Deliverable 3

Mariana-Ionela Muntian

May 12, 2025

I. Design Model

1. Dynamic Behavior

We present two interaction diagrams (one sequence and one communication diagram) for two relevant scenarios:

Scenario 1: User Sign-Up

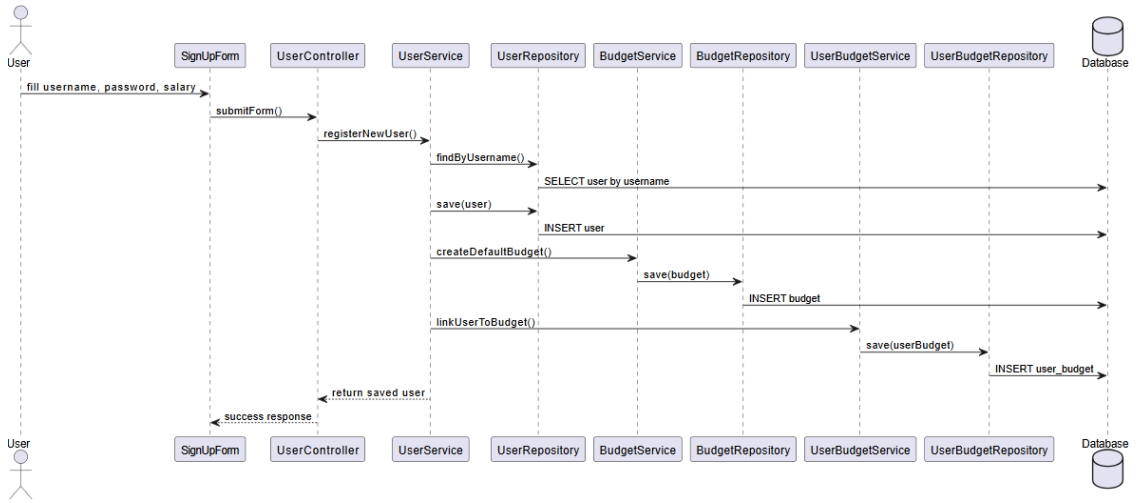

Figure 1: Sequence Diagram for User Sign-Up

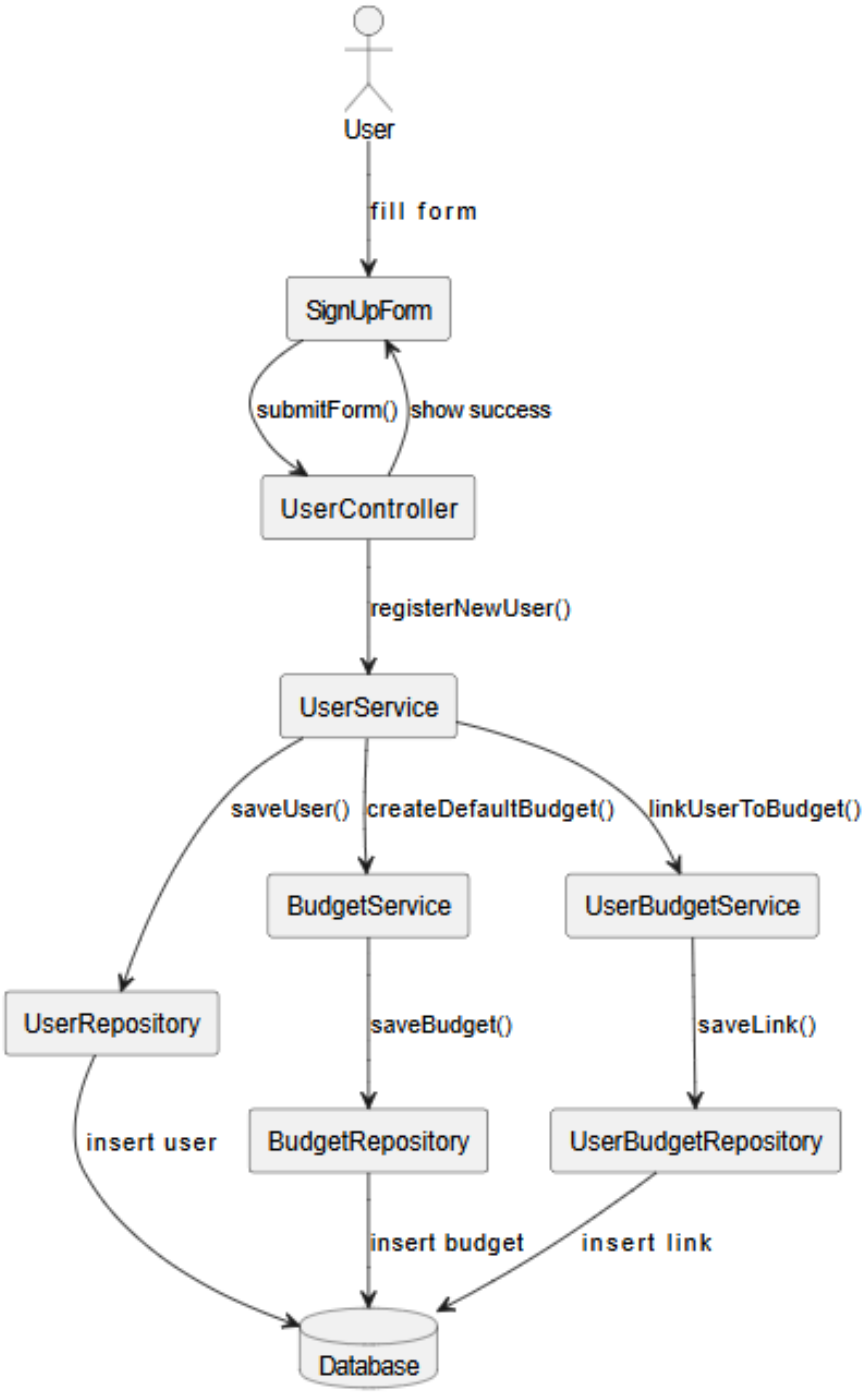

Figure 2: Communication Diagram for User Sign-Up

Scenario 2: Manage Expenses – Add Product

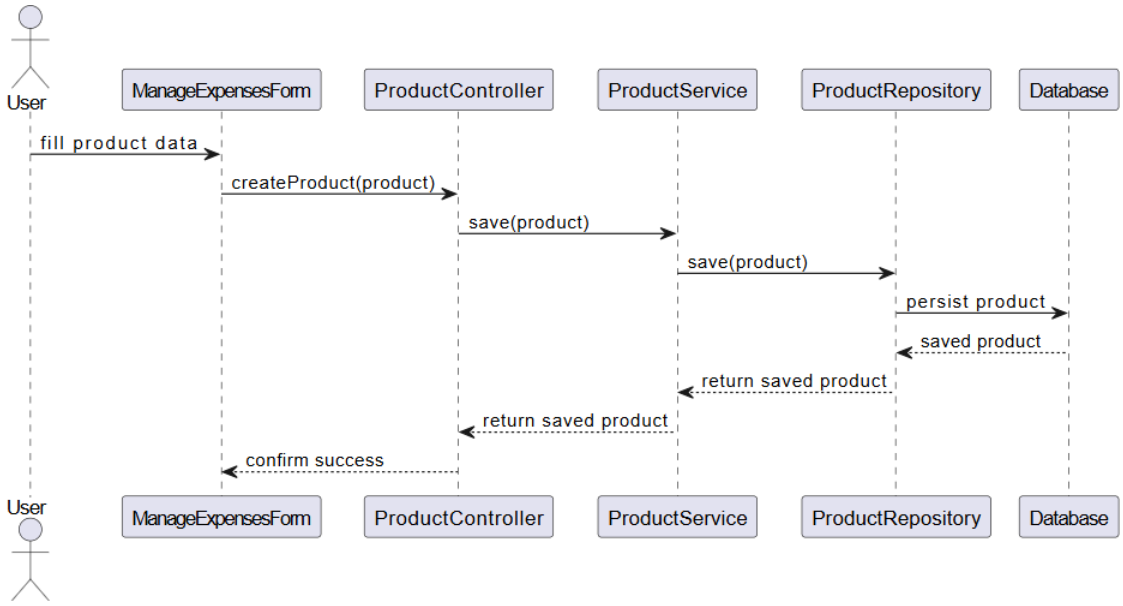

Figure 3: Sequence Diagram for Adding a Product

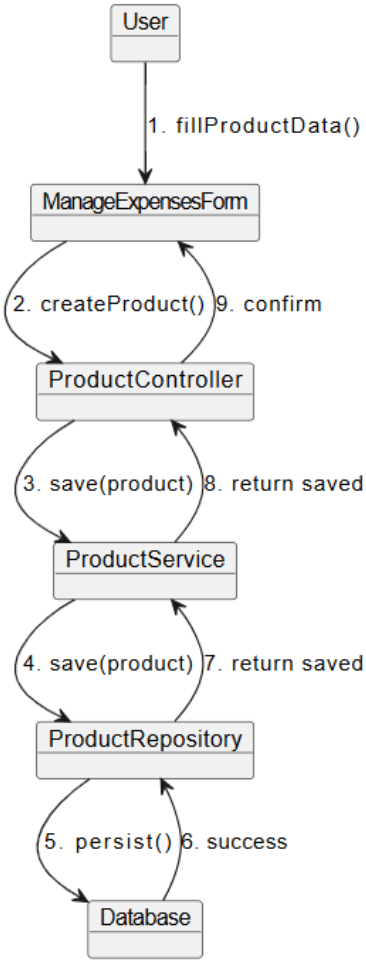

Figure 4: Communication Diagram for Adding a Product

2. Class Diagram

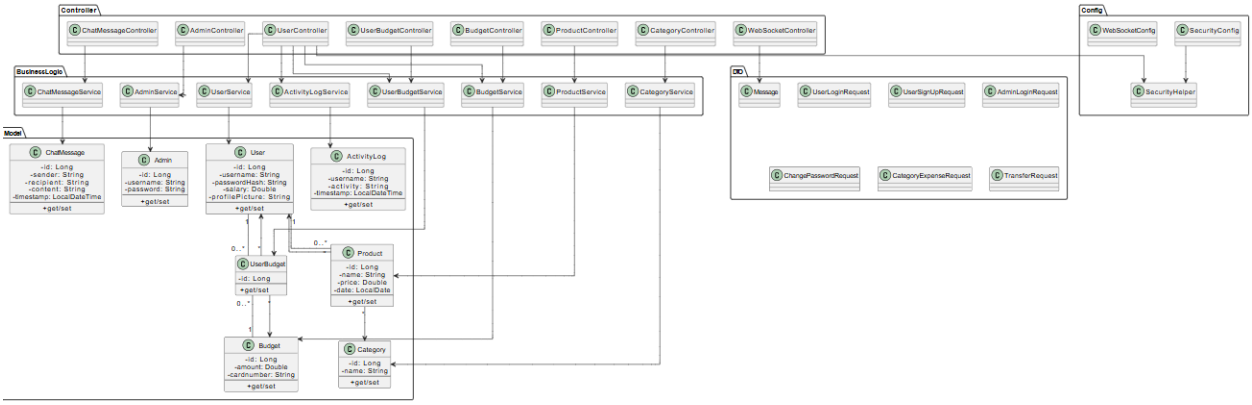

Figure 5: UML Class Diagram of the Application

The class diagram above represents the structure of the system, including core entities (such as `User`, `Product`, `Budget`, `Category`, etc.), service classes that encapsulate business logic, DTOs for data transfer, and controller classes responsible for handling incoming requests.

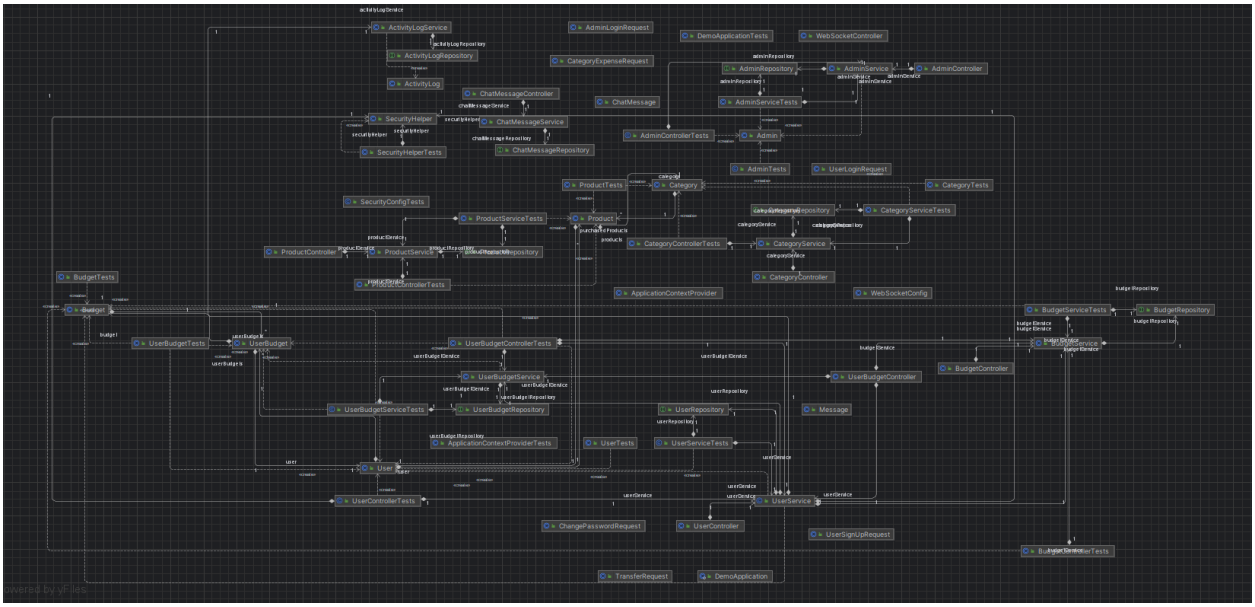

Figure 6: Detailed UML Class Diagram of the Application

2.1. Applied GoF Pattern: Strategy Pattern

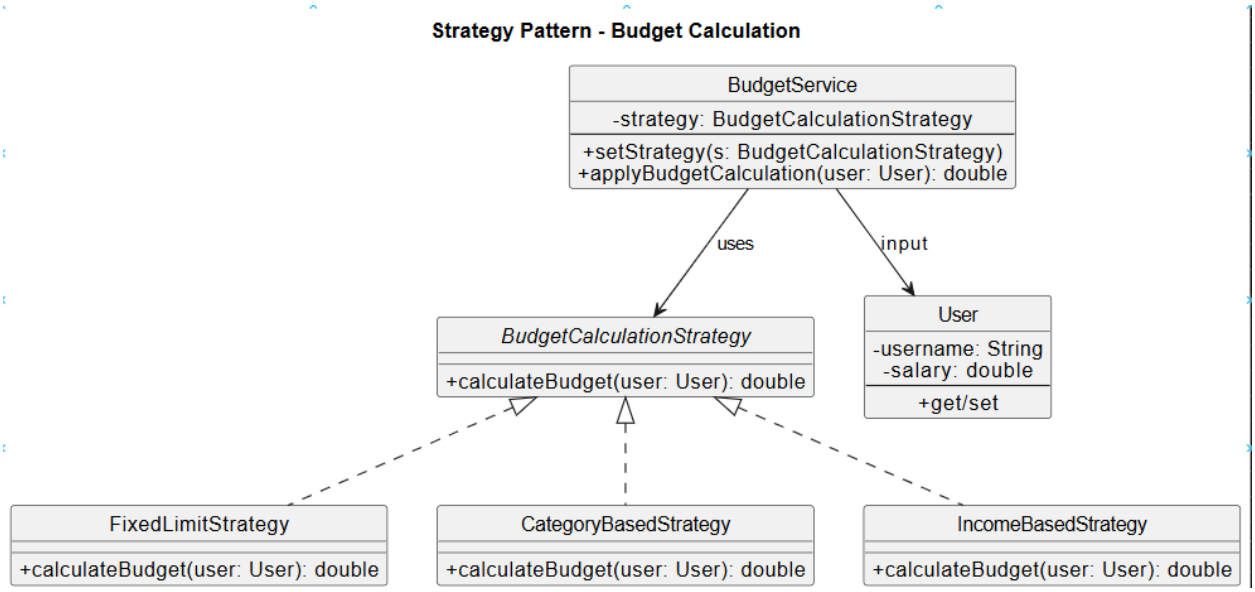

Figure 7: GoF Pattern

To ensure flexibility in handling different budgeting strategies (e.g., monthly limits, category-based allocations, income-based adjustments), the `BudgetService` can be extended using the *Strategy* design pattern. This allows different budget computation or validation strategies to be injected at runtime without modifying the service logic.

- **Context:** `BudgetService`
- **Strategy Interface:** `BudgetCalculationStrategy`
- **Concrete Strategies:** `FixedLimitStrategy`, `CategoryBasedStrategy`, etc.

Motivation: Applying the Strategy pattern makes the application more modular and scalable. It supports the Open/Closed Principle by allowing new strategies to be added without altering existing code and improves maintainability as the budgeting logic becomes decoupled from the service.

2.2. Applied GoF Pattern: Singleton Pattern

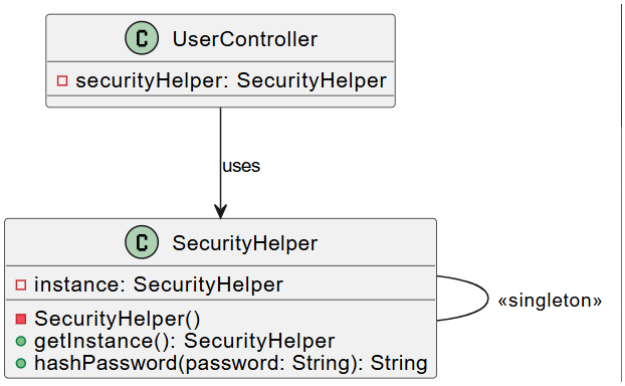

Figure 8: Singleton Pattern

The `SecurityHelper` class is a perfect candidate for the Singleton pattern since it provides stateless utility methods for password hashing. Singleton ensures that only one instance of `SecurityHelper` is used throughout the system.

- **Singleton Class:** `SecurityHelper`
- **Usage:** Shared by `UserController` and other services for password operations.

Motivation: Ensures consistent hashing logic and efficient resource usage by avoiding redundant instantiations.

2.3. Applied GoF Pattern: Factory Method Pattern

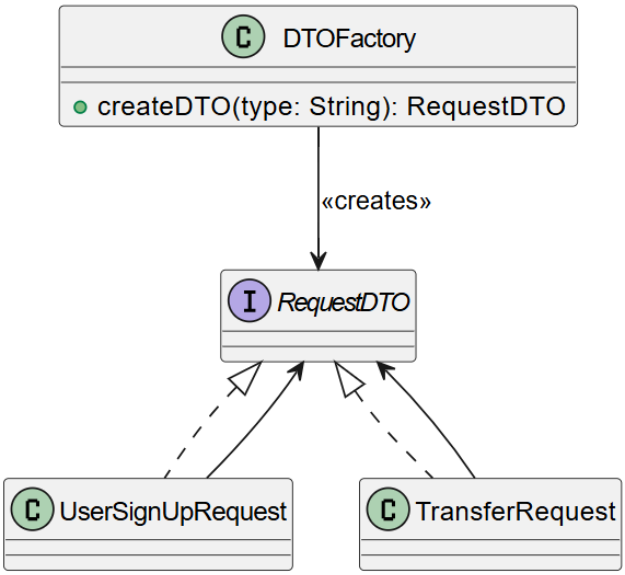

Figure 9: Factory Pattern

The Factory Method pattern can be applied to create different types of request or message objects in a centralized way. For instance, a future `DTOFactory` could produce DTOs based on the type of user interaction (chat, transfer, signup).

- **Creator:** `DTOFactory`
- **Product Interface:** `RequestDTO`
- **Concrete Products:** `UserSignUpRequest`, `TransferRequest`, etc.

Motivation: This pattern decouples the instantiation of DTOs from the controller logic, improving cohesion and reducing duplication.

2.4. Classes

a. Model Package

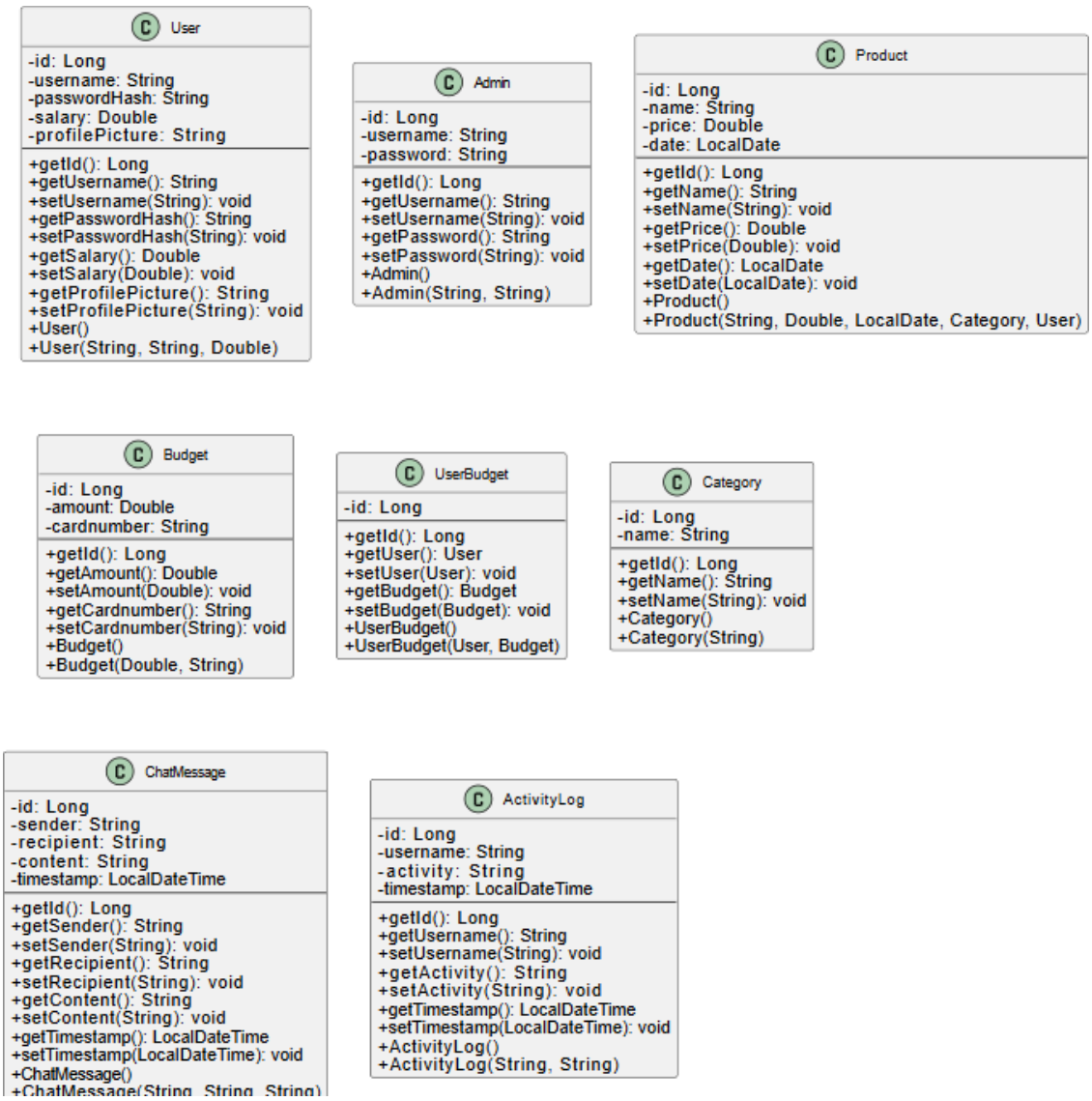

Figure 10: Model Classes

b. Business Logic (Service) Package

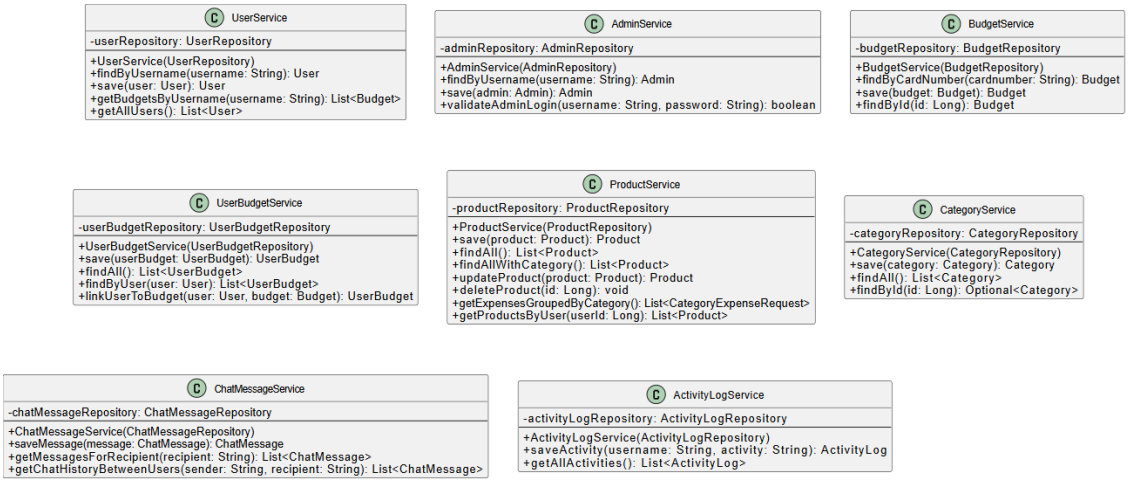

Figure 11: Bussines Logic Classes

c. Controllers Package

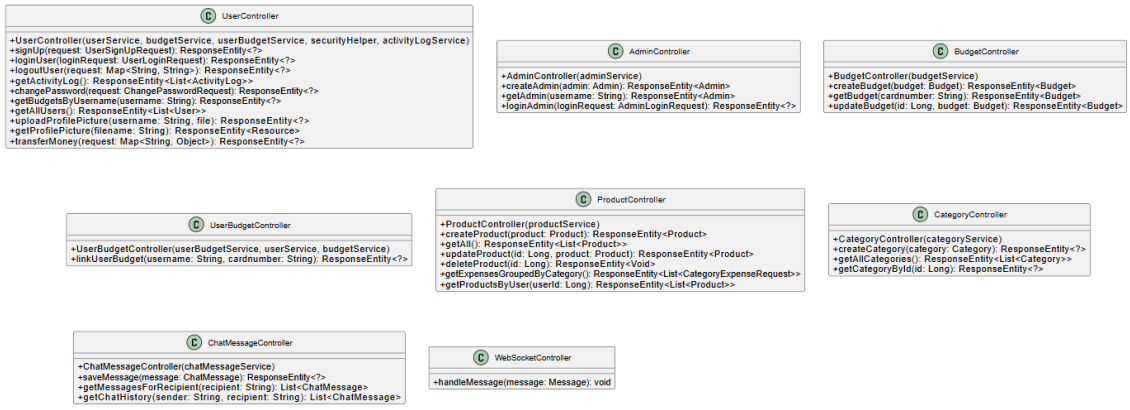

Figure 12: Controllers

d. DTO Package

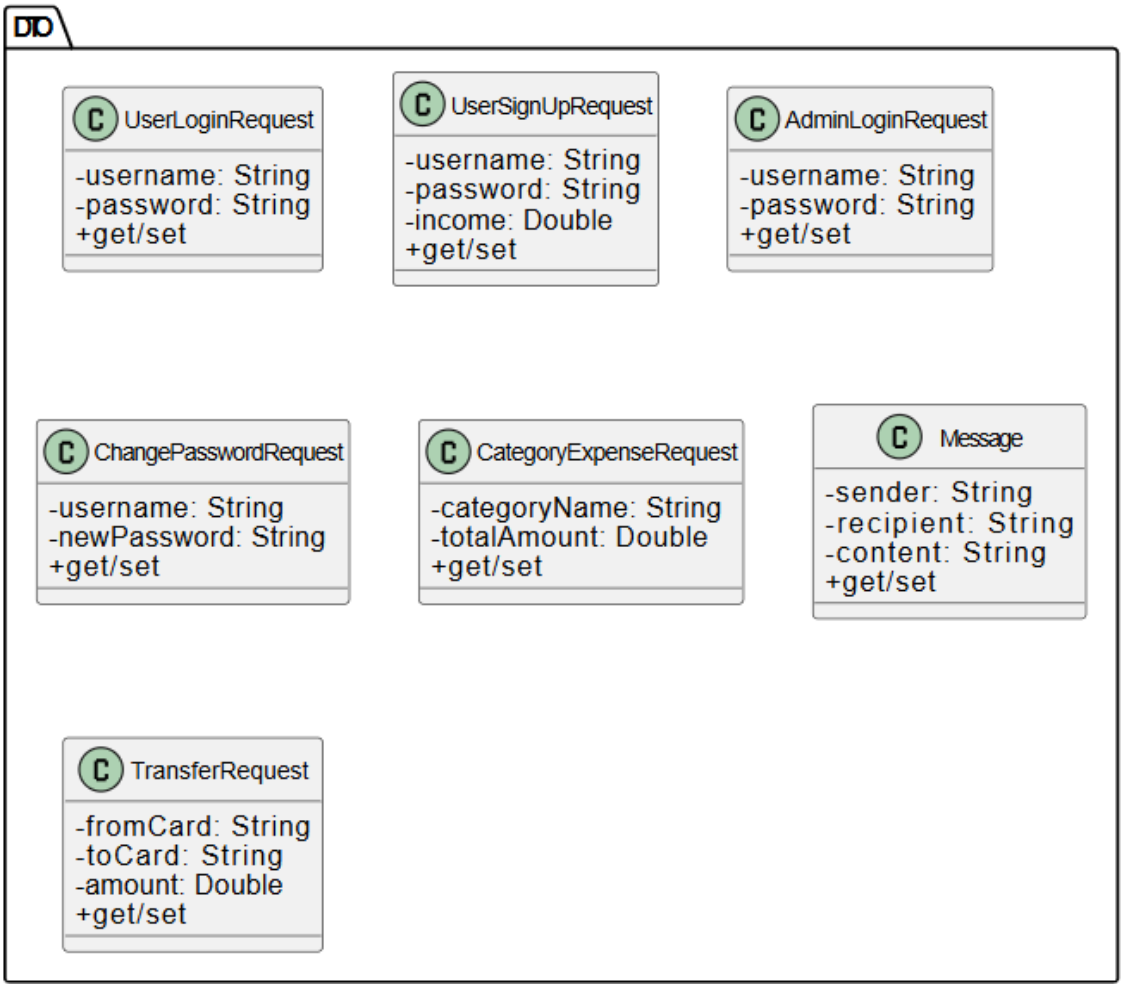

Figure 13: DTO

e. Configuration Package

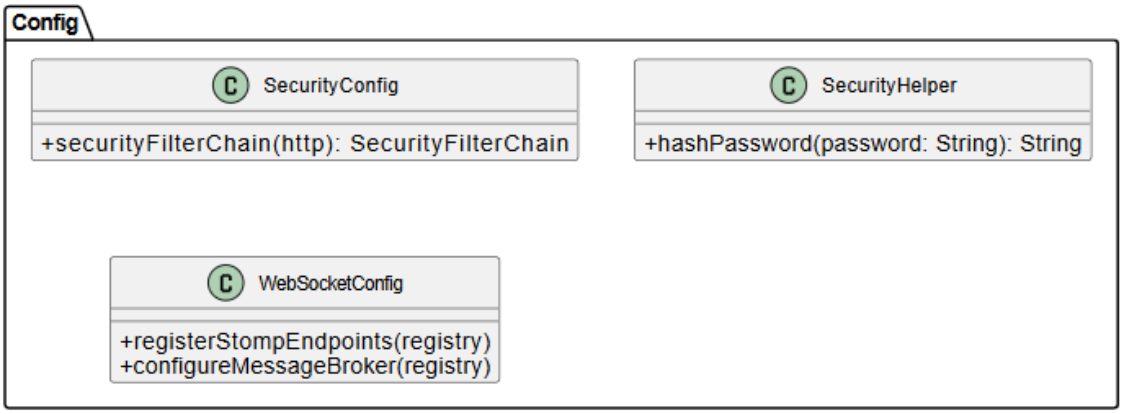

Figure 14: Config

f. Repository Package

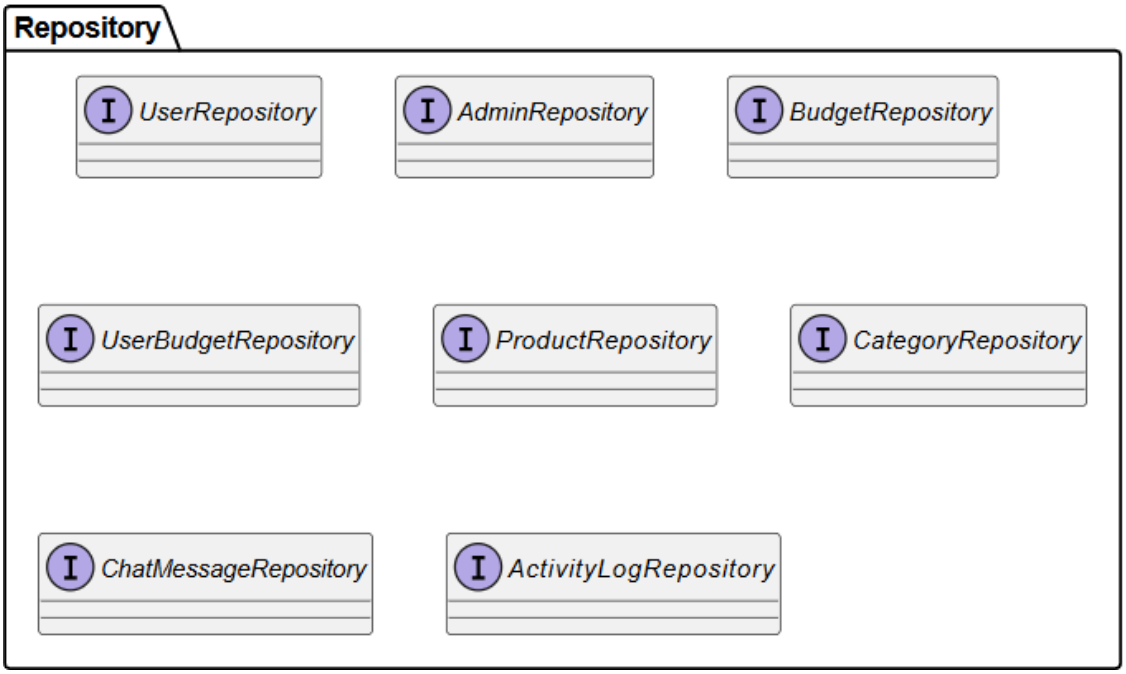

Figure 15: Repositories

II. Data Model

1. Architecture Style and Patterns

This system follows a layered (multitier) architecture pattern and adheres to the Model-View-Controller (MVC) paradigm:

- **Model Layer:** Contains domain entities such as `User`, `Product`, `Budget`, etc.
- **Service Layer:** Encapsulates business logic. Each service class (e.g., `UserService`, `ProductService`) handles core operations and interacts with the data access layer.
- **Controller Layer:** Handles HTTP requests and coordinates responses. Controllers act as bridges between user interfaces and backend logic.

Motivation: This separation of concerns promotes maintainability, testing, and scalability. Each layer can be modified independently and reused across different components or interfaces.

2. Data Model View

The data model represents the structure of the relational database that underpins the application's functionality. The system uses the following key entities:

- **User** – Stores information about each registered user, including username, password hash, income, and profile picture path.
- **Admin** – Stores administrator credentials.
- **Product** – Represents an expense or product added by a user, including name, price, date, and category.
- **Category** – Stores types of products (e.g., Food, Transport, Entertainment) and is linked to Products.
- **Budget** – Represents a budget associated with a card number and amount of money.
- **UserBudget** – A junction table linking Users and Budgets (many-to-many).
- **ChatMessage** – Contains chat messages between users, including sender, recipient, content, and timestamp.
- **ActivityLog** – Logs user actions such as login, logout, and password changes.

Entity Relationships:

- One `User` can be linked to many `Product`s.
- One `Category` can be linked to many `Product`s.
- Many `User`s can be linked to many `Budget`s via `UserBudget`.
- One `User` can send or receive many `ChatMessage`s.
- One `User` can have many `ActivityLog` records.

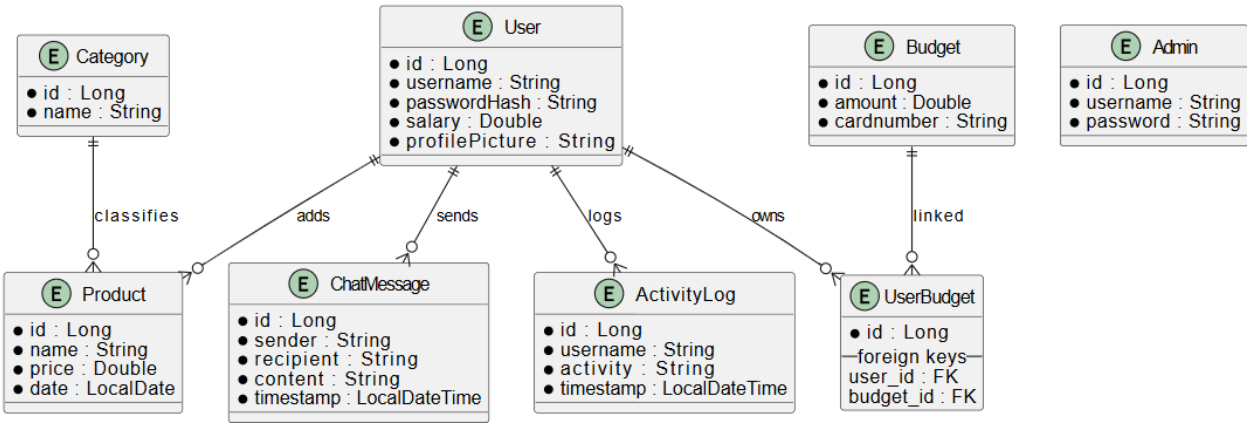

Figure 16: Entity-Relationship Diagram of the Application

III. System Testing

The system follows a layered architecture, and testing has been focused primarily on the business logic layer through unit tests. The testing strategy is based on the use of JUnit 5 and Mockito to validate service behavior independently from the database.

1. Testing Strategy

The application uses:

- **JUnit 5** for writing and executing unit tests.
- **Mockito** for mocking dependencies such as repositories, ensuring isolation of the service logic.
- **Arrange-Act-Assert (AAA)** testing structure to ensure clarity and modularity of each test.

The focus of the tests includes:

- Verifying the interaction between services and repositories.
- Ensuring correct data is returned or persisted.
- Handling edge cases (e.g., non-existent users, null values).

2. Test Case Examples

A. UserService Tests

- *testFindByUsername()* – Verifies that a user can be retrieved by username and the correct repository method is called.

```java
@Test
public void testFindByUsername() {
    String username = "testUser";
    User user = new User();
    user.setUsername(username);

    when(userRepository.findByUsername(username)).thenReturn(user);

    User result = userService.findByUsername(username);

    assertNotNull(result);
    assertEquals(username, result.getUsername());

    verify(userRepository, times( wantedNumberOfInvocations: 1)).findByUsername(username);
}
```

Figure 17: Test 1

- *testSave()* – Confirms that a new user is correctly saved and returned with the expected username.

```java
@Test
public void testSave() {
    User user = new User();
    user.setUsername("testUser");

    when(userRepository.save(user)).thenReturn(user);

    User result = userService.save(user);
    assertNotNull(result);
    assertEquals( expected: "testUser", result.getUsername());

    verify(userRepository, times( wantedNumberOfInvocations: 1)).save(user);
}
```

Figure 18: Test 2

B. UserBudgetService Tests

- *testSave()* – Checks that a new UserBudget entry is saved through the repository.

```java
@Test
public void testSave() {
    UserBudget userBudget = new UserBudget();
    when(userBudgetRepository.save(userBudget)).thenReturn(userBudget);
    UserBudget result = userBudgetService.save(userBudget);
    assertNotNull(result);
    verify(userBudgetRepository, times( wantedNumberOfInvocations: 1)).save(userBudget);
}
```

Figure 19: Test 3

- *testFindAll()* – Verifies retrieval of all user-budget relationships.
- *testFindByUser()* – Validates filtering of budgets associated with a given user.
- *testLinkUserToBudget()* – Ensures that linking a user to a budget creates a correct mapping and saves it.

C. BudgetService, ProductService, AdminService, and CategoryService are tested similarly:

- Each service's core operations such as `save()`, `findById()`, and `delete()` are tested using mocks.
- Edge cases like saving null objects or requesting missing IDs are also handled.

3. Tools Used

- **Maven Surefire Plugin** to automate test runs.
- **MockitoJUnitRunner** and `@Mock`, `@InjectMocks` for dependency injection.

Conclusion: All core services are tested in isolation with comprehensive test coverage, ensuring correctness and resilience of business logic.

IV. Future Improvements

While the current application provides solid functionality for budget monitoring, expense tracking, and secure user communication, several improvements can be considered for future development:

- **Mobile Application Support:** Develop an Android/iOS version using a cross-platform framework (e.g., Flutter) to increase accessibility and user engagement.
- **Advanced Analytics:** Integrate a dashboard with expense trends, savings goals, and predictive analytics using machine learning models.
- **Multi-Currency Support:** Allow users to manage budgets and expenses in different currencies, with automatic exchange rate updates.
- **Recurring Expenses and Alerts:** Enable users to define recurring expenses and receive notifications when nearing budget limits.
- **Bank Integration:** Introduce API integration with banking platforms for real-time transaction import and budget synchronization.
- **Role-Based Access Control (RBAC):** Extend the current user system with multiple roles and permissions, especially for admin dashboards.
- **Dark Mode and Accessibility:** Improve usability with theming options and enhanced support for accessibility tools.

These improvements aim to make the application more versatile, user-centric, and scalable for real-world adoption.

V. Conclusion

The Budget Monitoring System project was developed to provide users with an intuitive and functional platform for managing personal finances, tracking expenses, and analyzing budget usage. From its inception, the system has evolved through a structured development process encompassing requirements gathering, design modeling, implementation, and testing.

The application is based on a multi-layered architecture and leverages the MVC design pattern to ensure modularity and clear separation of concerns. Business logic is encapsulated in service classes, while data access is managed through JPA repositories. Controllers facilitate communication between the front-end and back-end, ensuring RESTful compliance.

GoF design patterns—such as Strategy, Singleton, and Factory Method—were applied where appropriate to enhance maintainability, testability, and code reuse. Testing was a critical part of the development lifecycle, and unit tests were implemented to validate the functionality of key service components.

Overall, the project achieves its goal of delivering a scalable, extensible, and user-friendly budgeting solution. It also lays a solid foundation for future enhancements such as analytics, mobile support, and bank integration.

VI. Bibliography

- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Spring Framework Documentation: https://docs.spring.io/spring-framework/docs/current/reference/html/
- PlantUML Documentation: https://plantuml.com/class-diagram