SUMMER TRAINING/INTERNSHIP

PROJECT REPORT

(Term June-July 2025)

SMART EXPENSE TRACKER

Submitted by

Ben Gregory John
Registration Number: 12315900

Jerome Philip John
Registration Number: 12321851

Md Aquib Raza
Registration Number: 12324762

Course Code: CSE 343

Under the Guidance of
Mr. Sarvesh Chopra
(Asst. Professor)

School of Computer Science and Engineering

CERTIFICATE

ACKNOWLEDGEMENT

We would like to express our deepest gratitude to all those who supported and guided us throughout the development of this project, Smart Expense Tracker. This summer training experience has been a valuable milestone in our academic journey.

First and foremost, we are extremely thankful to **Mr. Sarvesh Chopra**, our mentor, for his continuous support, expert guidance, and insightful feedback throughout the training period. His encouragement and constructive criticism helped us refine our ideas and improve the project quality.

We would also like to thank the **faculty and staff of the School of Computer Science and Engineering** for providing us with the platform and resources to undertake this project. Their commitment to academic excellence has always inspired us.

Special thanks to our project team — **Ben Gregory John**, **Jerome Philip John**, and **Md Aquib Raza** — for their amazing collaboration, mutual respect, and the shared enthusiasm that made this project possible. Every brainstorming session, late-night debugging, and design decision was a learning experience.

We would also like to extend our gratitude to our friends and families who supported us emotionally and mentally during this phase. Their patience, motivation, and unwavering belief in us gave us the strength to complete this work with passion.

This project has not only enhanced our technical skills in Java, JavaFX, and database management but has also taught us the importance of collaboration, time management, and perseverance. We are truly grateful for this opportunity.

.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Company Profile

This project was undertaken under the aegis of the **School of Computer Science and Engineering**, as part of the academic curriculum intended to simulate industry-level software development. Although there is no external company involvement, the entire development process adhered to professional software engineering standards typically followed by leading IT firms.

The project, titled **Smart Expense Tracker**, mirrors the core functionalities found in modern fintech solutions such as Walnut, Goodbudget, and YNAB (You Need A Budget). Each team member assumed a well-defined role — ranging from system analyst, developer, designer, to quality assurance engineer — mimicking the collaborative dynamics of a real-world software development team.

This simulation fostered an environment that emphasized technical rigor, documentation discipline, code modularity, and proper version control, thereby enriching the learning experience. The team adopted Agile-like development practices, breaking the work into modules such as GUI design, database connectivity, and expense visualization, following iterative and incremental enhancements.

Through this project, we gained firsthand exposure to a wide array of technologies and tools commonly used in the software industry, including:

- **JavaFX** for front-end GUI development
- **MySQL** for data persistence
- **JDBC** for database connectivity
- **Maven** for dependency and build management

This approach also helped bridge the gap between theoretical coursework and practical, application-driven learning, making the experience both educational and industry-relevant.

Overview of Training Domain

The core technical training domain was **Java-based desktop application development**, with a strong emphasis on building scalable, user-interactive GUI applications. The focus remained on using **Core Java**, supplemented with modern Java tools and libraries to create a real-world desktop software solution.

The key domains covered include:

- **Object-Oriented Programming (OOP):**
  Core OOP principles such as encapsulation, inheritance, polymorphism, and abstraction were actively used to represent real-world entities like users, income, expense and budget models. This helped maintain code reusability and modularity.

- **JavaFX GUI Development:**
  The interface was built using **JavaFX**, utilizing features like FXML, CSS-based styling, and Scene Builder to deliver a professional and responsive user experience. Layout managers like VBox, HBox, GridPane, and BorderPane were employed for structured designs. User actions were handled using **event-driven programming**.

- **JDBC and Database Integration:**
  To ensure persistent data storage, the application integrated **MySQL** using **Java Database Connectivity (JDBC)**. We implemented complete **CRUD** (Create, Read, Update, Delete) operations securely with prepared statements, minimizing SQL injection risks and ensuring real-time synchronization of user data.

- **Multithreading:**
  Tasks such as background data processing, delayed alerts for budget overruns, and smooth chart rendering were managed using **Java's multithreading capabilities**.

This ensured the UI remained responsive during time-intensive operations Budget alert checks in BudgetService (checkBudgetAlerts) were optimized to run efficiently, updating the dashboard's budgetAlertLabel without blocking the main thread. This approach ensures smooth navigation and real-time feedback for users.

- **Maven Project Management:**
  We structured the project using **Apache Maven**, allowing for clean dependency management, modular code organization, and automated build processes. This made the codebase easier to scale and maintain.

- **Testing and Error Handling:**
  Rigorous testing strategies were applied using manual test cases to validate functionality. Robust exception handling was integrated to provide user-friendly error messages and system resilience.

- **Enum-Based Data Validation**:
  The application uses enums (IncomeSource, ExpenseCategory, BudgetPeriod) to enforce data consistency and simplify validation. For instance, IncomeSource (e.g., SALARY, FREELANCE) and ExpenseCategory (e.g., FOOD, TRANSPORT) restrict dropdown inputs to predefined values, reducing errors. BudgetPeriod (WEEKLY, MONTHLY) ensures budgets are set with valid timeframes. This approach minimizes invalid data entry and aligns with database schema constraints, enhancing data integrity.

- **Session Management**:
  A Session Manager **singleton class** was implemented to maintain user sessions across the application. Upon successful login, the loggedInUser is stored, enabling user-specific data retrieval (e.g., getExpensesByUser(SessionManager.getInstance().getLoggedInUser().getId())). This ensures secure, personalized access to financial data and prevents unauthorized access to other users' records.

- **Version Control and Documentation**:

The project is hosted on GitHub (https://github.com/BenGJ10/Smart-Expense-Tracker), utilizing Git for version control to track changes and maintain a commit history. Comprehensive documentation includes an updated README.md with setup instructions, feature descriptions, and screenshots, as well as inline code comments for clarity.

Overall, this domain of training mirrors real-life software engineering practices and simulates internship-level exposure, preparing us for professional roles in software development and system design.

Objective of the Project

The primary goal of this project is to design and develop a **Smart Expense Tracker** — a comprehensive, user-centric desktop application that helps users manage personal finances efficiently, offering practical utility and intuitive usability.

The modern digital lifestyle involves various types of expenses — subscriptions, utility bills, food, transport, entertainment, and others — often spread across multiple platforms. People often lose track of where their money goes. This application addresses this problem by providing a **centralized platform for expense and income management**, enabling users to understand, visualize, and control their financial behavior.

Key functional objectives:

- Enable users to **record income and expense entries** under various categories (e.g., food, rent, transport).
- Allow users to **set monthly or category-wise budgets**, and raise alerts when nearing or exceeding those limits.
- Generate **data-driven insights** via reports and **visual representations** such as pie charts and bar graphs using JavaFX's chart libraries.

- Store data persistently using a **MySQL backend**, ensuring that the application is not just session-based, but also practical for long-term financial tracking.
- Provide a clean, modern, and responsive **JavaFX interface** that improves the overall user experience and accessibility.

Broader Learning Objectives:

- Apply classroom knowledge in **real-world development scenarios**.
- Strengthen problem-solving and debugging skills during integration phases.
- Gain experience in **UI/UX design**, **data modelling**, and **multilayer architecture**.
- Understand and apply concepts of **modular development**, **separation of concerns**, and **user-centric design principles**.
- Prepare ourselves for industrial software development environments by simulating full-cycle project implementation.

In conclusion, the **Smart Expense Tracker** project not only fulfils a practical personal finance need but also serves as a learning platform that enhances our skills in programming, system architecture, and software engineering — laying a strong foundation for future careers in technology.

CHAPTER 2: TRAINING OVERVIEW

Tools & Technologies Used

The development of the **Smart Expense Tracker** leveraged a modern software stack to simulate real-world desktop application development practices. The following tools and technologies were used throughout the training and implementation phases:

- **Java 21**: The core programming language for logic implementation, chosen for its stability, cross-platform support, and robust object-oriented features.
- **JavaFX**: A GUI toolkit used to build the desktop interface with components such as buttons, text fields, tables, and charts. It supports both FXML-based and programmatic UI creation, enabling responsive and modern design patterns.
- **MySQL**: A widely used relational database management system that served as the backend storage solution. MySQL provided reliability, ACID compliance, and scalability for persistent user data.
- **JDBC (Java Database Connectivity)**: Enabled interaction between the Java application and the MySQL database. JDBC allowed for executing SQL queries, managing transactions, and performing CRUD operations.
- **IntelliJ IDEA**: A powerful Java IDE used for code development and debugging. IntelliJ's intelligent code suggestions, integrated version control, and project structure handling accelerated the development workflow.
- **Scene Builder**: A visual layout tool for JavaFX used to design user interfaces via drag-and-drop, allowing separation of GUI design from application logic using FXML files.
- **Apache Maven**: A build automation tool used to manage dependencies, standardize the project structure, and streamline the compilation, packaging, and deployment processes.

These technologies, when combined, provided a full-stack solution for desktop application development and helped create a modular, maintainable, and production-ready software prototype.

Areas Covered During Training

The training emphasized core software development concepts applicable to modern enterprise-level applications. The following skill areas were covered:

- **Core Java**: Fundamental Java concepts were extensively applied to build a robust application foundation. This included leveraging Java's type system, control structures (e.g., loops, conditionals), and collections framework (e.g., ArrayList, ObservableList) for managing data in memory. Classes and objects were used to model entities like User, Income, Expense, and Budget, with constructors and methods ensuring proper initialization and behavior. Java's access modifiers (e.g., private, public) were utilized to enforce encapsulation, while static members and singletons (e.g., SessionManager) facilitated shared resource management. The use of generics in service and DAO interfaces (e.g., List<Expense>) ensured type safety and code reusability.

- **GUI Design and Layouts with JavaFX**: We explored advanced layout structures (VBox, GridPane, StackPane, etc.), CSS styling for themes, and scene transitions. Using Scene Builder and FXML, we achieved clean separation between the logic and the UI, ensuring maintainable code.

- **OOP-Based Architecture**: Java's Object-Oriented features were thoroughly applied to model real-world entities. Classes such as User, Expense, Budget, and DAO interfaces were implemented, reinforcing encapsulation, inheritance, and polymorphism.

- **Exception Handling Techniques**: Robust error management was added using try-catch-finally blocks, custom exceptions, and user-friendly alerts. This improved application stability and prevented runtime crashes.

- **Multithreading in Java**: Threads were implemented for background tasks like database fetches and real-time budget checks, ensuring that the UI remained responsive during long-running operations.
- **Database Connection using JDBC**: We developed database handler classes to manage connections, SQL execution, and data retrieval. Use of PreparedStatement ensured protection against SQL injection.

These topics were not just studied theoretically but also applied practically during the project lifecycle, reinforcing hands-on understanding of key software engineering principles.

Weekly Work Summary

Week 1: Project Setup, Research, and Database Design

The first week laid the groundwork for the project. The team collaboratively:

- Defined the project scope: a JavaFX-based desktop application for managing user authentication, income, expenses, budgets, and financial summaries with budget alerts.
- Researched technologies, selecting Java 21, JavaFX for the UI, MySQL for data storage, and Maven for project management.
- Configured the development environment using IntelliJ IDEA, setting up Maven with dependencies for JavaFX (javafx-controls, javafx-fxml) and MySQL Connector (mysql-connector-java).
- Designed the MySQL database schema for core entities: users, incomes, expenses, and budgets. Tables were normalized with foreign key constraints (e.g., user_id in incomes referencing users) to ensure data integrity.
- Created SQL scripts to initialize the database (expense_tracker_db) and tables, including sample data for testing (e.g., a test user with username “testuser”).
- Implemented DatabaseConnection.java and db.properties for secure database connectivity, using JDBC to manage connections.

- Set up the project structure (controller, dao, model, service, util packages) and initialized GitHub repository. This week concluded with a functional database and a clear development roadmap.

Week 2: GUI Layout for Login, Dashboard and User Authentication

The second week focused on user interface design:

- Developed JavaFX scenes using FXML and Scene Builder for modularity, creating screens for login (login.fxml), registration (register.fxml), dashboard (dashboard.fxml), income management (add_income.fxml), expense management (add_expense.fxml), and profile management (profile.fxml).
- Applied a dark-themed CSS stylesheet (styles.css) with consistent styling (Roboto font, #2B2D42 background, #206dc5 buttons) for a modern, cohesive aesthetic.
- Implemented navigation logic between screens using controller classes (e.g., LoginController, DashboardController) and event handlers (e.g., handleBackToDashboard).
- Built user authentication with UserService and UserDAO, supporting login and registration with password hashing (PasswordUtil) and session management (SessionManager singleton).
- Added basic form validation in LoginController and RegisterController (e.g., checking for empty fields, duplicate usernames) with error messages displayed via errorLabel.
- Ensured responsive UI with VBox, HBox, and GridPane layouts, supporting keyboard and mouse interactions. By the end of this week, the application had a functional front end with smooth navigation and secure user authentication.

Week 3: Income and Expense Management with DAO Integration

The third week focused on implementing transaction management and database integration:

- Developed Income and Expense models with IncomeSource and ExpenseCategory enums for structured data entry.
- Created IncomeDAO and ExpenseDAO classes to handle CRUD operations (Create, Read, Update, Delete) using JDBC with prepared statements to prevent SQL injection (e.g., INSERT INTO incomes (user_id, amount, source, date, description) VALUES (?, ?, ?, ?, ?)).
- Implemented IncomeService and ExpenseService to validate inputs (e.g., positive amounts, non-null sources/categories) and interact with DAOs.
- Built AddIncomeController and AddExpenseController to support adding, viewing, and deleting income and expense records, with TableView displaying real-time data.
- Added CustomLogger to log user actions (e.g., "Added income: ₹10000") and errors (e.g., "Failed to load expenses") to logs.log for debugging.
- Integrated basic edit functionality for incomes and expenses, allowing users to modify existing records via table action buttons (Edit/Delete).
- Handled exceptions (InvalidInputException, DatabaseException) to display user-friendly error messages in the UI (e.g., "Invalid amount format"). This week achieved full-stack integration for income and expense management, with robust error handling and logging.

Week 4: Final Logic, Charts Integration, and Testing

The final week completed the core functionality, testing, and documentation:

- Finalized budget management by implementing Budget model with BudgetPeriod enum (WEEKLY, MONTHLY) and completing AddBudgetController, BudgetService, and BudgetDAO for full CRUD operations.
- Added real-time budget alerts in DashboardController using BudgetService.checkBudgetAlerts, displaying warnings (e.g., "Budget exceeded for Food (MONTHLY): ₹8000/₹5000") when expenses exceed budgets.
- Enhanced edit functionality for incomes, expenses, and budgets, ensuring TableView updates in real-time after modifications and fixing issues with new entry creation during edits.

- Used JavaFX's Task for background processing of budget alert checks to maintain UI responsiveness.
- Conducted manual testing for all features (login, dashboard, income, expense, budget, profile, alerts), covering edge cases (e.g., invalid inputs, database errors) and verifying table updates.
- Wrote unit tests for DAOs (UserDAOTest, IncomeDAOTest, ExpenseDAOTest, BudgetDAOTest) using JUnit to validate database operations.
- Finalized documentation, including:
    - Updated README.md with setup instructions, feature descriptions, and screenshots (docs/screenshots/).
    - Created a demo video (docs/demo.mp4) showcasing the application and code walkthrough.
    - Added inline code comments and a project report detailing technical domains (OOP, JavaFX, JDBC, etc.).

CHAPTER 3: PROJECT DETAILS

Title of the Project: Smart Expense Tracker

This project is a Java-based desktop application designed to assist individuals in managing their personal finances effectively. Built with modern technologies such as JavaFX, JDBC, and MySQL, the application focuses on usability, reliability, and insightful analytics for tracking and budgeting daily expenses.

Problem Definition

In an increasingly digital and fast-paced world, individuals face growing complexity in managing their personal finances. Traditional methods such as writing expenses in notebooks or using unsynchronized spreadsheets are:

- **Manual and error-prone**

- **Difficult to update and maintain**
- **Lacking in real-time alerts and reports**
- **Not scalable or accessible across time periods**

Such practices often lead to financial disorganization, unplanned spending, and missed savings targets. Furthermore, users typically lack visibility into where their money goes, making it difficult to assess or correct spending behavior.

There is a pressing need for a **smart and intuitive personal finance solution** that offers:

- **Automated transaction logging**
- **Real-time budget tracking**
- **Visual financial analytics**
- **Persistent and secure storage**

The **Smart Expense Tracker** addresses these needs by providing an end-to-end solution that combines an intuitive GUI, database connectivity, automated calculations, and category-wise budgeting. Users are empowered with tools to **log transactions**, **set financial goals**, and **gain visual insights into their spending behavior** — all within a structured and responsive desktop application.

Scope and Objectives

The **Smart Expense Tracker** is designed with the aim of enhancing users' financial awareness and discipline. It provides a complete solution for tracking, analyzing, and managing personal finances efficiently.

Scope of the Project

The scope of this project is limited to a **desktop-based single-user application**. It supports:

- Entry and management of income and expenses

- Category-based budget creation and alerts
- Visual financial summaries using charts
- Persistent data storage via MySQL

The application focuses on **simplicity, privacy, and offline accessibility**, catering especially to users who prefer local desktop tools over mobile apps or cloud-based platforms.

Objectives

- **Track Income and Expenses:**
  The application allows users to log all financial transactions, whether income (salary, freelance, etc.) or expenses (rent, food, subscriptions), with attributes such as amount, date, category, and description. This structured logging helps users maintain a comprehensive financial record.
- **Create and Monitor Monthly Budgets:**
  Users can define budgets either by category (e.g., ₹5000 for food) or for overall monthly spending. The system automatically monitors actual expenses against these limits and raises alerts when usage exceeds safe thresholds.
- **Ensure Data Persistence and Security:**
  All transactions are stored in a **MySQL database** using **JDBC connectivity**, ensuring that data is saved securely and remains available even after application restarts. Database access is abstracted through DAO classes, and only authenticated users can interact with financial data.
- **Deliver a Smooth and Intuitive User Experience:**
  The application leverages **JavaFX** to provide a modern, responsive, and interactive GUI. Through the use of **FXML layouts**, **CSS styling**, and proper event handling, the application ensures that users can easily navigate between screens and interact with the application without needing technical knowledge.

System Requirements

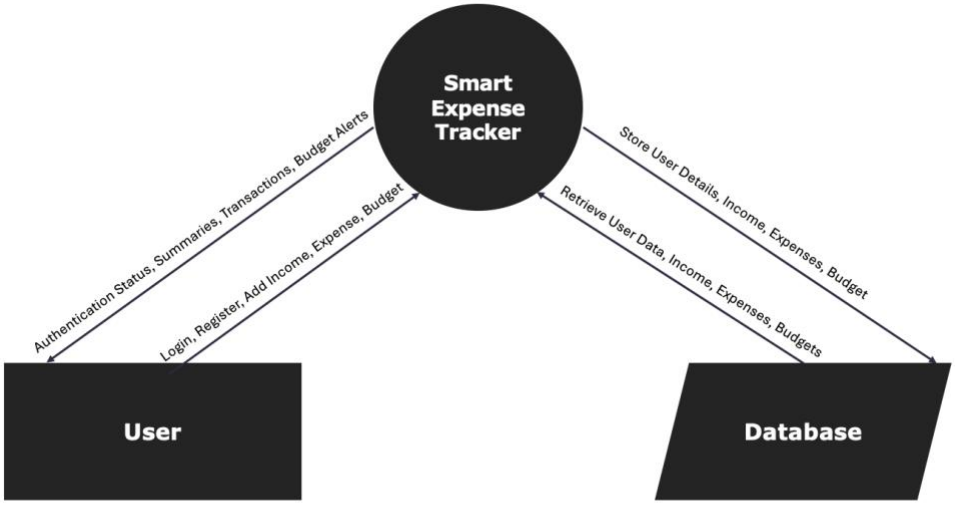
Software Requirements:

Component	Details
Programming Language	Java 21 or above
GUI Framework	JavaFX SDK
Database	MySQL Server (v8.x recommended)
Build Tool	Apache Maven
IDE	IntelliJ IDEA (or Eclipse)
Additional Tools	Scene Builder (for GUI design)

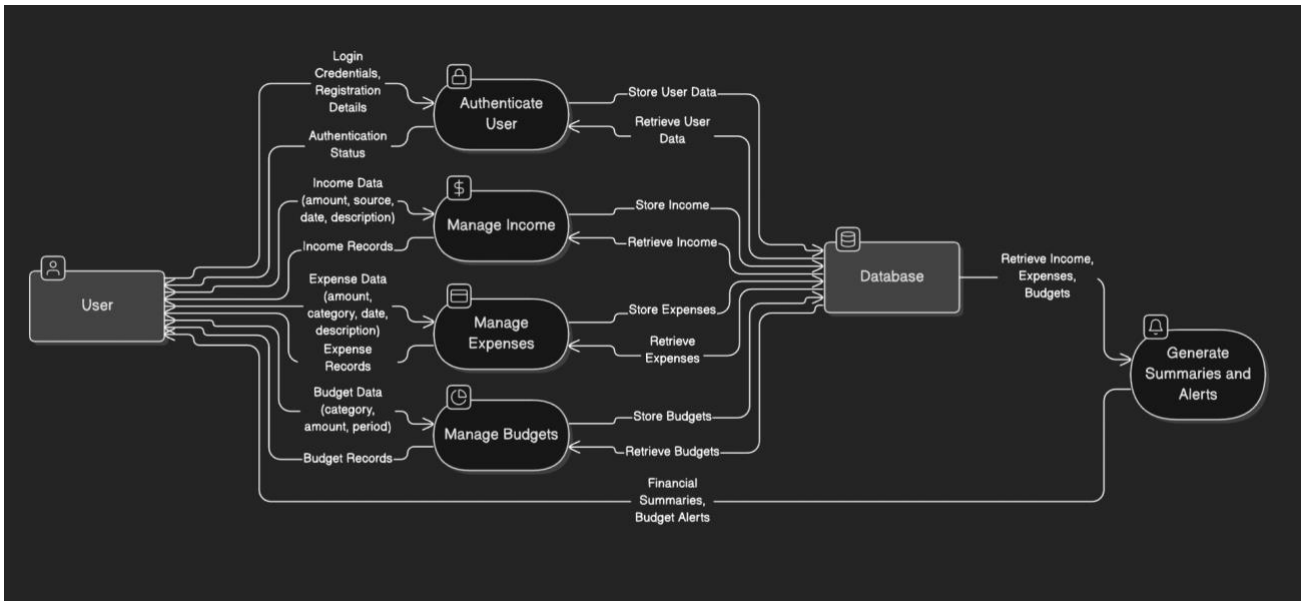
Hardware Requirements:

Component	Minimum Requirement
RAM	4 GB or more
Processor	Dual-core 2.0 GHz or above
Storage	200 MB for project files and DB
Display	Minimum 1024x768 resolution
Operating System	Windows 10 / Linux / macOS

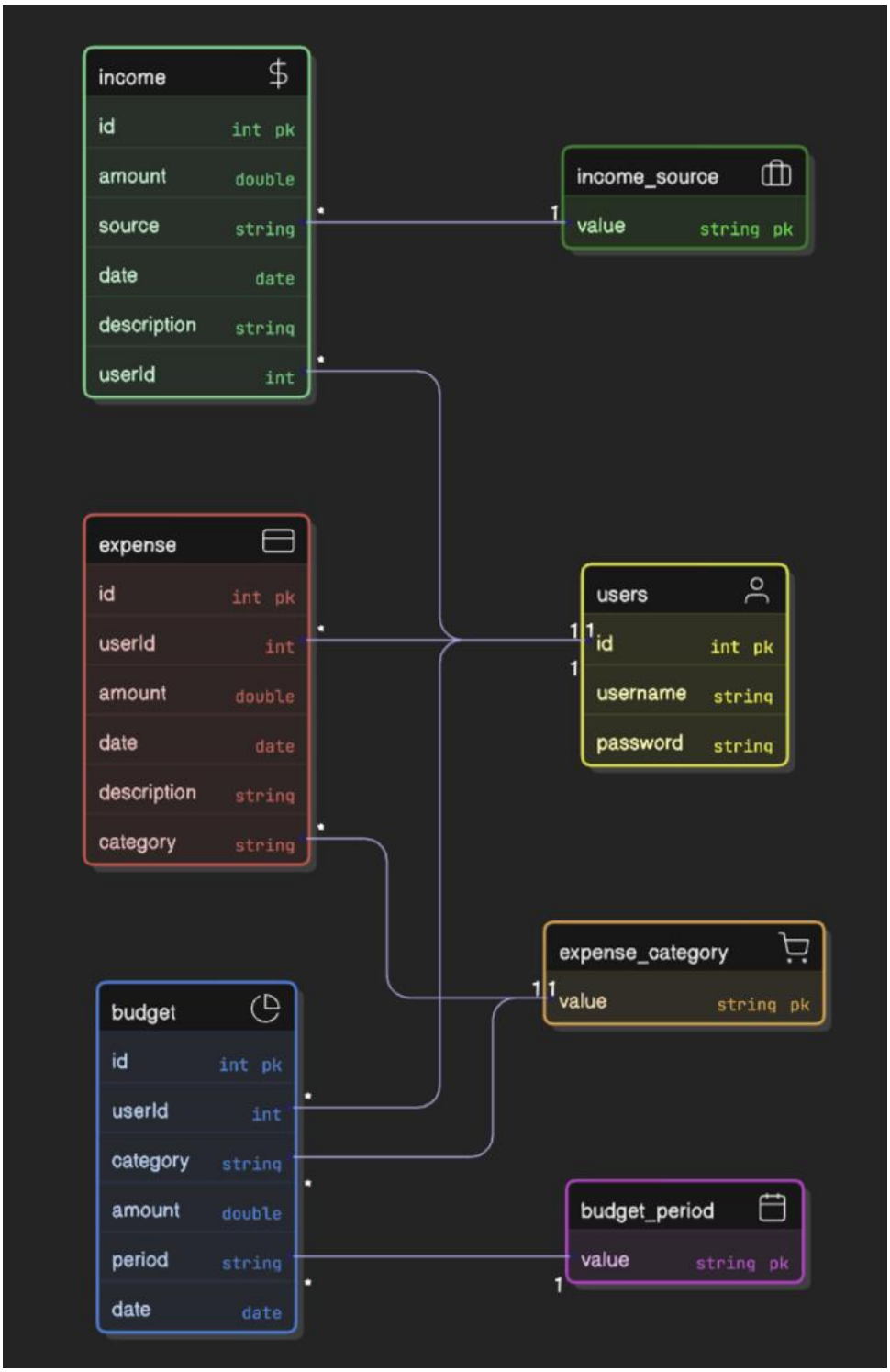
Level 0 Data Flow Diagram



Level 1 Data Flow Diagram



UML Use Case Diagram



CHAPTER 4: IMPLEMENTATION

Tools Used

The development of the Smart Expense Tracker utilized a range of industry-standard tools and platforms. Each tool served a specific purpose in the software development lifecycle and contributed to the robustness, usability, and maintainability of the application.

Tool	Purpose
IntelliJ IDEA	Integrated Development Environment (IDE) for writing and debugging Java code.
JavaFX SDK	Framework for building desktop GUI applications with modern UI elements.
Scene Builder	Visual design tool to create FXML layouts for JavaFX interfaces.
MySQL	Relational database management system used for persistent data storage.
MySQL Workbench	Tool for designing, visualizing, and managing the database schema.
JDBC	Java API for database connectivity and executing SQL queries securely.
Maven	Build automation and dependency management tool for Java projects.
Git	Version control system used for managing project source code and collaboration.

This diverse toolset provided a complete development environment capable of supporting full-stack desktop application development.

Methodology

The Smart Expense Tracker was developed using the **Model-View-Controller (MVC)** design pattern, a well-established architectural paradigm that enhances **code organization**, **separation of concerns**, and **scalability**.

MVC Breakdown:

- **Model:**
  The model layer contains all the business logic and data structures. It includes classes such as:
    - User.java: Represents application users with attributes like username, password, and registration date.
    - Expense.java: Encapsulates information such as amount, category, description, and date.
    - Budget.java: Holds data about budget limits and current usage.
- **View:**
  The view layer consists of all **FXML files and GUI elements** created using **JavaFX** and **Scene Builder**. It defines the layout and appearance of:
    - Login and registration screens
    - Dashboard and chart display
    - Expense entry forms
    - Budget tracking and report panels
- **Controller:**
  Controllers manage user interaction and act as a bridge between the view and model. Each screen (FXML file) has an associated controller class (e.g., LoginController.java, ExpenseController.java) that:

    - Handles button clicks and form submissions
    - Validates inputs
    - Calls appropriate model or DAO methods
    - Updates UI elements based on changes

- **DAO (Data Access Object):**
  This additional layer abstracts database interaction, enabling **loose coupling** between business logic and SQL code. Classes like ExpenseDAO.java and UserDAO.java provide reusable methods such as addExpense(), getExpensesByUser(), validateLogin() etc., using **JDBC and prepared statements** for secure data access.

This architecture ensures **modular development**, simplifies unit testing, and makes the codebase easier to extend in the future (e.g., adding multi-user support or cloud sync).

Best Practices Followed:

- **Modular Coding:** Code divided into logical packages: model, controller, dao, and util.
- **Exception Handling:** Robust try-catch-finally blocks and custom error alerts ensure fault tolerance.
- **Version Control with Git:** Regular commits tracked development progress, making rollback and collaboration easier.

Smart Expense Tracker Screenshots

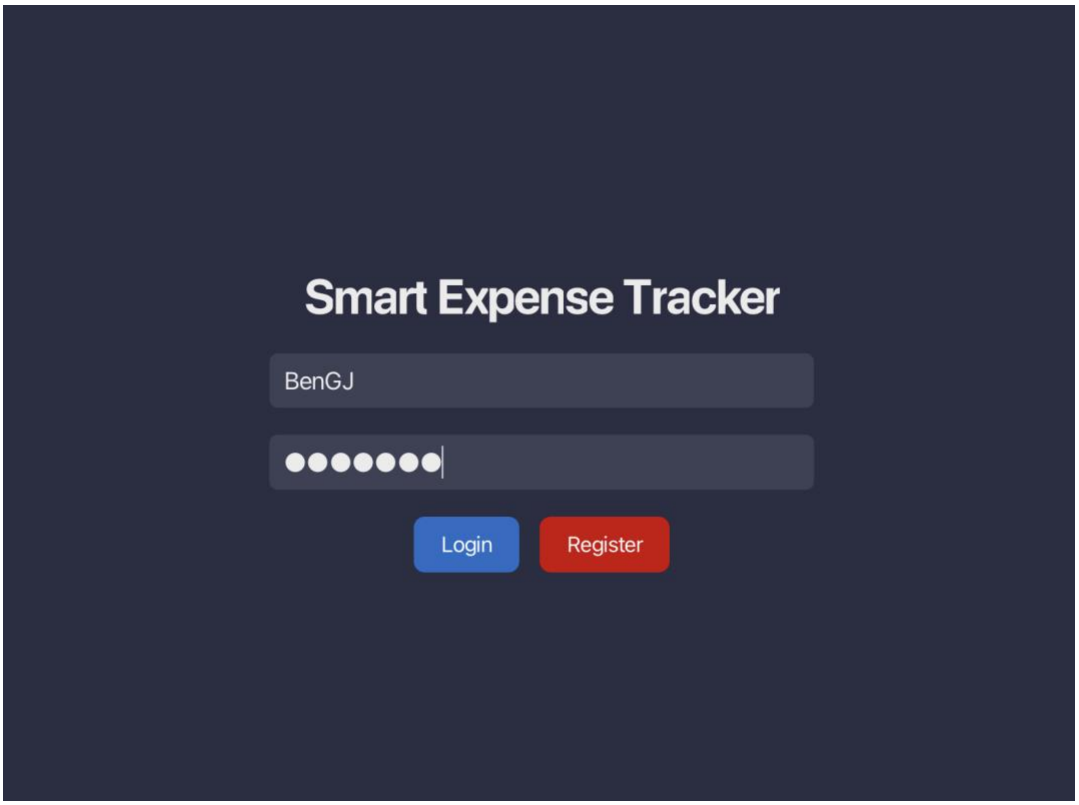


Fig 1.1 Login Page

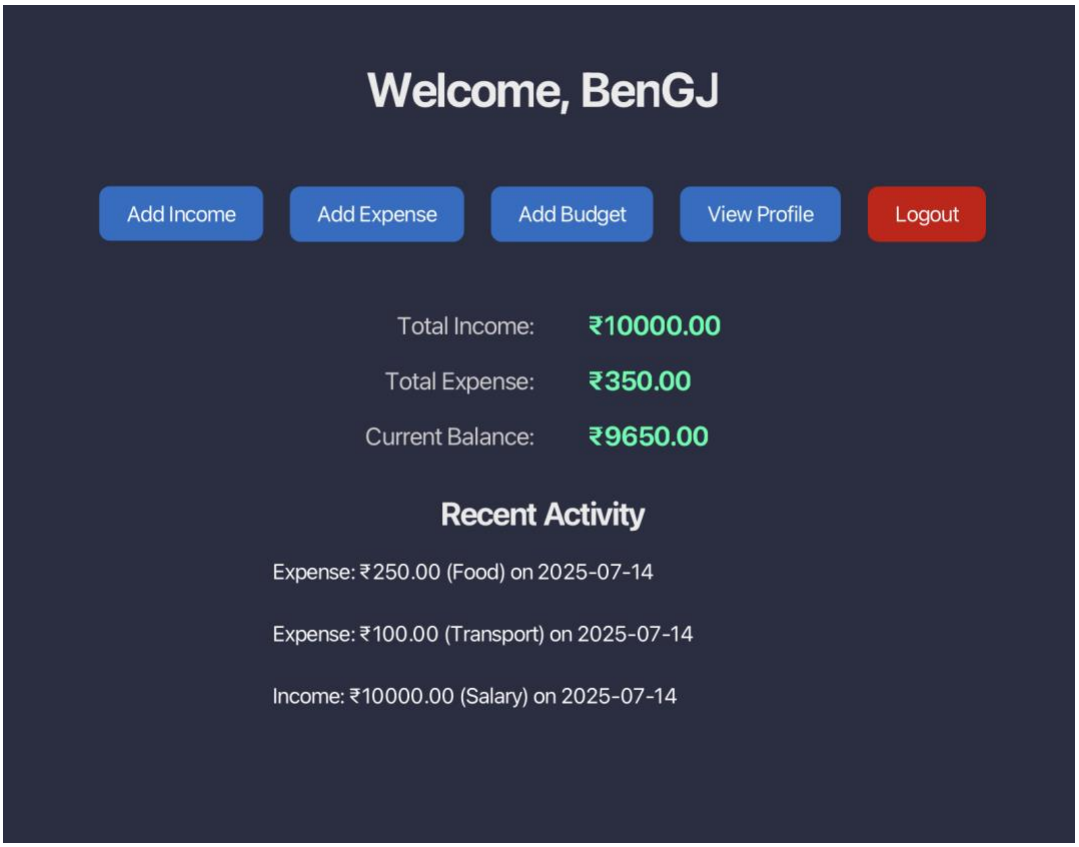


Fig 1.2 Dashboard containing recent activity and account summary

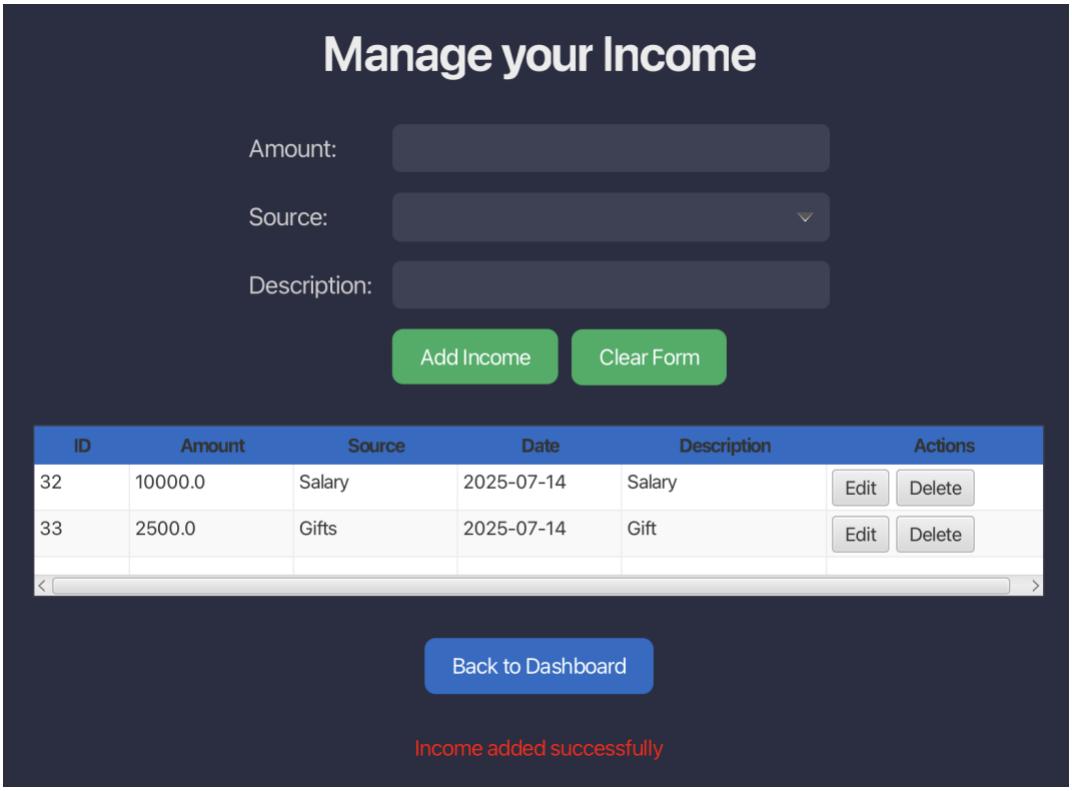


Fig 1.3 Adding Income into the Income table

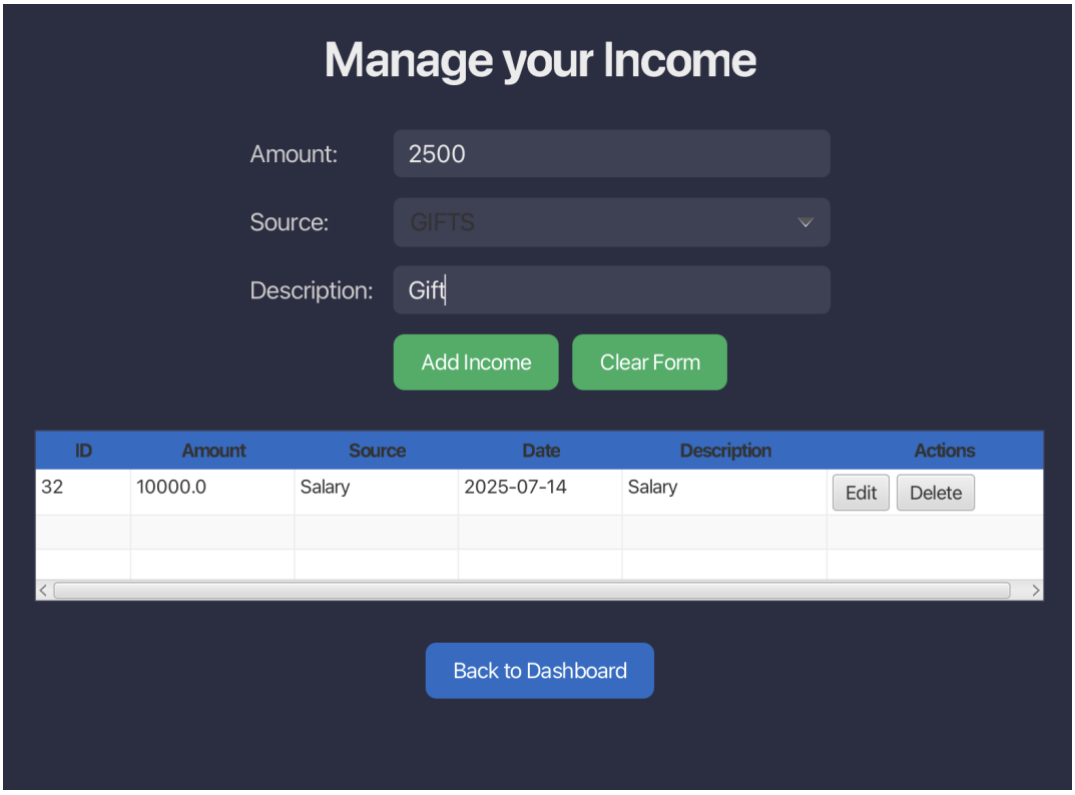


Fig 1.4 Added 2500 as Gifts

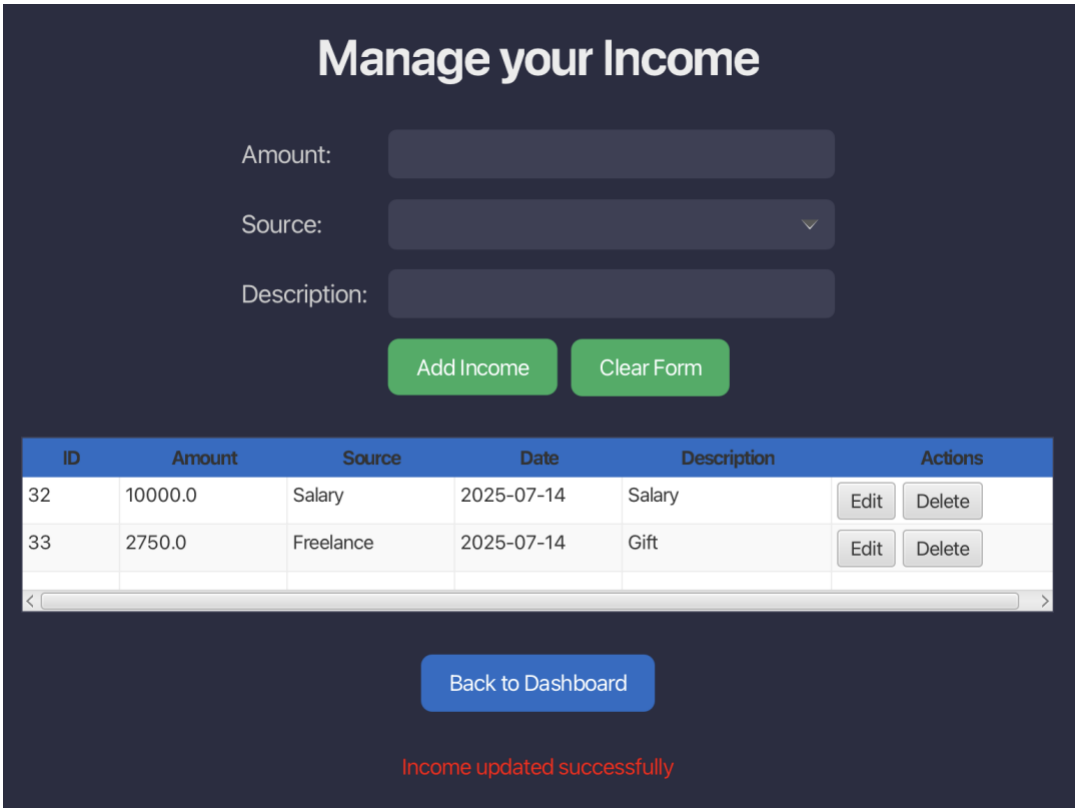


Fig 1.5 We changed amount and source to 2750 and Freelance

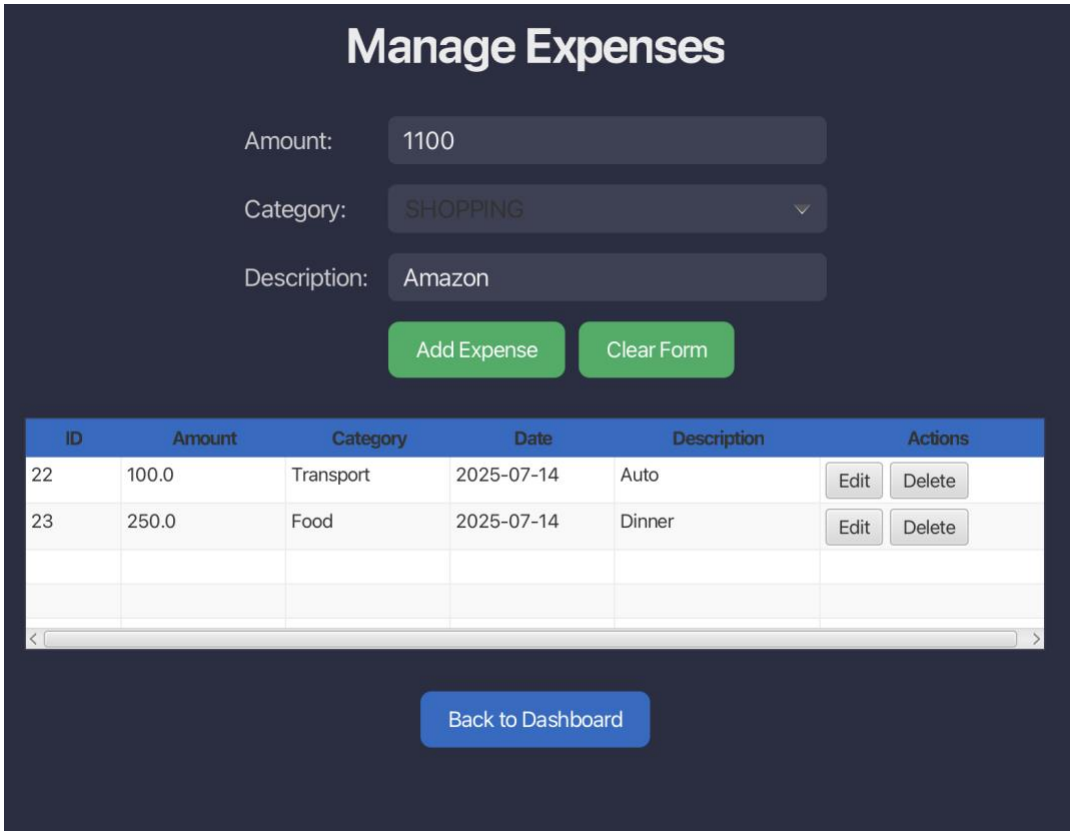


Fig 1.6 Adding Expenses into the Expense Table

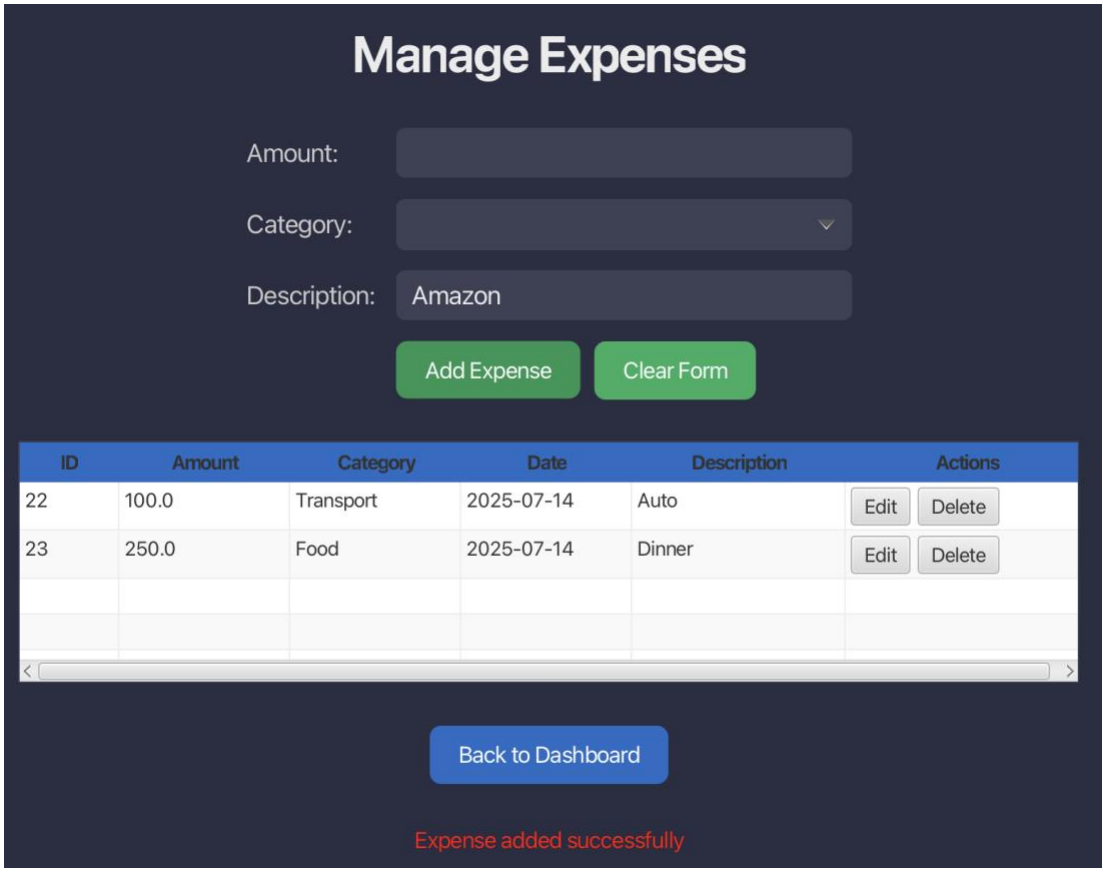


Fig 1.7 Expense added successfully

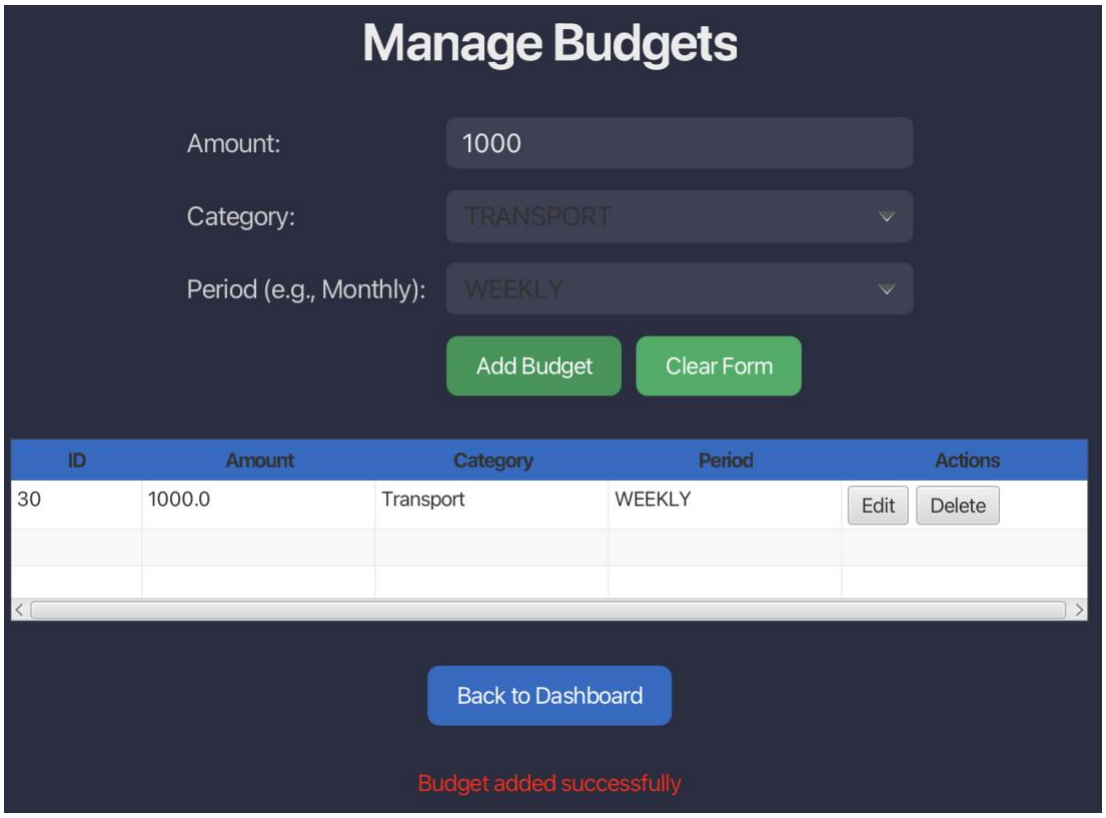


Fig 1.8 Adding Weekly Transport Budgets into the Budget Table

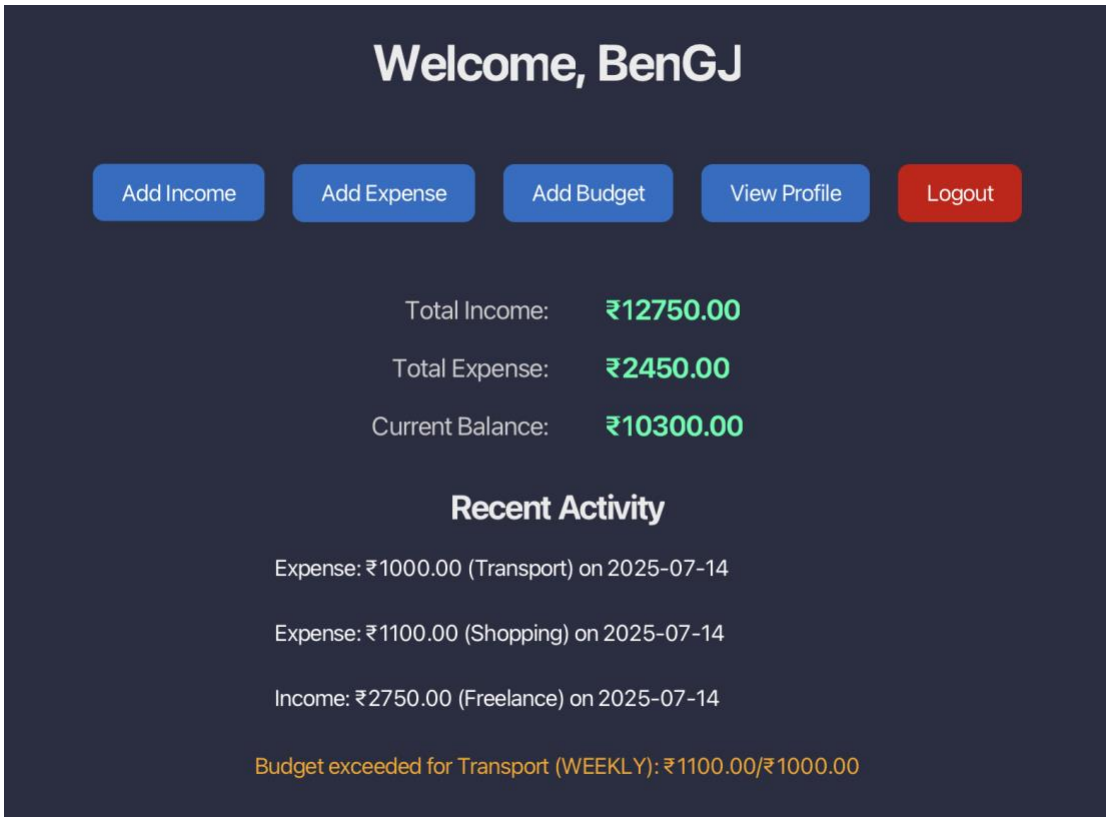


Fig 1.9 Budget Alert as Transport budget is exceeded

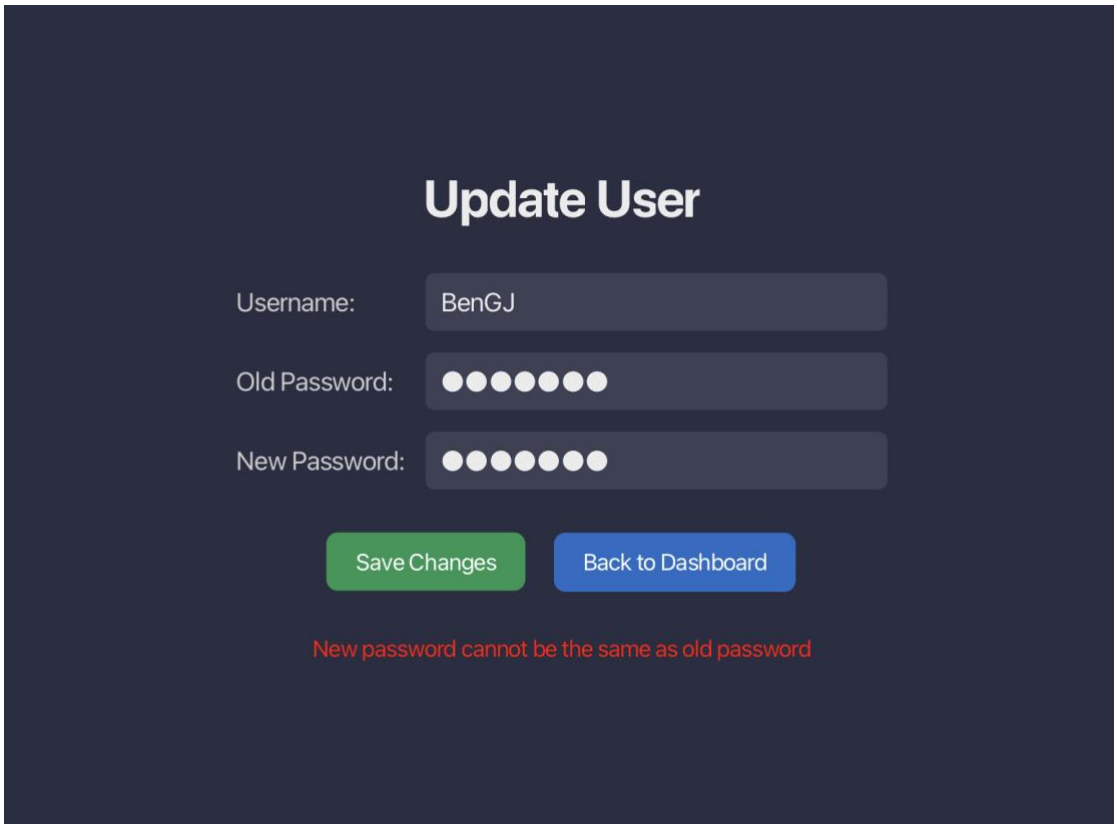


Fig 1.10 Trying to change Password for the logged in user

Code Snippets

```java
public static Connection getConnection() throws DatabaseException {  18 usages  BenGJ10
    Properties props = new Properties(); // Creates a Properties object to load key-value pairs from the file

    try (FileInputStream fis = new FileInputStream(PROPERTIES_FILE)) {
        props.load(fis); // loads all the values into the props object
        String url = props.getProperty("db.url");
        String user = props.getProperty("db.username");
        String password = props.getProperty("db.password");

        // Uses JDBC's DriverManager to establish a connection to the database, returns a Connection object that can be used in DAO classes.
        return DriverManager.getConnection(url, user, password);

    } catch (IOException | SQLException e) {
        throw new DatabaseException("Failed to connect to database: " + e.getMessage(), e);
    }
}
```

Fig 2.1 Database Connection using JDBC

```java
public class SessionManager {  BenGJ10
    private static SessionManager instance; // This holds the one and only instance of SessionManager  3 usages
    private User loggedInUser; // Stores current user who logged in  3 usages

    private SessionManager() {}  1 usage  BenGJ10

    public static SessionManager getInstance() {  BenGJ10
        if (instance == null) { // If it hasn't been created yet (null), it creates one
            instance = new SessionManager();
        }
        return instance; // Otherwise, it returns the already-created instance
    }

    public void setLoggedInUser(User user) { this.loggedInUser = user; }

    public User getLoggedInUser() { // Returns the currently logged-in user  BenGJ10
        return loggedInUser;
    }
}
```

Fig 2.2 Session Manager using Singleton instance for one active user at a time

```java
public class ValidationUtil { 22 usages  BenGJ10
    public static void validateUsername(String username) throws InvalidInputException { 3 usages  BenGJ10
        if (username == null || username.trim().isEmpty()) {
            throw new InvalidInputException("Username cannot be empty");
        }
        if (username.length() > 50) {
            throw new InvalidInputException("Username cannot exceed 50 characters");
        }
        if (!username.matches( regex: "^[a-zA-Z0-9]+$")) {
            throw new InvalidInputException("Username must be alphanumeric");
        }
    }

    public static void validatePassword(String password) throws InvalidInputException { 3 usages  BenGJ10
        if (password == null || password.isEmpty()) {
            throw new InvalidInputException("Password cannot be empty");
        }
        if (password.length() < 6) {
            throw new InvalidInputException("Password must be at least 6 characters");
        }
        if (!password.matches( regex: ".*[A-Z].*")) {
            throw new InvalidInputException("Password must contain at least one uppercase letter");
        }
        if (!password.matches( regex: ".*[a-z].*")) {
            throw new InvalidInputException("Password must contain at least one lowercase letter");
        }
        if (!password.matches( regex: ".*\\d.*")) {
            throw new InvalidInputException("Password must contain at least one digit");
        }
        if (password.length() > 30) {
            throw new InvalidInputException("Password cannot exceed 30 characters");
        }
    }
```

Fig 2.3 Username and Password Validator

```java
public class PasswordUtil { 5 usages  BenGJ10
    public static String hashPassword(String password) throws DatabaseException { 3 usages  BenGJ10
        try {
            // A core Java class used for one-way cryptographic hashing.
            MessageDigest digest = MessageDigest.getInstance( algorithm: "SHA-256"); // A secure hashing algorithm from the SHA-2 family.

            /* The resulting SHA-256 hash is a byte array (raw data).
            We convert it to a human-readable hex string (base-16), which can be stored in text format in a database.
            Each byte becomes a 2-digit hex representation. */

            byte[] hash = digest.digest(password.getBytes());
            StringBuilder hexString = new StringBuilder();
            for (byte b : hash) {
                String hex = Integer.toHexString( i: 0xff & b);
                if (hex.length() == 1) hexString.append('0');
                hexString.append(hex);
            }
            return hexString.toString();
        } catch (NoSuchAlgorithmException e) {
            throw new DatabaseException("Failed to hash password", e);
        }
    }
```

Fig 2.4 Hashing Password to store it safely in the database.

```java
@Override 16 usages  BenGJ10
// Returns a User object upon successful login
public User loginUser(String username, String password) throws InvalidInputException, DatabaseException {
    ValidationUtil.validateUsername(username);
    ValidationUtil.validatePassword(password);

    // SQL command to fetch user details from the database by username
    String sql = "SELECT id, username, password FROM users WHERE username = ?";

    try (Connection conn = DatabaseConnection.getConnection();
         PreparedStatement stmt = conn.prepareStatement(sql)) {
        stmt.setString( parameterIndex: 1, username.trim());

        // ResultSet is a Java object that holds the data retrieved from a database after executing a SQL SELECT query
        ResultSet rs = stmt.executeQuery();
        if (rs.next()) { // Checks if a user with that username was found, then it moves the cursor to the first result row.
            String storedHash = rs.getString( columnLabel: "password"); // Retrieves the hashed password

            if (PasswordUtil.verifyPassword(password, storedHash)) { // Compares it with the input password
                logger.info("User logged in: " + username); // Logs successful login
                return new User(rs.getInt( columnLabel: "id"), rs.getString( columnLabel: "username"), storedHash);
            }
        }
        logger.warning("Invalid login attempt for user: " + username);
        throw new InvalidInputException("Invalid username or password");
    } catch (SQLException e) {
        logger.error( message: "Failed to login user: " + username, e);
        throw new DatabaseException("Failed to login user", e);
    }
}
```

Fig 2.5 UserDAO file for user login.

```java
@Override 1 usage  BenGJ10
public void updateUser(User user) throws InvalidInputException, DatabaseException {
    String username = user.getUsername().trim();
    String password = user.getPassword();

    ValidationUtil.validateUsername(username);
    ValidationUtil.validatePassword(password);

    String hashedPassword = PasswordUtil.hashPassword(password);

    String sql = "UPDATE users SET username = ?, password = ? WHERE id = ?";

    try (Connection conn = DatabaseConnection.getConnection();
         PreparedStatement stmt = conn.prepareStatement(sql)) {

        stmt.setString( parameterIndex: 1, username);
        stmt.setString( parameterIndex: 2, hashedPassword);
        stmt.setInt( parameterIndex: 3, user.getId());

        int rowsAffected = stmt.executeUpdate();
        if (rowsAffected == 0) {
            logger.warning("No user found to update with ID: " + user.getId());
            throw new DatabaseException("No user found to update");
        }
        logger.info("Updated user profile: " + username);

    } catch (SQLException e) {
        if (e.getSQLState().equals("23000")) { // likely duplicate username
            logger.warning("Duplicate username on update attempt: " + username);
            throw new InvalidInputException("Username already exists");
```

Fig 2.6 UserDAO file for user login.

```java
@Override 5 usages BenGJ10 +1
// Validates the input and performs an INSERT operation in the `income` table.
public void addIncome(Income income) throws InvalidInputException, DatabaseException {
    ValidationUtil.validateAmount(income.getAmount());
    ValidationUtil.validateCategory(income.getSource().getDisplayName());

    if (income.getDateTime() == null) throw new InvalidInputException("Date cannot be null");

    // Ensure a user is logged in
    if (SessionManager.getInstance().getLoggedInUser() == null) {
        logger.warning("Attempt to add income without logged-in user");
        throw new DatabaseException("No user logged in");
    }

    String sql = "INSERT INTO income (user_id, amount, source, date_time, description) VALUES (?, ?, ?, ?, ?)";
    try (Connection conn = DatabaseConnection.getConnection(); // connection object to database
         PreparedStatement stmt = conn.prepareStatement(sql)) { // precompiled SQL command with placeholders
        stmt.setInt( parameterIndex: 1, SessionManager.getInstance().getLoggedInUser().getId());
        stmt.setDouble( parameterIndex: 2, income.getAmount());
        stmt.setString( parameterIndex: 3, income.getSource().name());
        stmt.setObject( parameterIndex: 4, income.getDateTime());
        stmt.setString( parameterIndex: 5, income.getDescription());
        stmt.executeUpdate();
        logger.info("Income added for user ID: " + SessionManager.getInstance().getLoggedInUser().getId());
    } catch (SQLException e) {
        logger.error( message: "Failed to add income", e);
        throw new DatabaseException("Failed to add income", e);
    }
}
```

Fig 2.7 IncomeDAO for adding income.

```java
@Override 2 usages BenGJ10
// Deletes an income record for the logged-in user by ID.
public void deleteIncome(int incomeId) throws DatabaseException {
    String sql = "DELETE FROM income WHERE id = ? AND user_id = ?";

    try (Connection conn = DatabaseConnection.getConnection();
         PreparedStatement stmt = conn.prepareStatement(sql)) {
        stmt.setInt( parameterIndex: 1, incomeId);
        stmt.setInt( parameterIndex: 2, SessionManager.getInstance().getLoggedInUser().getId());
        int rows = stmt.executeUpdate();

        if (rows == 0) {
            logger.warning("No income found to delete with ID: " + incomeId);
            throw new DatabaseException("Income not found or not owned by user");
        }
        logger.info("Income deleted: ID " + incomeId);
    } catch (SQLException e) {
        logger.error( message: "Failed to delete income ID: " + incomeId, e);
        throw new DatabaseException("Failed to delete income", e);
    }
}
```

Fig 2.8 IncomeDAO for deleting income.

```java
@Override 3 usages  BenGJ10
public void addExpense(Expense expense) throws InvalidInputException, DatabaseException {
    ValidationUtil.validateAmount(expense.getAmount());
    ValidationUtil.validateCategory(expense.getCategory().getDisplayName());
    ValidationUtil.validateDescription(expense.getDescription());

    if (expense.getDateTime() == null)
        throw new InvalidInputException("Date cannot be null");

    if (SessionManager.getInstance().getLoggedInUser() == null) {
        logger.warning("Attempt to add expense without logged-in user");
        throw new DatabaseException("No user logged in");
    }

    final String sql = "INSERT INTO expenses (user_id, amount, category, date_time, description) VALUES (?, ?, ?, ?, ?)";
    try (Connection conn = DatabaseConnection.getConnection();
         PreparedStatement stmt = conn.prepareStatement(sql)) {

        stmt.setInt( parameterIndex: 1, SessionManager.getInstance().getLoggedInUser().getId());
        stmt.setDouble( parameterIndex: 2, expense.getAmount());
        stmt.setString( parameterIndex: 3, expense.getCategory().name());
        stmt.setObject( parameterIndex: 4, expense.getDateTime());
        stmt.setString( parameterIndex: 5, expense.getDescription());

        stmt.executeUpdate();
        logger.info("Expense added for user ID: " + SessionManager.getInstance().getLoggedInUser().getId());
```

Fig 2.9 ExpenseDAO for adding expense.

```java
@Override 5 usages  BenGJ10
public List<Expense> getExpensesByUserId(int userId) throws DatabaseException {
    List<Expense> expenses = new ArrayList<>();
    final String sql = "SELECT id, amount, category, date_time, description FROM expenses WHERE user_id = ?";

    try (Connection conn = DatabaseConnection.getConnection();
         PreparedStatement stmt = conn.prepareStatement(sql)) {

        stmt.setInt( parameterIndex: 1, userId);
        ResultSet rs = stmt.executeQuery();

        while (rs.next()) {
            expenses.add(new Expense(
                    rs.getInt( columnLabel: "id"),
                    userId,
                    rs.getDouble( columnLabel: "amount"),
                    ExpenseCategory.valueOf(rs.getString( columnLabel: "category")),
                    rs.getObject( columnLabel: "date_time", LocalDateTime.class),
                    rs.getString( columnLabel: "description")
            ));
        }

        logger.info("Retrieved " + expenses.size() + " expenses for user ID: " + userId);
        return expenses;

    } catch (SQLException | InvalidInputException e) {
        logger.error( message: "Failed to retrieve expenses for user ID: " + userId, e);
        throw new DatabaseException("Failed to retrieve expenses", e);
    }
}
```

Fig 2.10 ExpenseDAO for retrieving income by userID.

```java
@Override 6 usages jerome784
public void addBudget(Budget budget) throws InvalidInputException, DatabaseException {
    ValidationUtil.validateAmount(budget.getAmount());
    ValidationUtil.validateCategory(budget.getCategory() != null ? budget.getCategory().name() : null);

    if (budget.getPeriod() == null || budget.getStartDate() == null) {
        throw new InvalidInputException("Period and start date must not be null");
    }

    final String sql = "INSERT INTO budgets (user_id, category, amount, period, start_date) VALUES (?, ?, ?, ?, ?)";

    try (Connection conn = DatabaseConnection.getConnection();
         PreparedStatement stmt = conn.prepareStatement(sql)) {

        stmt.setInt( parameterIndex: 1, budget.getUserId());
        if (budget.getCategory() != null) {
            stmt.setString( parameterIndex: 2, budget.getCategory().name());
        } else {
            stmt.setNull( parameterIndex: 2, Types.VARCHAR); // Nullable category for overall budget
        }
        stmt.setDouble( parameterIndex: 3, budget.getAmount());
        stmt.setString( parameterIndex: 4, budget.getPeriod());
        stmt.setObject( parameterIndex: 5, budget.getStartDate());

        stmt.executeUpdate();
        logger.info("Budget added for user ID: " + budget.getUserId());
```

Fig 2.11 BudgetDAO for adding budgets.

```java
@Override 2 usages jerome784
public void updateBudget(Budget budget) throws InvalidInputException, DatabaseException {
    ValidationUtil.validateAmount(budget.getAmount());
    ValidationUtil.validateCategory(budget.getCategory() != null ? budget.getCategory().name() : null);

    if (budget.getPeriod() == null || budget.getStartDate() == null) {
        throw new InvalidInputException("Period and start date must not be null");
    }

    final String sql = "UPDATE budgets SET category = ?, amount = ?, period = ?, start_date = ? WHERE id = ? AND user_id = ?";

    try (Connection conn = DatabaseConnection.getConnection();
         PreparedStatement stmt = conn.prepareStatement(sql)) {

        if (budget.getCategory() != null) {
            stmt.setString( parameterIndex: 1, budget.getCategory().name());
        } else {
            stmt.setNull( parameterIndex: 1, Types.VARCHAR);
        }
        stmt.setDouble( parameterIndex: 2, budget.getAmount());
        stmt.setString( parameterIndex: 3, budget.getPeriod());
        stmt.setObject( parameterIndex: 4, budget.getStartDate());
        stmt.setInt( parameterIndex: 5, budget.getId());
        stmt.setInt( parameterIndex: 6, budget.getUserId());

        int rows = stmt.executeUpdate();
        if (rows == 0) {
            logger.warning("No budget found to update with ID: " + budget.getId());
            throw new DatabaseException("Budget not found or not owned by user");
        }
```

Fig 2.12 BudgetDAO for updating budgets.

CHAPTER 5: RESULTS AND DISCUSSION

Output / Report

The Smart Expense Tracker was successfully implemented as a comprehensive desktop-based expense management application. It fulfilled all core functionalities outlined in the initial objectives and demonstrated stable performance across various use cases and test scenarios.

Functional Achievements

- **User Authentication (Login and Registration):**
  A secure and reliable login system was implemented, allowing users to register with unique credentials. Input validations ensure correctness and prevent duplication or weak password entries. Upon successful login, users are redirected to the main dashboard.

- **Income and Expense Management:**
  Users can add, edit, and delete financial transactions with relevant details such as amount, category, date, and description. These entries are accurately stored and retrieved from the backend database, enabling users to maintain a detailed financial history.

- **Database Connectivity and Persistence:**
  The application uses JDBC to securely connect to a MySQL database. All transactions are persistently stored and efficiently retrieved. The use of prepared statements ensures secure execution of SQL queries and protection against injection attacks.

- **Budget Management:**
  Users can set monthly or category-specific budget limits. The application continuously monitors expenses and notifies users when spending approaches or exceeds the set limits. Budget data is maintained in the database and dynamically updated as new expenses are added.

- **Reporting and Analytics:**
  The application includes a reporting section where users can view summaries based on date ranges and categories. Reports are dynamically generated using JavaFX charts such as pie charts for category distribution and bar charts for monthly comparisons. This enhances financial visibility and helps users make informed decisions.

- **User Interface and Responsiveness:**
  Built using JavaFX, the application delivers a modern and intuitive interface. The UI supports smooth navigation between screens, responsive layouts, consistent styling, and real-time interaction, providing an effective user experience.

These outputs confirm that the application meets its intended objectives of tracking, budgeting, visualizing, and managing personal finances.

Challenges Faced

During the development process, several challenges were encountered that required additional effort, research, and problem-solving. Overcoming these obstacles contributed to a deeper understanding of practical software development.

- **Cross-Platform Compatibility:**
  Since development was carried out on both macOS and Windows, managing file paths, configuring the JavaFX SDK, and setting up environment variables presented compatibility issues. Adjustments had to be made to ensure consistent performance on different operating systems.

- **UI Freezing During Database Operations:**
  Initial implementations ran JDBC operations on the JavaFX Application Thread, which caused the user interface to freeze during data processing. This issue was resolved by offloading database operations to background threads using JavaFX concurrency classes such as Task and Service.

- **Scene Builder and FXML Integration:**
  Integrating Scene Builder with IntelliJ IDEA posed difficulties, especially in

ensuring correct linkage between FXML files and their respective controllers. These were resolved through proper configuration and adherence to FXML structure guidelines.

- **Designing an Effective MVC Architecture:**
  Maintaining strict separation of model, view, and controller layers required careful planning and refactoring. Ensuring low coupling and high cohesion across modules was necessary to maintain scalability and readability.

- **Data Validation and Exception Handling:**
  Implementing consistent input validation across forms and ensuring exception handling for both user input and database operations was challenging. Each input field had to be validated for correctness, and appropriate error messages were displayed to guide users.

These challenges, although complex, provided valuable exposure to common issues faced in real-world software development and significantly enhanced the team's problem-solving capabilities

Learnings

The project provided an opportunity to gain practical experience in desktop application development using modern tools and technologies. The following key areas of learning were achieved through this project.

- **JavaFX User Interface Development:**
  Experience was gained in creating dynamic and responsive desktop applications using JavaFX. Understanding layout structures, styling with CSS, and managing FXML files enhanced GUI development skills.

- **Database Programming with JDBC:**
  A strong understanding of connecting Java applications to a relational database using JDBC was developed. This included performing CRUD operations, using prepared statements, and handling exceptions related to database connectivity.

- **Project Management with Maven:**
  The use of Maven for dependency management and project structuring improved the overall build process and facilitated collaboration. Maven ensured consistent project configuration across different systems.

- **Object-Oriented Programming and Architecture:**
  The project reinforced the use of OOP principles such as encapsulation, inheritance, and abstraction. These principles were applied in designing model classes, controllers, and data access layers, resulting in a modular and maintainable codebase.

- **Multithreading and Concurrency:**
  Practical knowledge was gained in implementing multithreading in Java, particularly in the context of JavaFX. Database operations were executed in the background using Task and Service classes to maintain UI responsiveness.

- **Collaborative and Cross-Platform Development:**
  Working on different platforms improved adaptability and understanding of environment-specific configurations. Team collaboration, version control using Git, and task division were essential in delivering the project within the defined timeline.

This project not only improved technical proficiency but also enhanced soft skills such as teamwork, communication, and time management, providing a well-rounded learning experience.

CHAPTER 6: CONCLUSION

Summary

The Smart Expense Tracker project successfully fulfilled its core objectives by delivering a functional, intuitive, and modular desktop application for personal finance management. Developed using JavaFX for the graphical user interface and MySQL for persistent data storage, the application allows users to register, log in, and manage their income and expenses with ease. Core features such as budget creation, expense categorization, and dynamic data visualization through charts significantly enhance the application's practicality and user engagement.

Throughout the development process, several industry-relevant practices were applied. The software architecture adhered to the Model-View-Controller (MVC) design pattern, ensuring clean separation of concerns and promoting maintainability. The integration of the Data Access Object (DAO) pattern abstracted database operations, while multithreading techniques were employed to maintain interface responsiveness during background tasks. JDBC was used extensively to manage secure and efficient communication between the application and the MySQL database.

Development was facilitated by professional tools such as IntelliJ IDEA for coding and debugging, Scene Builder for user interface design, and MySQL Workbench for database visualization and management. These tools streamlined the end-to-end software development lifecycle, from design and implementation to testing and debugging.

In addition to technical learning outcomes, the project fostered critical soft skills including collaboration, time management, and analytical problem-solving. The development process deepened the team's understanding of core Java programming, object-oriented design, and real-world application development practices.

Moving forward, the project has significant scope for expansion. Potential enhancements include:

- Integration with a mobile application for cross-platform support
- Cloud-based synchronization for multi-device access
- Application of machine learning algorithms to enable predictive budget forecasts and personalized financial insights

With these improvements, the Smart Expense Tracker can evolve into a comprehensive and intelligent personal finance management tool, offering value to a wide range of users seeking financial clarity and control.