Department of Computer Science
Technical University of Cluj-Napoca

# Software Design

*Assignment 1*

Name: Sofia Dobra
Group: 30435
Email:
dobrasofia219@gmail.com

2025

# Contents

# 1 Requirements Analysis

## 1.1 Assignment Specification

The assignment requires the development of a backend application with at least three entities, including a user entity supporting different user types (e.g., admin, customer). The backend must be implemented using an appropriate ORM with proper repository patterns, annotations, and relationship mappings (one-to-many, many-to-many). CRUD operations should be fully implemented for these entities, including creating, reading, updating, and deleting records, while ensuring relevant validations and handling edge cases.

A minimal frontend should also be developed to function as an admin panel, allowing CRUD operations for the user entity. This interface should display all users in a table, enable the creation of new users, and allow updates or deletions. While aesthetics are not a priority, the frontend must be functional and correctly reflect backend changes.

**Resort Management System**

The application is designed to manage a resort environment, where users can own horses and register in various activities. It includes three main entities: user, horse and activity.

## 1.2 Functional Requirements

The application has the following functional requirements:

1. The system should support three types of users: admin, student and instructor.
2. The administrator should be able to view, update and delete users.
3. The system allows registration of users as participants to activities.
4. The system should allow adding new horses, viewing horse details, updating information, and deleting horses.
5. Each horse should be assigned to its owner.
6. A user can own multiple horses, but a horse can belong to only one user.
7. The system must allow creating, viewing, updating, and deleting activities from the database.
8. Users can register for multiple activities, and in an activity can be involved several participants.

## 1.3 Non-functional Requirements

The application has the following non-functional requirements:

1. Implement validation to ensure that all inputs are valid and adhere to the specified format.
2. Use an ORM to handle database interactions.
3. Use a database to store all the data.
4. Ensure the application follows a modular architecture with separate layers for controllers, services, and repositories.

5. The system should be scalable, allowing future extensions such as additional entities or features.
6. Ensure proper error handling and logging to facilitate debugging and maintenance.
7. The backend must expose a RESTful API that the frontend can interact with.

# 2 Use-Case Model

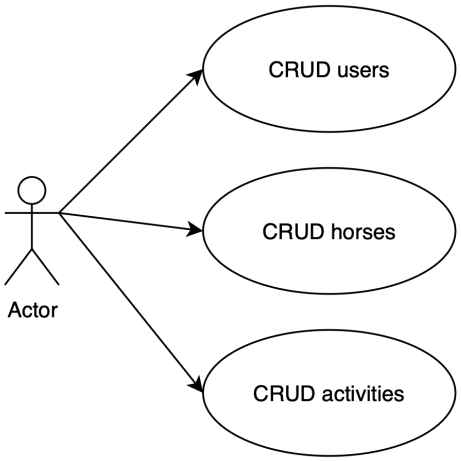


Figure 1: Use case diagram.

**Description of the *Create user* use-case**

- Use case goal: Creating a new user instance in the database and displaying it on screen as a new row of the table
- Primary actor: The user of the application
- Main success scenario: Database connection succeeds, the new user entity is created successfully, inserted in the database, and appears as a new row in the table.
- Extentions:
  - Failure: Either the database connection fails, the input from the user is invalid, or the application crashes unexpectedly. The user will receive a corresponding message in the front-end.

**Description of the *Read user* use-case**

- Use case goal: Fetching user instances from the database and displaying them on screen in a tabular form
- Primary actor: The user of the application
- Main success scenario: Database connection succeeds, data is fetched correctly and is displayed in the expected manner.
- Extentions:

- Alternate scenario of success: Database connection succeeds, data is fetched correctly, but no rows are displayed in the table because the table in the database is empty. The home screen dispays an informing message.
- Failure: Either the database connection fails, or the application crashes unexpectedly. The user will receive a corresponding message in the front-end.

Description of the ***Update user*** use-case

- Use case goal: Updating an existing user instance from the database and displaying the updated information on the screen on the corresponding user position.
- Primary actor: The user of the application
- Main success scenario: Database connection succeeds, the user instance is updated correctly and the updates are visible in the Users table.
- Extentions:
  - Failure: Either the database connection fails, the input from the user is invalid, or the application crashes unexpectedly. The user will receive a corresponding message in the front-end.

Description of the ***Delete user*** use-case

- Use case goal: Deleting an existing user instance from the database and displaying the result - the corresponding row is removed from the Users table.
- Primary actor: The user of the application
- Main success scenario: Database connection succeeds, the user instance is removed correctly, and the user instance information is no longer visible in the table.
- Extentions:
  - Failure: Either the database connection fails, the input from the user is invalid, or the application crashes unexpectedly. The user will receive a corresponding message in the front-end.

The other use cases (CRUD horses, CRUD activities) work in a similar manner, the only difference being the database entities they operate on.

# 3 System Architectural Design

## 3.1 Architectural Pattern Description

Implementation of the ***Presentation*** Tier

The **Presentation Tier** (Frontend) was built using **Vite + React + TypeScript**, to create a simple Admin Panel. It communicates with the backend via **RESTful API calls** (HTTP requests). The frontend currently displays **users** in a table (`UserTable`) and allows CRUD operations (buttons for Create, Update and Delete). Upon clicking on

the Create/Update button a pop-up window appears with the corresponding user fields (`UserModal`). When clicking on the corresponding button (Add/Update, depending on the type of action), a method in `useUserActions` is called (e.g., ***`handleAddUser`***, which calls the proper method in `UserService` (e.g., ***`addUser`***).

**Implementation of the *Logic* Tier**

The **Logic/Application Tier** (Backend) is developed using **Spring Boot** to handle business logic and API requests. It follows a **layered architecture**:

- **Controller Layer**: Handles API requests (e.g., `UserController`).
- **Service Layer**: Implements business logic (`UserService`).
- **Repository Layer**: Uses Spring Data JPA to interact with the database (`UserRepository`).

The backend implements **ORM (Object-Relational Mapping)** with **Hibernate** and supports CRUD operations for `User`, `Horse`, and `Activity`. One many-to-many relationship (`User - Activity`) and one many-to-one relationship (`Horse - User`) are handled with proper JPA annotations.
User iput is validated at this level and data consistency is ensured on two sides:

- Invalid user input: handled by annotations (e.g., `NotBlank`, `NotEmpty` etc.) and a method in the `GlobalExceptionHandler` class.
- Database related invalidations: handled explicitly in code (e.g., check explicitly before adding a user if another user in the database has the same email address). In this case also, there is a corresponding method in the handler class.

**Implementation of the *Data* Tier**

The **Data Tier (Database)** is implemented as a relational database in **MySQL**. It stores `user`, `horse`, and `activity` data persistently. The relationships between entities include:

- `user-horse`: one-to-many relationship; the `horse` table has an additional column in which the `owner_id` is stored.
- `user-activity`: many-to-many relationship; there is a join table (`user_activity`) which has a compound primary key, made from ***`activity_id`*** and ***`user_id`***.

## 3.2 Diagrams

This architecture represents a three-tier system designed for a Resort Management application, consisting of the **Presentation**, **Application/Logic**, and **Data** layers. The **Presentation** layer is responsible for handling user interactions, and it communicates with the **Application/Logic** layer via REST API calls. The **Application/Logic** layer contains three primary components: **Controller**, **Service**, and **Repository**, each responsible for managing different aspects of the business logic. The **Controller** handles incoming requests from the frontend, delegates processing to the **Service** layer, which contains the core business logic, and then

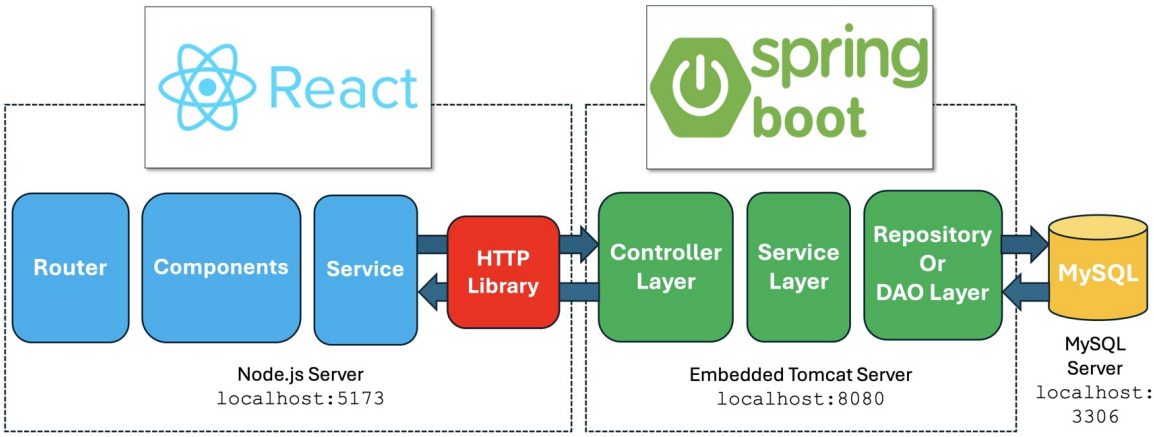


Figure 2: Three Tier architecture

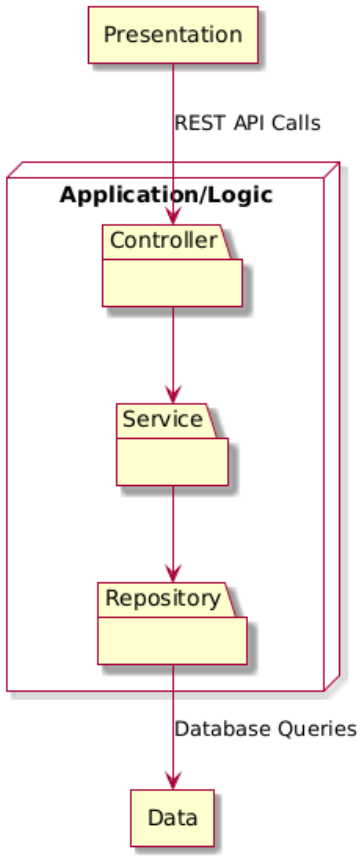


Figure 3: Layered architecture

interacts with the **Repository** for data access. The **Repository** handles database interactions, ensuring that data is retrieved or stored in the **Data** layer (the database). The flow of data starts from the **Presentation** layer, passes through the **Controller** to the **Service** and **Repository** layers, and ultimately interacts with the **Data** layer for persistent storage, ensuring a structured and efficient separation of concerns.

# 4 Class Design

## 4.1 Package + Class Diagram

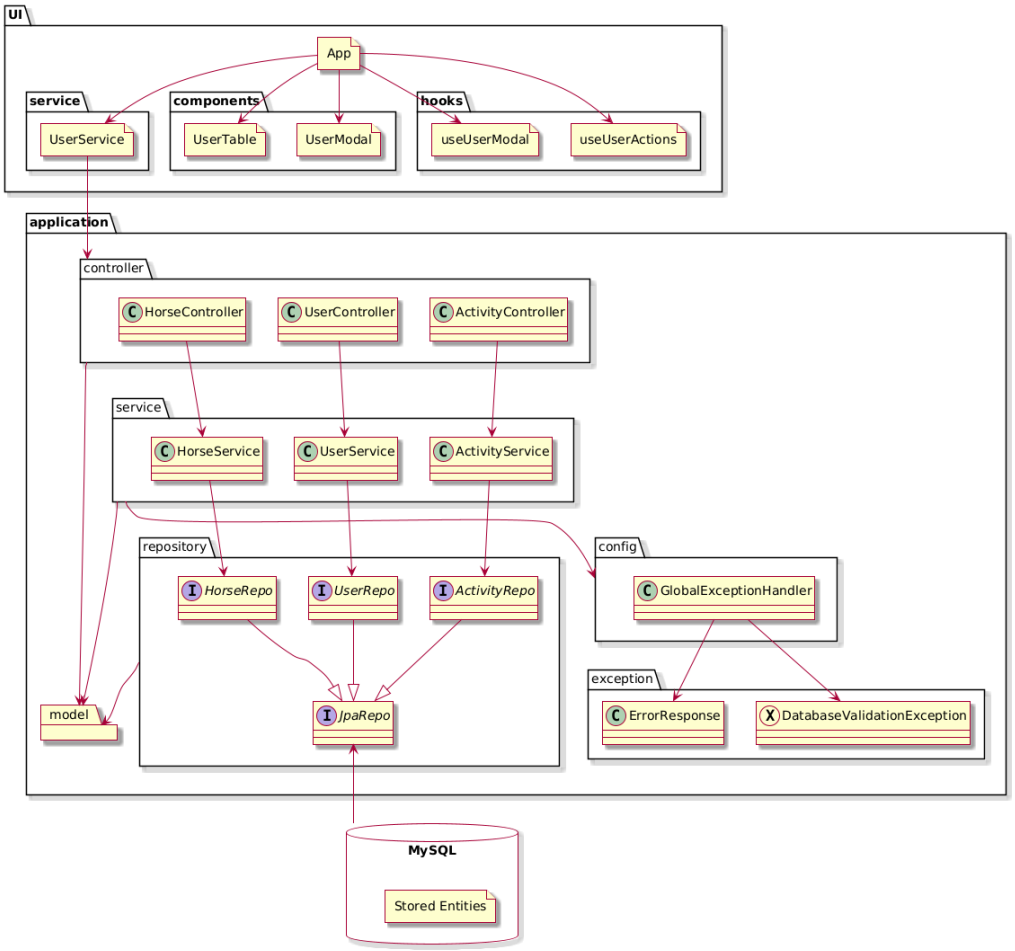


Figure 4: Package + Class Diagram

# 5 Data Model

The `user` table stores the core information about each user in the system, including unique identifiers like *`user_id`*, *`username`*, and *`email`*. This table is used to manage user data and supports relationships with other entities. Each horse in the system is linked to a user through a one-to-many relationship, where the *`user_id`* in the `horse` table serves as a foreign key referencing the `user` table.

The `activity` table contains details about various activities offered at the resort, such as the activity name, description, and date. The `user` and `activity` entities are connected through a many-to-many relationship, represented by the `user_activity` join table. This table stores combinations of `user_id` and `activity_id`, along with the participation date, establishing a link between users and activities.

These relationships are enforced through `foreign keys`, ensuring referential integrity in the database. The `user_activity` table is critical for managing the many-to-many relationship between users and activities, while the `horse` table maintains a clear one-to-many link with the `user` table. This design facilitates efficient data management and retrieval within the resort management system.

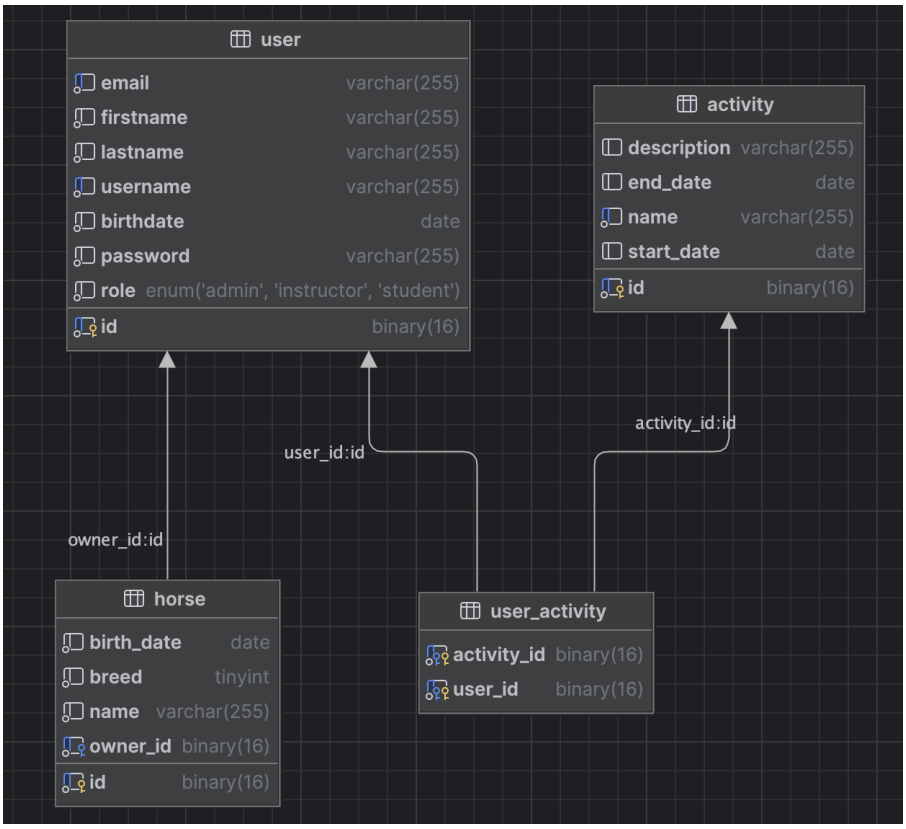


Figure 5: Database diagram