CruxBeta

Analysis and Design Document: Assignment 1

Student: Duțu Ana Sofia

Group: 30435

Table of Contents

1.Requirements Analysis

1.1. Assignment Specification

This project represents the foundation of a climbing platform, aiming to support climbers by providing essential features such as user management, product ordering, and, in future iterations, location-based climbing spot listings.

1.1.1. Current Functionality

- Entities Implemented: Users, Admins, Products (climbing-related), and Orders.
- Database Connection: The backend is fully integrated with a database to manage entity relationships.
- Backend Framework: Spring Boot, implementing RESTful services.
- Frontend Framework: React, displaying user data in a structured table.
- CRUD Operations: Basic operations (Create, Read, Update, Delete) are implemented for users.

1.1.2. Future Goals

- Location Listings: Integrate climbing spots in Cluj, allowing users to explore available locations.
- Expanded Product Ordering: Enable purchasing and managing climbing-related products directly from the platform.
- User Roles & Permissions: Implement role-based access for administrators and customers.

1.2. Functional Requirements

➔ User Management

- Support admins and customers.
- CRUD operations for users.
- Display all users in a table (React frontend).

➔ Product Management

- CRUD operations for climbing products.

➔ Order Management

- CRUD operations for orders.
- Users can place orders for climbing products.
- 1:N (users-orders), N:M (orders-products) relationships.

➔ Database & Frontend

- Spring Boot backend with ORM for database integration.
- React frontend for managing users.

1.3 Non-functional Requirements

The application has the following non-functional requirements:

- Validation: Ensure valid input for all entities (e.g., unique emails, valid product details).
- ORM Usage: Uses JPA with Hibernate for database interactions.
- Dependency Injection: Utilizes Spring Boot's DI container for maintainability.
- Layered Architecture: Implements Controller-Service-Repository separation.
- Database Persistence: Stores users, products, and orders in a PostgreSQL database.
- Error Handling: Provides meaningful error messages for database failures and invalid input.

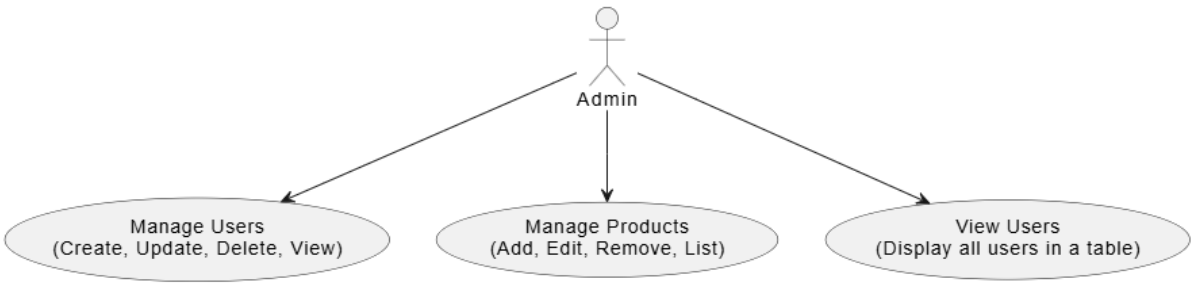
2. Use-Case Model

The system includes three main actors:

1. Admin – Manages users and products.
2. Customer – Browses products and places orders.
3. System – Handles database operations and enforces validation.

Use Cases

- Manage Users (Admin)
    - Create, update, delete, and view users.
- Manage Products (Admin)
    - Add, edit, remove, and list climbing products.
- Place Orders (Customer)
    - Select products and submit orders.
- View Users (Admin)
    - Display all users in a table (frontend).
- Data Persistence (System)
    - Store and manage entities using ORM.



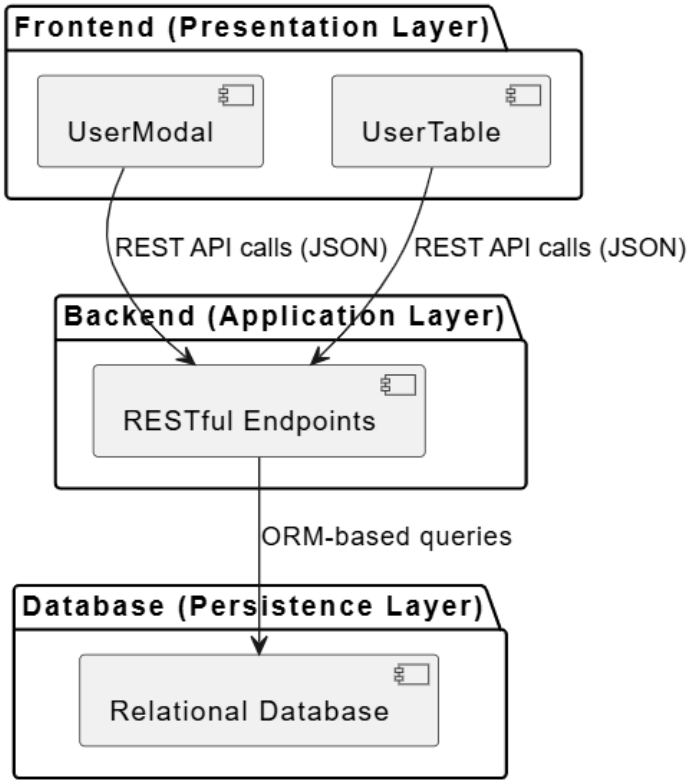
3. System Architectural Design

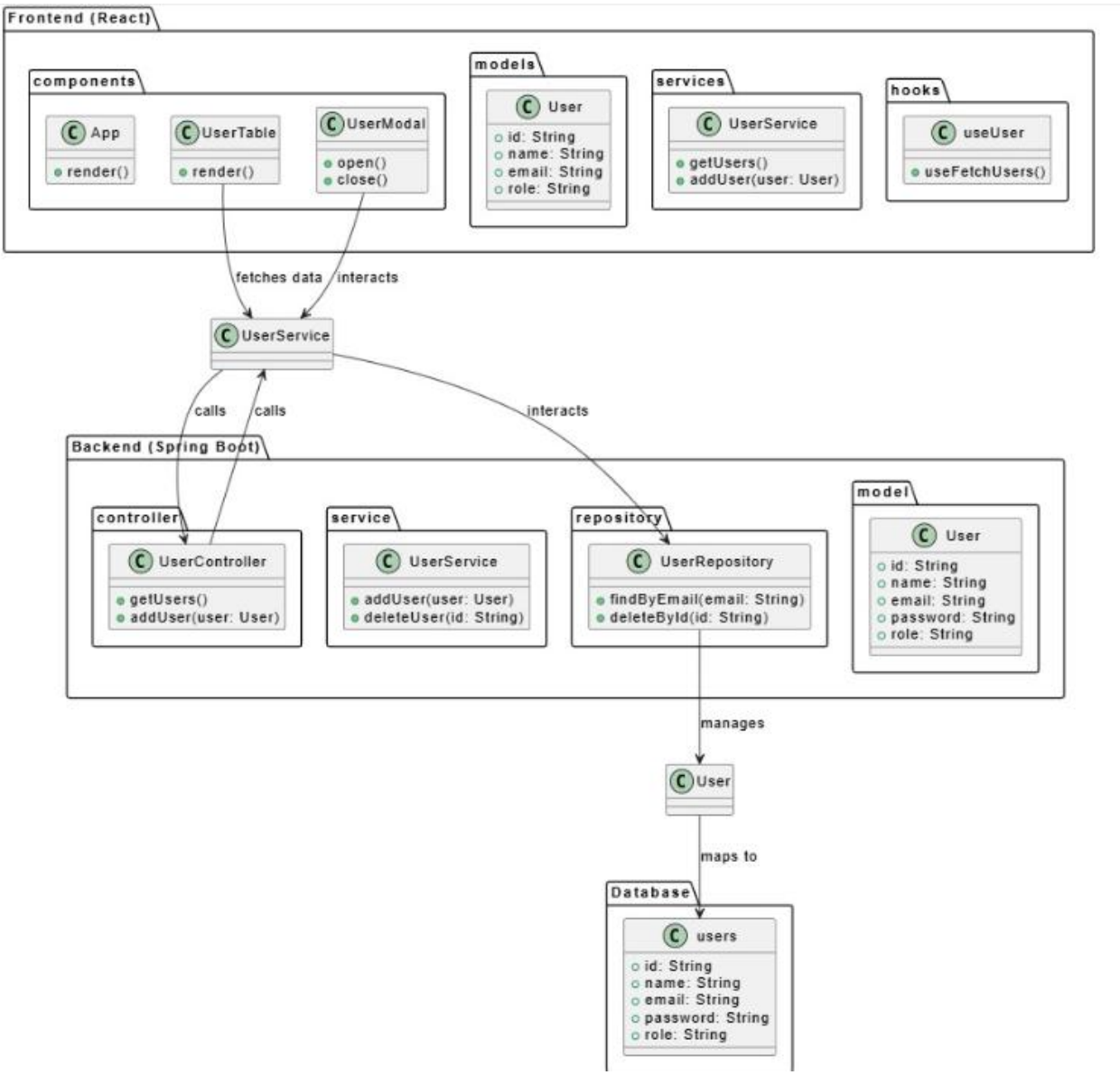
The system follows a three-tier architecture , ensuring modularity, scalability, and maintainability. It consists of:

1. Frontend (Presentation Layer) :
    - Built with React for a responsive and interactive user interface.
    - Communicates with the backend via RESTful APIs using Axios .
    - Modular components like UserModal and UserTable ensure reusability.
2. Backend (Application Layer) :
    - Developed using Spring Boot to handle business logic and API requests.
    - RESTful endpoints (e.g., GET /users, POST /users) manage CRUD operations.
    - Spring Data JPA integrates with the database using ORM for efficient data access.
3. Database (Persistence Layer) :
    - A relational database (e.g., PostgreSQL , MySQL) stores application data.

- Proper indexing, constraints, and ORM mapping ensure performance and data integrity.

4. Communication :
    - Frontend ↔ Backend: JSON-based REST APIs over HTTP/HTTPS.
    - Backend ↔ Database: ORM (Hibernate) abstracts SQL queries.
5. Scalability :
    - Designed for horizontal scaling and cloud deployment.
    - Supports containerization with Docker for easy deployment.
6. Technology Stack :
    - Frontend: React, Axios, TypeScript.
    - Backend: Spring Boot, Java, Hibernate.
    - Database: PostgreSQL/MySQL.




Frontend (React)
components
App
render()
UserTable
render()
UserModal
open()
close()
models
User
id: String
name: String
email: String
role: String
services
UserService
getUsers()
addUser(user: User)
hooks
useUser
useFetchUsers()
fetches data
interacts
UserService
calls
calls
interacts
Backend (Spring Boot)
controller
UserController
getUsers()
addUser(user: User)
service
UserService
addUser(user: User)
deleteUser(id: String)
repository
UserRepository
findByEmail(email: String)
deleteById(id: String)
model
User
id: String
name: String
email: String
password: String
role: String
manages
User
maps to
Database
users
id: String
name: String
email: String
password: String
role: String

4. Component Design

4.1. Backend Overview

Controller Layer

- **OrderController** : Exposes RESTful endpoints for managing orders, such as creating, updating, and deleting orders.
- **ProductController** : Handles product-related endpoints, enabling CRUD operations on products.
- **UserController** : Manages user-related endpoints, including user creation, updates, and deletions.

Model Layer

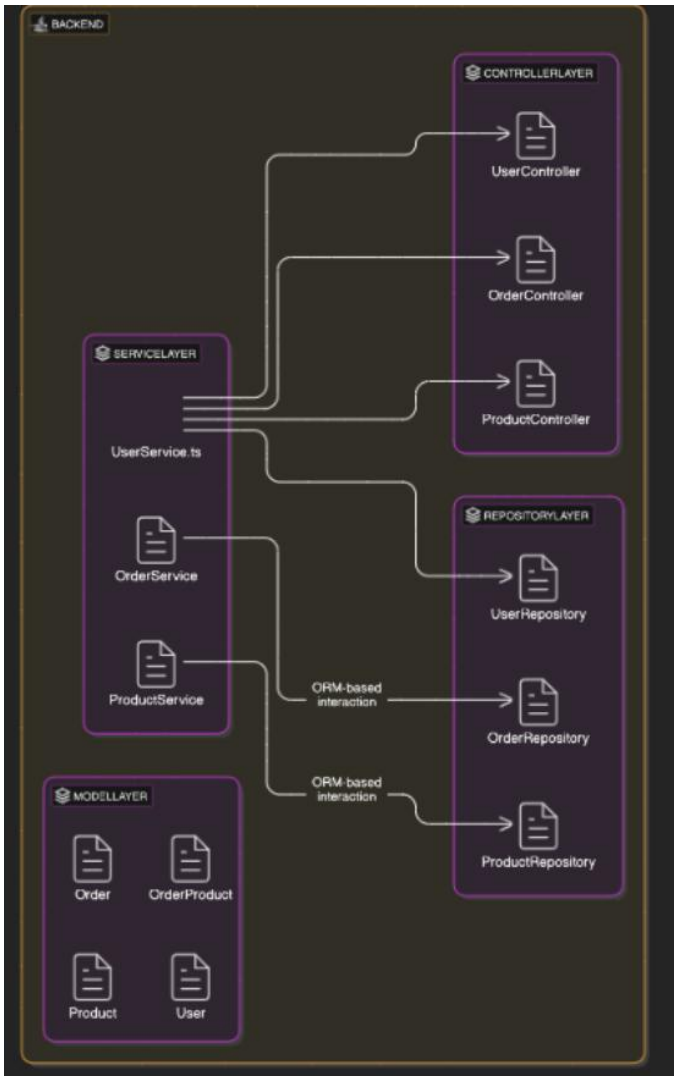
- **Order** : Represents an order with fields like **id**, **totalPrice**, **status**, and a list of **OrderProduct** items.
- **OrderProduct** : Links orders and products, storing details like **quantity** and **price** for each product in an order.
- **Product** : Represents a product with attributes like **name**, **description**, **price**, and **stock**.
- **User** : Models a user with fields like **id**, **name**, **email**, **password**, and **role**.

Repository Layer

- **OrderRepository** : Provides database access for orders, supporting custom queries like **findByUserId**.
- **ProductRepository** : Enables querying products by attributes like **name** or **category**.
- **UserRepository** : Offers methods to find users by **email** or **role**, extending **JpaRepository**.

Service Layer

- **OrderService** : Implements business logic for order management, such as calculating totals and validating order data.
- **ProductService** : Handles product-related operations, including stock updates and price adjustments.
- **UserService** : Manages user-related operations, such as authentication and role-based access control.



4.2.Frontend Overview

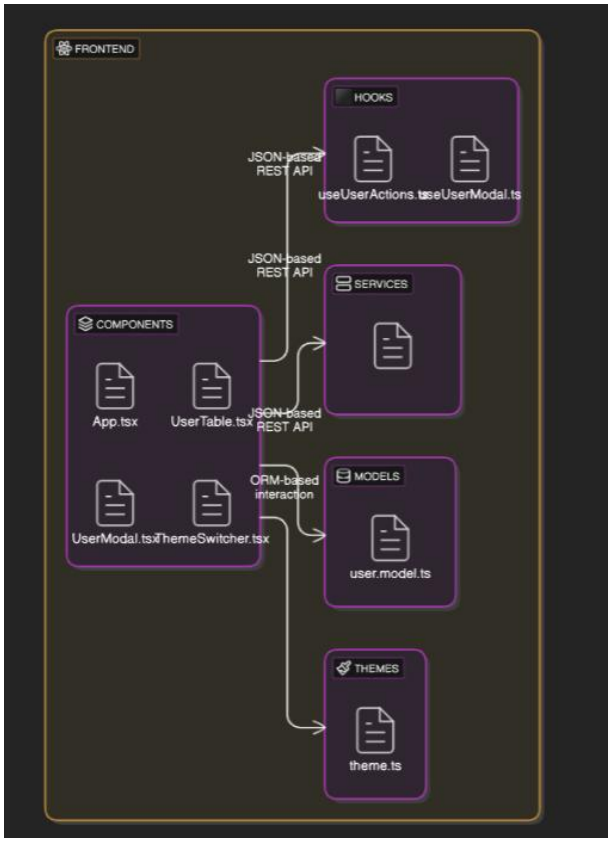
The frontend is built with React and follows a modular structure. Key components include:

- **App.tsx** : The main entry point integrating all components.
- **UserTable.tsx** : Displays users in a table with sorting and filtering.
- **UserModal.tsx** : Handles adding or updating user details via a modal.
- **ThemeSwitcher.tsx** : Allows toggling between light and dark themes.

Reusable logic is managed through:

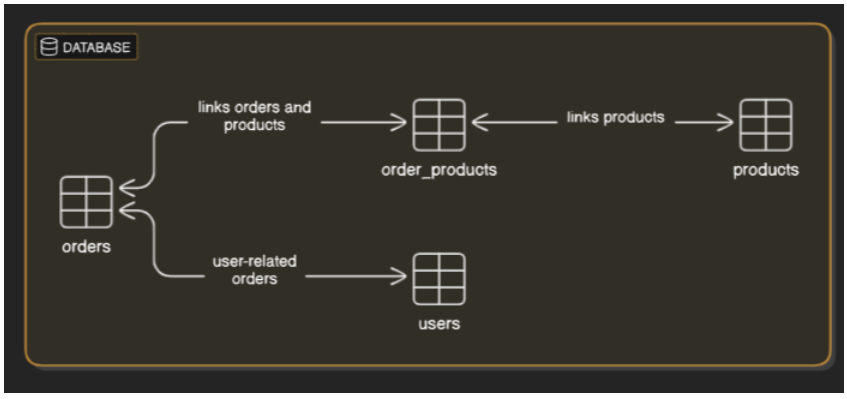
- Hooks : **useUserActions.ts** for CRUD operations, **useUserModal.ts** for modal state.
- Services : **UserService.ts** handles API calls to the backend.
- Models : **user.model.ts** defines the **User** interface for type safety.

This structure ensures a clean, maintainable, and scalable frontend.



4.3.Relationships Between Classes and Packages

1. Frontend ↔ Backend :
    - The **UserService** in the frontend communicates with the **UserController** in the backend via RESTful APIs.
    - JSON payloads are exchanged between the frontend and backend.
2. Backend Layers :
    - **UserController** delegates requests to **UserService**.
    - **UserService** interacts with **UserRepository** to perform database operations.
    - **UserRepository** uses ORM (Hibernate) to query the **users** table in the database.
3. Database :
    - The **User** entity class in the backend maps to the **users** table in the database, enabling seamless data persistence and retrieval.



Benefits of This Structure

- Modularity : Each package has a clear responsibility, making the codebase easier to understand and maintain.

- Reusability : Components like **UserTable** and services like **UserService** can be reused across different parts of the application.
- Scalability : The layered architecture allows for easy extension, such as adding new features or integrating additional modules.
- Separation of Concerns : Clear boundaries between the frontend, backend, and database ensure that changes in one layer have minimal impact on others.