basic navigation*
SPRINT 3	*Income/Expense recording, dashboard implementation*
SPRINT 4	*Reporting features, savings goals functionality*
SPRINT 5	*Advanced features, testing, documentation*
SPRINT 6	*Final integration, bug fixes, presentation preparation*

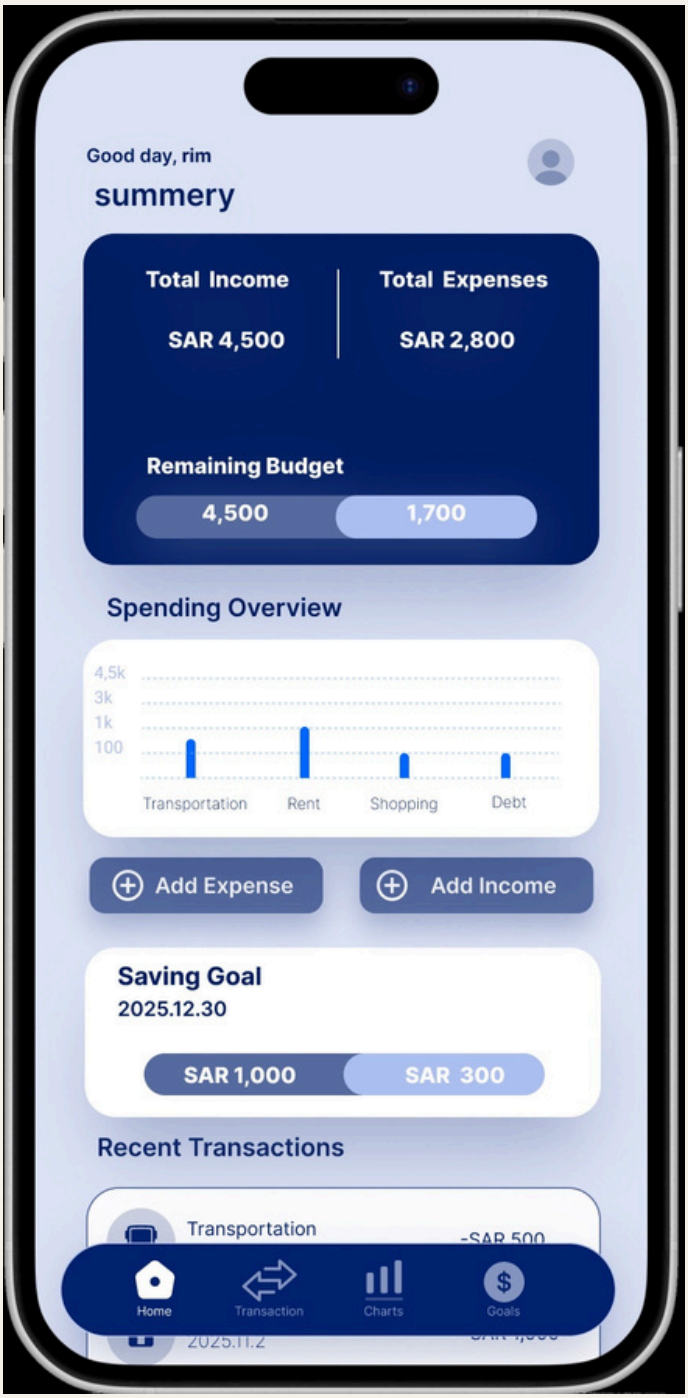
Interface application

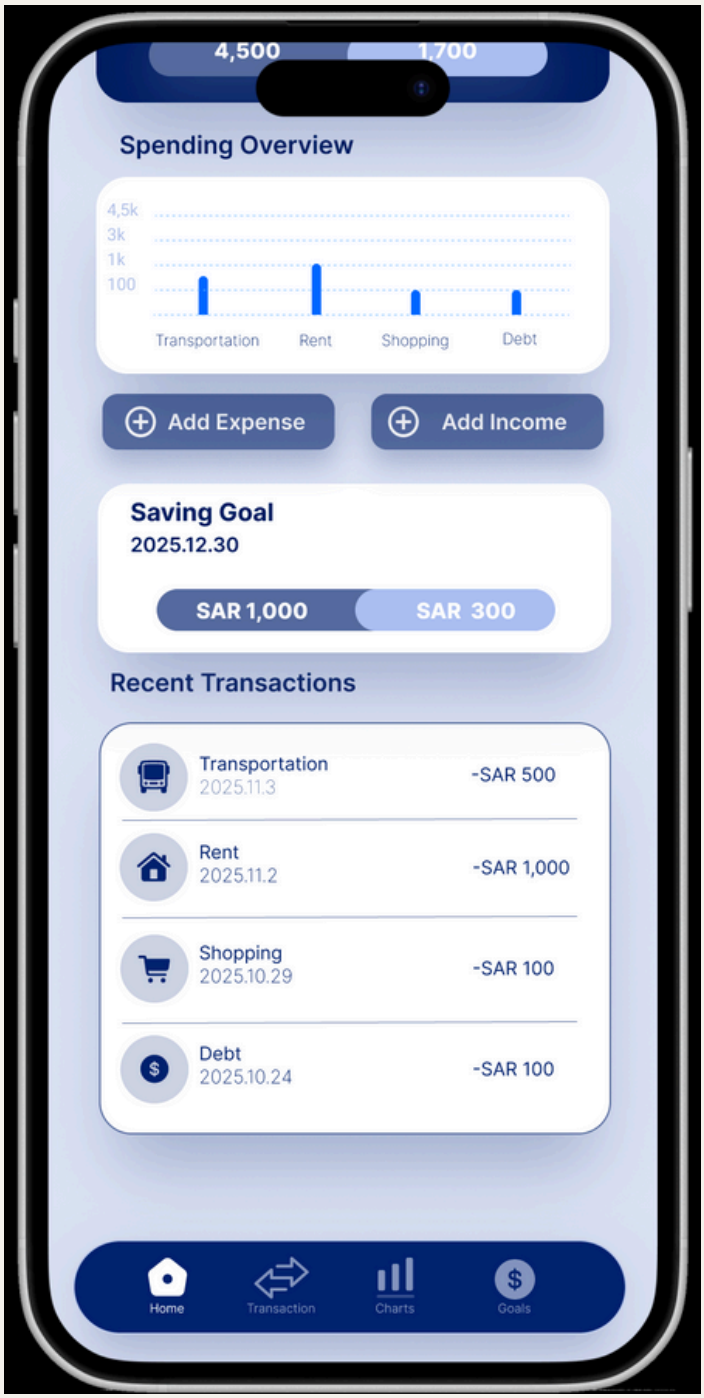




Clear pictures here :

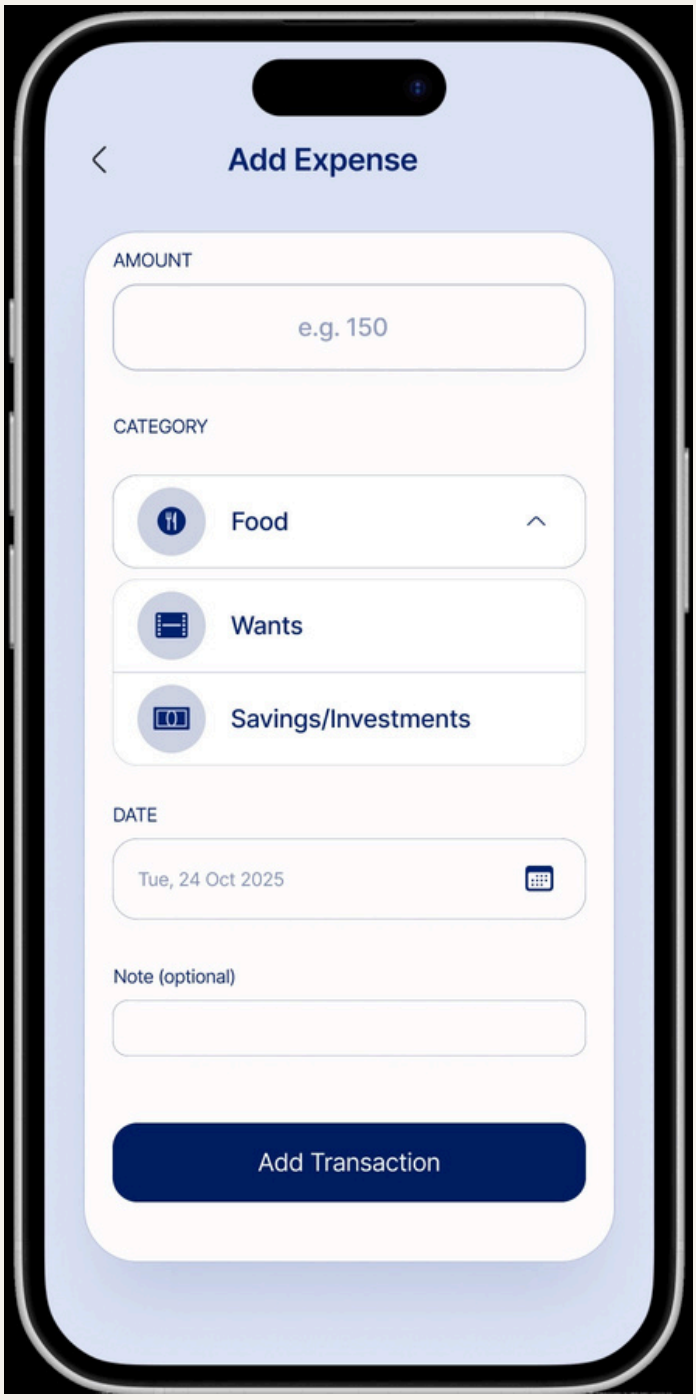
Interface application

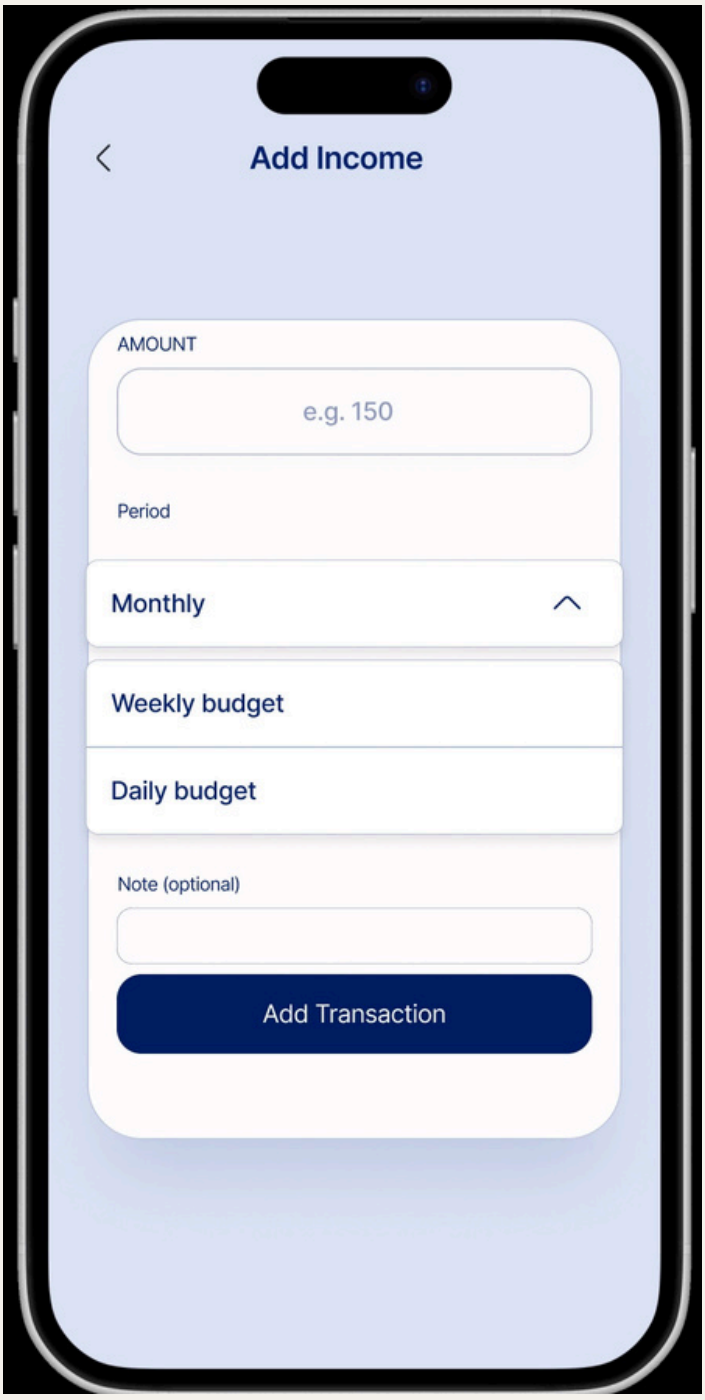




Clear pictures here :

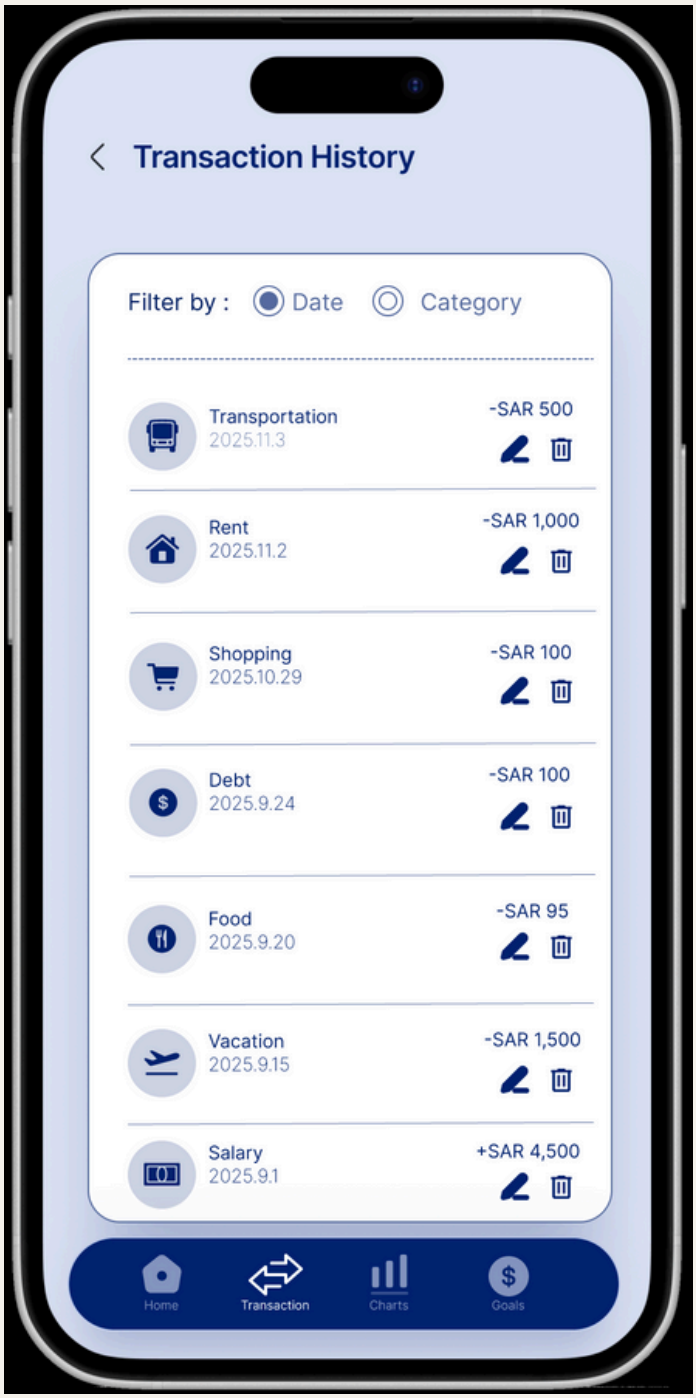
Interface application

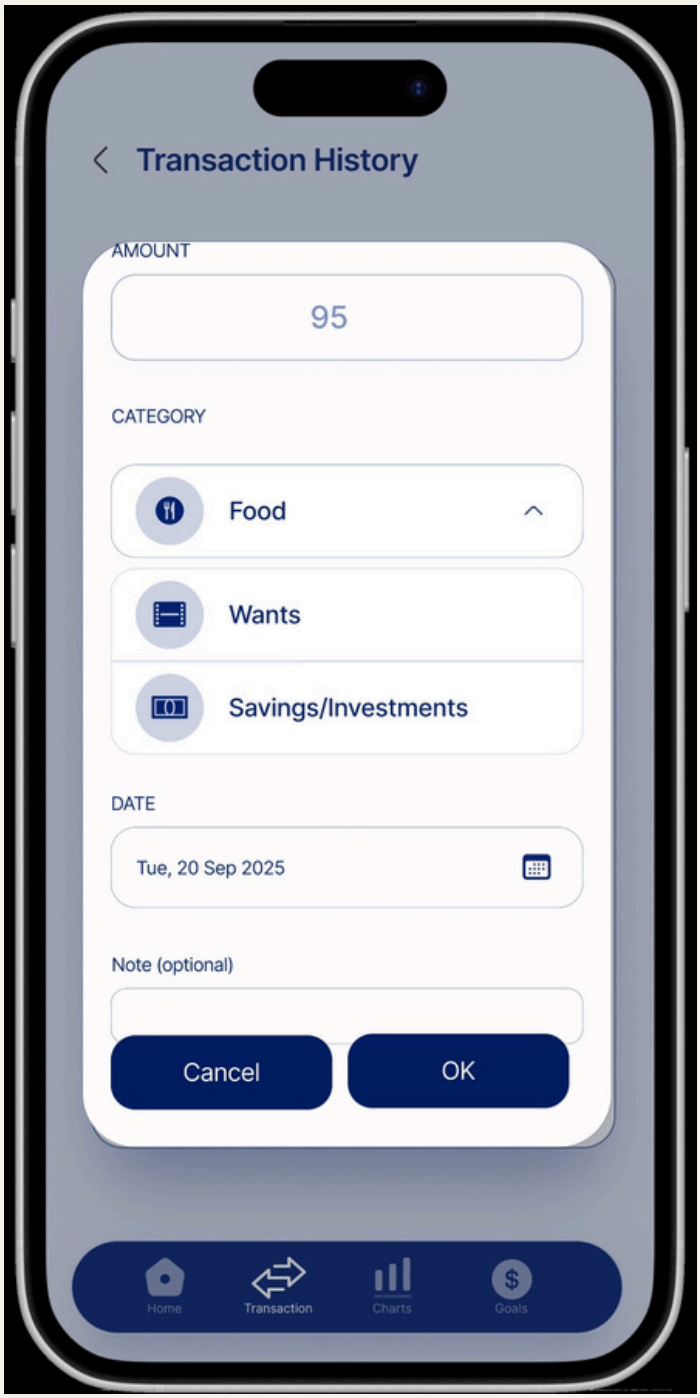




Clear pictures here :

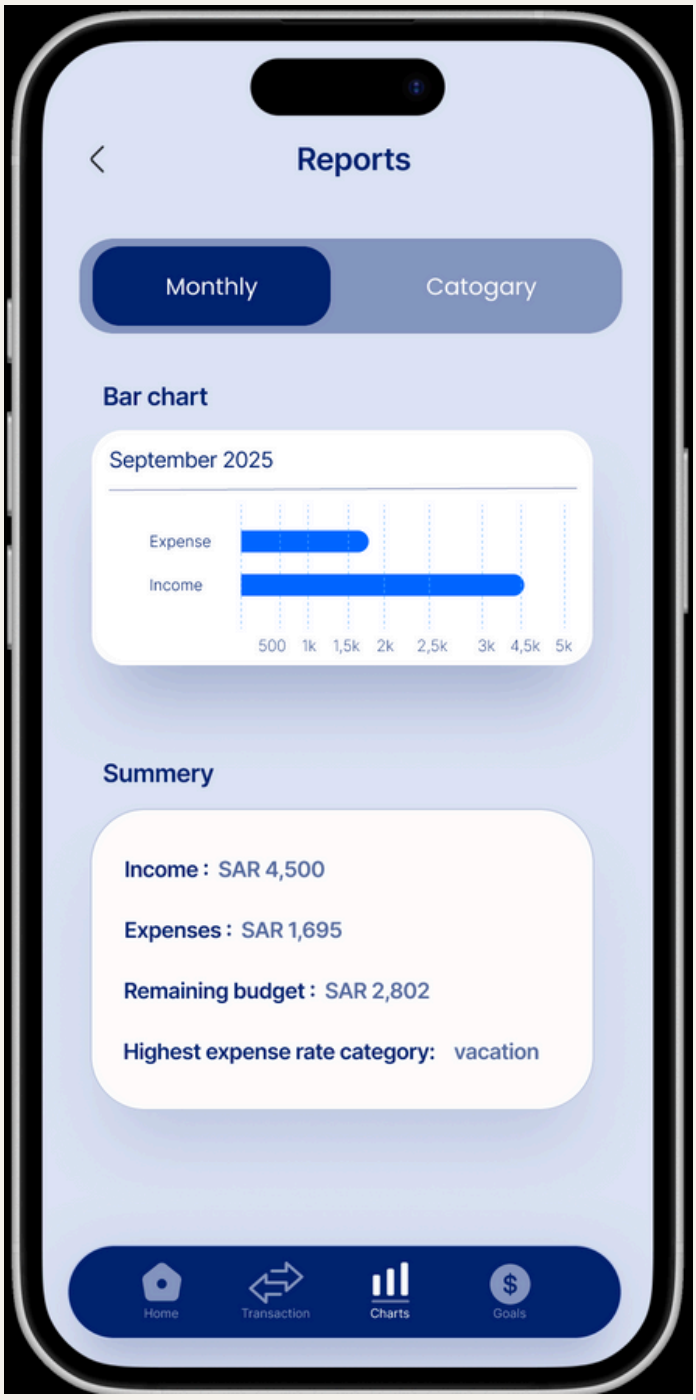
Interface application





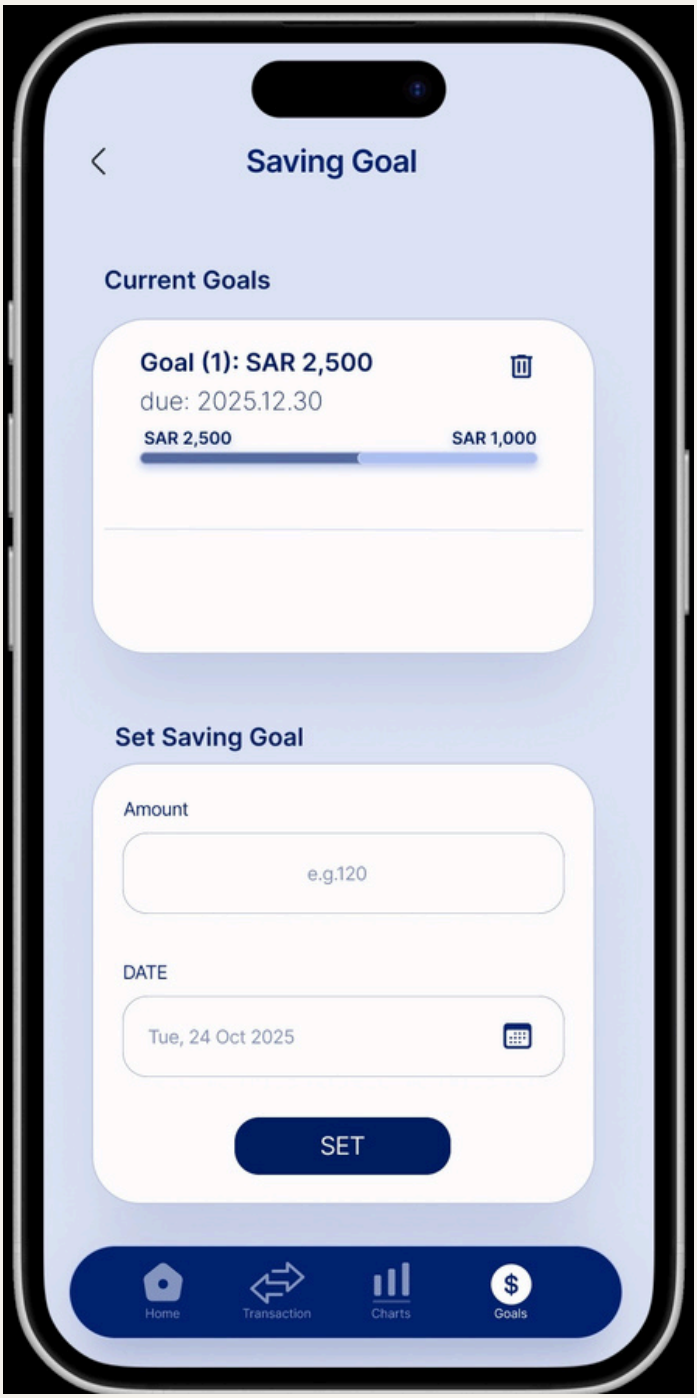
Clear pictures here :

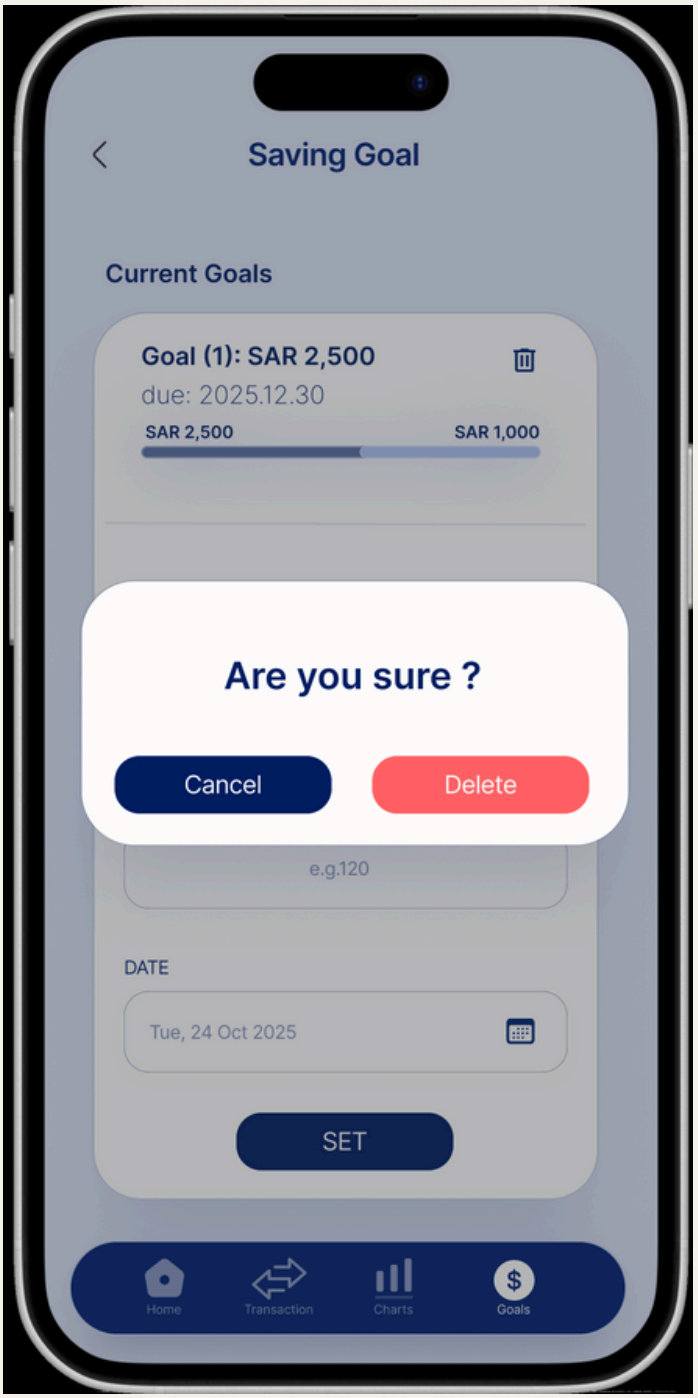
Interface application



Clear pictures here :

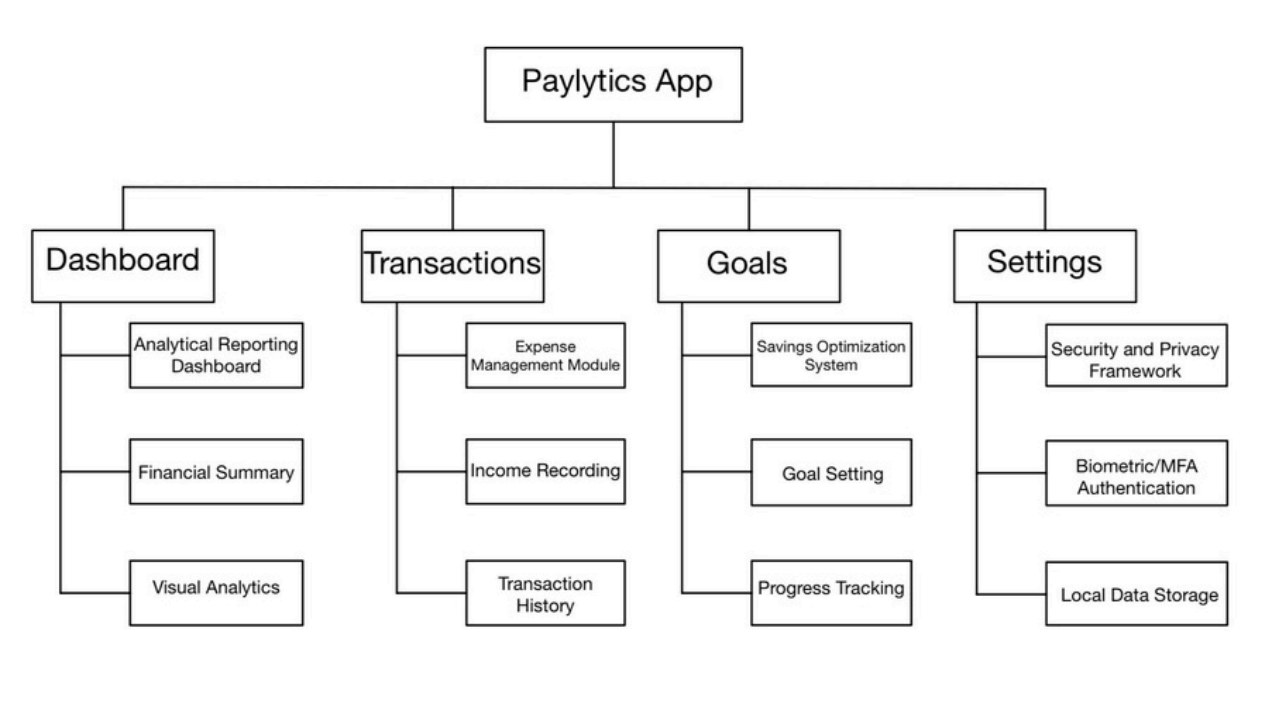
Interface application





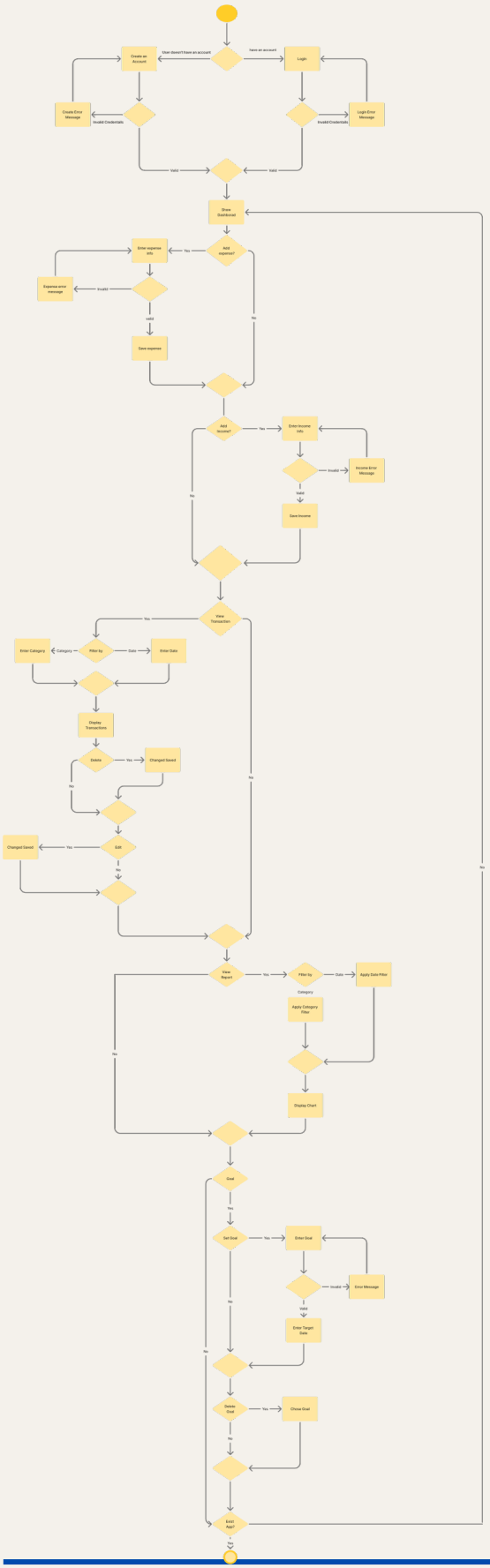
Clear pictures here :

architecture illustrate



The diagram illustrates the main architecture of the Paylytics App by dividing it into four fundamental functional domains. These primary sections include the Dashboard for displaying analytics, Transactions for recording the movement of funds, Goals for managing savings and financial planning, and finally Settings for managing security and local data. Specialized working units branch off from each section to execute the system's full range of functions.

Activity diagram

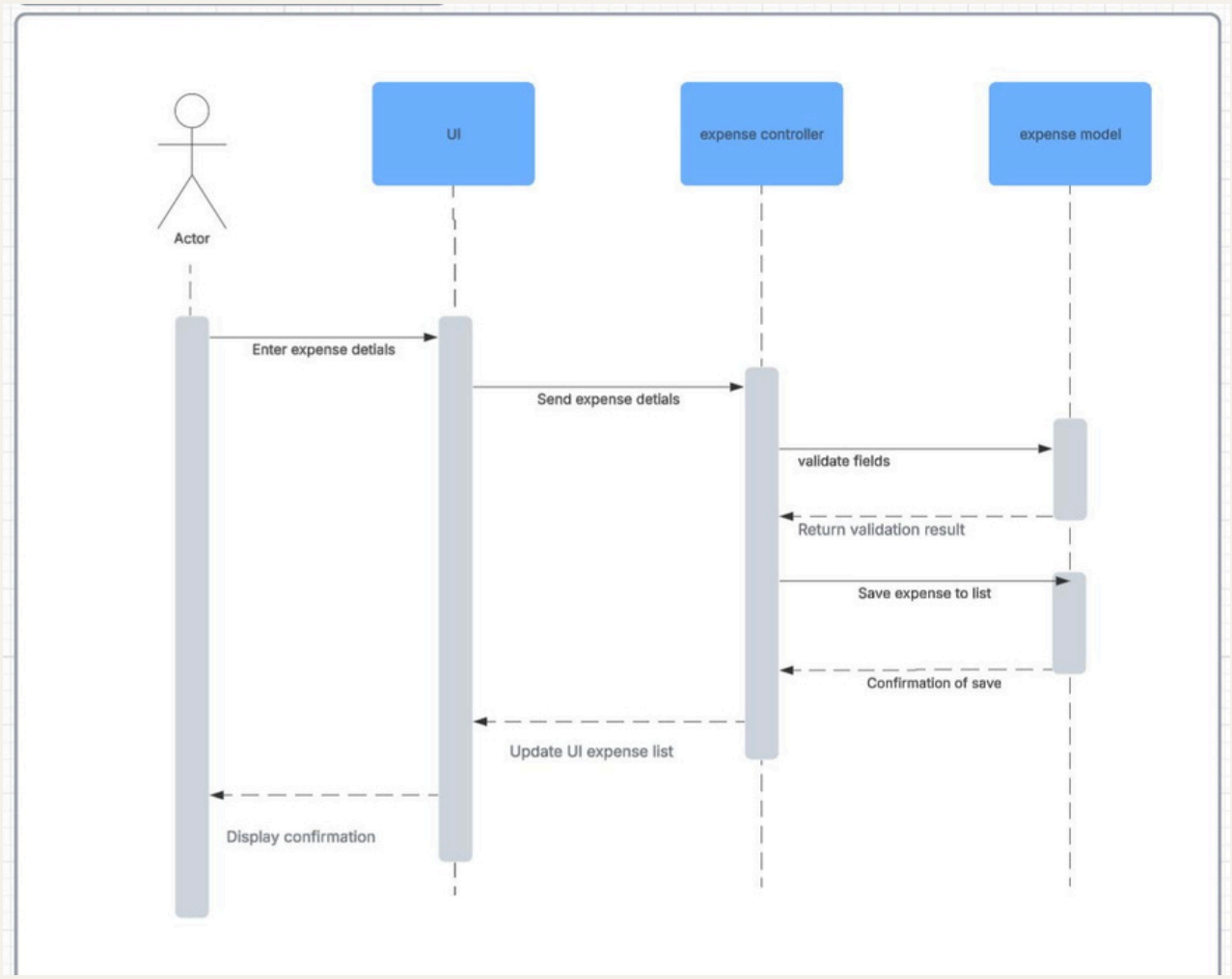


This activity diagram shows how user interact with the app. The user start by logging in or creating an account, and the system guides them if something goes wrong. Once in, users can add expenses or income, review transactions and reports, filter transactions, and set or delete goals.

Clear pictures here :

Sequence diagram

Sequence Diagrams

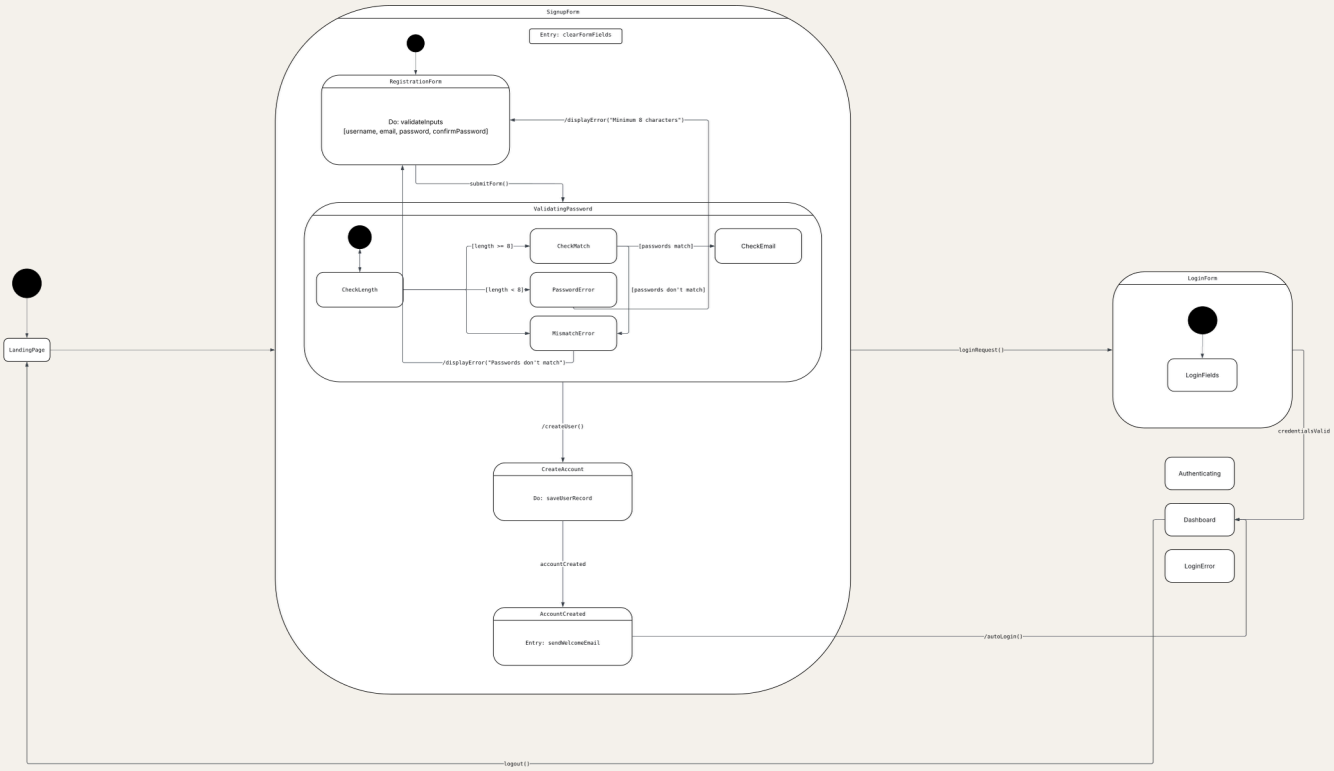


This sequence diagram describes the process of adding a new expense to the system.
The user begins by entering the expense details through the UI and UI sends these details to the Expense Controller, which forwards the data to the Expense Model for field validation.
The model validates the fields and returns the validation result to the controller, the controller instructs the model to save the expense to the list. After saving, the model returns a confirmation message to the controller

Clear pictures here :

State Diagram

State Diagram: Authentication State Diagram

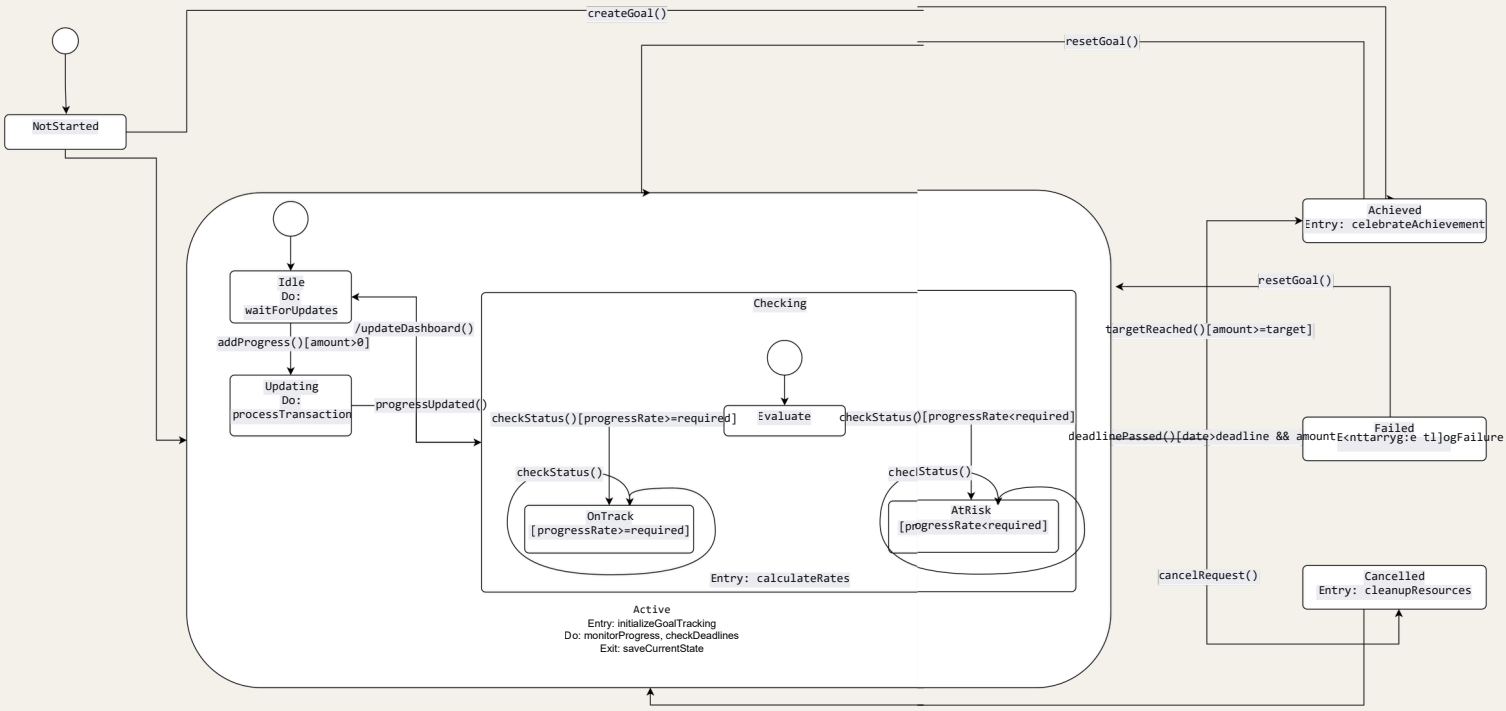


This state diagram describes the authentication flow for account creation and login. The process begins on the LandingPage, where users choose to sign up or log in. New users move to the SignupForm, which validates password length and confirmation before creating an account. Existing users go to the LoginForm, which handles authentication. After successful signup or login, the user enters the Dashboard, the main application interface.

During signup, the system checks password length through CheckLength and ensures both password fields match through CheckMatch, with error states providing clear feedback when needed. A successful registration automatically logs the user in. The diagram supports FR-1 by enabling account creation, FR-2 by validating password length, and FR-3 by verifying password matching.

State Diagram

State Diagram: Savings Goal State Diagram

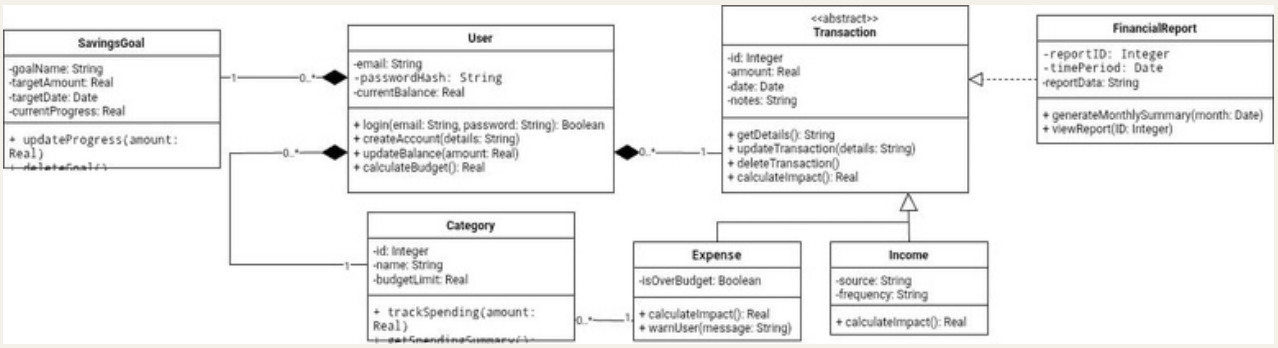


This state diagram shows how the system manages a savings goal from creation to completion or failure. It begins in NotStarted before the goal is created, then moves to Active, where progress is tracked. Within Active, the goal may be Idle while waiting for updates, Updating when new progress is added, or Checking when the system evaluates performance.

During checking, the system calculates progressRate = currentAmount / daysElapsed and compares it to requiredRate = targetAmount / totalDays. If the progressRate meets or exceeds the requiredRate, the goal is OnTrack; otherwise, it becomes AtRisk.

The goal enters Achieved when targetReached() confirms the amount has reached the target, or Failed when deadlinePassed() occurs and the amount is still below the target. The user may also trigger cancelRequest() to move it to Cancelled. From any outcome, resetGoal() can return the system to Active to start again.

Class Diagram



The Class Diagram above models the static structure of the Paylytics application, defining the system's core entities, their properties (attributes and operations), and the relationships that govern data integrity and system behavior.

1. Core Entities and Traceability

The system is organized around seven main classes:

- User: The central class, modeling the client application user account and maintaining the overall balance.
- Transaction: An abstract superclass designed to hold common data (amount, date) shared by all financial movements. This enforces structural consistency between Income and Expense records.
- Income and Expense: Concrete subclasses of Transaction that implement specific functionality for earning and spending money.
- Category and SavingsGoal: Classes modeling key user-defined financial structures, fulfilling the budget classification (FR.5) and goal-tracking (FR.8, FR.9) requirements.
- FinancialReport: The class responsible for generating and summarizing complex data structures (FR.7, FR.6) based on the recorded transactions.

Clear pictures here :

2. Key Relationships and Multiplicity

The design uses three critical types of relationships to ensure data integrity and traceability:

A. Generalization (Inheritance)

- The Income and Expense classes inherit all properties from the Transaction superclass. This is represented by the solid line with an unfilled triangle.

B. Composition (Strong Ownership)

- The Composition relationship (represented by the filled diamond) is used to model the system's data architecture, where the User is the primary owner.
- This relationship signifies that the data entities (Transaction, SavingsGoal, Category) cannot exist outside of the User object. If a user record is deleted, all associated financial data must be deleted instantly.
- The multiplicity is One-to-Many: One user owns zero or more of these data records.

C. Association (Classification)

- A simple solid line connects Expense and Category to model the requirement that every expense must be classified.
- The multiplicity is enforcing the rule that every expense belongs to exactly one category, and a category can contain many expenses.

D. Dependency (Usage)

- A dashed line with an open arrow is drawn from the FinancialReport class to the Transaction class. This signifies that the report component relies on the transaction data to execute its functions, but it does not own the data.