**MINISTRY OF EDUCATION AND TRAINING**

**CMC UNIVERSITY**



**ASSIGNMENT REPORT**

**COURSE: Advanced Programming**

**Project Title : Design and Development of a Personal Finance Management Website**

**Group Member:**

**Trần Duy Anh - BIT230009**

**Trần Quỳnh Anh - BIT230008**

**Nguyễn Hải Anh - BIT230500**

**Trần Phúc Công - BIT230061**

**Đoàn Anh Vũ - BIT230458**

**Hà Nội, tháng năm 2024**

# INTRODUCTION

Managing personal finances is an important skill in our daily lives [1]. When people manage their income and expenses well, they can save money for future needs, avoid unnecessary spending, and feel more secure during emergencies. However, many people still do not have the habit of tracking their financial activities [2]. This often leads to overspending, lack of savings, and poor financial decisions [3].

To help solve this problem, our team developed a personal finance management website using Java and the Spring Boot framework. The website allows users to record their income and expenses, manage monthly budgets, and set financial goals.

With a clear layout, intuitive navigation, and visually organized charts, the interface allows users to quickly track spending patterns and understand their financial situation. This helps them make informed decisions and gradually build better money management habits.

# TABLE OF CONTENTS

[INTRODUCTION 2](#_Toc205891481)

[LIST OF ABBREVIATION 4](#_Toc205891483)

[IMPLEMENTATION PLAN 7](#_Toc205891484)

[CHAPTER I: PROBLEM STATEMENT AND OBJECTIVES 9](#_Toc205891485)

[CHAPTER II: REQUIREMENTS ANALYSIS 10](#_Toc205891486)

[1. System Requirements 10](#_Toc205891487)

[1.1. Functional Requirements 10](#_Toc205891488)

[1.2. Non-functional Requirements 10](#_Toc205891489)

[CHAPTER III: TECHNOLOGY USED 12](#_Toc205891490)

[1. Technologies Used 12](#_Toc205891491)

[1.1. Backend / API / Database Services: 12](#_Toc205891492)

[1.2. Frontend 12](#_Toc205891493)

[2. Development Tools & Frameworks 12](#_Toc205891494)

[3. Development Methodology 13](#_Toc205891495)

[CHAPTER IV: SYSTEM DESIGN 14](#_Toc205891496)

[1. Use Case Diagrams 14](#_Toc205891497)

[2. System Architecture 15](#_Toc205891498)

[3. Database Design 15](#_Toc205891499)

[4. API Design 19](#_Toc205891500)

[CHAPTER V: USER INTERFACE DESIGN 21](#_Toc205891501)

[CHAPTER VII: LIMITATIONS AND FUTURE WORK 43](#_Toc205891502)

[CONCLUSION 43](#_Toc205891503)

[REFERENCES 44](#_Toc205891504)

# LIST OF ABBREVIATION

|  |  |
| --- | --- |
| **Abbreviation** | **Meaning** |
| API | Application Programming Interface |
| DAO | Data Access Object – a design pattern for interacting with the database |
| DTO | Data Transfer Object – an object used to pass data between layers |
| SQL | Structured Query Language |
| SP | Stored Procedure – a precompiled SQL code stored in the database |
| ID | Identifier (usually a unique numeric or string value) |
| UI | User Interface |
| JDBC | Java Database Connectivity – Java API for database access |
| GET | HTTP method used to retrieve data from the server |
| @Param / @RequestParam | Annotation in Spring Boot for mapping HTTP request parameters |
| JSON | JavaScript Object Notation – a lightweight data format for exchanging information |
| COALESCE | SQL function returning the first non-null value from its arguments |
| SUM | SQL aggregate function that returns the total sum of a numeric column |
| UC | Use Case – a description of a system’s behavior in response to user interaction |
| CRUD | Create, Read, Update, Delete – the four basic operations of persistent storage |
| ERD | Entitiy Relationship Diagram |

**LIST OF FIGURES**

[Figure 1: Use Case Diagram 11](#_Toc205891346)

[Figure 2: Entitiy Relationship Diagram 15](#_Toc205891347)

[Figure 3: Database Diagram 16](#_Toc205891348)

[Figure 4: Login page 18](#_Toc205891349)

[Figure 5: Sign up page 18](#_Toc205891350)

[Figure 6: Dashboard page 19](#_Toc205891351)

[Figure 7: Add New Transaction page 19](#_Toc205891352)

[Figure 8: Transaction History page 20](#_Toc205891353)

[Figure 9: Budget Management page 20](#_Toc205891354)

[Figure 10: Visualized Chart page 21](#_Toc205891355)

[Figure 11: Category Management page 21](#_Toc205891356)

[Figure 12: Stored Procedure GetBudgetsWithSpending 23](#_Toc205891357)

[Figure 13: Java Code: BudgetRowMapper 24](#_Toc205891358)

[Figure 14: Budget Management Frontend 24](#_Toc205891359)

[Figure 15: Stored Procedure ApplyUserPrefixesToBudget 25](#_Toc205891360)

[Figure 16: Function createTransaction in TransactionController 27](#_Toc205891361)

[Figure 17: Function createTransaction in transactionService 27](#_Toc205891362)

[Figure 18: Retrieve user transaction history by month (Java DAO) 29](#_Toc205891363)

[Figure 19: Stored Procedure GetUserRecentTransactionsByMonth 30](#_Toc205891364)

[Figure 20: Stored Procedure GetMonthlySummary 31](#_Toc205891365)

[Figure 21: Retrieve monthly transaction summary (Java DAO) 32](#_Toc205891366)

[Figure 22: Stored Procedure GetTotalExpenseOfCategory 33](#_Toc205891367)

[Figure 23: Fetching spending categories via stored procedure in Java 34](#_Toc205891368)

[Figure 24: Service layer method delegating spending category retrieval to DAO 34](#_Toc205891369)

[Figure 25: Spring Boot REST endpoint for fetching spending categories 34](#_Toc205891370)

[Figure 26: Stored Procedure GetUserTotalBalance 35](#_Toc205891371)

[Figure 27: Retrieve total user balance via stored procedure in Java 36](#_Toc205891372)

[Figure 28: Service method retrieving total balance from DAO 36](#_Toc205891373)

[Figure 29: Spring Boot GET endpoint for retrieving total balance 36](#_Toc205891374)

[Figure 30: Stored Procedure GetUserTransactions 37](#_Toc205891375)

[Figure 31: Mapping and retrieving user transaction history via stored procedure in Java 37](#_Toc205891376)

[Figure 32:Transaction History Frontend 38](#_Toc205891377)

[Figure 33: Stored Procedure GetFilteredTransactionHistory 38](#_Toc205891378)

[Figure 34: DAO method for retrieving filtered transaction history with SQL Server date handling 39](#_Toc205891379)

**LIST OF TABLES**

[Table 1: Developments Tools and Frameworks Table 12](#_Toc205891819)

[Table 2: Users Table 15](#_Toc205891820)

[Table 3: Categories Table 15](#_Toc205891821)

[Table 4: Transactions Table 15](#_Toc205891822)

[Table 5: Budgets Table 16](#_Toc205891823)

[Table 6: Prefixes Table 16](#_Toc205891824)

[Table 7: API Design Table 18](#_Toc205891825)

# IMPLEMENTATION PLAN

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **No.** | **Task** | **Description** | **Main Responsible Member** | **Time (Week)** | **Deliverables** |
| 1 | Requirement Analysis | Identify core features and design user flow | Trần Quỳnh Anh | Week 1 (30/06 – 06/07) | List of planned features, use-case diagram |
| 2 | Wireframe Design | Create low-fidelity screen sketches | Đoàn Anh Vũ | Complete wireframes for all main screens |
| 3 | Database Design | Define tables, data models, and relationships | Nguyễn Hải Anh | Week 2 (07/07 – 13/07) | ERD diagram, SQL schema or model classes |
| 4 | Spring Boot Project Setup | Initialize project structure and dependencies | Nguyễn Hải Anh | Spring Boot project with configured Maven/Gradle, base folder structure, Git repo |
| **Backend** | | | | | |
| 5 | Implement Category Management API | CRUD API for Expense Categories | Trần Duy Anh | Week 3 (14/07 – 20/07) | REST APIs for categories |
| 6 | Implement Income/Expense Transaction API | CRUD API for Transactions | Trần Quỳnh Anh | REST APIs for income/expenses |
| 7 | Calculate Balance and Top Spending Categories | API to get current balance and top spending categories | Trần Quỳnh Anh | Endpoint for total income - expenses |
| 8 | Set Budget Plan API | Budget Plan CRUD | Nguyễn Hải Anh | REST APIs for budgets |
| 9 | Implement Budget Template Feature | Allow users to set fixed spending limits per category | Nguyễn Hải Anh | Budget template API + UI for setting/viewing per-category limits |
| 10 | Implement Search/Filter Transaction API | Search by keyword, date, category | Đoàn Anh Vũ | Search and filter API for transactions |
| 11 | Implement JWT Authentication | Auth API: Login, Register, JWT | Trần Phúc Công | Working login/register API with JWT |
| **Frontend** | | | | | |
| 12 | Build Login & Register pages | UI for login and signup | Trần Phúc Công | Week 4 (21/07 – 27/07) | Responsive login/register forms |
| 13 | Build Dashboard page | Show balance overview | Trần Quỳnh Anh | Dashboard UI with balance info |
| 14 | Build Category Management UI | Manage categories (CRUD) | Trần Duy Anh | Category list, add/edit/delete |
| 15 | Build Transaction UI | Add transactions | Trần Quỳnh Anh | Transaction form and listing |
| 16 | Build Transaction History page | Full list with filter options | Đoàn Anh Vũ | Table with filter for history |
| 17 | Display Spending Statistics | Charts for category/month | Trần Quỳnh Anh | Charts using Chart.js/ApexCharts |
| **Testing & Finalization** | | | | | |
| 18 | Test API and Features | Manual testing, bug fixing | Trần Duy Anh | Week 5 (28/07 – 03/08) | Test report, bug list |
| 19 | Deploy website to domain | Host the web application on a domain | Trần Duy Anh | Live website accessible via domain |

# CHAPTER I: PROBLEM STATEMENT AND OBJECTIVES

1. **Objectives of the application**

The main objectives of the application are to help users record their income and expenses quickly and easily, allow them to create and manage monthly budgets, and provide charts and summaries of their financial activities. By doing so, the application aims to make users more aware of their spending habits and encourage better financial planning and saving.

1. **Target Users**

The application is designed for a broad range of users, including students who are learning to manage their personal finances independently, young professionals who want to monitor their income and spending habits, individuals or families aiming to plan budgets and save for future goals, and anyone seeking a simple yet efficient tool to improve their financial awareness and discipline.

# CHAPTER II: REQUIREMENTS ANALYSIS

1. System Requirements

## Functional Requirements

The core functional requirements of the application include:

* **User Registration and Login:**
  + Users can securely create an account, log in, and log out.
* **Transaction Management:**
  + Add, edit, or delete income and expense transactions.
  + Assign categories to each transaction.
  + Include notes and select the transaction date.
* **Category Management:**
  + Create, update, and delete income or expense categories.
  + Clearly separate income and expense category types.
* **Budget Planning:**
  + Allow users to set monthly budget limits for each category.
  + Apply budget plan template for the next months.
* **Transaction History:**
  + Display a list of transactions filtered by date, category, or type.
  + Support searching and filtering for easier tracking.
* **Dashboard Overview:**
  + Provide charts and summaries for income, expenses, and budget status.
  + Show real-time updates of financial activities.
* **Data Security and Storage:**
  + Ensure secure data storage via backend services.
  + Maintain personalized data per user account.

## Non-functional Requirements

While functional requirements describe what the system should do, non-functional requirements explain how the system should perform. For our personal finance management web application, the non-functional requirements are:

* Performance: The application should respond quickly. Pages should load in under 5 seconds.
* Usability: The interface should be simple and easy to understand. First-time users should understand the basic features without needing training.
* Reliability: The system should work correctly and be available at all times. If there is a problem, it should recover quickly. Data should not be lost.
* Security: User data must be protected. Passwords should be stored securely. Only the user should be able to view and edit their personal financial data.
* Maintainability: The system should be easy to update. Developers should be able to fix bugs or add new features without changing the whole system.
* Compatibility: The web application should work on different browsers such as Chrome, Firefox, and Edge. It should also display well on both desktop and mobile screens.

# CHAPTER III: TECHNOLOGY USED

1. Technologies Used
   1. Backend / API / Database Services:

The backend of this system is built using Java Spring Boot. It handles the logic of the application, such as adding a transaction, viewing reports, or calculating the budget.

The system stores all information (users, categories, transactions, and budgets) in a SQL Server database. We use Spring Data JPA to connect Java code with the database.

The backend also provides RESTful APIs, which return data in JSON format. These APIs are used by the frontend to get or send information.

## Frontend

* The frontend is created using HTML, CSS, and JavaScript. These pages send requests to the backend APIs to fetch data or submit forms.
* We use fetch API in JavaScript to call the backend endpoints. The layout and design are made with plain CSS for simplicity and easy maintenance.

1. Development Tools & Frameworks

Table 1: Developments Tools and Frameworks Table

|  |  |
| --- | --- |
| **Tool/Framework** | **Purpose** |
| Apache NetBeans | Used for writing and running backend code |
| Visual Studio Code | Used to write and edit HTML, CSS, and JavaScript for the frontend |
| Git & GitHub | Version control and collaborative coding |
| Spring Boot | Java framework used to build the backend and create RESTful APIs |
| Postman | API testing and documentation |
| SQL Server Management Studio (SSMS) | Used to create and manage the SQL Server database |

Table 1 presents the development tools and frameworks utilized in the project, covering both backend and frontend development. It includes programming environments such as Apache NetBeans and Visual Studio Code, version control with Git and GitHub, the Spring Boot framework for building RESTful APIs, Postman for API testing and documentation, and SQL Server Management Studio for database creation and management. These tools collectively supported efficient development, testing, and maintenance of the system.

1. Development Methodology

We followed the Agile software development process. The work is divided into short stages called sprints. Each sprint focuses on a small part of the project, including:

* Understanding the task
* Designing the solution
* Writing the code
* Testing and fixing bugs
* Reviewing and improving

# CHAPTER IV: SYSTEM DESIGN

1. Use Case Diagrams

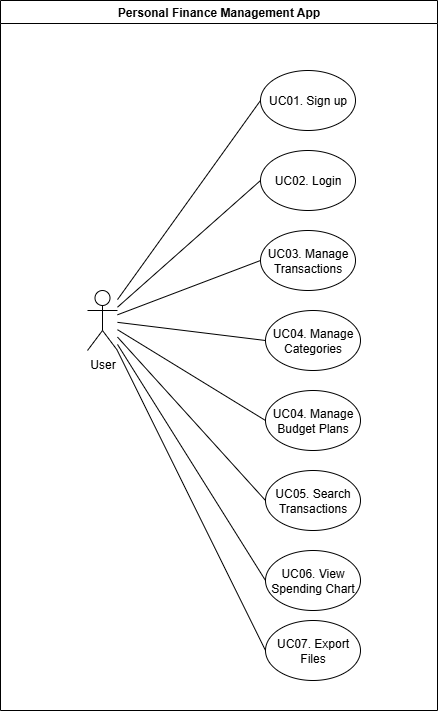


Figure 1: Use Case Diagram

**Use Case Description**

* **UC01 – Sign Up**: Allows a new user to create an account by providing necessary personal details such as username, email, and password. This step is required before accessing other features.
* **UC02 – Login**: Enables existing users to securely access their account by entering their registered credentials.
* **UC03 – Manage Transactions**: Lets users add delete income or expense records. This helps them keep accurate and up-to-date financial data
* **UC04 – Manage Categories**: Allows users to create, edit, or remove expense and income categories, making it easier to organize financial records.
* **UC04 – Manage Budget Plans**: Enables users to set monthly spending limits or budget plans for specific categories, helping them control expenses.
* **UC05 – Search Transactions**: Provides a search and filter function to quickly locate transactions based on keywords, date ranges, or categories.
* **UC06 – View Spending Chart**: Displays visual charts summarizing spending and income trends over time, giving users a better understanding of their financial habits.

1. System Architecture

The system uses a 3-tier architecture, which separates the application into three main layers:

* Presentation Layer: This is the frontend of the system. It is built using HTML and CSS. It allows users to interact with the system, such as adding transactions, viewing budgets, and checking reports.
* Business Logic Layer This is the backend, created using Spring Boot. It handles all the logic of the application. It receives requests from the frontend, processes them, and returns the results. It also communicates with the database through services and controllers.
* Data Layer: This layer is responsible for storing and retrieving data. The system uses SQL Server as the main database. Data is accessed using Spring Data JPA.

1. Database Design

In this project, we use SQL Server for database. There are in total 5 tables used for the project:

Table 2: Users Table

|  |  |  |
| --- | --- | --- |
| **Field’s name** | **Data Type** | **Description** |
| **UserID** | **INT** | **Store auto generated user’s ID** |
| **Username** | **NVARCHAR** | **Store user’s username for displaying** |
| **Email** | **NVARCHAR** | **Store user’s email for login** |
| **Password** | **NVARCHAR** | **Store user’s password** |
| **CreatedAt** | **DATETIME** | **Time that created account, will not be used by users.** |

**Table 1 contains user’s account basic information.**

Table 3: Categories Table

|  |  |  |
| --- | --- | --- |
| **Field’s name** | **Data Type** | **Description** |
| **CategoryID** | **INT** | **Store auto generated category’s ID** |
| **UserID** | **INT** | **Foreign key from user’s table, define that each category only belong to one user.** |
| **Name** | **NVARCHAR** | **Store category’s name.** |
| **Type** | **NVARCHAR** | **Store category’s type (Expense or Income)** |
| **IconCode** | **INT** | **Store category’s picked icon’s code by the user** |
| **ColorCodeHex** | **NVARCHAR** | **Store category’s picked color’s coded (in HEX code) by the user** |
| **CreatedAt** | **DATETIME** | **Time that created the category, will not be used by users.** |

Table 2 contains categories of each user, each category belongs to only one user. Can be set to Expense or Income.

Table 4: Transactions Table

|  |  |  |
| --- | --- | --- |
| **Field’s name** | **Data Type** | **Description** |
| **TransactionID** | **INT** | **Store auto generated transaction’s ID** |
| **UserID** | **INT** | **Foreign key to Users table** |
| **CategoryID** | **INT** | **Foreign key to Categories table** |
| **Amount** | **DOUBLE** | **Store the amount of the transaction** |
| **TransactionDate** | **DATETIME** | **Store transaction’s date. This is different from CreatedAt, as the user can define it by themselves.** |
| **Note** | **NVARCHAR** | **Store transaction’s note** |
| **CreatedAt** | **DATETIME** | **Time that created transaction, will not be used by users.** |

Table 3 contains transactions of each user for a single category of them.

Table 5: Budgets Table

|  |  |  |
| --- | --- | --- |
| **Field’s name** | **Data Type** | **Description** |
| **BudgetID** | **INT** | **Store auto generated budget’s ID** |
| **UserID** | **INT** | **Foreign key to Users table** |
| **CategoryID** | **INT** | **Foreign key to Categories table** |
| **Amount** | **DOUBLE** | **Store budget’s amount for a month** |
| **Month** | **INT** | **Along with Year field to define budget’s month** |
| **Year** | **INT** | **Along with Month field to define budget’s month** |
| **CreatedAt** | **DATETIME** | **Time that created budget, will not be used by users.** |

Table 4 contains budget limit for each month of a user. Each category can only have 1 budget set for each month

Table 6: Prefixes Table

|  |  |  |
| --- | --- | --- |
| **Field’s name** | **Data Type** | **Description** |
| **PrefixID** | **INT** | **Store auto generated prefix’s ID** |
| **UserID** | **INT** | **Foreign key to Users table** |
| **CategoryID** | **INT** | **Foreign key to Categories table** |
| **Amount** | **DOUBLE** | **Store prefix’s amount** |

Table 5 contains budget’s template created by user, can be apply for a selected month.

**Entity Relationship Diagram (ERD):**

**A diagram of a computer

AI-generated content may be incorrect.**

Figure 2: Entitiy Relationship Diagram

Figure 2 is an ERD which shows the database structure of a financial management system, including tables for Users, Transactions, Categories, Budgets, and Prefixes, with primary keys for unique identification and foreign keys to define relationships, enabling the system to track user accounts, categorize and record transactions, manage budgets, and store predefined recurring amounts.

**Database Diagram:**

**A screenshot of a computer

AI-generated content may be incorrect.**

Figure 3: Database Diagram

1. API Design

Table 7: API Design Table

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **API Endpoint** | **Function Description** | **HTTP Method** |
| 1 | /api/budget/showByMonth | Get budget list by user, month, year | GET |
| 2 | /api/budget/update | Update existing budget amount | PUT |
| 3 | /api/budget/delete | Delete a budget item | DELETE |
| 4 | /api/categories/simpleCategoryExpense | Get available categories for expense (for budget) | GET |
| 5 | /api/budget/insert | Add new budget | POST |
| 6 | /api/categories/simpleCategoryForPrefix | Get available categories for prefix planning | GET |
| 7 | /api/prefix/insert | Add new prefix plan | POST |
| 8 | /api/prefix/show | Get all prefix plans of user | GET |
| 9 | /api/prefix/delete | Delete a prefix plan | DELETE |
| 10 | /api/prefix/applyPrefix | Apply prefix to a new month | POST |
| 11 | /api/categories/income | Get user’s income categories | GET |
| 12 | /api/categories/expense | Get user’s expense categories | GET |
| 13 | /api/transactions/transactionHistory | Get user's transaction history | GET |
| 14 | /api/transactions | Create a new transaction | POST |
| 15 | /api/transactions/recentTransaction | Get user’s recent transaction summary | GET |
| 16 | /api/transactions/{transactionId} | Delete transaction by ID | DELETE |
| 17 | /api/monthlyreport/top3MonthlyExpense | Get top 3 monthly expenses | GET |
| 18 | /api/monthlyreport/monthlySummary | Get overall summary for the month | GET |
| 19 | /api/users/register | Create a new user account | POST |
| 20 | /api/users/login | Authenticate user and return token/session | POST |
| 21 | /api/users/profile | Get logged-in user's profile details | GET |
| 22 | /api/users/update | Update user profile details | PUT |
| 23 | /api/transactions/update | Update an existing transaction | PUT |
| 24 | /api/categories/all | Get all categories of a user (both income and expense) | GET |
| 25 | /api/categories/create | Add a new category | POST |
| 26 | /api/categories/update | Update an existing category | PUT |
| 27 | /api/categories/delete/{categoryId} | Delete category by ID | DELETE |
| 28 | /api/statistics/totalBalance | Get total current balance of the user | GET |
| 29 | /api/statistics/yearlySummary | Get income & expense summary for a year | GET |
| 30 | /api/statistics/categorySummary | Get spending summary by category | GET |

# CHAPTER V: USER INTERFACE DESIGN

A login screen with blue and white text

AI-generated content may be incorrect.

Figure 4: Login page

A screen shot of a login form

AI-generated content may be incorrect.

Figure 5: Sign up page

A screenshot of a computer

AI-generated content may be incorrect.

Figure 6: Dashboard page

A screenshot of a computer

AI-generated content may be incorrect.

Figure 7: Add New Transaction page

A screenshot of a computer

AI-generated content may be incorrect.

Figure 8: Transaction History page

A screenshot of a computer

AI-generated content may be incorrect.

Figure 9: Budget Management page

A graph on a white background

AI-generated content may be incorrect.

Figure 10: Visualized Chart page

A screenshot of a computer

AI-generated content may be incorrect.

Figure 11: Category Management page

**CHAPTER VI: IMPLEMENTATION**

Backend Code Explanation

The project’s function mostly goes around using basic CRUD method, which already exist in Spring’s Repository. However, some of them are more troublesome and require more than just basic method. We will go into detailed with some of them in this part.

**1. Budget:**Budget require some extra fields that its own table doesn’t own. Further than that, it also needs some calculation to know how much the user has spent on the category. To do this, we use Store Procedure in SQL Server and then call it with proper input information to receive the expected information. Following is the SP we used for showing detailed budget:  
A screenshot of a computer code

AI-generated content may be incorrect.

Figure 12: Stored Procedure GetBudgetsWithSpending

The SP require user’s ID along with month and year for finding the correct transactions. The extra fields that the SP return are Category’s name, Icon Code and the Total spent amount of the category in the selected month. Total Spent is calculated by finding all transaction of type “Expense” (Because budget can only be set for Expense categories) in the selected date range, and then summing them up while grouping by the category’s ID.

The result is then processed by the DAO (BudgetDAO), which is then mapped to the DTO (BudgetWithSpendingDTO). Following is the detailed code in BudgetDAO class

**A screen shot of a computer code

AI-generated content may be incorrect.**

Figure 13: Java Code: BudgetRowMapper

Front end will receive the resulted list and generate cards that each resemble a budget

A screenshot of a computer

AI-generated content may be incorrect.

Figure 14: Budget Management Frontend

1. **Prefix:**

Adding and deleting a prefix is no problem, the problem only appears when trying to apply them. The first idea was to call add budget API with each of them. But obviously that will slow down the system by a lot. That’s why we decided to process them in database using SP.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 15: Stored Procedure ApplyUserPrefixesToBudget

Obviously, the core is still using adding new budget multiple times, but by calling them directly in SQL Server, the performance will be greatly improved. With the given user’s ID and selected month – year, the SP will find Prefixes that created by the user, then will determine action using MERGE command. The budget is considered already existed if there is a budget of the same category ID from the user. If it already exists, it will be override by the set amount in prefix. If not, it will be added accordingly.

Since the SP doesn’t return anything, we simply make a DAO class (PrefixDAO) that execute the SP, and it is all set. Now the user can apply all their prefix in one click without delaying or stay in stale state of the program.

1. **Add Transaction**
   1. **Page Initialization**

* When the page loads (DOMContentLoaded), the script gets the user ID from the authentication token and stores it in userID.
* The default transaction type (e.g., *Income* or *Expense*) is read from the dropdown (transactionType).
* The script calls loadCategories(type) to fetch relevant categories for the selected transaction type from the endpoint: GET /api/categories/{type}?userID={userID}
* The server returns a list of categories for that user and type.
* These categories are displayed in the Category dropdown.
  1. **Dynamic Category Loading:** If the user changes the transaction type dropdown, the change event triggers another call to loadCategories() to refresh the category list.
  2. **Form Submission (Frontend):** When the **Add Transaction** form is submitted:
* The default page reload is prevented (e.preventDefault()).
* Input values (type, category, amount, date, note) are collected.
* The date is formatted into an ISO-like string with "T00:00:00" for the time part.
* All data is packaged into a JSON object like:

{

"transactionType": "Expense",

"categoryId": 3,

"amount": 50.0,

"transactionDate": "2025-08-11T00:00:00",

"note": "Lunch with friends",

"userId": 1

}

* 1. **API Call to Backend**
* The script sends a **POST** request to: POST /api/transactions
  + Headers include Content-Type: application/json and Authorization: Bearer {token}.
  + The body contains the transaction data in JSON.
  1. **Backend Processing (TransactionController)**
* The request is received by:

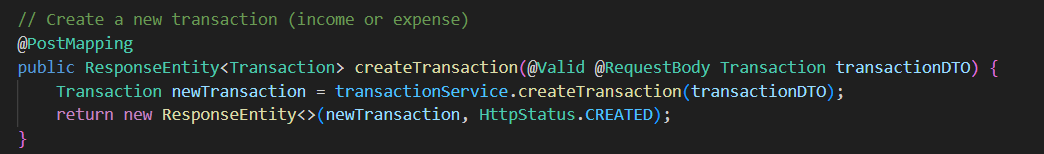


Figure 16: Function createTransaction in TransactionController

* The backend:
  + Validates the transaction data.
  + Passes it to transactionService.createTransaction(transactionDTO) for saving in the database.

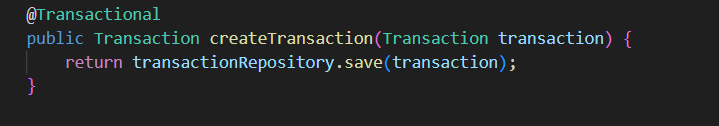


Figure 17: Function createTransaction in transactionService

* + Returns the saved transaction with HTTP **201 Created** if successful.
  1. **Response Handling (Frontend)**
* If the response is OK:
  + A success message ("Transaction added successfully!") is shown.
  + The form is reset.
  + Categories are reloaded for the selected transaction type.
* If there’s an error:
  + An error message ("Failed to save. Please check the input.") is shown.
  + The console logs the detailed error.

1. **Dashboard:**
   1. **Page initialization & auth**

* Browser loads the page → DOMContentLoaded runs.
* Script calls getToken() and parseJwt(token) to decode token and extract userId. **Security note:** Do not trust client-supplied userID alone on the server; backend must validate token and match subject/claims to requested userId (or derive userId from token server-side).
* Script queries GET /api/months with the Authorization header.
  + Purpose: provide the month select options (e.g. "2025-07" with "July 2025" label).
  + Example response:

[

{ "value": "2025-07", "label": "July 2025" },

{ "value": "2025-08", "label": "August 2025" }

]

* + Backend implementation idea: return last N months or months that have data for that user (service queries transactions or returns static options).
* After /api/months returns:
* Populate <select id="monthSelect"> with options.
* Set default monthSelect.value to current month (constructed as YYYY-MM).
* Immediately call three functions with the current month/year:
  + loadTransactionsByMonth(month, year)
  + loadMonthlySummary(month, year)
  + loadSpendingCategories(month, year)
* Also separately loadTotalBalance(userId) is called on DOM ready.
  1. **loadTransactionsByMonth(month, year)**
* **Frontend request**
* GET /api/transactions/recentTransactionByMonth?userID={userId}&month={month}&year={year}
* Authorization header included.
* **Expected response shape:**

[

{

"transactionId": 123,

"categoryName": "Food",

"categoryType": "Expense",

"amount": -120000,

"transactionDate": "2025-08-11T08:00:00"

},

...

]

* **Backend flow (service + repository)**
* Controller receives request and calls a service method like GetUserRecentTransactionsByMonths(userId, month, year).
* The service directly delegates to the DAO method: transactionDAO.GetUserTransactionHistoryByMonth(userId, month, year)
* The DAO builds and executes the SQL call to the stored procedure:

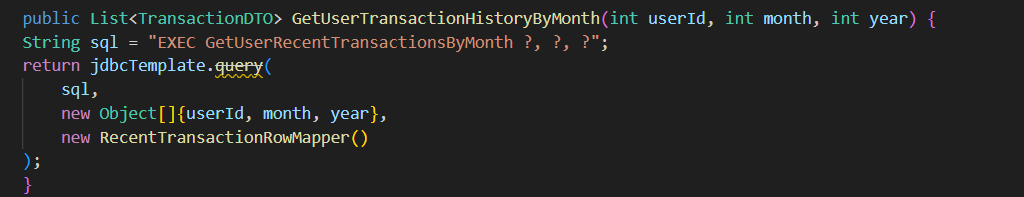


Figure 18: Retrieve user transaction history by month (Java DAO)

Here, RecentTransactionRowMapper maps each row from the SQL result set into a TransactionDTO.  
The TransactionDTO contains only the relevant fields: amount, transactionDate, categoryName, and categoryType.

* **Stored procedure (GetUserRecentTransactionsByMonth)**

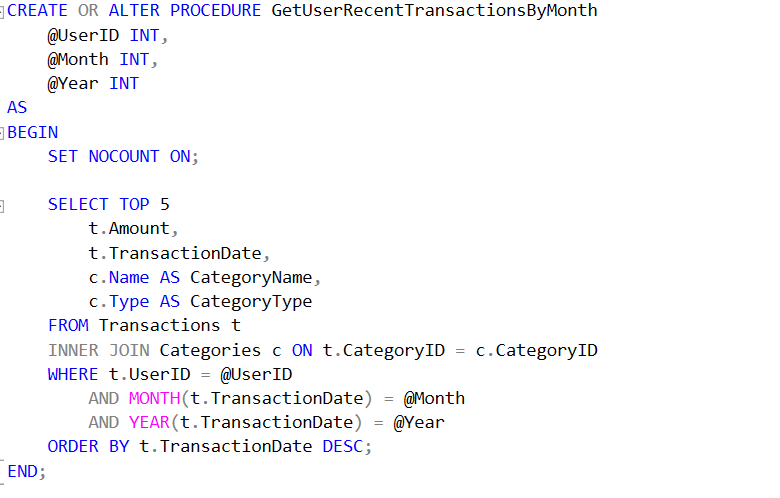


Figure 19: Stored Procedure GetUserRecentTransactionsByMonth

The procedure accepts @UserID, @Month, and @Year, and returns the top 5 most recent transactions for that user in the given month and year.  
It joins the Transactions table with the Categories table to include category name and type.  
Results are ordered by TransactionDate DESC so the newest appear first.

* **Response**: The list of DTOs is returned up the chain — DAO → Service → Controller — and finally serialized as JSON in the HTTP response.
* **Frontend rendering**
* If the returned list is empty, the UI shows a message such as **“No transactions yet”**.
* If there are transactions:
* Iterate through the list and create DOM elements for each.
* Display a colored dot (red for Expense, blue for Income).
* Format the amount using Vietnamese currency format: amount.toLocaleString('vi-VN', { style: 'currency', currency: 'VND' })
* Format the date for display using toLocaleDateString().
  1. **loadMonthlySummary(month, year)**
* **Frontend request:** GET /api/transactions/summaryTransactionByMonth?userID={userId}&month={month}&year={year}
* **Expected response shape:**

[

{ "income": 5000000, "expense": 3200000 }

]

* **Backend flow**
* The stored procedure GetMonthlySummary calculates TotalIncome and TotalExpense for the given user and month:

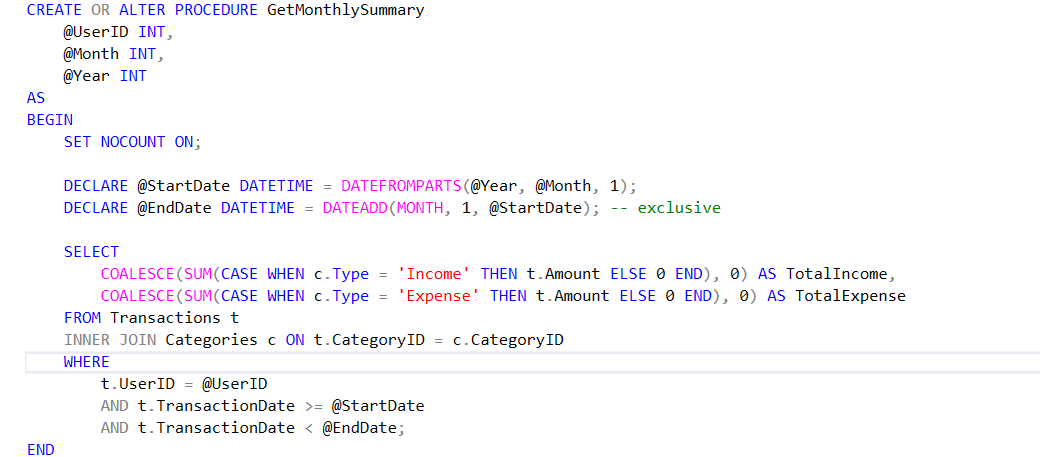


Figure 20: Stored Procedure GetMonthlySummary

* + Determines @StartDate (first day of month) and @EndDate (first day of next month).
  + Joins Transactions with Categories to classify as "Income" or "Expense".
  + Uses SUM + CASE + COALESCE to aggregate values and replace NULL with 0.
* **DAO method runs the procedure via:**

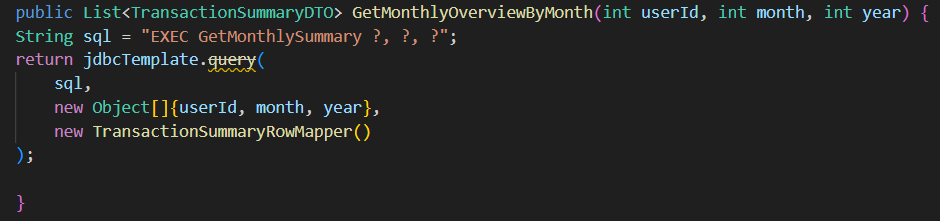


Figure 21: Retrieve monthly transaction summary (Java DAO)

* Service method simply calls the DAO and passes the result to the controller.
* Controller method @GetMapping("/summaryTransactionByMonth") exposes the endpoint, returning the result as JSON to the frontend.
* **Frontend**
* Receives data[0] from the API and extracts income and expense.
* Formats amounts, calculates balance = income - expense.
* Displays balance with conditional colors (green for positive, red for negative, black for zero).
* Defaults missing values to 0 to prevent calculation errors.
  1. **loadSpendingCategories(month, year)**
* Frontend request: GET /api/categories/getSpendingCate?userID={userId}&month={month}&year={year}
* Expected response:

[

{ "categoryId": 5, "categoryName": "Food", "amount": 1200000 },

{ "categoryId": 2, "categoryName": "Transport", "amount": 500000 }

]

* **Backend flow**

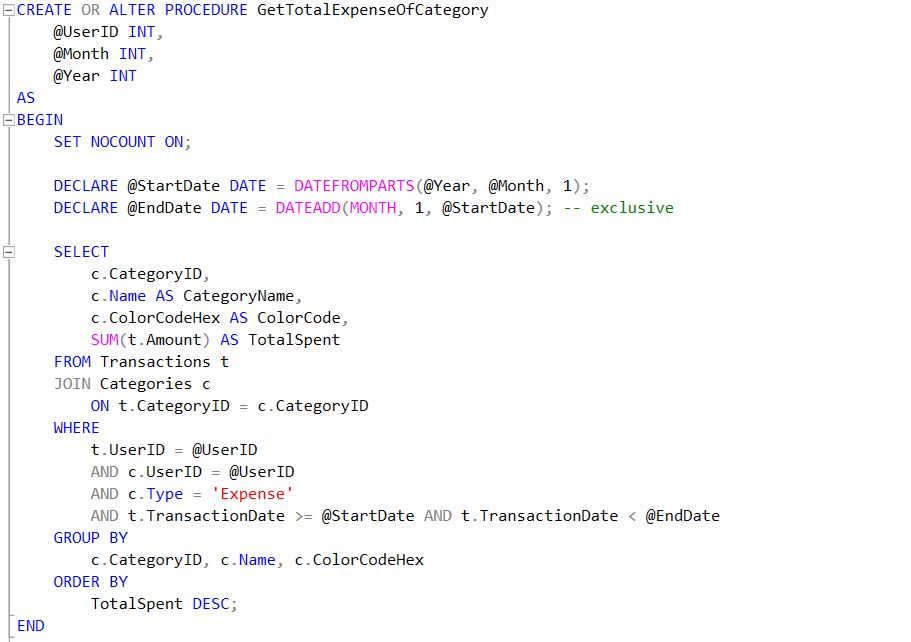


Figure 22: Stored Procedure GetTotalExpenseOfCategory

The stored procedure GetTotalExpenseOfCategory is executed with parameters userID, month, and year. This procedure calculates the total spending for each expense category in the given month by:

* Determining the date range from the first day of the month to the start of the next month.
* Joining the Transactions and Categories tables.
* Filtering by the given userID, category type = 'Expense', and transaction date within the calculated range.
* Grouping the results by category to sum up the total spent in each.
* Sorting results in descending order by the total amount spent.

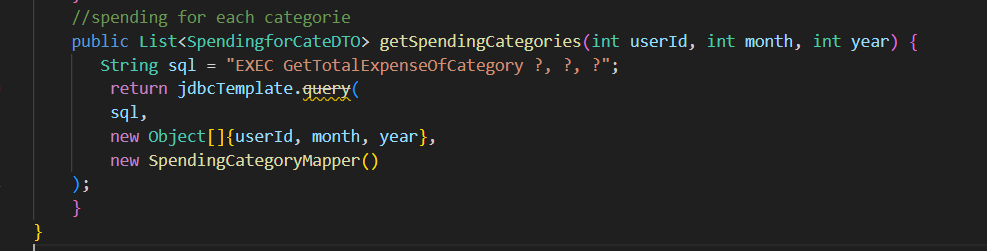
In Java, the method 

Figure 23: Fetching spending categories via stored procedure in Java

runs the SQL command EXEC GetTotalExpenseOfCategory ?, ?, ? using JdbcTemplate, and maps the results to a list of SpendingforCateDTO objects through SpendingCategoryMapper.

The service method

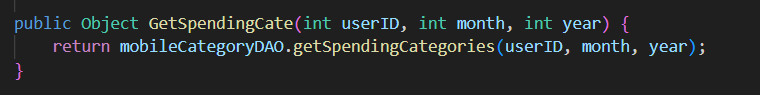


Figure 24: Service layer method delegating spending category retrieval to DAO

simply calls the DAO method and returns the list of DTOs to the controller.

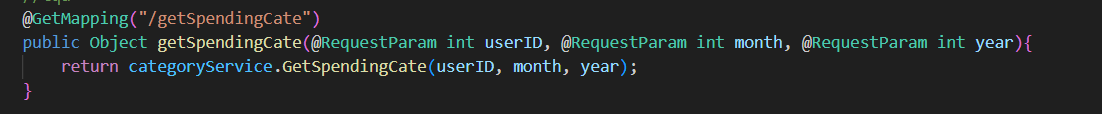
The controller endpoint 

Figure 25: Spring Boot REST endpoint for fetching spending categories

accepts the parameters from the frontend request and returns the result of the service method.

* **Frontend rendering**
  + If no categories are returned, display a “no data” message.
  + Otherwise, calculate the totalExpense = sum(amount) from all categories.
  + Render the categories list in HTML, ensuring category names are safely escaped to prevent XSS.
  + For each category, also calculate the spending percentage using (amount / totalExpense \* 100).toFixed(1).
  1. **loadTotalBalance(userId)**
* Frontend request: GET /api/transactions/totalBalance?userID={userId}
* Expected response:

[

{ "totalBalance": 12345678 }

]

* Backend flow

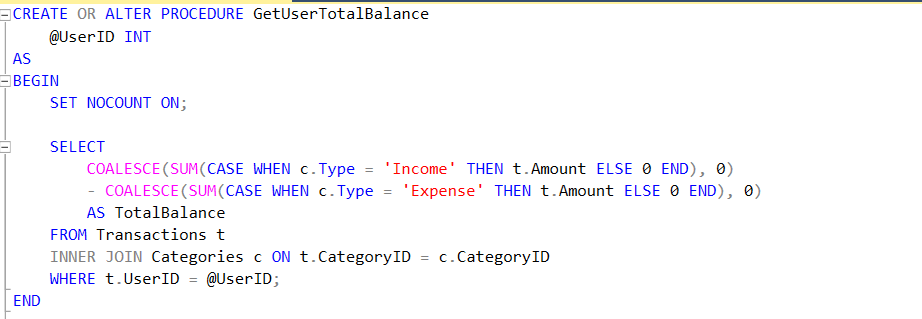
****

Figure 26: Stored Procedure GetUserTotalBalance

* The stored procedure GetUserTotalBalance takes @UserID as input.
* It calculates **total balance** as: SUM(Income) - SUM(Expense) by joining the Transactions table with the Categories table to determine the transaction type.
* COALESCE is used to ensure that NULL sums are treated as 0.
* The result is returned as a single value TotalBalance.
* In Java, the DAO method

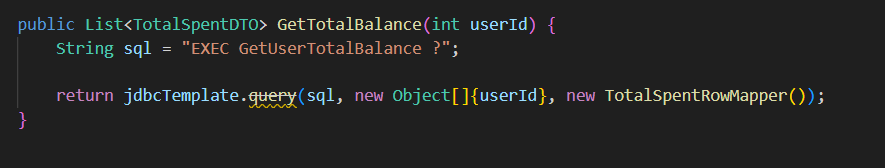


Figure 27: Retrieve total user balance via stored procedure in Java

runs the SQL command EXEC GetUserTotalBalance ? using JdbcTemplate, mapping the result to a list of TotalSpentDTO via TotalSpentRowMapper.

* The service method

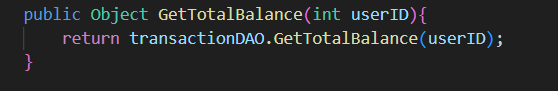
****

Figure 28: Service method retrieving total balance from DAO

simply delegates to the DAO method and returns the result.

* The controller endpoint

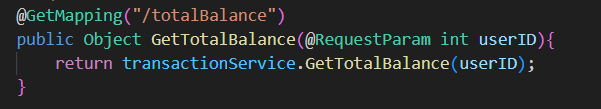


Figure 29: Spring Boot GET endpoint for retrieving total balance

receives the request parameter from the frontend and calls the service method.

* **Frontend rendering**
* The frontend reads totalBalance from the first item in the returned array.
* If no data is returned, it defaults the displayed balance to 0.

1. **Transaction History**

Transaction history requires some extra fields that its own table does not own.To do this, we use Store Procedure in SQL Server and then call it with proper input information to receive the expected information. Following is the SP we used for showing detailed history transaction:

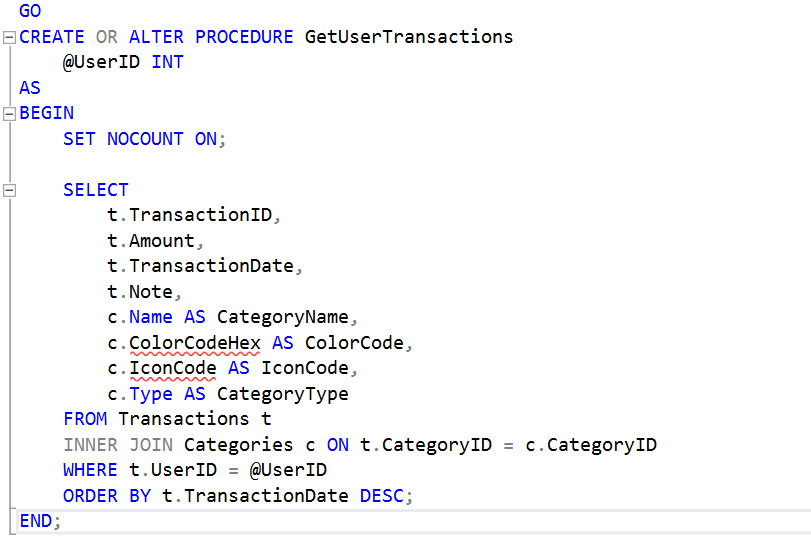


Figure 30: Stored Procedure GetUserTransactions

The SP requires a user's ID for finding the correct history transactions. The extra fields that the SP return are Category’s name, Icon Code and Category’s type.

The result is then processed by the DAO (TransactionDAO), which is then mapped to the DTO (TransactionHistoryDTO). Following is the detailed code in Transaction DAO class

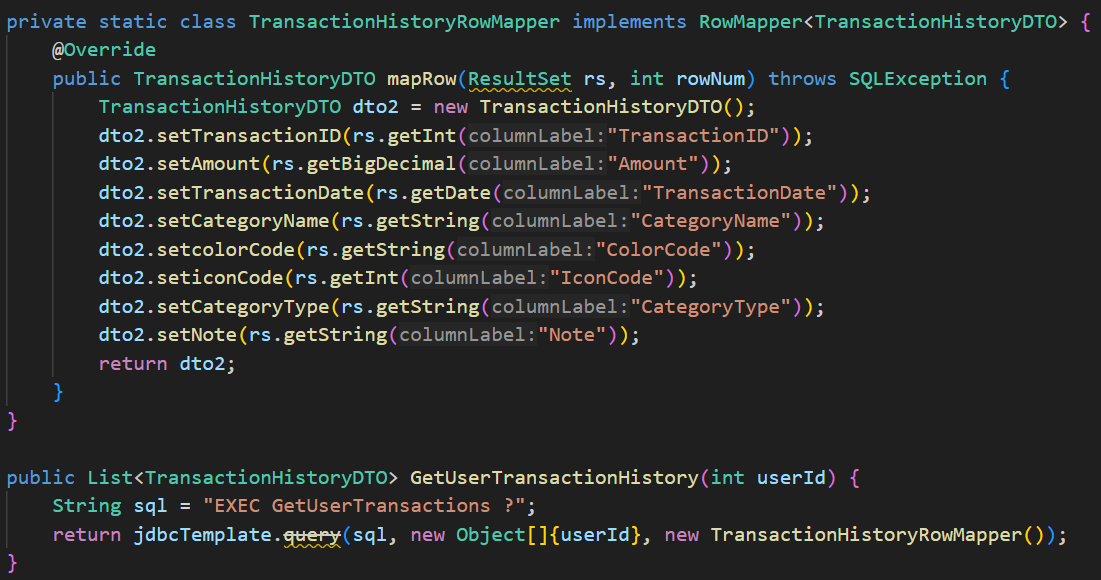


Figure 31: Mapping and retrieving user transaction history via stored procedure in Java

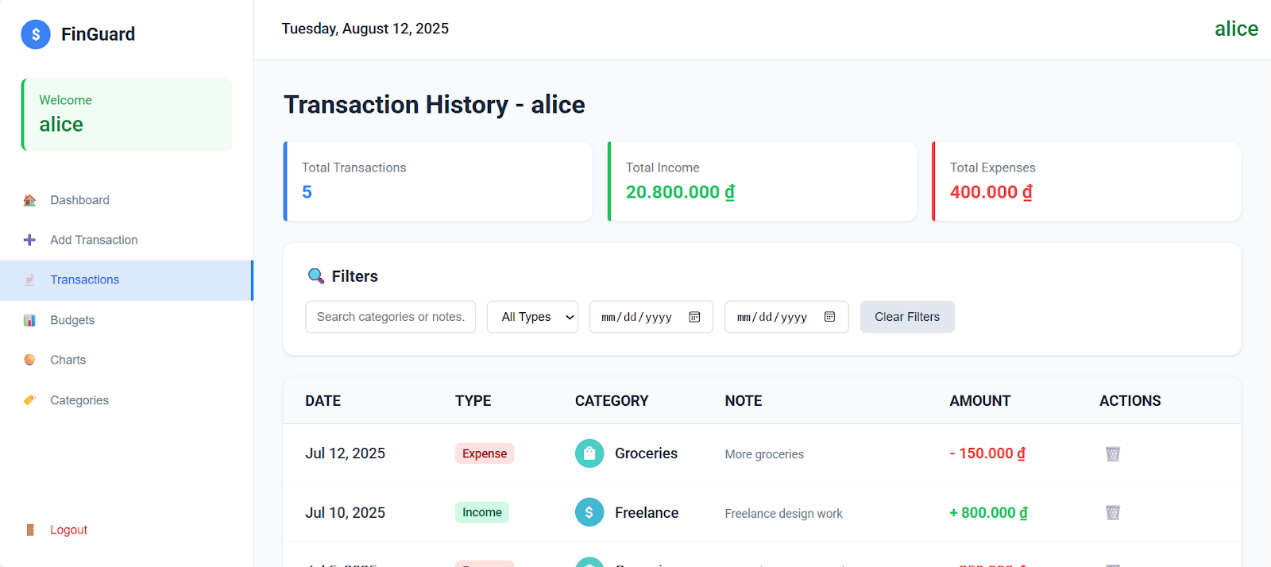
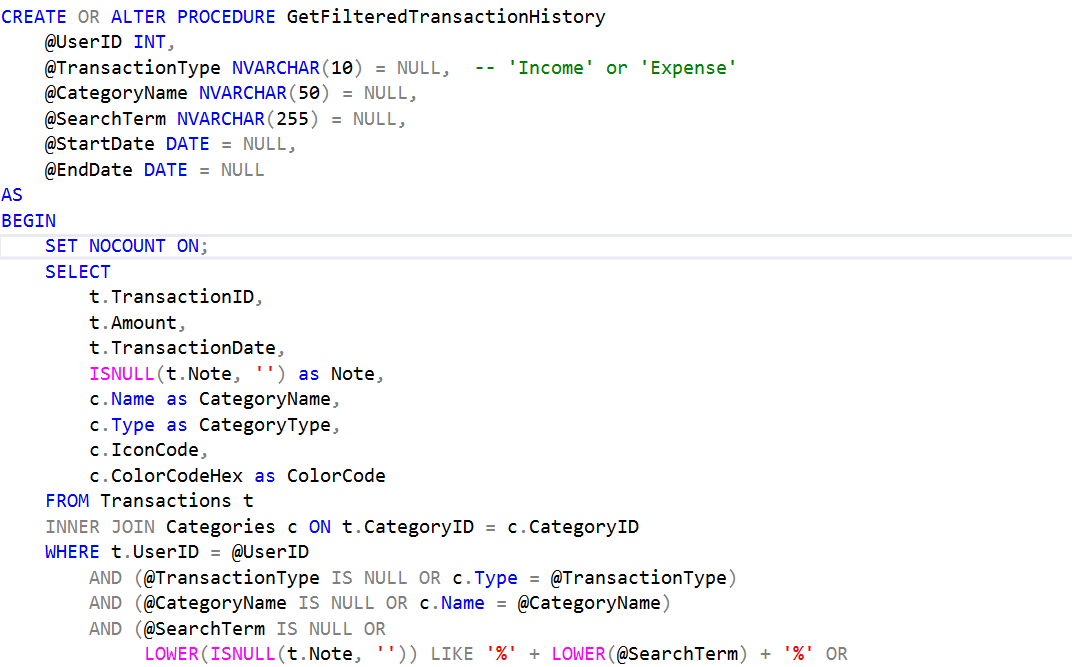
Front end will receive the resulted list and show on the screen  


Figure 32:Transaction History Frontend

1. **Filter (Transaction History)**

To make it easier for users to retrieve data when the database contains a large amount of information, we decided to implement filtering on the backend instead of handling it directly on the frontend.We use Store Procedure in SQL Server and then call it with proper input information to receive the expected information



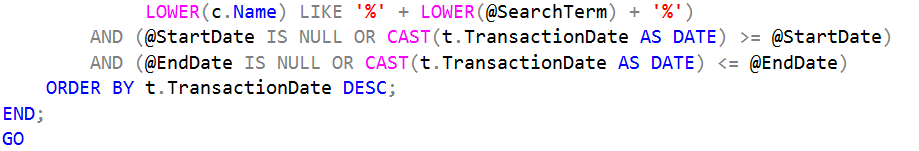


Figure 33: Stored Procedure GetFilteredTransactionHistory

This SP is designed to retrieve a filtered list of transaction history for a specific user. It requires the UserID parameter and allows optional filters such as transaction type (Income or Expense), category name, search term, and date range.

The result is then processed by the DAO (TransactionDAO). After receiving the parameters from the upper layer, the method logs them for debugging purposes and processes the input data, such as converting LocalDate to java.sql.Date for compatibility with SQL Server and replacing empty strings with null so that the Stored Procedure interprets them as no filter applied. Next, the method prepares the SQL statement to call the Stored Procedure with six parameters, then executes it using jdbcTemplate.query and maps the returned results to a list of DTOs via TransactionHistoryRowMapper. Finally, the result is returned to the calling layer.

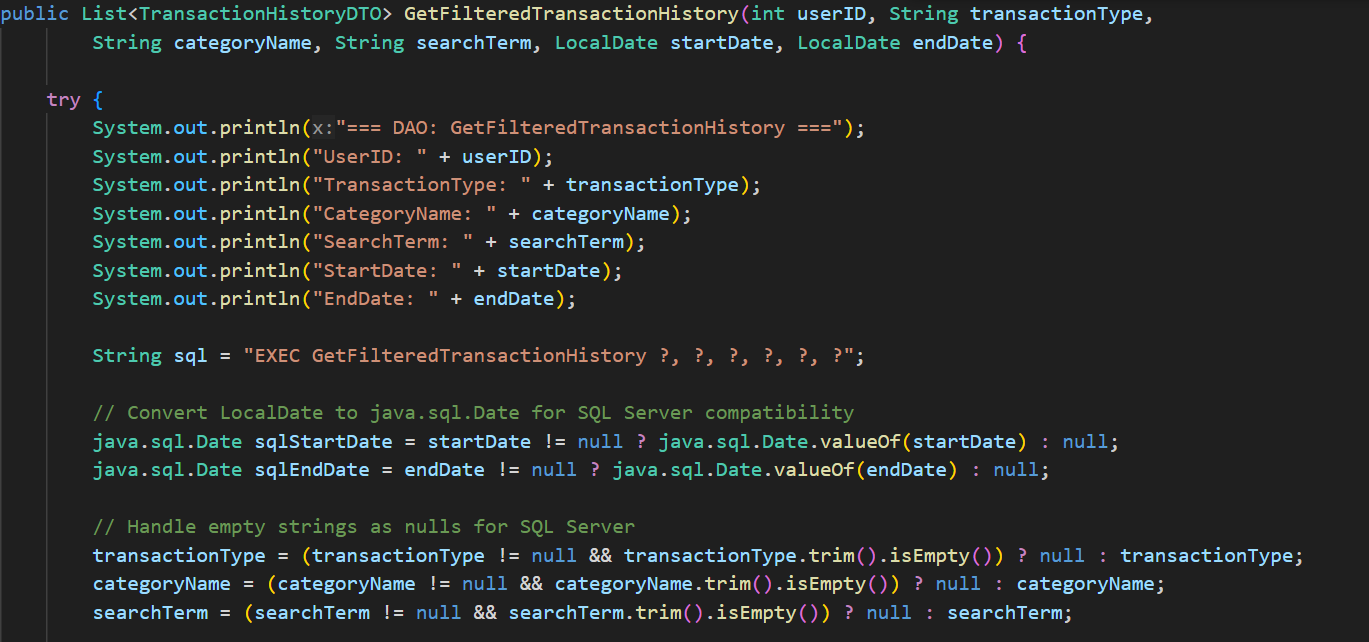




Figure 34: DAO method for retrieving filtered transaction history with SQL Server date handling

1. **Category:**

Category Management allows users to add, edit, delete, and view income and expense categories. When the page loads, the frontend calls

**GET /api/categories?userId={userId}**

with a JWT token to retrieve the category list from the database through the Controller → Service → Repository layers, then renders them into two columns (Income and Expense). When adding a new category, the user enters the name, type, color, and icon; the frontend sends a

**POST /api/categories/insert**

request with the data and token. The Controller receives the request, the Service processes the logic (including checking for duplicates within the same type), saves it to the database, and returns a result so the UI can refresh the list. For editing, the process is similar but uses

**PUT /api/categories/update;**

the existing data is pre-filled into the modal, and the Service updates the record in the database. For deletion, the frontend sends

**DELETE /api/categories/delete?id={id},**

the Service removes the record, returns the result, and the UI shows a confirmation message before reloading the list.

# CHAPTER VII: LIMITATIONS AND FUTURE WORK

This study and system implementation, while functional, still have certain limitations. The current system focuses primarily on core budgeting, transaction tracking, and category management features, without incorporating advanced analytics, AI-driven insights, or predictive financial planning. Additionally, the performance and scalability have not been extensively tested for very large datasets or high concurrent user loads. Some aspects, such as security measures and data backup strategies, can be further enhanced to meet enterprise-level standards. In future work, the system could be expanded with mobile application support, real-time synchronization across devices, and integration with external financial APIs to automatically import bank transactions. Furthermore, implementing machine learning models to provide spending forecasts, personalized saving tips, and anomaly detection would significantly improve user experience and decision-making support.

# CONCLUSION

The developed personal finance management system successfully meets its primary objectives of enabling users to track transactions, manage budgets, categorize expenses and income, and generate monthly financial summaries. Through a well-structured database design, clear API endpoints, and efficient data access methods, the system provides a reliable and user-friendly platform for managing personal finances. The modular architecture allows for easy maintenance and future feature expansion, while the integration of budget planning and prefix templates helps users save time and maintain consistent financial control. Although there are areas for improvement, such as enhancing security, performance optimization, and introducing predictive analytics, the current implementation offers a solid foundation for further development and demonstrates the feasibility and effectiveness of the proposed approach.

# REFERENCES

|  |  |
| --- | --- |
| [1] | Lusardi, Annamaria and Mitchell, Olivia S, "The economic importance of financial literacy: Theory and evidence," *American Economic Journal: Journal of Economic Literature,* vol. 52, pp. 5--44, 2014. |
| [2] | Angrisani, Marco and Kapteyn, Arie and Lusardi, Annamaria, "The national financial capability study: empirical findings from the American Life Panel Survey," *FINRA report,* vol. 43, 2016. |
| [3] | Lusardi, Annamaria, "Financial literacy and the need for financial education: evidence and implications," *Swiss journal of economics and statistics,* vol. 155, pp. 1--8, 2019. |