Department of Computer Science
Technical University of Cluj-Napoca

Software Design Project
Laboratory activity 2025

Name: Suciu Andrei
Group: 30431
Email: suciu.se.an@student.utcluj.ro

Software Design

Contents

1 Deliverable 1

1.1 Project Specification

Macro Buddy is a calorie tracking application designed to help users monitor their nutritional intake and achieve their dietary goals. The system allows users to track their daily food consumption by recording entries in a personal journal, organized by date and meal type.

The application is built using Java Spring framework with a PostgreSQL database for data persistence, containerized using Docker. The current implementation includes a mock frontend developed with Java Swing, following the Model-View-Controller architectural pattern.

Macro Buddy supports two types of users: regular users and administrators. Regular users can manage their personal food entries and customize their nutritional goals, while administrators have additional privileges to create and manage the food database accessible to all users.

1.2 Functional Requirements

1. **User Authentication and Authorization**
    - Users must be able to register with a unique username and email
    - Users must be able to log in securely with their credentials
    - The system must distinguish between regular users and administrators
    - Access to certain functions must be restricted based on user role
2. **Food Database Management**
    - Administrators must be able to create new food entries with nutritional information
    - The system must store comprehensive nutritional data for each food item
    - Users must be able to search and browse the food database
3. **Personal Journal Management**
    - Users must be able to add and remove food entries to their personal journal
    - Users must be able to specify date, meal type, and quantity for each entry
    - Users must be able to view their entries filtered by date
4. **Nutritional Goal Setting**
    - Users must be able to set personal nutritional goals for calories, protein, fat, and carbohydrates
    - The system must provide default values for new users
    - Users must be able to update their goals at any time
5. **Nutritional Analysis**
    - The system must calculate and display daily totals of nutritional intake
    - The system must compare daily totals against user goals
    - The system must provide visual feedback on goal progress

1.3 Use Case Model

1.3.1 Use Cases Identification

Use-Case: User Registration
Level: User goal
Primary Actor: Unregistered User
Main success scenario:

1. User provides username, email, and password
2. System validates input data
3. System creates new user account with regular user role
4. System initializes default nutritional goals for the user
5. System confirms successful registration

Extensions:

- Invalid or duplicate username/email: System notifies user and requests different input

Use-Case: User Login
Level: User goal
Primary Actor: Registered User
Main success scenario:

1. User provides username/email and password
2. System authenticates credentials
3. System grants access appropriate to user role

Extensions:

- Invalid credentials: System notifies user and allows retry

Use-Case: Add Food Entry
Level: User goal
Primary Actor: Regular User
Main success scenario:

1. User selects date and meal type
2. User searches for food item from database
3. User specifies quantity consumed
4. System calculates nutritional values based on quantity
5. System adds entry to user's journal
6. System updates daily nutritional totals

Extensions:

- Food item not found: User can request admin to add new food

Use-Case: Create New Food Item
Level: User goal
Primary Actor: Administrator
Main success scenario:

1. Admin provides food name, producer, and serving information
2. Admin inputs nutritional data (calories, protein, fat, carbs)
3. System validates input data
4. System adds new food item to database
5. System confirms successful addition

Extensions:

- Invalid or incomplete data: System highlights issues and requests corrections

Use-Case: Modify Nutritional Goals
Level: User goal
Primary Actor: Regular User
Main success scenario:

1. User accesses personal settings
2. User modifies calorie, protein, fat, and/or carb goals
3. System validates input values
4. System updates user's nutritional goals
5. System recalculates progress based on new goals

Extensions:

- Invalid values: System explains valid ranges and requests corrections

1.3.2 UML Use Case Diagrams

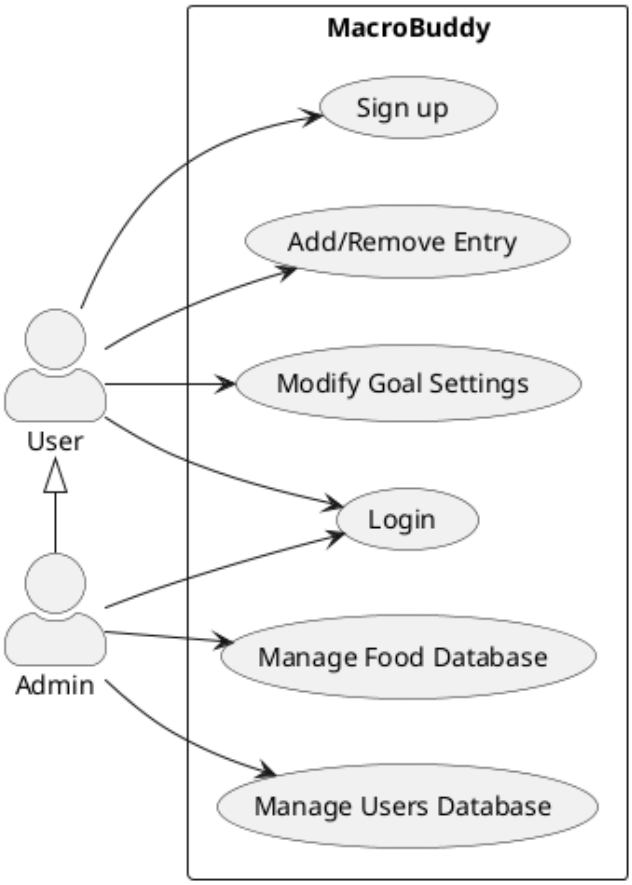


Figure 1: Macro Buddy Use Case Diagram

1.4 Supplementary Specifiaction

1.4.1 Non-Functional Requirements

1. Security
The system must securely store user credentials and personal data. Passwords must be stored using cryptographic hashing algorithms. Access to user journals and personal data must be restricted to the respective users only. Administrator privileges must be strictly controlled.

This requirement is suitable for implementation because:

- Nutritional tracking applications contain sensitive personal health information
- Spring Security framework provides robust and easy to use security features

2. Usability
The user interface must be intuitive and require minimal training. Common tasks should be accomplishable in three clicks or less. The system must provide meaningful feedback for user actions and clear error messages when needed.

This requirement is suitable for implementation because:

- Daily use applications must have low friction to encourage consistent use
- Java Swing allows for fast development of clean and intuitive interfaces

3. Performance
The system must be highly respond to user input under normal operating conditions. The application must operate efficiently with at least 100 concurrent users, with minimal overhead on the users.

This requirement is suitable for implementation because:

- Response time directly usability
- Spring and PostgreSQL can be configured for high performance
- Database indexing and query optimization can be implemented

4. Maintainability
The codebase must follow object-oriented design principles and established design patterns. The system architecture must support future extension of features without major refactoring.

This requirement is suitable for implementation because:

- The current package structure already supports separation of concerns
- Service-oriented architecture allows for modular development
- Unit tests provide a foundation for sustainable maintenance
- Java Spring supports extensible application design

1.4.2 Design Constraints

Technology Stack
The system must be implemented using Java Spring framework for the backend services and PostgreSQL for data persistence. The current implementation requires Java Swing for the frontend interface.

Architectural Patterns
The backend must implement a layered architecture with clear separation between controllers, services, and repositories. This ensures maintainability and supports potential future migration to a web-based interface.

Development Process
Development must follow test-driven development practices, with unit tests required for all service classes. Mockito and JUnit Jupiter are the preferred testing frameworks.

Deployment Environment
The database must be containerized using Docker to ensure consistent development and deployment environments. The application must be configurable to connect to either a local or remote PostgreSQL instance.

1.5 Glossary

Entry A record of food consumption by a user, including date, meal type, food item, and quantity.

- Format: Object with date (timestamp), meal (string), quantity (float), food reference, and user reference
- Validation: Quantity must be positive

Food Item A nutritional data record for a specific food, including macronutrient information.

- Format: Object with name, producer, serving information, and nutritional values
- Validation: Name is required, nutritional values must be non-negative

Macronutrients The primary nutritional components tracked by the system: protein, fat, and carbohydrates.

- Format: Floating-point values measured in the food's respective serving units
- Validation: Values must be non-negative

Nutritional Goals User-defined targets for daily intake of calories and macronutrients.

- Format: Integer for calories, floating-point for protein, fat, and carbohydrates
- Validation: Values must be positive

Meal Type A categorization of food entries based on time of day or purpose.

- Format: String value (e.g., "Breakfast", "Lunch", "Dinner", "Snack")
- Validation: Must be one of the predefined meal types

Regular User A standard user who can manage their personal journal and settings.

- Format: User object with role set to "ROLE_USER"
- Validation: Must have valid user credentials

Administrator A user with elevated privileges who can manage the food database.

- Format: User object with role set to "ROLE_ADMIN"
- Validation: Must have valid user credentials

2 Deliverable 2

2.1 Domain Model

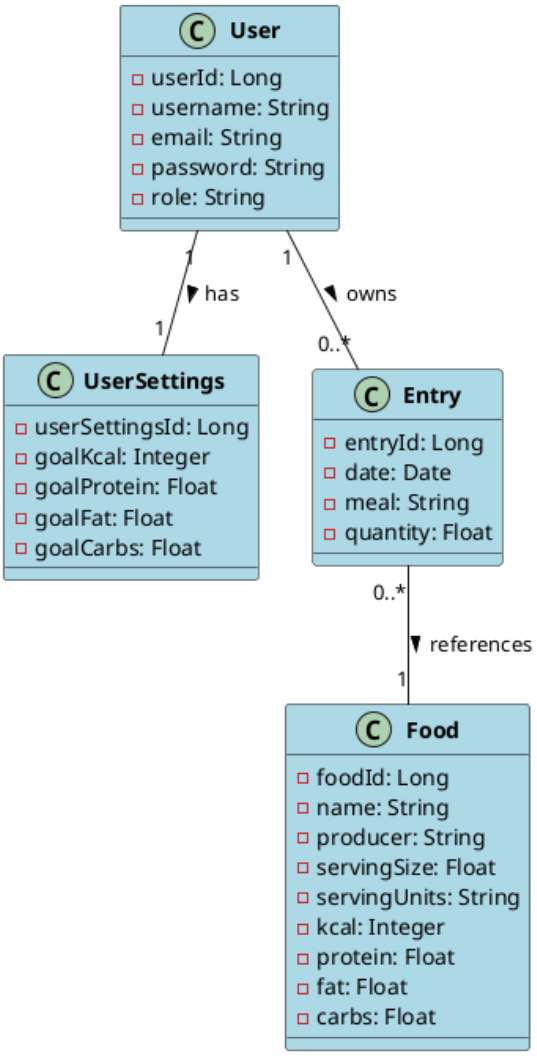


Figure 2: Macro Buddy Domain Model

The domain model illustrates the core entities of the system:

- **User**: Represents application users with authentication information and role designation (regular user or administrator).
- **UserSettings**: Contains the nutritional goals and other settings for a specific user.
- **Food**: Represents food items in the database with their nutritional information.

- **Entry**: Represents a food consumption record in a user's journal.

Key relationships:

- Each User has exactly one UserSettings profile.
- A User can have multiple Entry records (their food journal).
- Each Entry references exactly one Food item.

2.2 Architectural Design

2.2.1 Conceptual Architecture

The application implements client-server architecture with separation between the frontend and backend components. The backend follows a layered architecture pattern, while the frontend implements its own React-based architectural pattern. Together, they form a comprehensive web application with a React TypeScript frontend and a Java Spring RESTful API backend.

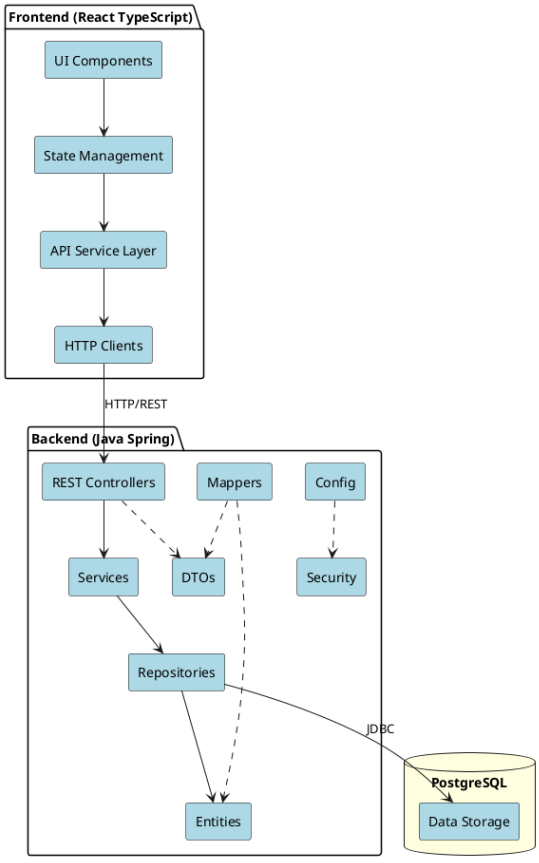


Figure 3: Conceptual Architecture

Architectural Style Justification:
The client-server architecture with layered backend was chosen for the following reasons:

1. **Clear Separation of Concerns**: The architecture separates the frontend (React) from the backend (Spring) with a well-defined REST API interface between them. Within each part, there are further separations (UI components/services in frontend, controllers/services/repositories in backend).

2. **Enhanced Security**: The architecture implements modern security practices with JWT for stateless authentication, CSRF protection for request verification, and secure password encryption.

3. **Maintainability and Testability**: Each layer has clear responsibilities, making it easier to test components in isolation and update specific parts without affecting the entire system.

The use of DTOs (Data Transfer Objects) and Mappers further enhances the architecture by:

- Providing a clear contract between frontend and backend
- Protecting internal domain objects from direct exposure
- Allowing for different representations of the same data for different use cases

2.2.2 Package Design

The package structure reflects the layered architecture and organizes code by technical responsibility.

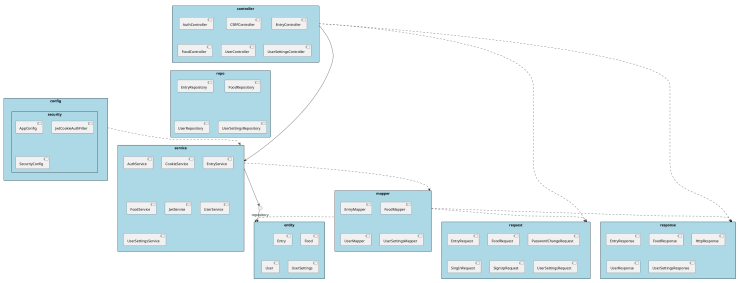


Figure 4: Package Diagram

The package design follows a clear dependency direction, where higher-level packages (controller) depend on lower-level packages (service, repository), maintaining the layered architecture principles:

- **config**: Contains application configuration for security features (JWT, CSRF, password encryption)
- **controller**: REST API endpoints that handle HTTP requests and responses

- **service**: Business logic implementation
- **repo**: Data access layer for interacting with the database
- **entity**: JPA entity classes that map to database tables
- **mapper**: Converts between entities and DTOs
- **request/response**: DTOs for API input and output

2.2.3 Component and Deployment Diagram

The component and deployment diagrams illustrate the runtime architecture and physical deployment structure of the system.

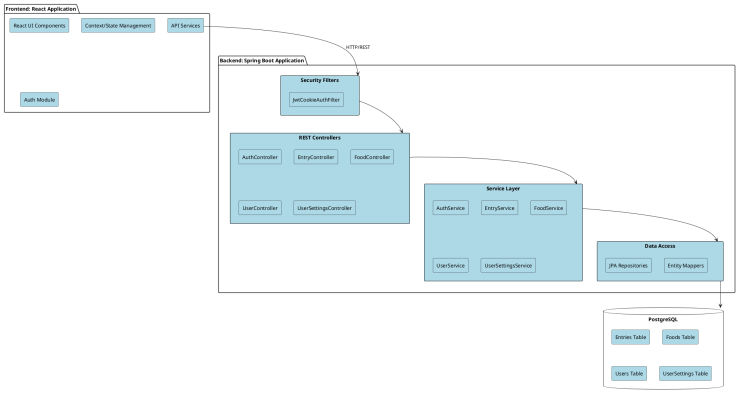


Figure 5: Component Diagram

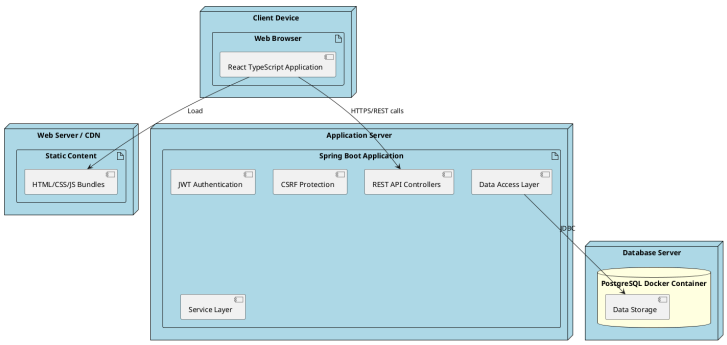


Figure 6: Deployment Diagram

The component diagram shows the internal structure of the application with a focus on the logical components and their interactions, while the deployment diagram illustrates how these components are distributed across physical hardware or containerized environments.
Key aspects of the deployment architecture:

- The frontend is a TypeScript React web application loaded in the client's browser.
- Static content (compiled React code) is served via a web server or CDN for optimal performance.
- The Spring Boot backend provides secure REST API endpoints with JWT authentication and CSRF protection.
- Business logic is implemented in the service layer, with clear separation from the data access layer.
- The PostgreSQL database runs in a Docker container for consistency across development and production environments.
- All client-server communication occurs via HTTP/S using RESTful conventions.

3 Deliverable 3

3.1 Design Model

3.1.1 Dynamic behavior

Sequence Diagram: Adding a Food Entry
The sequence diagram illustrates the process of a user adding a food entry to their journal, showcasing the interactions between the frontend components and backend services.

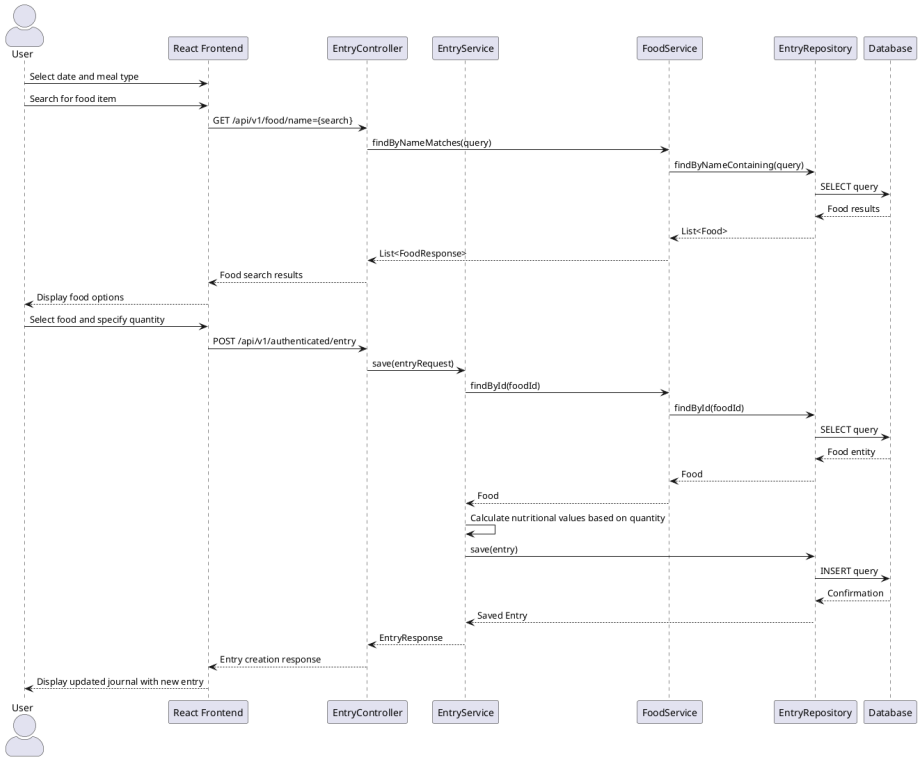


Figure 7: Sequence Diagram

Communication Diagram: Creating a New Food
The communication diagram illustrates the process of an admin creating a new food. It shows the transformation between DTOs and entity objects, and demonstrates the separation of the services.

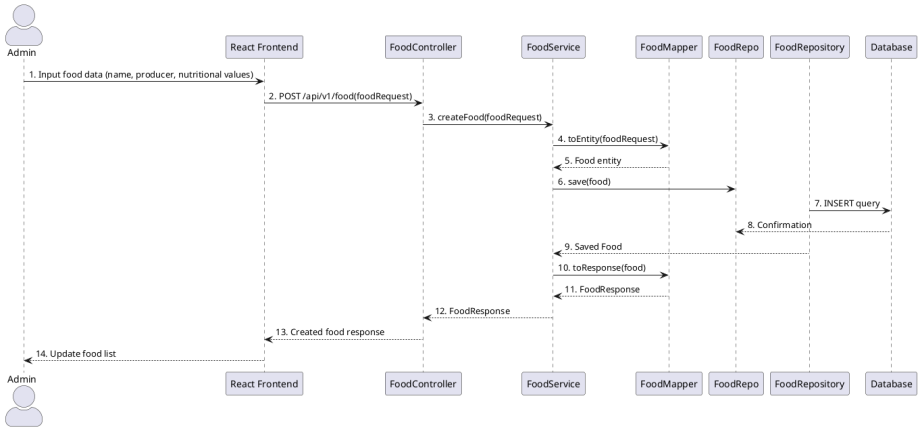
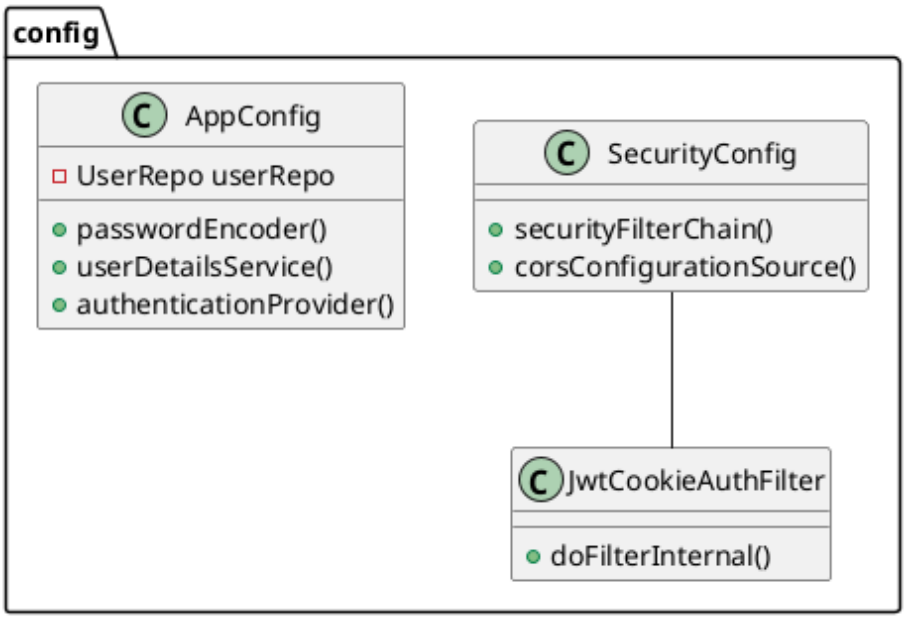
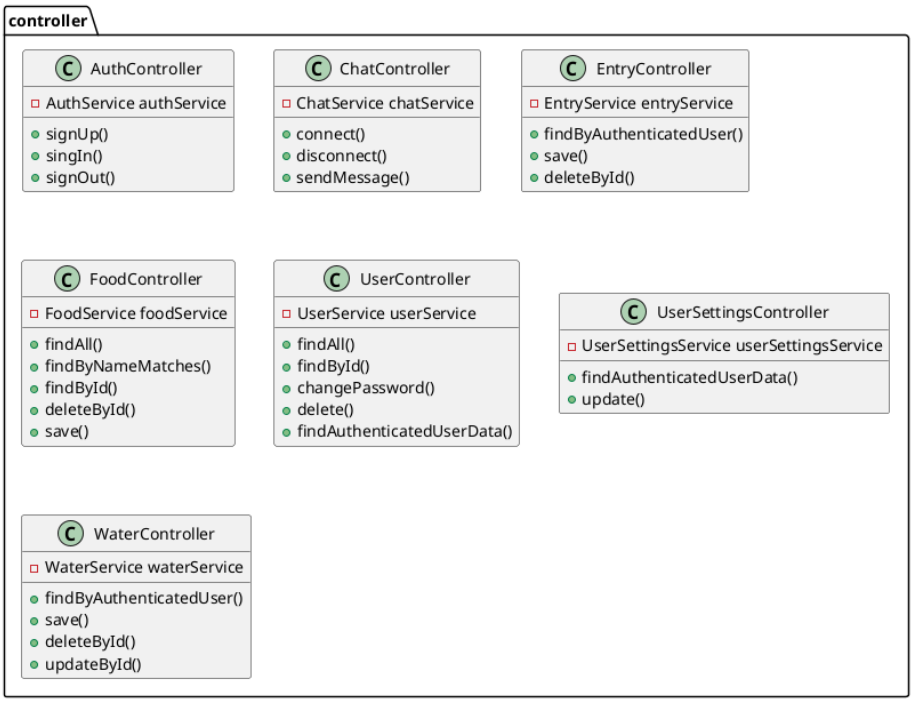


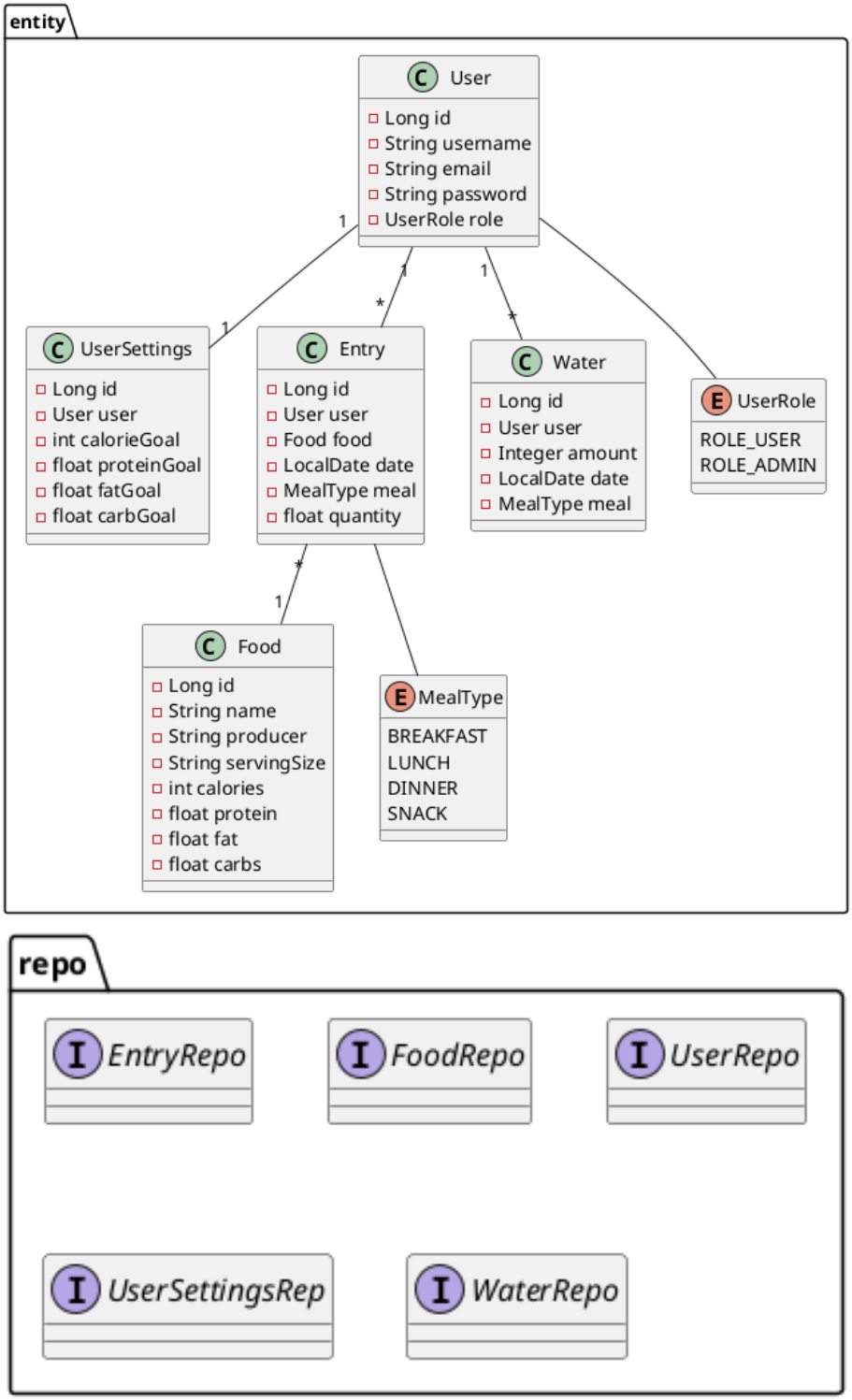
Figure 8: Communication Diagram

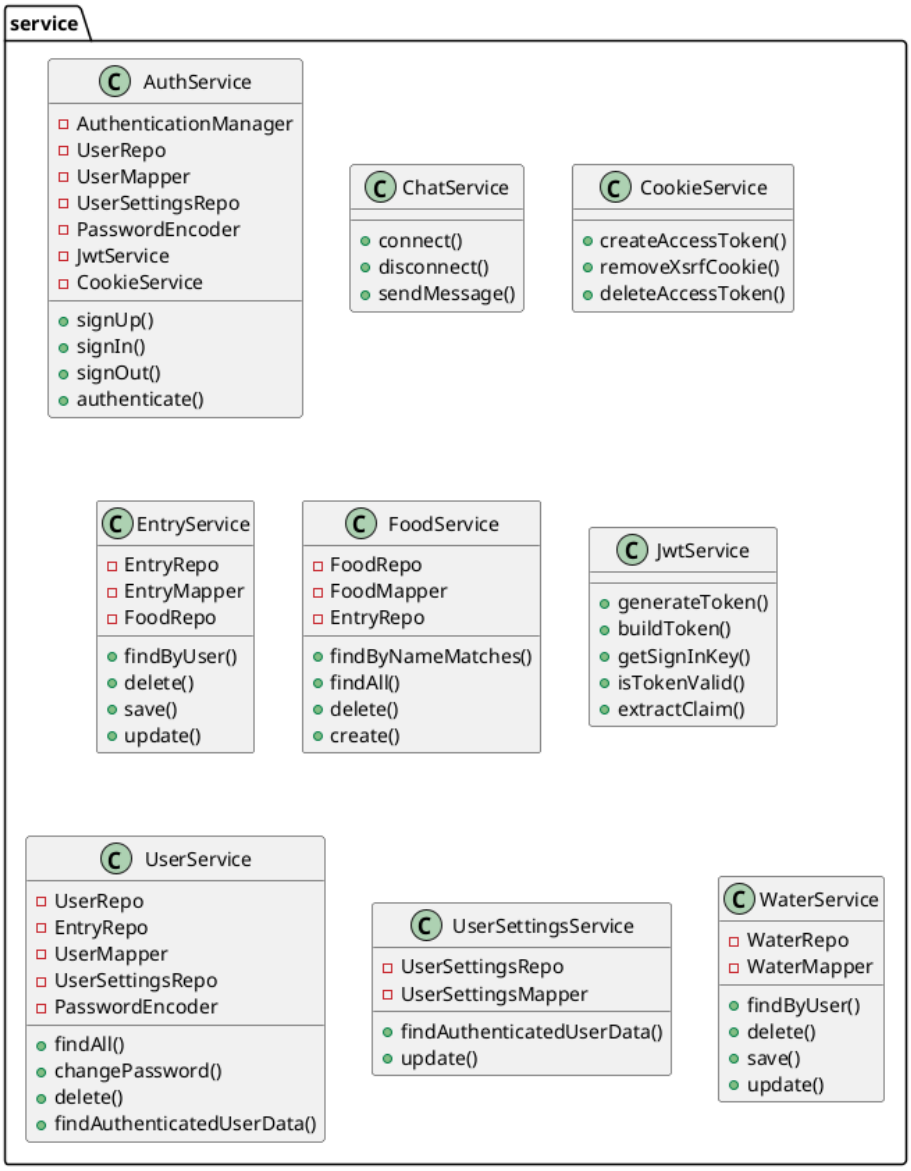
3.1.2 Class Diagram

The following class diagrams illustrate the structure of the Macro Buddy application.




controller
AuthController
- AuthService authService
+ signUp()
+ singIn()
+ signOut()
ChatController
- ChatService chatService
+ connect()
+ disconnect()
+ sendMessage()
EntryController
- EntryService entryService
+ findByAuthenticatedUser()
+ save()
+ deleteById()
FoodController
- FoodService foodService
+ findAll()
+ findByNameMatches()
+ findById()
+ deleteById()
+ save()
UserController
- UserService userService
+ findAll()
+ findById()
+ changePassword()
+ delete()
+ findAuthenticatedUserData()
UserSettingsController
- UserSettingsService userSettingsService
+ findAuthenticatedUserData()
+ update()
WaterController
- WaterService waterService
+ findByAuthenticatedUser()
+ save()
+ deleteById()
+ updateById()


entity
User
Long id
String username
String email
String password
UserRole role
UserSettings
Long id
User user
int calorieGoal
float proteinGoal
float fatGoal
float carbGoal
Entry
Long id
User user
Food food
LocalDate date
MealType meal
float quantity
Water
Long id
User user
Integer amount
LocalDate date
MealType meal
UserRole
ROLE_USER
ROLE_ADMIN
Food
Long id
String name
String producer
String servingSize
int calories
float protein
float fat
float carbs
MealType
BREAKFAST
LUNCH
DINNER
SNACK
repo
EntryRepo
FoodRepo
UserRepo
UserSettingsRep
WaterRepo



The class diagram incorporates several Gang of Four (GoF) design patterns to enhance the application's architecture:

1. **Strategy Pattern**: The service interfaces (UserService, FoodService, EntryService) with their implementations (UserServiceImpl, FoodServiceImpl, EntryServiceImpl) demonstrate the Strategy pattern. This allows for different implementations of the same interface, making the system more flexible and testable.

2. **Factory Method Pattern**: The JWT token provider uses the Factory Method pattern to create authentication tokens, encapsulating the token creation logic within a specialized class.

3. **Singleton Pattern**: The SecurityConfig and other configuration classes

are implemented as Spring @Configuration beans, which are effectively singletons managed by the Spring container.

4. **Adapter Pattern**: The Mapper classes (not fully shown) implement the Adapter pattern by converting between DTOs and entity objects, bridging the gap between the presentation layer and the domain model.

5. **Facade Pattern**: The service layer acts as a facade to the underlying repository operations, providing a simplified interface to the controllers and hiding the complexity of data access and business logic.

These design patterns were used to promote separation of concerns, enhance maintainability, and improve testability. The Strategy pattern, in particular, facilitates mock implementations for testing, while the Adapter pattern ensures that the internal domain model remains decoupled from the external API representation.

3.2 Data Model

The data model for Macro Buddy is implemented using JPA entities mapped to PostgreSQL tables. The following diagram illustrates the relational database schema:

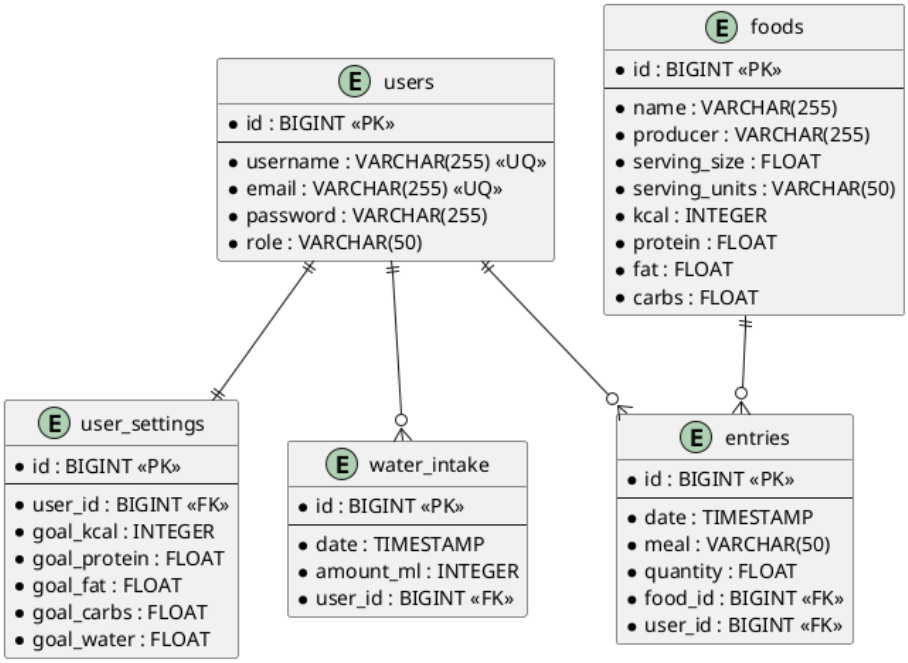


Figure 9: Data Model

The database schema is designed with the following considerations:

1. **Normalization**: The schema follows normal form conventions to minimize data redundancy and improve data integrity.
2. **Constraints**:
   - Foreign key constraints ensure referential integrity
   - Unique constraints on username and email prevent duplicates
   - Check constraints ensure that nutritional values are non-negative

JPA Implementation The Java entities are mapped to this schema using JPA annotations:

- @Entity, @Table annotations map classes to database tables
- @Id, @GeneratedValue define primary keys and their generation strategy
- @Column specifies column details including constraints
- @ManyToOne, @OneToMany, @OneToOne establish entity relationships

- @JoinColumn defines foreign key columns
- @Enumerated maps Java enums to database columns

Hibernate is configured to automatically validate the schema against the entity definitions during application startup, ensuring consistency between the code and database schema.

3.3 System Testing

The testing strategy for Macro Buddy encompasses multiple testing levels to ensure the application's quality, reliability, and security. The following approach is implemented:

Unit Testing:

- JUnit is used as the primary testing framework
- Mockito provides mocking capabilities for isolating components
- Each service class has dedicated test cases

3.4 Future Improvements

Based on the current implementation of Macro Buddy, several potential improvements and feature extensions could enhance the application's functionality and user experience:

Functional Enhancements

1. **Recipe Management**:
    - Allow users to create custom recipes composed of multiple food items
    - Calculate nutritional information for entire recipes
    - Enable sharing recipes between users
2. **Integration Capabilities**:
    - API for integration with fitness tracking applications
    - Import nutritional data from external databases
    - Barcode scanning functionality for quick food entry
    - Export capabilities for nutritional data

Technical Improvements

1. **Enhanced Security**:
    - Support for multi-factor authentication
    - Support for Google Captcha
    - Encryption of sensitive user data
2. **Mobile Support**:
    - Develop a dedicated mobile application
    - Implement offline functionality for journal entries

User Experience Enhancements

1. **Personalization**:
    - Customizable dashboard with widgets
    - Multiple themes for the entire application
2. **Social Features**:
    - Achievement system for meeting nutritional goals
    - Optional sharing of progress on social platforms
3. **Educational Content**:
    - Integrated nutritional articles and guides
    - Tips for maintaining a balanced diet
    - Explanations of nutritional concepts and terminology

These improvements would significantly enhance Macro Buddy, transforming it from a basic calorie tracker into a comprehensive nutritional management platform.