**CS673 Software Engineering** 

**Team 1 - PennyWise**

**Software Design Document**

| Team Member | Role(s) | Signature | Date |
| --- | --- | --- | --- |
| Brain Fenstermacher | Team leader | *Brian Fenstermacher* | 9/17/23 |
| Jisoo Lee | Requirement leader | *Jisoo Lee* | 09/14/23 |
| Sherif Zeyada | Design and Implementation leader | *Sherif Zeyada* | 09/17/23 |
| Mali Rivera | QA leader | *Mali Rivera* | 9/9/23 |
| Clyde Yeung | Configuration leader | *Clyde Yeung* | 9/9/23 |
| Sean Rawson | Security leader | *Sean Rawson* | 9/22/23 |
|  |  |  |  |
|  |  |  |  |

**Revision history**

| **Version** | **Author** | **Date** | **Change** |
| --- | --- | --- | --- |
| **1.1** | **Sean Rawson** | **10/16/23** | **Architecture and authentication changes** |
| **1.2** | **Chaitanya Saraogi** | **10/16/23** | **Business logic for Budget Notification added** |
| **1.3** | **Sean Rawson** | **10/17/23** | **Update class diagram** |

[Introduction](#_heading=h.gjdgxs)

[Software Architecture](#_heading=h.30j0zll)

[Class Diagram](#_heading=h.1fob9te)

[UI Design (if applicable)](#_heading=h.3znysh7)

[Database Design (if applicable)](#_heading=h.2et92p0)

[Security Design](#_heading=h.tyjcwt)

[Business Logic and/or Key Algorithms](#_heading=h.3dy6vkm)

[Design Patterns](#_heading=h.1t3h5sf)

[Any Additional Topics you would like to include.](#_heading=h.4d34og8)

[References](#_heading=h.2s8eyo1)

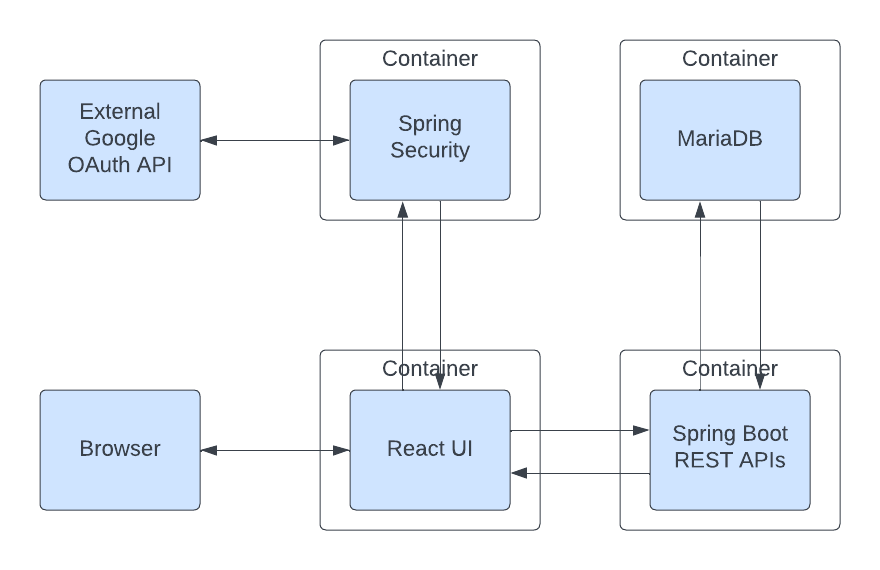
[Glossary](#_heading=h.17dp8vu)

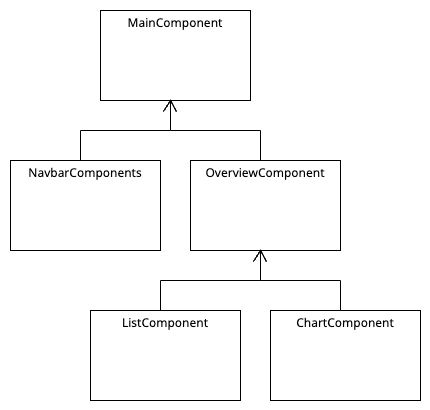
# Introduction (Jisoo)

PennyWise is a robust application built on a client-server architecture with user-centric front-end and robust back-end. Our system is designed to be easily expanded and scalable. Each of our components will be within its container which reduces coupling and leads to future improvements. The Spring Boot back-end serves as a core in our application; it handles data processing and communication with the database. For the data integrity and persistence, we are using the MariaDB database instance. Data integrity and persistence is crucial for our application because we are managing income and expenditure data from the user. PennyWise application is containerized using Docker. This approach will streamline deployment and configuration management. Well-defined communication will exist in between front-end, back-end, and the database. Security is one of the top priorities, with protecting user financial data, preventing unauthorized access, and mitigating potential threats.

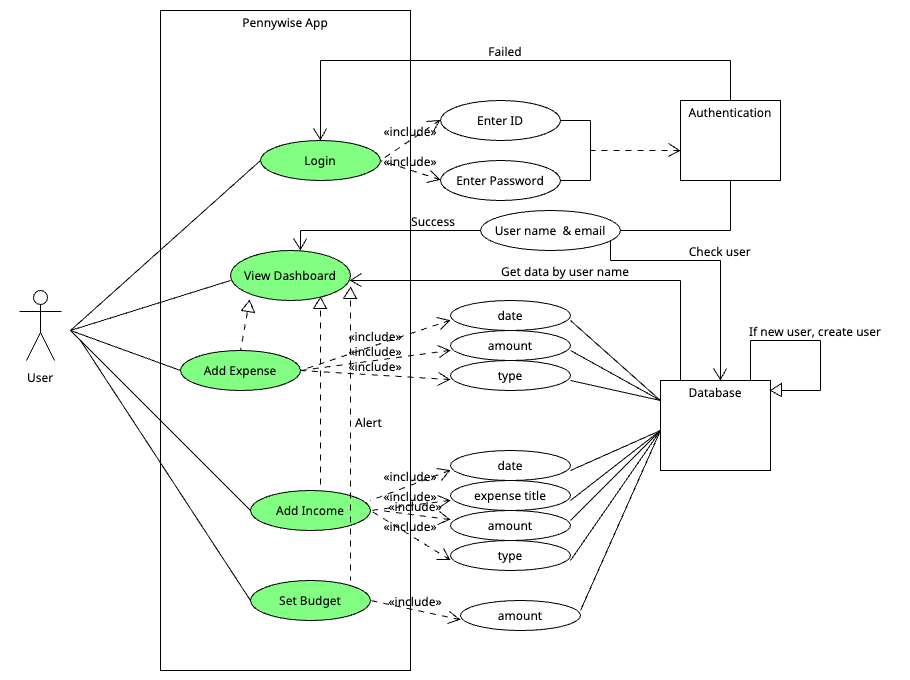
# Software Architecture

As stated in the introduction PennyWise is built on a client-server architecture, but has elements of a tiered architecture as well. The main components are a React front-end, a Spring Boot back-end, and a MariaDB instance. There are clearly defined communication pathways between the parts of the system. Users interact with the front-end, they are externally authenticated by Google, data flows from the front-end to the back-end, and the back-end communicates with the database. Each component will be containerized to reduce coupling between the subsystems and better manage dependencies and configurations.

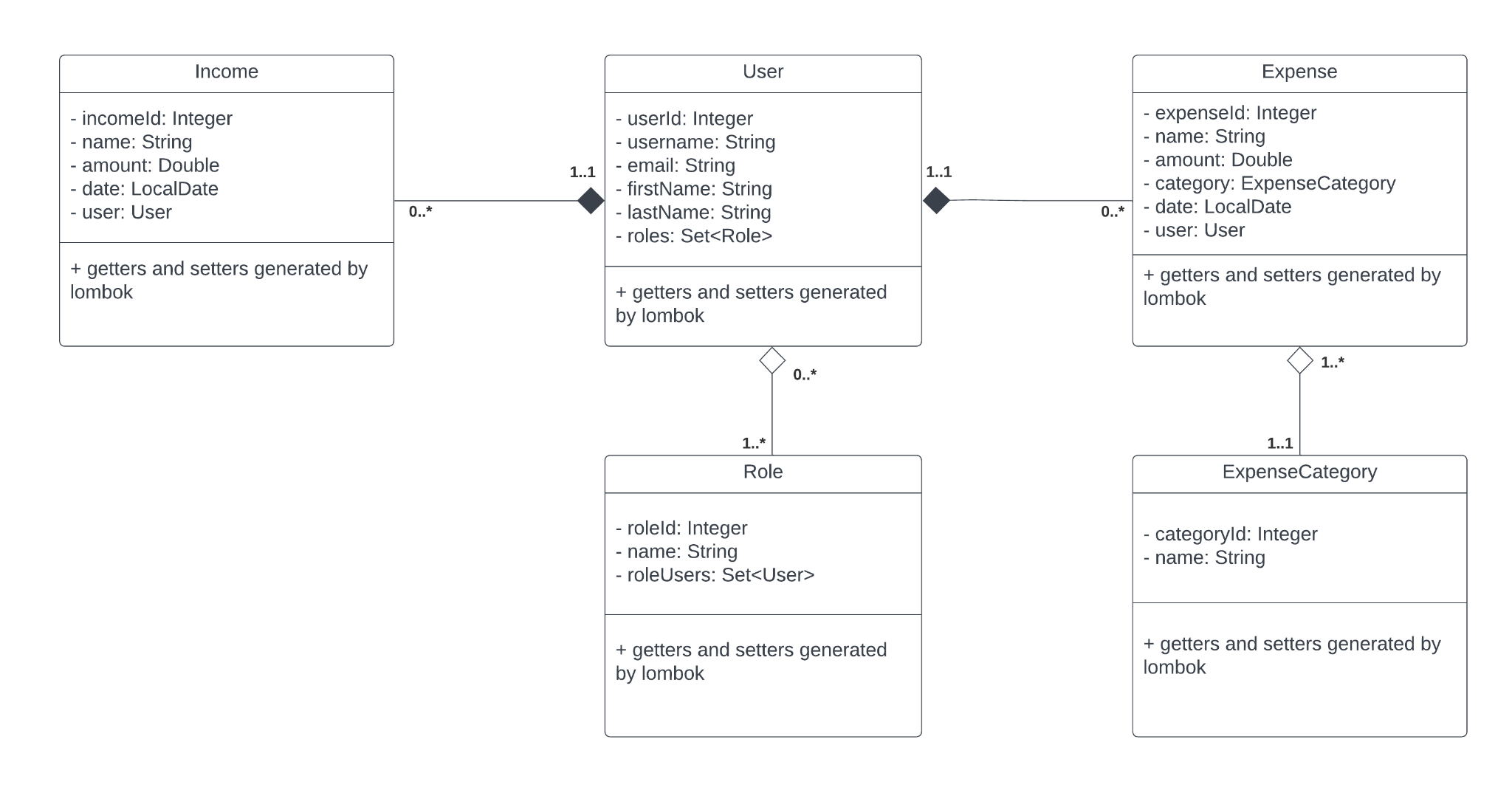
Within the back-end, there are two separate components that work to tie together user authentication, the React front end, and the database. Authentication is handled by Spring Security in one container, completely separate from the API and database layers. Once a user is authenticated, the front end will interact directly with the API endpoints. In this way, different functionalities are kept separate and managing or changing them is easier. The authentication mechanism could be changed out for our own authorization server, without affecting the front end or API.

The PennyWise front-end has a component-based architecture, with separate UI elements allowing for composition and reuse. These pieces are organized into a hierarchy which makes it easier to develop and refactor as the application grows in complexity, while still maintaining logical groupings on screen. The navbar and expense overview are the two main components of a user’s home screen. The expense overview is further broken down into list and chart components for showing different presentations of a user’s budget breakdown. With these components separated, adding or removing elements of the expense overview will be much easier. Additionally, the navbar can vary independently of the overview component if new pages or features are added.

The business logic for the Spring Boot back-end is contained in a series of standard Spring components, rest controllers, services, and repositories. The rest controllers provide the public interface that the front-end will interact with. Any operation that PennyWise needs to support can be exposed by one or many endpoints, with associated processing backing them. All of the required processing is performed by the service layer, including any data transformations or calculations. Once data processing in the service layer is complete, entities that can be saved in the database are passed along to the repository layer. Backing the repository classes is the Hibernate object-relational mapper, which will use the class definitions below to generate the SQL necessary to persist and retrieve data from the database.



# Class Design

The domain model of PennyWise ended up being quite simple, with only five entity classes. These classes are User, Role, Income, Expense, and ExpenseCategory. The main interactions between classes center around the User, because almost every operation in the application is from the perspective of a user. The income and expense classes are directly related to the User class, and instances of those two classes cannot exist without an associated user. Therefore we represent these relationships as compositional. Roles in the application exist independently of users, but each user has a role (usually “USER”), so this relationship is represented as an aggregation. Similarly, every expense must belong to a category, but expense categories exist independently of expenses so this relationship is also an aggregation. 

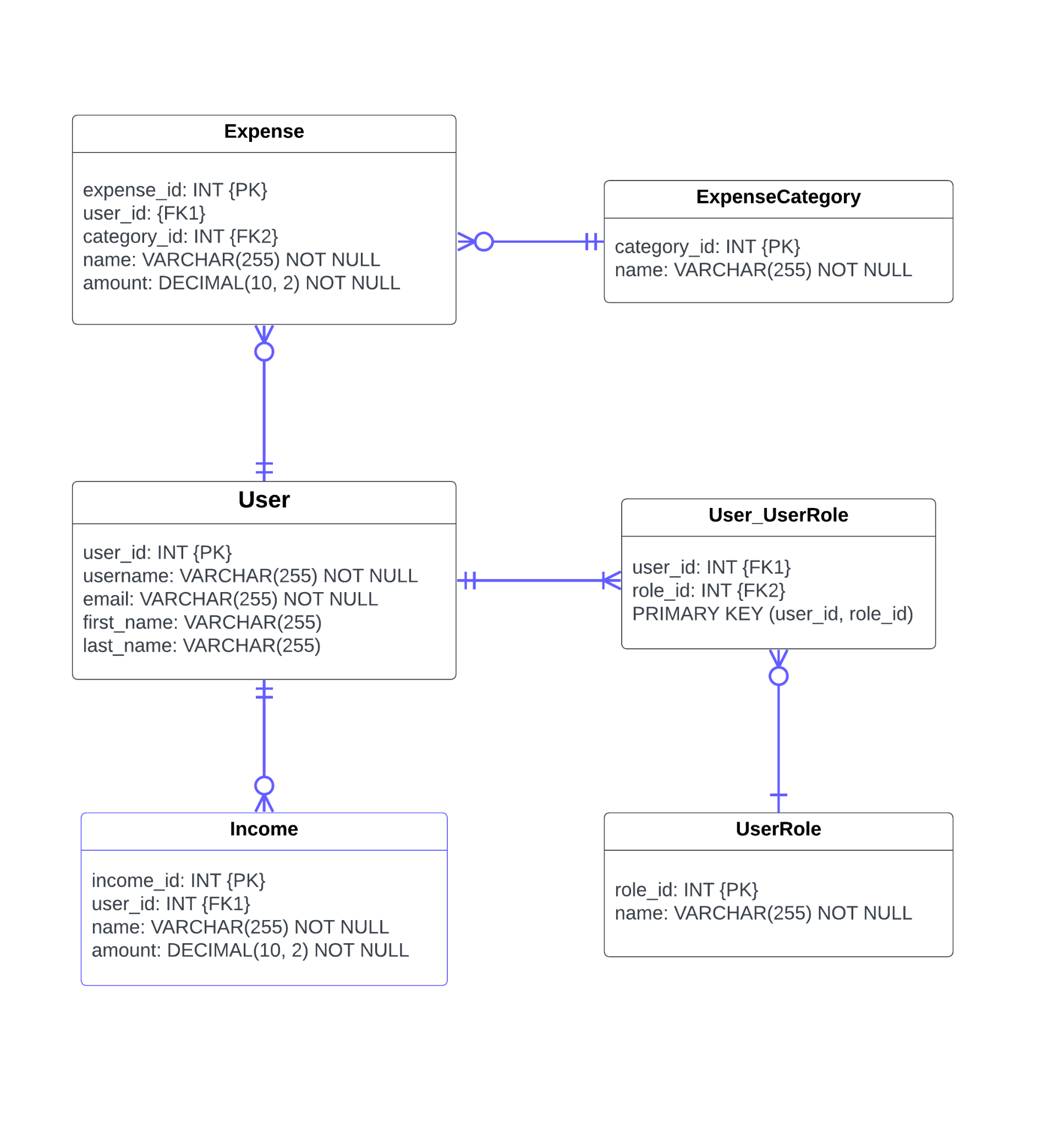
# UI Design (if applicable) (Brian)

Our plan for UI is to be straightforward and easy for our users to work with. By using a clean color palette that reflects the purpose of the product (finance/budgeting) we intend to create a platform that instills trust and security to the user. We are utilizing large buttons to indicate where users can interact on screen and directional cues for areas that can be clicked to open more information. In the following video, you will see a basic breakdown of how a user will navigate and interact with our application. Please note that the design is subject to change based on time restrictions and overall functionality requirements, however, we will do our best to ensure that our deliverable product will operate in a similar fashion to this.

<https://drive.google.com/file/d/1msDx58ORpdw2c-uavcsztuwuvuWDrD4P/view?usp=drive_link>

Additionally, we’ve decided to use the Material UI (MUI) as our react framework. MUI provides prebuilt components that can be dropped into our .js files and allow us to accelerate our frontend development process.

# Database Design (if applicable) (Clyde)

The database design for PennyWise closely mirrors the class diagram. Operations center around the user, and most joins will be against the User table. From the perspective of a user, income and expenses are optional. This is represented by the crow’s foot with an open circle. Users could theoretically sign in and never add anything. Likewise, expense categories and user roles do not require that expenses and users exist, respectively. For the user/role join table, there are two sides with different requirements. From the perspective of a user, a role must always be assigned, so this relation is mandatory. From the perspective of a role, there could be many users or no users with that role, so UserRole has an optional relation to the join table.

# Security Design (Sean)

Our application will have several components, front-end, back-end, and database, that need to work well together and maintain secure communications with external entities. In the end we chose to use Google as our authentication provider, as this removed the need for us to deal with user passwords.

For the main application containers, they are deployed to Heroku, which handles SSL certificates for us. An out-of-the-box solution made it easier to concentrate on features without worrying about certificate signing and having to register with a provider.

Deploying to Heroku is a key part of our process, and that requires API keys, database passwords, and other secrets. To that end, the team has made use of Github secrets to keep those sensitive pieces of information out of source control.

# Business Logic and/or Key Algorithms (Chaitanya & Sherif)

In this section, you shall describe any key algorithms used in your software system, either in terms of pseudocode or flowchart, or sequence diagrams.

**Algorithm 1(Budget Notification Algorithm):**

The Budget Notification System is an integral component of our application designed to provide users with real-time expense tracking and timely alerts when exceeding predefined budget threshold

**Pseudocode for Budget Notification System**

Procedure addUserExpense(ExpenseDto expenseDto):

// Save the provided expense and calculate the current monthly expenses and budget status

result = expenseService.save(expenseDto)

budget = getUserBudget(expenseDto)

currentMonthlyExp = getCurrentMonthlyExp(expenseDto) + result.getAmount()

// Determine if the user is over budget

if (budget is not null) and (currentMonthlyExp > budget):

result.setIsOverBudget(true)

else:

result.setIsOverBudget(false)

// Return the result as a ResponseEntity

Return ResponseEntity with result and HttpStatus.CREATED

Function getUserBudget(ExpenseDto expenseDto):

// Retrieve the user's budget from the database

user = userService.findByUsername(expenseDto.getUsername)

Return user.getBudget()

Function getCurrentMonthlyExp(ExpenseDto expenseDto):

// Query the expenses table for the user's expenses in the same month as the provided expense

expenses = expenseService.findAllExpensesByUsername(expenseDto.getUsername)

expenseMonth = expenseDto.getDate().getMonth()

expenseYear = expenseDto.getDate().getYear()

sum = 0

// Calculate the sum of expenses for the same month and year

for each expense in expenses:

if expense.getDate().getMonth() == expenseMonth and expense.getDate().getYear() == expenseYear:

sum += expense.getAmount()

Return sum

**User Benefits**

The Budget Notification System offers several advantages to users:

* **Real-time Monitoring:** Users can track their budget and expenses as they are added, promoting financial awareness and responsible spending.
* **Timely Alerts:** The system provides budget notifications, keeping users informed when they exceed budget thresholds, enabling them to make informed financial decisions.

**Conclusion**

The Budget Notification System is a critical feature of our application, ensuring users can manage their finances effectively. By providing real-time expense tracking and alerts. it empowers users to maintain control over their expenses and make informed financial decisions. This system contributes to a positive user experience and encourages financial responsibility.

**Algorithm 2(Expense Tracking Algorithm):**

We will use this algorithm To provide real-time expense tracking, this algorithm continuously updates the user's remaining budget based on income, expenses, and budgeted amounts. It alerts users when they are close to exceeding their budget in a specific category.

Pseudocode for Expense Tracking Algorithm:

1. Initialize variables:

- TotalIncome = user's monthly income

- BudgetedExpenses = predefined budget allocation for expense categories

- CurrentExpenses = user's current expenses (initially empty)

- Budget = TotalIncome - sum of CurrentExpenses

2. Loop (while user continues to add expenses):

a. Prompt user to enter a new expense:

- ExpenseName

- ExpenseAmount

- ExpenseCategory

b. Add the entered expense to CurrentExpenses:

- Append ExpenseName, ExpenseAmount, and ExpenseCategory to

CurrentExpenses

c. Update Budget:

- Subtract ExpenseAmount from Budget

d. Display Budget to the user.

3. Users can continue adding expenses or choose to exit.

4. Optionally, if user-defined categories are used:

a. Calculate the total expenses for each user-defined category.

5. Display a summary of expenses:

- TotalIncome

- BudgetedExpenses

- ActualExpenses (total of all expenses entered)

- Budget

6. End of algorithm.

# Design Patterns (Sherif)

In this section, you shall describe any design patterns used in your software system.

**Pattern Name:** Observer Pattern

Implementation in PennyWise:

In the PennyWise project, we've applied the Observer pattern as follows:

**Subject**: The category class acts as the Subject. It maintains a list of Observers (Users) who are interested in tracking changes to expenses.

**Observer**: The User class implements the Observer interface. Users are interested in being notified when new expenses are added or when changes occur in their budget.

**Concrete Observer**: Each individual User instance is a Concrete Observer. They register with the Expenses to receive notifications about expense updates.

**Update/Notify**: The Expenses provides methods for Users to register themselves as Observers and notifies them whenever an expense is added or updated.

**Use Case in PennyWise:**

For example, when a User adds a new expense, the Expenses notifies all registered Users about this change. Users can then update their budgeting and expenditure tracking based on this new information, ensuring that they have an up-to-date view of their financial situation.

The Observer pattern enhances the flexibility and maintainability of PennyWise by decoupling the expense management logic from the user interface. It allows multiple Users to stay synchronized with changes to their expenses without the need for complex polling or manual updates.

# Any Additional Topics you would like to include.

# References

* Component hierarchy

(<https://dev.mobify.com/v2.x/get-started/architecture/react-component-hierarchy>)

* Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.

# Glossary