Department of Computer Science
Technical University of Cluj-Napoca

Software Design

Semester Project

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

2025

Contents

1 Requirements Analysis

1.1 First 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 Second Assignment Specification

The assignment requires the implementation of both backend and frontend applications. The backend must include unit and integration tests for classes that manipulate user data, as well as for at least three other classes that involve complex business logic. Each controller should have integration tests using a database stub instead of mocks. All classes with public methods, excluding POJOs, DTOs, and entities, should have unit tests to ensure high code coverage.

The frontend should feature multiple pages for basic CRUD operations on the core entities, with user access differentiated by role. One entity must support backend-powered searching, ordering, and filtering. A login page must be implemented to authenticate users, redirecting them to different pages based on their roles. Additionally, the application must incorporate at least one non-trivial use case involving business logic, and the design pattern used should be documented.

1.3 Third Assignment Specification

The assignment requires implementing authentication using sessions and JWTs. All backend endpoints, except login, must be protected by JWTs, with different tokens for admin and customer roles. The frontend must also enforce route-level protection, preventing unauthorized users from accessing restricted pages like /admin or /dashboard without authentication. JWTs are to be stored in the session, and the integration tests must be updated to reflect these changes. Additionally, passwords must be securely handled on the backend through hashing, ensuring they are never stored in plaintext. The frontend must be adjusted accordingly to disable direct password updates in the user entity.

A password reset feature must be introduced, allowing users to regain access via a secure verification method such as email, SMS, WhatsApp, or an authenticator app. A new “forgot password” page should collect identifiable information and a new password (entered twice), and the user must then verify the reset via a received code. Upon correct verification, the password is updated. Furthermore, the backend must be refactored using a functional programming style, applying constructs like streams, immutability, optionals, and method references to improve code clarity and maintainability.

1.4 Functional Requirements

The application has the following functional requirements:

1. The system should support three types of users: admin, student and instructor. Basic authentication should be provided for all users. Upon a successful login they will be redirected to the corresponding page.
2. The administrator should be able to view, update and delete users and activities.
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.
9. Each user can reset his/her account password.

1.5 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.
8. Security is ensured at route level by means of JWTs. These are specific to user roles (i.e. a token generated for a student can't be used for completing successfully an action in the application allowed only for the admin).
9. Passwords are hashed in the backend.

2 Use-Case Model

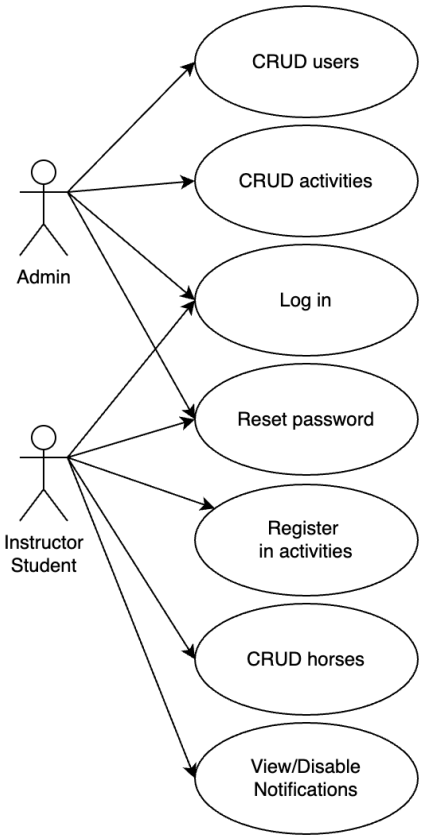

Figure 1: Use case diagram.

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 logged in admin 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 logged in admin 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 logged in admin 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 logged in admin 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.

Description of the ***Login*** use-case

- Use case goal: Authenticate the user based on the provided credentials and redirect them to the appropriate page based on their role.
- Primary actor: The user of the application
- Main success scenario: The user provides a valid username and password. The backend verifies the credentials, and the user is successfully authenticated. The user is redirected to the appropriate page according to their role (e.g., admin or regular user).

- Extensions:
  - Alternate scenario of success: The user provides valid credentials, but the user role cannot be determined due to a database issue. The user is shown a message indicating the issue.
  - Failure: The user provides incorrect credentials (wrong username or password). The frontend displays an error message informing the user that the login attempt was unsuccessful. Another failure occurs if there is a network error or the backend is unreachable; in this case, an error message is displayed on the frontend.

Description of the ***Register in Activity*** use-case

- Use case goal: Allow a logged-in instructor or student to register for a specific activity.
- Primary actor: The logged-in user (either an instructor or a student)
- Preconditions: The user must be logged in as either an instructor or a student.
- Main success scenario: The user selects an activity from a list of available activities and submits a registration request. The system verifies the user's eligibility (instructor or student) and successfully registers them for the activity, displaying a confirmation message.

Description of the ***View/Disable Notifications*** use-case

- Use case goal: Allow the logged-in user to view and disable notifications for specific activities.
- Primary actor: The logged-in user (either an instructor or a student)
- Preconditions: The user must be logged in as either an instructor or a student.
- Main success scenario: The user views a list of activities for which notifications are currently active. The user selects an activity and disables notifications for it. Afterward, the user will no longer receive notifications for that specific activity.
- Extensions:
  - Alternate scenario of success: The user disables notifications for an activity, but the system confirms the action and continues to show a list of remaining active notifications for other activities.
  - Failure: The user is not logged in as an instructor or student. The system prompts the user to log in with the correct role. Alternatively, if the system encounters an error when processing the request, the user receives a corresponding error message indicating the failure.
- Postconditions:
  - Once notifications are disabled for a specific activity, the user will no longer see notifications for that activity.

- While notifications are still active for other activities, they will continue to update based on the upcoming event time until the user either disables them or the event occurs.

Description of the ***Reset Password*** use-case

- Use case goal: Allow a user who has forgotten their password to securely reset it using a verification method (e.g., email, SMS, authenticator).
- Primary actor: The user of the application (any role: admin, instructor, or student)
- Preconditions:
  - The user has previously registered in the system (has an account).
  - The user has access to a communication method registered with their account (e.g., email, phone).
- Main success scenario:
  - The user navigates to the *Forgot Password* page.
  - The user enters a unique identifier (e.g., email or phone number) and the new password twice.
  - The system sends a verification code through the chosen method.
  - The user enters the received code.
  - The system verifies the code and resets the password.
  - The user is informed of the successful password reset and can now log in with the new credentials.
- Extensions:
  - Alternate scenario of success: The user resets their password and is immediately redirected to the login page with a success message.
  - Failure:
    * The user enters incorrect or non-matching information (e.g., wrong email or mismatched passwords). The system prompts the user to correct the input.
    * The verification code is invalid or expired. The system informs the user and offers to resend the code.
    * A system error occurs (e.g., messaging service unavailable). The user receives a corresponding error message.
- Postconditions:
  - The user's password is updated securely in the system.
  - The user can log in with the new password.
  - Any existing session tokens are invalidated to ensure security.

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`*.

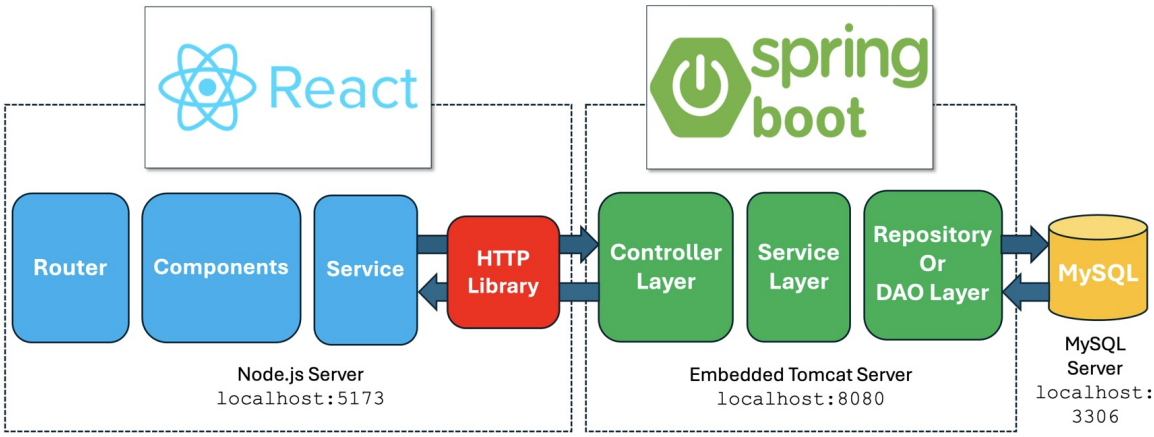

Figure 2: Three Tier architecture

3.2 Diagrams

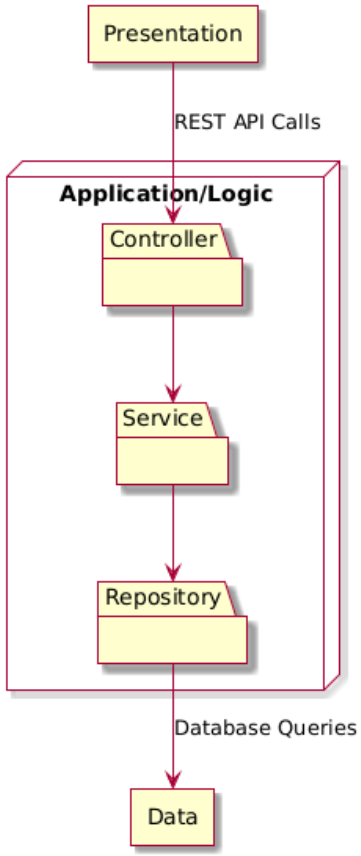

Figure 3: Layered architecture

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 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

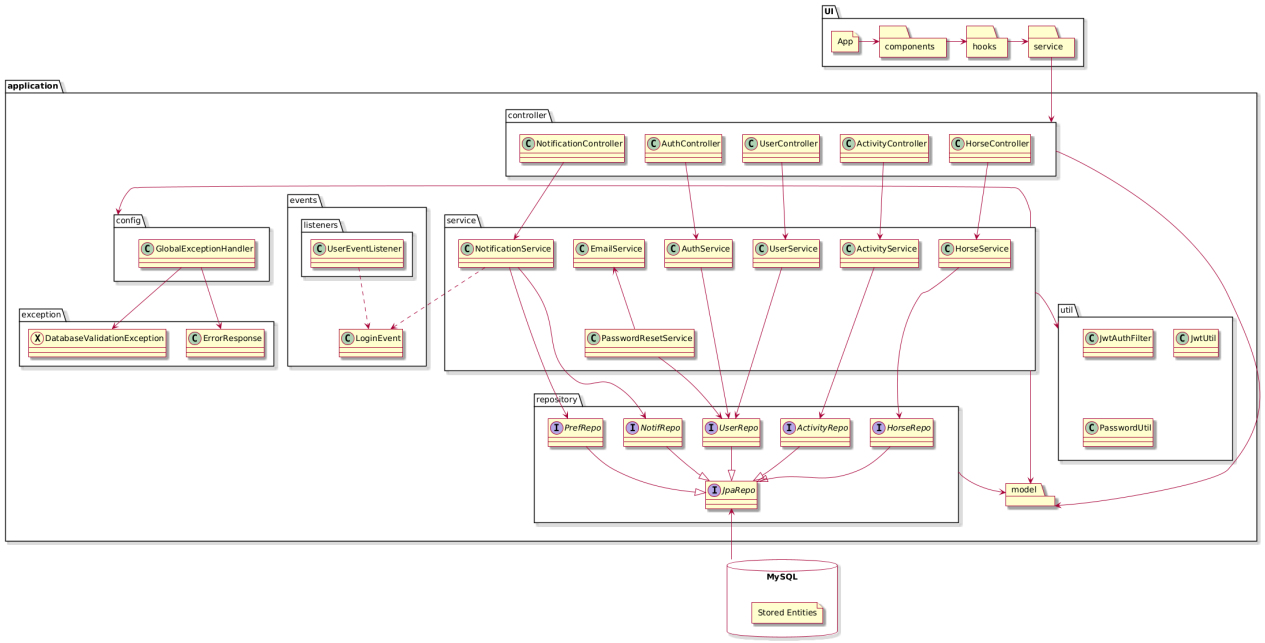

Figure 4: Package + Class Diagram

4.2 Non-trivial flow

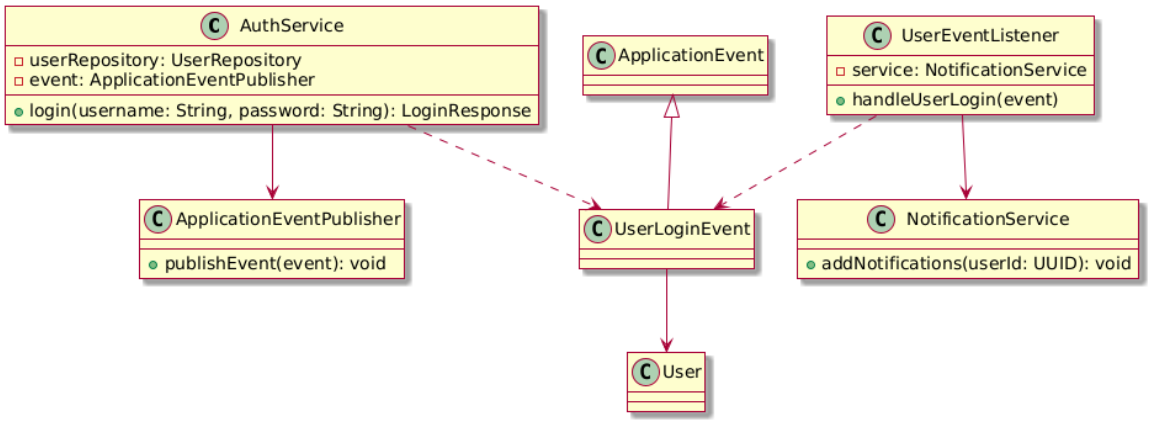

Figure 5: Class Diagram

The Observer Pattern is a behavioral design pattern that allows an object (the `Subject`) to notify a list of dependent objects (the `Observers`) about state changes, without needing to

know who or what those objects are. In the context of this application, the flow of the system involves multiple components reacting to a user login event, which demonstrates the Observer Pattern in action.

The non-trivial flow begins when a user attempts to log in through the `AuthService` component. Upon successful login, the system generates a `UserLoginEvent`, which contains the logged-in user's data. This event is published by the `AuthService`, acting as the `Subject`. The `AuthService` uses the `ApplicationEventPublisher` to publish the `UserLoginEvent`, notifying any interested observers of the event.

The primary `Observer` in this scenario is the `UserEventListener`, which listens for the `UserLoginEvent`. Upon receiving this event, the `UserEventListener` triggers the service, `NotificationService` class, to add notifications for the user. The `NotificationService` interacts with the user data to provide the necessary updates, which are stored in the system and presented to the user.

4.3 Password Reset Functionality

Frontend Implementation A dedicated page titled *Forgot Password* is accessible from the login screen. The form on this page requires users to input their registered email address. Upon submission, the system initiates the verification process.

Verification Mechanism Upon receiving the form data, the backend generates a time-sensitive verification code and sends it to the user via email. This is implemented using a messaging service (SMTP server). The verification code is temporarily stored in a secure format on the server, associated with the user's email.

Backend Logic The backend exposes a RESTful API endpoint, `/auth/request-reset`, to receive password reset requests. If the identifier is valid, the verification code is dispatched. A second endpoint, `/auth/verify-code`, accepts the code. The backend validates the code and generates a token for accessing the third endpoint (where the actual password reset takes place). The final endpoint in this flow, `/auth/reset-password`, takes the new password and then updates the user record in the database.

Security Measures Verification codes expire after a short duration (5 minutes) and can only be used once. Unlike the request and verification routes, the reset routes is protected by JWT. The token is generated if and only if the email address has been successfully verified.

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.

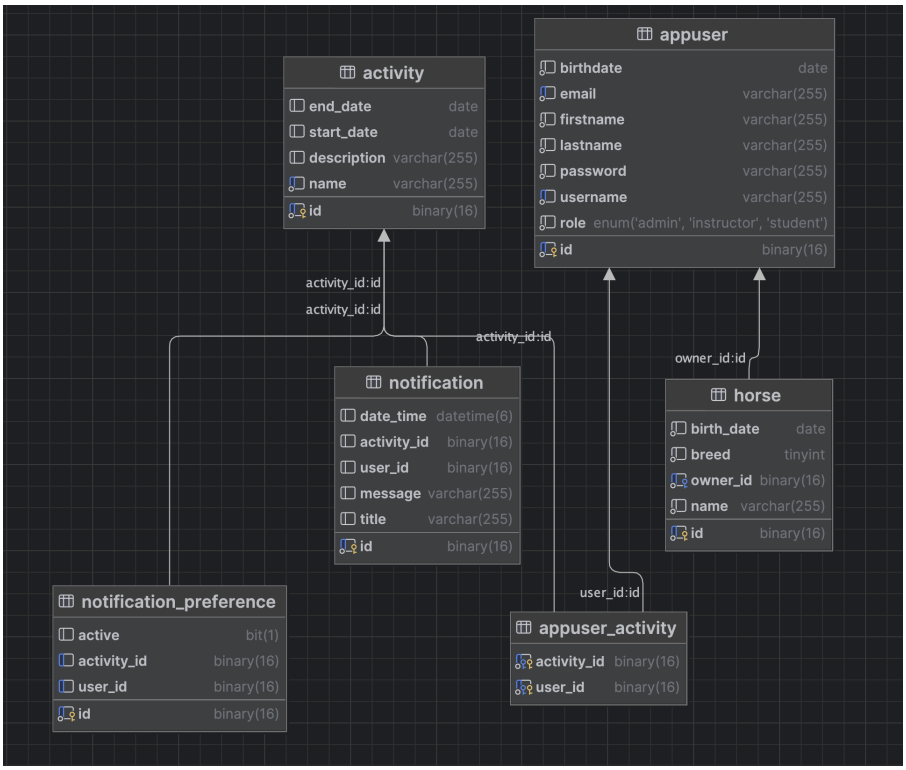

Figure 6: Database diagram

6 Changes for the project: Security bulletproofing

This section outlines the key code modifications and additions made to mitigate the most critical security vulnerabilities identified in the OWASP Top 10. Each enhancement is designed to strengthen the application's defense mechanisms and promote secure coding practices.

6.1 Broken Access Control

Broken Access Control occurs when applications do not properly enforce restrictions on what authenticated users are allowed to do. This can lead to unauthorized access to resources, such as viewing or modifying data belonging to other users, performing admin-level functions, or accessing sensitive operations.

To mitigate this threat, developers must ensure that access control checks are consistently enforced server-side, based on user roles and permissions, rather than relying solely on client-side logic or obscurity.

In our application, access control is enforced using two main strategies:

- **JWT Authentication Tokens:** Only authenticated users (i.e., users with valid JWTs) can access protected endpoints. Public routes such as `/login`, `/request-reset`, and

`/verify-code` are explicitly excluded from this requirement. (The last two deal with the password reset functionality.)

- **Role and Ownership Checks using `@PreAuthorize`:** Access to methods is restricted further using Spring Security's `@PreAuthorize` annotation to ensure users can only access allowed resources based on both property rights and role.

```
@GetMapping("horse/owner/{ownerId}")  E1Raton
@PreAuthorize("#ownerId.toString() == authentication.details")
public List<Horse> getHorsesByOwnerId(@PathVariable UUID ownerId) {
    return horseService.getHorsesByOwnerId(ownerId);
}
```

Figure 7: Broken Access Control Prevention

In this example (Fig. 7), even though the user is authenticated via JWT, the method additionally checks whether the `ownerId` in the URL matches the ID associated with the currently authenticated user (`authentication.details`). This prevents users from accessing other users' data even if they try to tamper with the URL.

6.2 Cryptographic Failures (Sensitive Data Exposure)

Cryptographic Failures, formerly known as Sensitive Data Exposure in the OWASP Top 10, refer to issues where sensitive data (such as passwords, tokens, credit card numbers, or personal information) is not properly protected in storage or in transit. This may result in attackers gaining unauthorized access to critical data due to weak or missing encryption, improper key management, or insecure transport protocols.

To prevent this threat, all sensitive data must be encrypted using strong, industry-standard algorithms (e.g., AES-256 for storage, TLS 1.2+ for transmission), and secrets such as keys or passwords should never be stored in plain text.

Our application handles sensitive data securely through:

- **Strong hashing for passwords:** Passwords are not stored in the database, in plaintext. We use algorithms like BCrypt to store passwords safely.
- **TLS (Transport Layer Security) for data in transit:** All HTTP traffic is encrypted using HTTPS (HTTP + TLS). In this way we ensure the following:
  - *Encryption:* Data sent between client and server is encrypted using TLS. This means even if someone intercepts the traffic (e.g., on public Wi-Fi), they cannot read or tamper with the data.
  - *Integrity:* TLS ensures that the data has not been altered in transit.
  - *Authentication:* HTTPS uses certificates issued by trusted authorities to prove that the server you're communicating with is the one you expect; it prevents man-in-the-middle (MITM) attacks where a malicious actor could impersonate a trusted server.
- **Encryption of critical fields:** Sensitive fields such as email addresses are encrypted before storage.

6.3 Injection

Injection flaws occur when untrusted input is sent to an interpreter as part of a command or query. This allows attackers to execute arbitrary commands or access data without proper authorization. Common types include SQL Injection, OS Command Injection, and LDAP Injection.

To prevent injection:

- Always use **parameterized queries** or **prepared statements** for database access.
- Never concatenate user input directly into queries or commands.
- Use **input validation** and **encoding** when passing data to interpreters.
- Apply the **principle of least privilege** — databases should run with restricted access rights.

SQL Injection Protection via ORM: In case of Spring Data JPA repositories (see Fig. 8) with parameters, Spring automatically uses prepared statements under the hood. This protects against SQL injection by ensuring that user input is bound safely.

```
public interface UserRepository extends JpaRepository<User, UUID> {  E1Raton
    Optional<User> findByEmail(String email);  5 usages  E1Raton

    Optional<User> findByUsername(String username);  9 usages  E1Raton
}
```

Figure 8: JPA repository

6.4 Insecure Design

Insecure Design refers to flaws in the architecture or design of an application that create security risks, regardless of how securely the code is implemented. Unlike vulnerabilities introduced by coding mistakes, insecure design results from failing to consider security early in the software development lifecycle (SDLC).

The backend application is organized into well-defined packages that follow a layered architecture, promoting both maintainability and security by design. This separation of concerns ensures that logic is properly encapsulated and that security rules can be consistently enforced at different levels of the application.

- `controller` → `service` → `repository` workflow: This layered workflow enforces separation of concerns. Controllers handle HTTP requests and perform input validation; services encapsulate business logic and enforce security rules like access control or ownership verification; repositories interact with the database using safe, parameterized queries (via JPA or Spring Data). This structure minimizes the risk of logic duplication or security oversight across layers.
- `model` – Contains domain entities (e.g., `User`, `Horse`), enabling centralized validation and mapping between objects and database tables. In addition, this package includes Data Transfer Objects (DTOs) such as wrappers for entities, as well as classes modeling

incoming requests and outgoing responses. Using DTOs prevents overexposing internal entity structures and allows validation or transformation of data before it reaches sensitive layers.

- `config` – Includes security-related and global configurations like:
  - `SecurityConfig` – Defines authentication and authorization rules, including filter chains, endpoint protection, CORS settings, and JWT integration.
  - `GlobalExceptionHandler` – Provides centralized error handling, reducing information leakage and ensuring consistent responses to API consumers.
  - `AuditConfig` – Configures audit logging using aspects or listeners to track sensitive actions such as logins, data modifications, or privilege changes, improving traceability and accountability.
- `aspect` – Implements cross-cutting concerns such as auditing (`AuditAspect`), allowing transparent logging of security-relevant actions without cluttering business logic. This aligns with the principle of separation of concerns and supports compliance requirements (e.g., audit trails).
- `events` – Supports an event-driven design (e.g., `UserLoginEvent`), which decouples logic and makes the system more secure and scalable by isolating side-effects (e.g., logging, notifications) from core functionality. Listeners respond to published events asynchronously or post-process without interrupting primary flows.
- `util` – Includes utilities for cryptographic operations that help secure sensitive data at rest and in transit. Also includes security filters like `JwtAuthFilter`, which verifies JWT tokens and ensures that only authenticated requests reach protected resources.
- `annotation` – Custom annotations, likely used in conjunction with AOP (aspect-oriented programming), providing reusable and declarative security-related functionality (e.g., tagging methods for auditing or authorization checks).
- `exception` – Centralized definition of application-specific exceptions, allowing fine-grained error control, better debugging, and the prevention of information leakage (e.g., avoiding stack traces or raw database messages in API responses).

6.5 Security Misconfiguration

Security misconfiguration is ranked as the fifth most critical security risk in the OWASP Top 10 and refers to insecure default configurations, incomplete or ad-hoc configurations, and unnecessary enabled features. In Spring Boot applications, this often manifests as exposed management endpoints, default credentials, excessive error details, or overly permissive CORS policies.

This Spring Boot application adheres to layered architecture and incorporates multiple measures to prevent security misconfiguration, including:

- **Actuator endpoints disabled or excluded:** Although Spring Boot's Actuator is commonly used for application monitoring, this application does not expose or depend on Actuator endpoints in production. Therefore, the actuator dependency is unused, eliminating potential information leakage (e.g., via `/actuator/env` or `/actuator/mappings`).

- **Error detail suppression:** Default error responses are configured to exclude sensitive debugging information such as exception messages and stack traces. This is done to avoid leaking internal logic to users or attackers.

```
server.error.include-message=never
server.error.include-stacktrace=never
```

- **Centralized CORS configuration:** All controllers previously used `@CrossOrigin`, which permitted requests from any origin. This was replaced by a centralized CORS policy in `SecurityConfig.java`, restricting access to trusted origins only:

```
config.setAllowedOrigins(List.of("https://localhost:3000"));
config.setAllowedMethods(List.of("GET", "POST", "PUT", ...));
config.setAllowedHeaders(List.of("*"));
config.setAllowCredentials(true);
```

  This ensures that only the designated frontend can make cross-origin requests and prevents unauthorized JavaScript from external origins from interacting with the backend.

- **No default or hardcoded credentials:** Credentials and secrets are managed securely using environment variables rather than being hardcoded into source files or properties. For example, JWT and encryption keys are read from external configurations:

```
@Value("${JWT_SECRET}")
private String secretKey;
```

- **HTTP security headers enforced:** The application configures essential HTTP security headers to mitigate common web vulnerabilities. These are set in the Spring Security filter chain and include:
  - `Strict-Transport-Security` – Forces use of HTTPS and protects against protocol downgrade attacks.
  - `X-Content-Type-Options: nosniff` – Prevents MIME-type sniffing, reducing the risk of drive-by downloads.
  - `X-Frame-Options: DENY` – Protects against clickjacking by disallowing the site to be embedded in an iframe.
  - `X-XSS-Protection` – Enables basic cross-site scripting (XSS) filtering in browsers.
  - `Content-Security-Policy (CSP)` – Restricts the sources from which content (scripts, styles, etc.) can be loaded, helping to prevent XSS and data injection attacks.

  These headers are configured programmatically in `SecurityConfig.java` using the `headers()` DSL, ensuring that all HTTP responses are consistently secured.

```
.headers( HeadersConfigurer<HttpSecurity> headers -> headers
        .contentSecurityPolicy( ContentSecurityPolicyConfig csp -> csp
                .policyDirectives("default-src 'self'; script-src 'self'; object-src 'none';")
        )
        .frameOptions(HeadersConfigurer.FrameOptionsConfig::deny
        )
        .httpStrictTransportSecurity( HstsConfig hsts -> hsts
                .includeSubDomains(true)
                .maxAgeInSeconds(31536000)
        )
        .referrerPolicy( ReferrerPolicyConfig referrer -> referrer
                .policy(org.springframework.security.web.header.writers.ReferrerPolicyHeaderWriter.ReferrerPolicy.STRICT_ORIGIN_WHEN_CROSS_ORIGIN)
        )
)
```

Figure 9: Supplementary checks in `SecurityConfig`

6.6 Vulnerable and Outdated Components

Vulnerable and outdated components refer to third-party libraries, frameworks, or dependencies included in a software project that contain known security vulnerabilities or are no longer maintained. Attackers can exploit these weaknesses to gain unauthorized access, perform denial-of-service attacks, or compromise system integrity. Keeping dependencies up to date is a critical security practice.

The following steps were performed to address this issue:

1. **Run Security Check:** Execute the Maven command `mvn verify` to trigger OWASP Dependency-Check. This command performs compilation, testing, and security scanning.
2. **Review Report:** After execution, navigate to `target/dependency-check-report.html`. Open the report in a browser. It lists all dependencies with known CVEs (Common Vulnerabilities and Exposures), along with severity ratings, evidence count, and confidence level.
3. **Identify Critical Components:** Pay attention to entries marked with `CRITICAL` or `HIGH` severity. In our project, the following dependencies were flagged:
   - `spring-context:6.2.3` – *LOW severity*
   - `tomcat-embed-core:10.1.36` – *CRITICAL severity*
   - `spring-security-crypto:6.4.4` – *MEDIUM severity*
4. **Update vulnerable libraries:** If possible and really necessary, the problematic dependencies were updated accordingly. Finally, in this project only the vulnerability related to the Spring context remained (which had *LOW* severity).

6.7 Identification and Authentication Failures

Identification and authentication failures occur when an application incorrectly implements mechanisms that validate user identities or fails to properly protect authentication credentials. This may result in unauthorized access, session hijacking, or impersonation. Common causes include insecure password storage, improper session management, and failure to validate JWTs (JSON Web Tokens).

To mitigate this threat, several security measures have been implemented in both the backend and frontend of the application:

- **Secure Password Handling:** Passwords are hashed using `BCrypt` via a utility class `PasswordUtil`, ensuring that plaintext passwords are never stored in the database.
- **Token-Based Authentication:** Upon successful login, a JWT token is issued using a secure secret key. This token is signed and contains essential claims such as issuer, issued-at time, expiration time, user ID, and role.
- **Token Expiration:** Each token is issued with an expiration time. Tokens are considered invalid after this time and are rejected by the server and the authentication filter.
- **Database Token Validation:** In addition to verifying the JWT signature and claims, the backend also checks whether the token exists in the `auth_token` table (see Fig. 10). If it does not, the request is treated as unauthorized. This ensures tokens are invalidated upon logout.
- **JWT Filtering and Claim Validation:** A custom `JwtAuthFilter` is implemented, which intercepts requests, verifies the token signature and issuer, checks expiration and issued-at timestamps, and validates required claims (such as `userId` and `role`).
- **Frontend Session Handling:** The frontend stores the token securely in `sessionStorage` and attaches it to all API requests. If a token is expired or invalid, the backend responds with HTTP `401 Unauthorized`, which is intercepted globally.
- **Global Interceptor for Expired Sessions:** All requests from the frontend go through a custom `fetchWithAuth` interceptor. If an unauthorized response is detected, the user is notified with a custom modal saying *Session has expired. Please log in again.* Upon confirmation, the token is cleared and the user is redirected to the login page.
- **Logout and Token Revocation:** When the user logs out, their token is deleted from the `auth_token` table, ensuring it cannot be reused.

These mechanisms collectively ensure robust authentication and prevent unauthorized access by validating both the identity and the session state of users across the application.

```
AuthToken authToken = authTokenRepository.findByToken(token);
if (authToken == null) {
    log.error("Token not found in database (probably logged out)");
    response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
    return;
}

if (authToken.getExpiryDate().isBefore(LocalDateTime.now())) {
    log.error("Token in database has expired");
    response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
    return;
}
```

Figure 10: Token validation in the back-end

6.8 Software and Data Integrity failures

Software and data integrity failures refer to vulnerabilities that arise when an application does not adequately verify the trustworthiness of its software components, data sources, or update mechanisms. These failures can lead to unauthorized data manipulation, execution of untrusted code, or exploitation of flawed dependencies, thereby compromising the security and reliability of the system.

In the **Horseland** project, several strategies were implemented to minimize such risks and ensure the integrity of both software and data throughout the stack.

1. **Backend Measures (Spring Boot):**
   - **Input Validation:** The backend uses Spring's `javax.validation` API to enforce constraints on input data at the entity and controller levels, reducing the risk of malformed or malicious data being processed.
   - **ORM Data Handling:** JPA/Hibernate is used with parameterized queries to prevent injection attacks and maintain consistency in data transactions.
   - **Transactional Safety:** The use of `@Transactional` annotations ensures atomic operations, preventing partial updates or inconsistent states in the event of failures.
   - **Authentication Integrity:** JWT-based authentication prevents session tampering and ensures that all requests are verified and originate from authenticated users.

2. **Frontend Measures (React + TypeScript):**
   - **Type Safety:** TypeScript enforces static typing, reducing common programming errors that could lead to corrupt logic or data mishandling.
   - **Axios Interceptors:** Secure HTTP communication is managed using Axios interceptors that automatically attach JWT tokens and handle expired session scenarios securely.
   - **Client-side Validation:** Form data is validated client-side before submission to reduce erroneous or malformed input reaching the backend.

3. **Dependency and Configuration Integrity:**
   - **Maven Dependency Management:** All third-party libraries are declared with fixed versions in `pom.xml`, and regularly updated to patch known vulnerabilities.
   - **Environment Configuration:** Sensitive credentials and environment configurations are separated from source code using Spring Boot profiles and environment variables.

These combined approaches significantly reduce the surface area for software and data integrity failures, ensuring that the Horseland system maintains a secure, consistent, and trustworthy operational state across both frontend and backend components.

6.9 Security Logging and Monitoring Failures

Security Logging and Monitoring Failures occur when an application lacks adequate logging of security-relevant events, or when it fails to integrate with monitoring systems. This can result in undetected security breaches, unauthorized access, or difficulties during forensic analysis.

In application, mitigation of this risk was achieved on two sides:

1. Audit Logging

An `AuditLog` entity (see Fig. 11) records critical operations and user actions, helping to track changes over time. It stores the username of the user responsible for the action (if available), a brief description of the operation (e.g., *CREATE*, *UPDATE*, *LOG IN* etc.), the timestamp (formatted as `yyyy/mm/ddThh:mm:ss`), and the entity that was affected or on which the action was performed.

```java
@Entity  E1Raton
@Data
@Table(name = "audit_log")
public class AuditLog {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private UUID id;

    private String username;
    private String operation;
    private LocalDateTime timestamp;
    private String entity;
}
```

Figure 11: The `AuditLog` class

These logs are displayed in the front-end (see Fig. 12) only to administrator users. This enables traceability of user actions, enhances accountability, and facilitates post-incident investigation.

2. Structured Logging with `@Slf4j`

By using the Lombok `@Slf4j` annotation, structured and performant logging is enabled across various classes/components. These include: `GlobalExceptionHandler`, `JwtAuthFilter` etc.

6.10 Server-Side Request Forgery (SSRF)

Server-Side Request Forgery (SSRF) is a type of vulnerability where an attacker tricks the server into making unintended requests to internal or external systems. This can result in unauthorized access to internal services, cloud metadata, or protected resources not intended to be publicly exposed.

In the context of the **Horseland** project, the application architecture was deliberately designed to eliminate the risk of SSRF:

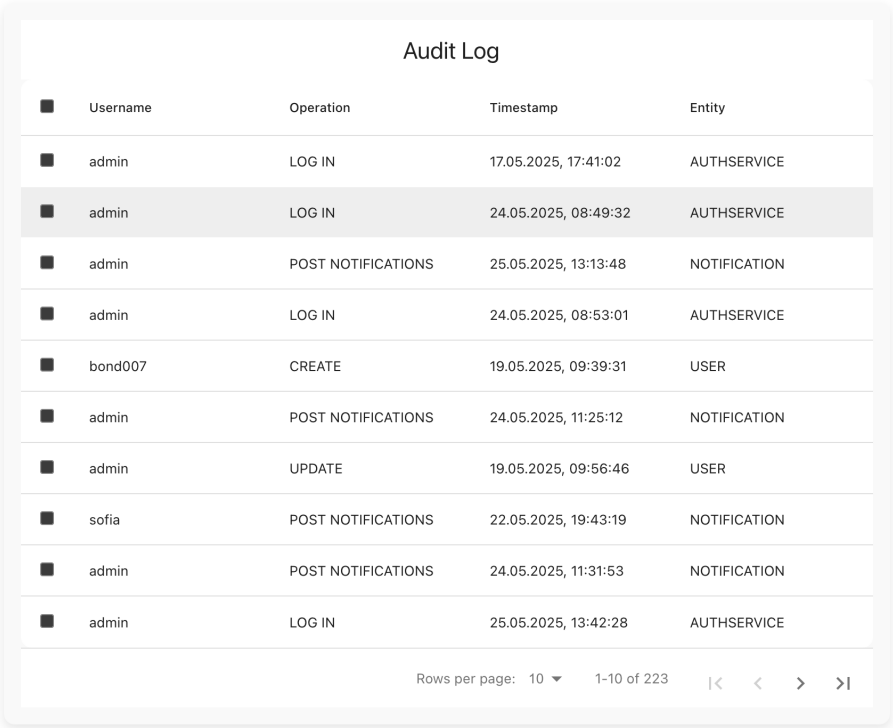

Figure 12: The Audit View in the front-end

- **No External API Calls:** The backend system, built with Spring Boot, does not initiate any outbound HTTP requests to external APIs or services. All functionality is internal to the system, based on direct database access and local service logic.
- **Strict Controller Routing:** All REST endpoints are tightly defined and bound to specific authenticated and authorized actions. The use of Spring Security's `@PreAuthorize` annotations ensures that user-controlled input cannot dynamically influence internal routing or trigger hidden endpoints.
- **No URL Fetching Based on User Input:** The application does not accept or process user-provided URLs. Thus, attackers cannot supply crafted URLs that would trick the server into initiating requests to internal resources.
- **Safe Use of HTTP Clients:** Although the project includes libraries like Axios on the frontend and may use HTTP clients internally (in future extensions), they are currently not employed to call external services, removing the typical vectors used in SSRF exploitation.

Through this careful design and strict avoidance of dynamic outbound network requests, the Horseland system effectively mitigates SSRF risks by architecture, not just configuration. Should future enhancements introduce integrations with third-party APIs, additional safeguards such as allow-lists and request sanitization would be applied.