**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. 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. This often leads to overspending, lack of savings, and poor financial decisions.

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 simple and user-friendly interface, users can easily understand how they spend their money and make better financial choices. This project not only gives users a useful tool but also encourages better habits in managing personal finances every day.

# TABLE OF CONTENTS

# 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 | 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 | 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 | 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 | API to get current balance | 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** | | | | | |
| 11 | Build Login & Register pages | UI for login and signup | Trần Phúc Công | Week 4 | Responsive login/register forms |
| 12 | Build Dashboard page | Show balance overview | Trần Quỳnh Anh | Dashboard UI with balance info |
| 13 | Build Category Management UI | Manage categories (CRUD) | Trần Duy Anh | Category list, add/edit/delete |
| 14 | Build Transaction UI | Add transactions | Trần Quỳnh Anh | Transaction form and listing |
| 15 | Build Transaction History page | Full list with filter options | Đoàn Anh Vũ | Table with filter for history |
| 16 | Display Spending Statistics | Charts for category/month | Trần Quỳnh Anh | Charts using Chart.js/ApexCharts |
| **Testing & Finalization** | | | | | |
| 17 | Test API and Features | Manual testing, bug fixing | Trần Duy Anh | Week 5 | Test report, bug list |
| 18 | Write Report | Final report and presentation | Trần Quỳnh Anh | Word file, presentation slides |

# CHAPTER I: PROJECT OVERVIEW

## Project Introduction

Personal finance management is an important skill for everyone. It helps people plan their budget, save money, and avoid unnecessary spending. However, many people still do not track their income and expenses regularly. This often leads to poor financial decisions and lack of savings.

In today’s digital age, a web-based system for personal finance management can be very helpful. Unlike mobile apps, a web system allows users to manage their finances from any device with an internet connection, such as a laptop or desktop computer. It is also easier to handle large amounts of data and generate detailed financial reports.

This project aims to build a simple and user-friendly web application to help users manage their money. The system allows users to add income and expenses, set monthly budgets, and view financial summaries. By using this system, users can better understand their spending habits and improve their financial planning.

## Problem Statement & Objectives

Managing money is not always easy. Many people spend without knowing how much they earn or how much they already spent. This can lead to overspending, not having enough savings, and stress when facing financial problems. Some people try to write down their expenses, but it is hard to keep track without a good system.

To solve this issue, we want to build a simple web application that helps users manage their personal finances. The system should be easy to use, even for people who are not good with computers. It should let users track their income and expenses, set budgets, and see their financial summary at any time.

### 2.1 Objectives of the application

* Help users record their income and expenses quickly and easily.
* Allow users to create and manage monthly budgets.
* Show users charts and summaries of their financial activities.
* Make users more aware of their spending habits.
* Encourage better financial planning and saving.

### 2.2 Target Users

The application is designed for a broad range of users, including:

* Students learning to manage their personal finances independently.
* Young professionals seeking to monitor their income and spending habits.
* Individuals or families who want to plan budgets and save for future goals.
* Anyone looking for a simple and 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

|  |  |
| --- | --- |
| **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 |

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

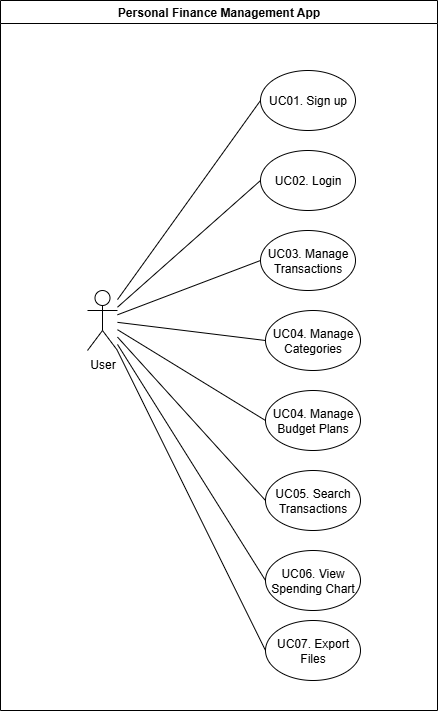
## Writing the code

## Testing and fixing bugs

## Reviewing and improving

# CHAPTER IV: SYSTEM DESIGN

1. Use Case Diagrams



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:

**Users Table:**

**Containing user’s account basic information.**

|  |  |  |
| --- | --- | --- |
| **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.** |

**Categories Table:**

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

|  |  |  |
| --- | --- | --- |
| **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.** |

**Transactions Table:**

Containing transactions of each user for a single category of them.

|  |  |  |
| --- | --- | --- |
| **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.** |

**Budgets Table:**

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

|  |  |  |
| --- | --- | --- |
| **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.** |

**Prefixes Table:**

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

|  |  |  |
| --- | --- | --- |
| **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** |

**Entity Relationship Diagram (ERD):**

**A diagram of a computer

AI-generated content may be incorrect.**

**Database Diagram:**

**A screenshot of a computer

AI-generated content may be incorrect.**

1. API Design

|  |  |  |  |
| --- | --- | --- | --- |
| **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 |

# CHAPTER V: USER INTERFACE DESIGN

A login screen with blue and white text

AI-generated content may be incorrect.

*Login page*

A screen shot of a login form

AI-generated content may be incorrect.

*Logout page*

A screenshot of a computer

AI-generated content may be incorrect.

*Dashboard page*

A screenshot of a computer

AI-generated content may be incorrect.

*New Transaction page*

A screenshot of a computer

AI-generated content may be incorrect.*Transaction History page*

A screenshot of a computer

AI-generated content may be incorrect.

*Budget Management page*

A graph on a white background

AI-generated content may be incorrect.*Visualized Chart page*

A screenshot of a computer

AI-generated content may be incorrect.

*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.

**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.

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.**

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.

**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.

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.

**Transaction:**

**Dashboard:**

**Category:**

**Testing**

# CHAPTER VII: LIMITATIONS AND FUTURE WORK

# PART VIII: CONCLUSION