**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 |  |  |
| Sean Rawson | Security leader | *Sean Rawson* | 9/22/23 |
|  |  |  |  |
|  |  |  |  |

**Revision history**

| **Version** | **Author** | **Date** | **Change** |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

[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)

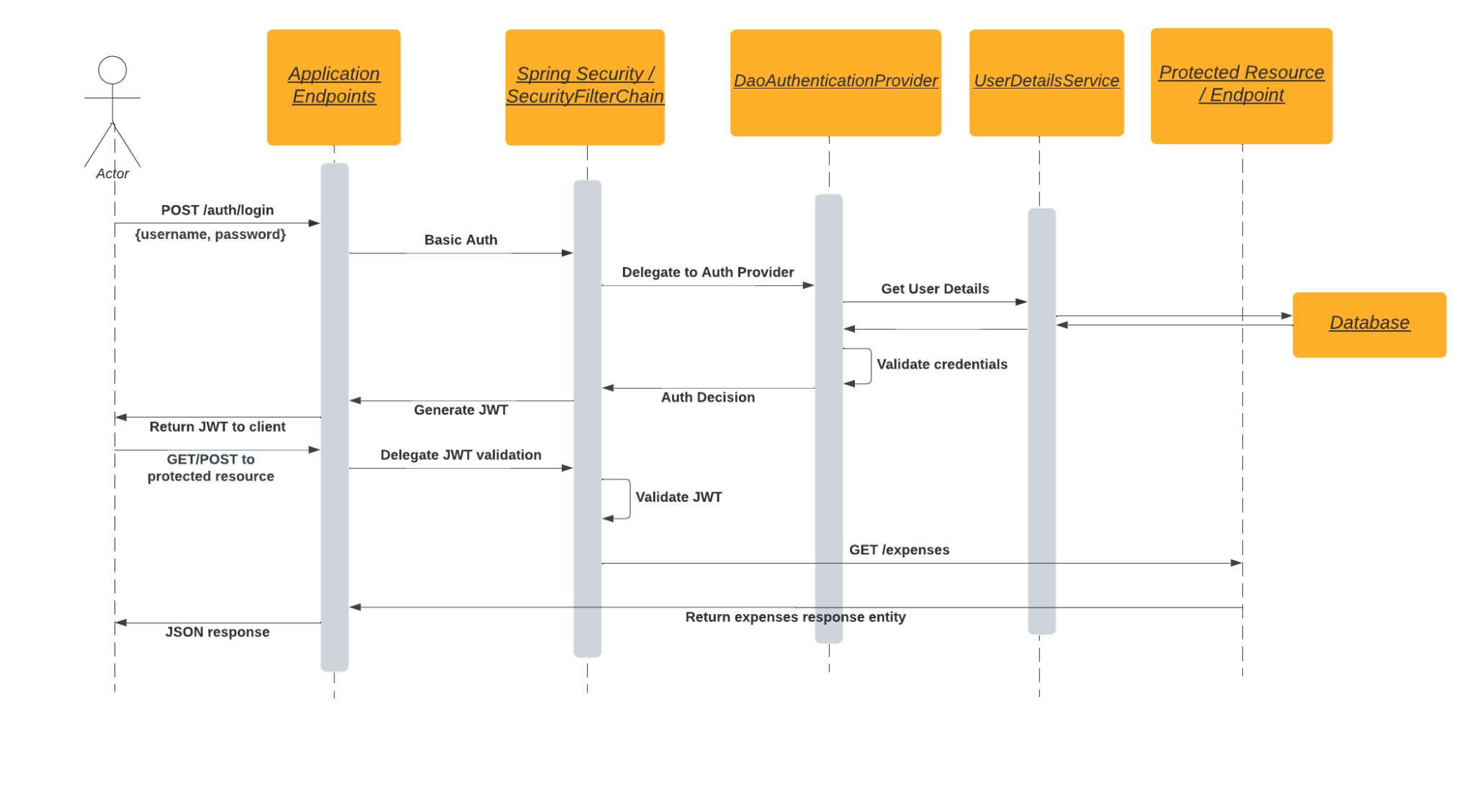
[Glossary](#_heading=h.17dp8vu)

# 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.

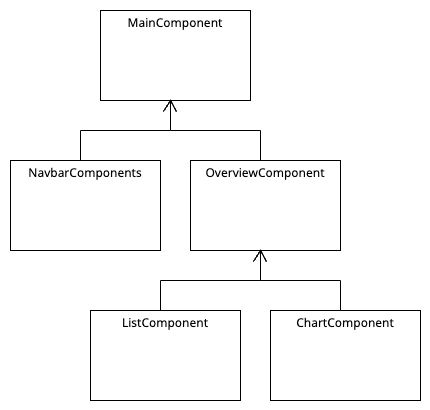
# Software Architecture \*

As stated in the introduction PennyWise is built on a client-server architecture. It could also be thought of as a layered architecture with three levels. 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, 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 together to communicate with both the front-end and database components. Spring Security is responsible for authenticating and authorizing all requests. On initial authentication, user details are retrieved from the database. On successful authentication, a JWT is generated and can be used for subsequent requests without retrieving user details again. Below is a diagram of a typical flow when authentication is required for a protected resource (a REST endpoint in our case).

DaoAuthenticationProvider and UserDetailsService are both elements of Spring Security, and are involved in the actual retrieval and validation of user details against supplied credentials. Once credentials are validated, a JWT is returned to the user and can be supplied with all future requests (until token expiry). All validation of the JWT is done in the background by Spring Security for each request.

Once a (non-admin) user is authenticated, they can begin to actually use PennyWise for managing their budget. Both the front-end React UI and the Spring Boot portion of the back-end are heavily involved in these use cases.

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.

# 

# 

# 

# 

# 

# 

# 

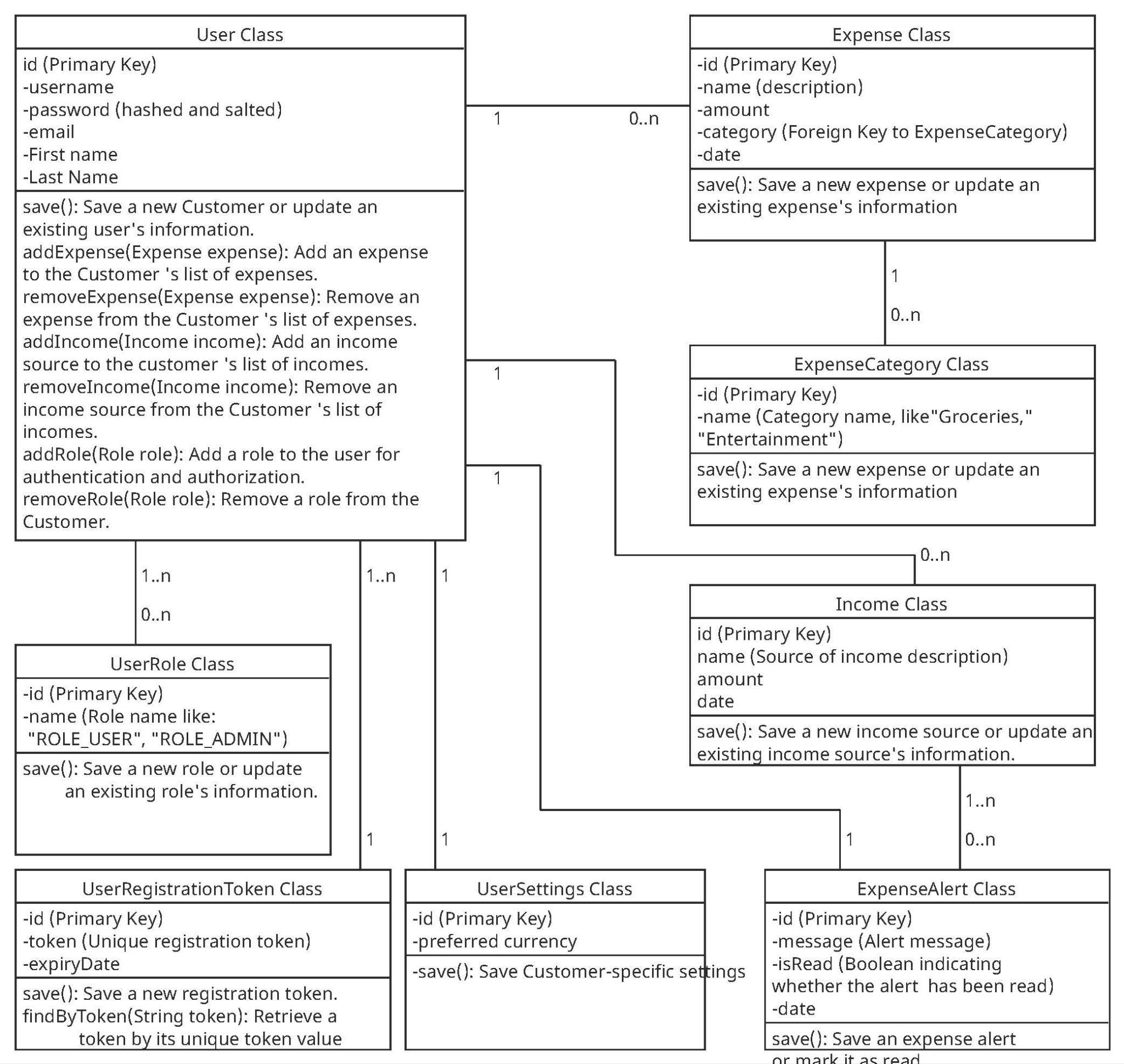
# 

# 

# 

# Class Diagram (Sherif)

In this section, you will provide a detailed description of each component (or package) and use one or multiple class diagrams to show the main classes and their relationships in each component.



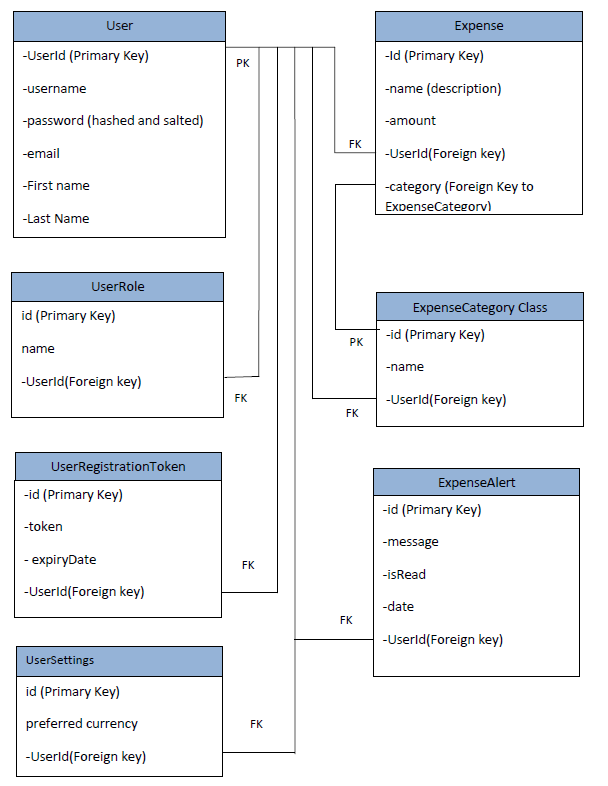
# UI Design (if applicable) (Brian)

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 prebuild components that can be dropped into our .js files and allow us to accelerate our frontend development process.

# Database Design (if applicable) (Clyde)



# 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. We will be using token-based authentication, and plan to choose either internal JWT management or Google OAuth2 sign in. Choosing Google could make the application more secure, as there will be less of a chance of incorrectly implementing security measures.

For communication between the back-end components, there will be the overlay network created by Docker compose. The container management traffic is encrypted by default on this network. The data traffic, however, is not. We plan to run Docker compose with the **–opt encrypted** flag to encrypt communications between the three main systems in the project.

# Business Logic and/or Key Algorithms (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.

The key Algorithm is (**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