instituto
superior de
engenharia
de lisboa

Department of Electronical Engineering, Telecommunications and Computers

Remote Lab

50565: Ângelo Filipe Maia Azevedo (a50565@alunos.isel.pt)
50539: António Miguel Alves (a50539@alunos.isel.pt)

Report for Project and Seminar Class
of Computer Science and Computer Engineering BSc

Advisor: Prof. Pedro Miguens Matutino

June 2025

Lisbon School of Engineering

Remote Lab

50565 Ângelo Filipe Maia Azevedo

____

50539 António Miguel Alves

____

Advisor: Prof. Pedro Miguens Matutino

____

Report for Project and Seminar Class of Computer Science and Computer Engineering
BSc

June 2025

Abstract

The design, development, implementation, and validation of digital systems require, in addition to simulators, the use of hardware for verification of their implementation in real devices. However, access to these real devices is sometimes restricted, not being available 24h/7. In the current teaching paradigm where face-to-face time is reduced and remote and autonomous work is increased, it is necessary to create alternatives to the usual model.

The Remote Lab project aims to provide an online laboratory with access to remote hardware. This remote workbench consists of a web application running on an embedded system. The web application, accessed through a website, aims to provide a dashboard where users can join a laboratory. This is where users can control the remote hardware. A hierarchy system will be implemented to provide different roles, each with their own permissions relative to how users can browse the information provided by the web application.

This project will implement the infrastructure to support the configuration, manipulation and visualization of remote hardware. Based on an architecture with back-end (database and Web API) and front-end (Web App, with a dashboard).

Resumo

A conceção, desenvolvimento, implementação, e por fim a validação de sistemas digitais requerem para além dos simuladores, a utilização de hardware para uma verificação da sua concretização em dispositivos reais. No entanto, o acesso a esses dispositivos reais é por vezes restrito, não estando acessíveis 24h/7. No atual paradigma de ensino em que se reduz o tempo de contacto presencial, aumentando-se o trabalho remoto e autónomo, é necessário criar alternativas ao modelo habitual.

O projeto Remote Lab tem como objetivo fornecer um laboratório online com acesso a hardware remoto. Este laboratório consiste numa aplicação web executada num sistema embebido. A aplicação web, acedida através de um website, visa fornecer um painel de controlo onde os utilizadores podem aderir a um laboratório. Os utilizadores podem controlar o hardware remoto. Será implementado um sistema hierárquico para fornecer diferentes funções, cada uma com as suas próprias permissões relativamente à forma como os utilizadores podem navegar pela informação fornecida pela aplicação web.

Este projeto implementará a infraestrutura de suporte à configuração, manipulação e visualização de hardware remoto. Baseado numa arquitetura com back-end (base de dados e Web API) e front-end (Web App, com um dashboard).

Contents

List of Figures

Listings

Acronyms

API Application Programming Interface

BSc Bachelor of Science

CI/CD Continuous Integration and Continuous Deployment

CRUD Create; Read; Update; Delete

ER Model Entity-Relationship Model

FPGA Field-Programmable Gate Array

HTTP HyperText Transfer Protocol

HTTPS HyperText Transfer Protocol Secure

IP Internet Protocol address

ISA iLab Shared Architecture

JDBC Java Database Connectivity

JDBI Java Database Interface

JSON JavaScript Object Notation

MAC Media Access Control address

MIT Massachusetts Institute of Technology

OAuth Open Authorization

OOP Object-Oriented Programming

RBAC Role-Based Access Control

REST Representational State Transfer

URI Uniform Resource Identifier

Chapter 1

Introduction

In recent years, the need for remote access to laboratory resources has grown significantly, driven by the expansion of online education, increased research collaboration, and the growing complexity of experimental setups. Traditional laboratories often require physical presence, which can limit accessibility and flexibility for students, researchers, and professionals. This limitation has become particularly evident in situations where geographical constraints, time restrictions, or extraordinary circumstances (such as global pandemics) prevent direct access to laboratory facilities.

The project described in this report addresses these challenges by developing a comprehensive solution for remote laboratory access that maintains the quality and integrity of hands-on experimentation while providing the flexibility of remote operation.

1.1 Objectives

Considering these emerging needs, a platform was proposed and implemented to enable secure, efficient, and user-friendly remote access to laboratory equipment and resources. The design of the platform was divided into two distinct phases. In the first phase, a database, an API, and a web application were designed and implemented. This phase also encompassed the deployment architecture of the platform, including containerization, orchestration, and other essential configurations. In the second phase, the communication protocols between the platform and laboratory hardware were designed and implemented.

To design and implement a scalable platform for remote laboratory access, the following main objectives were established:

- Web API to ensure comprehensive user, laboratory, and hardware management;
- Web application that provides an intuitive and user-friendly interface;
- Secure authentication and authorization mechanisms;
- Role-based access control (Role-Based Access Control (RBAC));

- Robust data persistence through a well-designed database system;
- Remote manipulation capabilities via terminal access to laboratory devices.

Additionally, several optional objectives were identified to enhance the system's functionality:

- Laboratory scheduling and reservation system;
- Real-time visual monitoring of laboratory hardware;

1.2 State of the Art

Numerous initiatives have emerged to provide remote access to laboratory resources, particularly in higher education and research contexts. Pioneering projects such as MIT's iLab [1] and LabShare [2] have demonstrated both the feasibility and substantial benefits of remote laboratories, enabling students and researchers to conduct experiments from anywhere in the world. These platforms typically emphasize secure access protocols, intelligent scheduling systems, and seamless integration with diverse laboratory equipment.

The existing literature underscores the critical importance of usability, scalability, and security in the design of remote laboratory systems. Key challenges identified include ensuring real-time interaction capabilities, maintaining hardware integration reliability, and providing adequate user support and training.

Several systems currently offer functionalities similar to those proposed in the Remote Lab project. The ISA [1], originally developed by MIT, represented an early successful implementation but is no longer operational and unavailable for public access. Similarly, LabShare [2], another notable platform, is currently inactive. Contemporary systems such as WebLab-Deusto [3] focus on specialized domains like electronics and instrumentation, providing tailored interfaces and tools for remote experimentation in specific fields.

Other significant contributions to the field include the Remote Laboratory Management System (RLMS) [4] and the Labshare Sahara framework [5], which have influenced modern approaches to remote laboratory architecture and user experience design. These systems serve as valuable references for the development of the Remote Lab platform, informing critical decisions related to system architecture, user experience optimization, and hardware integration strategies.

1.3 Document Structure

This report is organized as follows: Chapter 2 presents and describes the proposed system architecture, including detailed specifications and core functionalities. Building upon the understanding of system components, the database design, web API implementation, and web application development are described in Chapters 3, 4, and 5, respectively. Chapter 6 details the project organization methodology and deployment strategies employed. Chapter 7 presents comprehensive testing procedures and validation results for the implemented system. Finally, Chapter 8 provides conclusions regarding the system's performance and outlines potential directions for future development and enhancement.

Chapter 2

Proposed Architecture

With the objective of implementing a platform to provide remote laboratory access, Figure 2.1 presents a high-level architecture of the proposed system. A user can remotely access the platform through a web interface. The server hosting the platform communicates with an external authentication service to verify user credentials and establish secure sessions. Once authenticated, users can remotely manipulate laboratory hardware through the platform's interface.

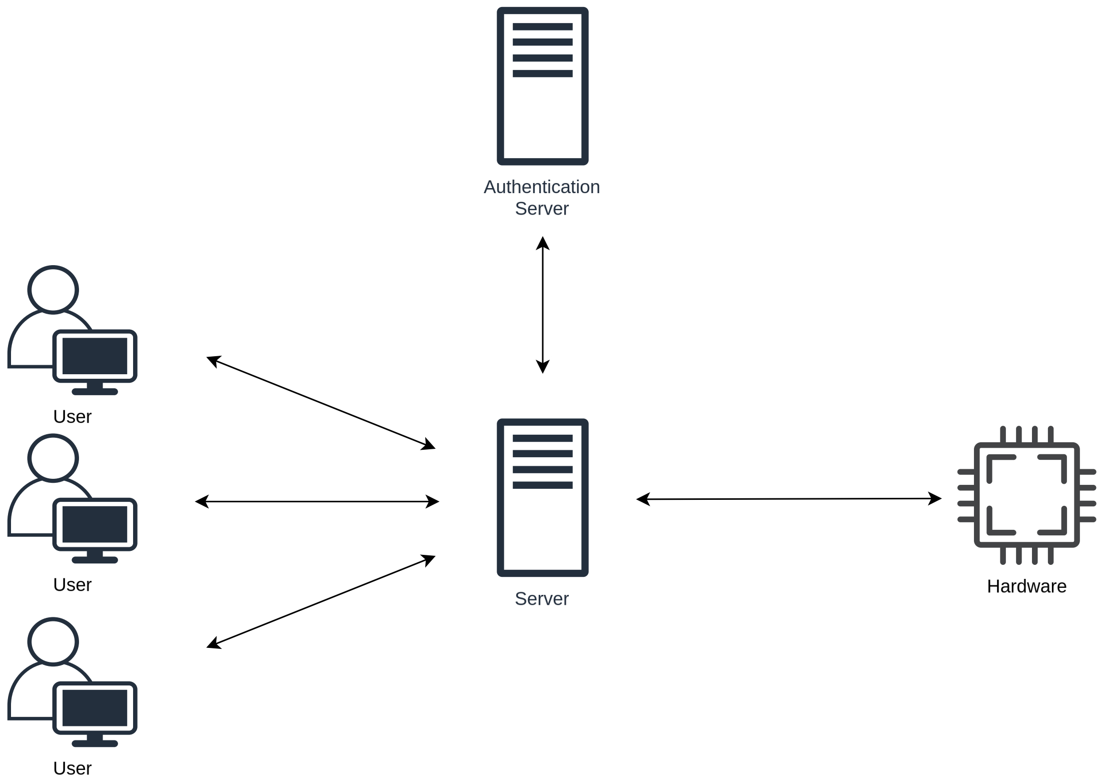


Figure 2.1: High-level Architecture

This chapter is structured to provide a comprehensive understanding of the system's design. Section 2.1 describes the different types of users and their respective interactions with the system. Section 2.2 introduces the core system functionalities and their implementation approach. Finally, Section 2.3 presents the detailed system architecture with an introduction to its individual components and the technologies employed.

2.1 System Users

To ensure proper access control and functionality separation, the system defines three distinct user roles, each with specific permissions and capabilities:

Student can access assigned laboratories and manipulate their associated hardware within defined parameters and time constraints.

Professor can create and configure laboratories, assign hardware resources, create and manage student groups, and monitor laboratory usage and performance.

Administrator can manage system users, configure system-wide settings, monitor platform performance, and handle administrative aspects such as user provisioning and system maintenance.

This hierarchical separation of user types provides reliable access control throughout the system, determining whether a user can perform specific actions based on their assigned role. This approach is further enhanced through the implementation of a hierarchy and a RBAC approach, which provides fine-grained control over user permissions and system resources.

2.2 System Functionalities

The system is built around several core functionalities that work together to provide a comprehensive remote laboratory experience. The platform supports robust user authentication and authorization mechanisms, ensuring secure access to laboratory resources. These security features are integrated throughout a Web API, which exposes well-defined endpoints that provide access to resources, implement resource management capabilities, and enforce business logic rules.

The Web API follows Representational State Transfer (REST) principles and exposes resources through standardized Uniform Resource Identifiers (URIs). It maintains bidirectional communication with a purpose-built database designed to meet the system's requirements for storing and managing all necessary information, including user data, laboratory configurations, and session logs.

A critical component of the system is the hardware abstraction layer, which communicates with the Web API to translate high-level user instructions into hardware-specific commands. This abstraction enables the platform to support diverse types of laboratory equipment while maintaining a consistent user interface.

The Web Application serves as the primary user interface, making HyperText Transfer Protocol (HTTP) requests to the Web API to provide users with comprehensive functionality, including:

- Secure user authentication and profile management;
- Laboratory discovery, access, and real-time interaction capabilities;
- Group creation and collaborative workspace management;
- Hardware configuration and monitoring interfaces;
- Comprehensive user and system administration tools;

2.3 System Components

The proposed system comprises four main components: a Web Application, Web API, Database, and Hardware Abstraction layer. Figure 2.2 illustrates the detailed architecture of the proposed system along with the specific technologies considered during the design phase:

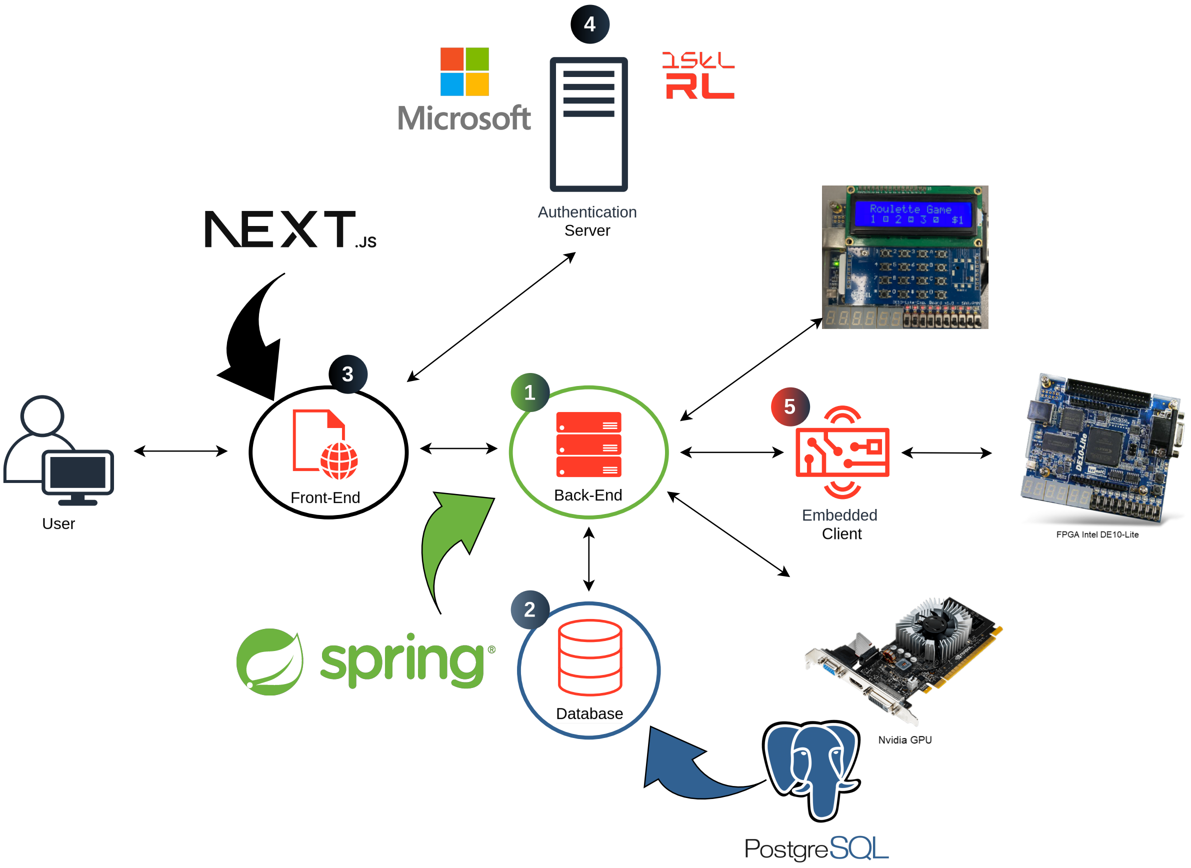


Figure 2.2: Detailed System Architecture

The server entity shown in Figure 2.2 is expanded in Figure 2.2 to reveal its internal components: the Back-End (1), Database (2), and Front-End (3). The architecture also illustrates the external Authentication Server (4) and the Embedded Client (5), demonstrating the system's integration capabilities.

Back-End Architecture

The Back-End (1) encompasses both the Web API and the Hardware Abstraction layer. The Web API utilizes HTTP as its primary communication protocol and is built using Spring Framework [6] as the main technology stack, with Kotlin [7] as the primary programming language. Chapter 4 provides comprehensive details about the Web API, including the rationale for choosing Spring [8] and Kotlin [7], architectural decisions, and implementation specifics.

Database Design

The Database (2) is designed to efficiently store and manage all system information, including user profiles, laboratory configurations, hardware specifications, and operational logs. PostgreSQL [9] was selected as the database management system. Chapter 3 describes the database entity-relationship model, explains the rationale behind choosing PostgreSQL [9], and provides detailed implementation information.

Front-End Implementation

The Front-End (3) is developed using Next.js [10], a modern React-based framework [11] that provides server-side rendering capabilities and excellent performance optimization features. It serves as the primary visual interface for users and handles all client-server communication by making HTTP requests to the Back-End (1) for data retrieval and manipulation. Chapter 5 provides an in-depth analysis of the Web Application design and implementation.

Authentication System

User authentication is implemented through the Web Application using NextAuth.js [12], a comprehensive authentication framework that facilitates secure communication with external authentication providers. The Authentication Server (4) is provided by Microsoft Open Authorization (OAuth) [13], enabling users to authenticate using their institutional credentials.

Hardware Integration

The Embedded Client (5) is illustrated to demonstrate the system's hardware communication capabilities. In the current implementation, the Embedded Client communicates with Field-Programmable Gate Array (FPGA), showcasing the platform's ability to interface with programmable hardware devices. The hardware abstraction layer within the Back-End (1) provides different instruction sets and interaction protocols depending on the specific hardware associated with each laboratory, ensuring compatibility with diverse equipment types while maintaining a consistent user experience.

Chapter 3

Database

The database serves as the foundational component of the system architecture. PostgreSQL was selected as the database management system due to its open-source nature and robust support for relational data models. This choice aligns with previous project implementations and provides the consistency and performance required for the system's operational needs.

This chapter presents an overview of the Entity-Relationship Model (ER Model) and critical implementation details. Complete technical documentation is provided in the accompanying appendix.

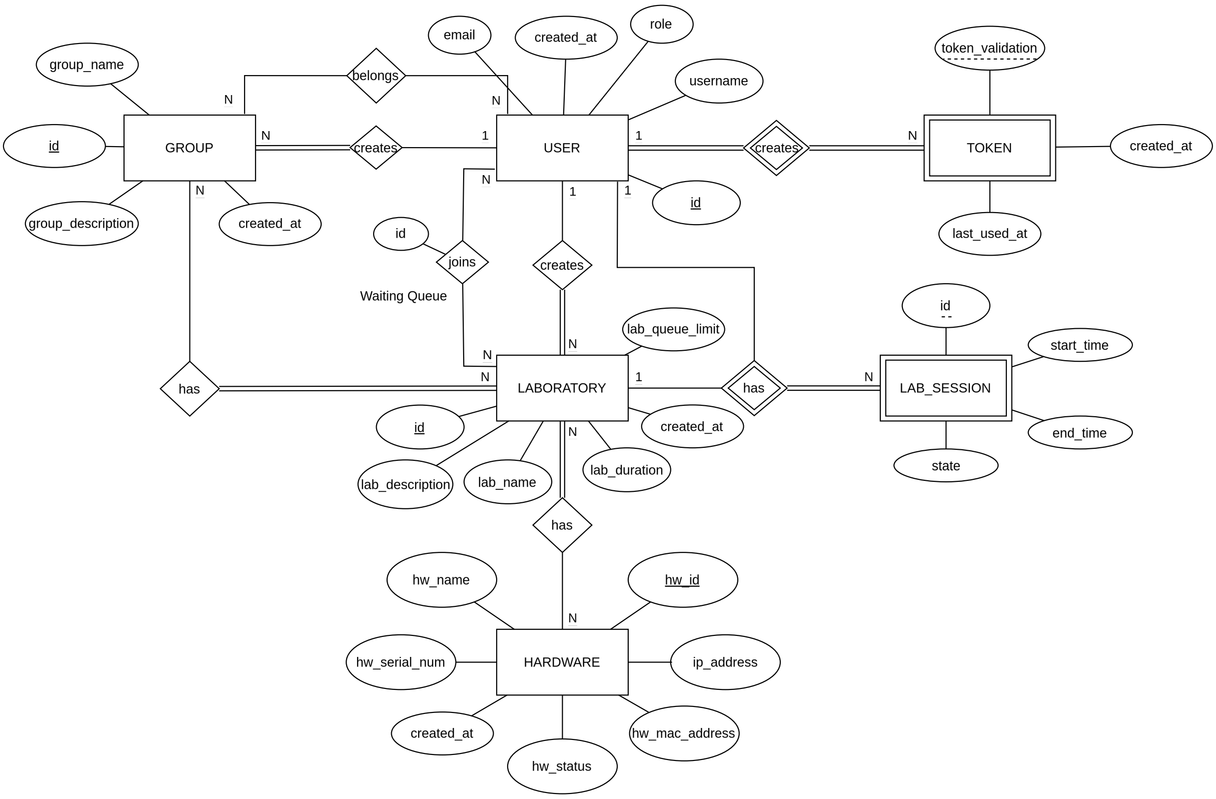


Figure 3.1: Entity-Relationship Model (ER Model)

The database design follows a normalized relational structure that supports user authentication, secure session management, and the remaining system functionalities. The ER Model encompasses the core entities required for system functionality while maintaining data integrity and scalability.

3.1 Core Entities

User

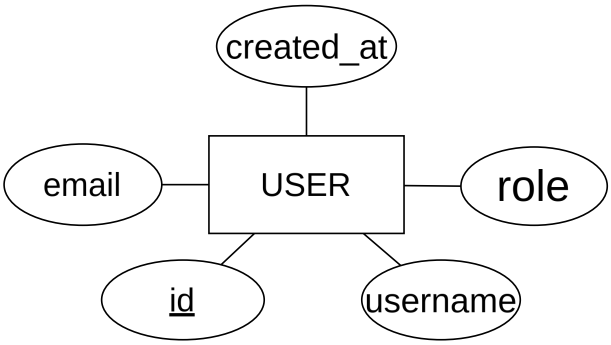


Figure 3.2: User Entity

The **User** entity represents a user in the system. The username and email attributes are provided by the authentication system. The role serves as descriminator attribute to identify whether the user is an administrator, professor or student.

Token

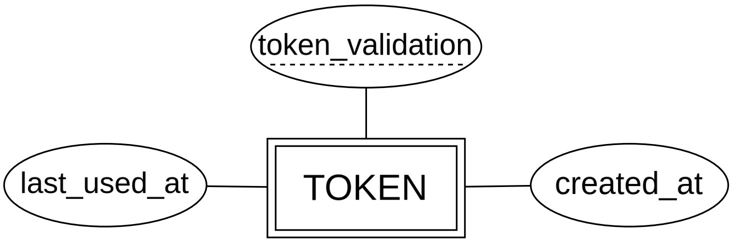


Figure 3.3: Token Entity

A user can create N tokens. The **Token** is a weak entity because it cannot be uniquely identified by its attributes alone and therefore requires a user, which is a strong entity, to be identified. A token is created by only one user.

This is a useful entity for authentication propuses. It was designed to hold a hash value in the *token_validation* attribute.

Authentication workflow:

1. Upon successful user login, a unique token is created with cryptographically secure values and stored in the database.

2. For subsequent authenticated operations, the system queries the database to verify the client-provided token against stored values.

3. Valid tokens enable secure user identification without transmitting unique identifiers.

The *last_used_at* and the *created_at* are useful for determining token expiration.

Laboratory

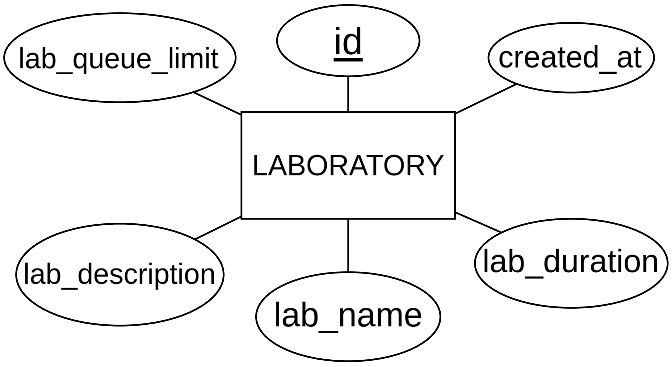


Figure 3.4: Laboratory Entity

A user, as an administrator or professor, can create N laboratories. When creating a **Laboratory**, the user can define the name (*lab_name*) and description (*lab_description*). They can also define the duration of a laboratory session (*lab_duration*) and its queue limit (*lab_queue_limit*).

Hardware

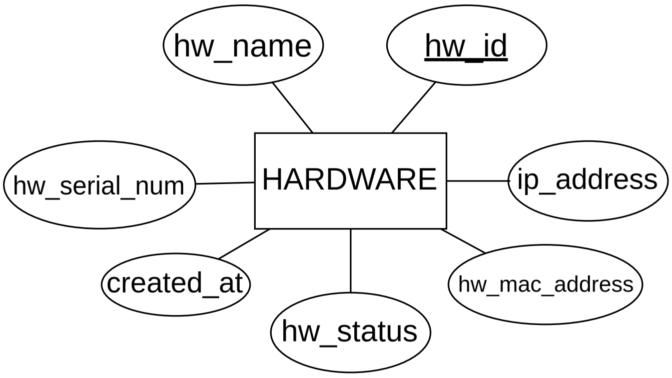


Figure 3.5: Hardware Entity

Upon successful laboratory creation, the user can associate **Hardware** to it, which must be created separately.

For the creation, it requires a name (*hw_name*), Internet Protocol address (IP) (*ip_address*) and Media Access Control address (MAC) (*mac_address*) addresses (which can be null depending on the hardware), a status (*hw_status*) to indicate whether the hardware is under maintenance, occupied, or available, and a serial number (*hw_serial_num*) to uniquely identify the hardware. Although it has an identifier, the serial number helps physically identify the hardware.

Group

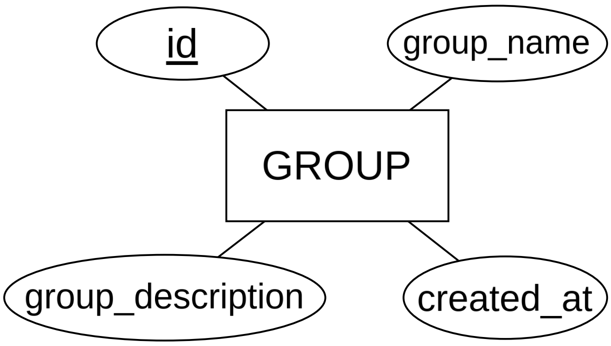


Figure 3.6: Group Entity

For a student to access a laboratory, they must be in a group that is associated with that laboratory. A professor can create a **Group** and associate users to it.

When creating a group, the user needs to name it (*group_name*) and, optionally, add a description (*group_description*) to it.

Lab Session

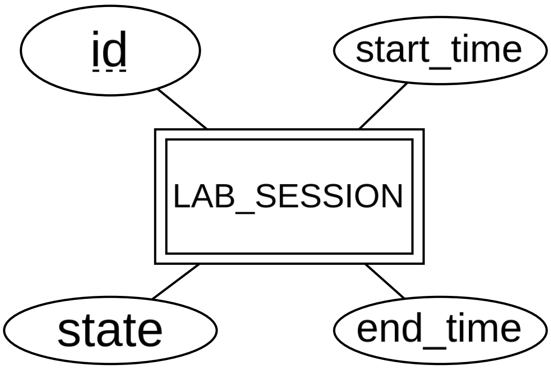


Figure 3.7: Lab Session Entity

Finally, a user can join a laboratory if they are in a group associated with it. If the laboratory is being used, the user enters a waiting queue; otherwise, a **Lab Session** is created.

Lab Session is a weak entity. It requires two strong entities to be identified: the **User** entity and the **Laboratory** entity. This is used to check whether a user is in a lab session or for statistical purposes. The *state* attribute indicates whether the session is over or still running. The *start_time* and *end_time* can be used for statistical details, such as determining how much time a user spent in a laboratory, or for future purposes, such as scheduling sessions.

3.2 Implementation Details

Although PostgreSQL [9] is being used for its functionalities, it was decided that all logic and verifications are implemented in the Web API, so that no triggers or complex constraints are implemented on the database side.

3.3 Summary

This chapter has provided an overview of the database architecture, implementation, and design decisions. It has also presented the ER Model of the database and described a typical user journey, explaining database interactions.

The documentation should be consulted for a comprehensive deep dive. It explains every entity, its attributes, and provides theoretical insights.

Chapter 4

Web API

The Web API provides endpoints for user management, authentication, authorization, and Create; Read; Update; Delete (CRUD) operations.

It is developed with Kotlin [7] and Spring Framework [6], and follows the Controller-Service-Repository pattern, which is prevalent in many Spring [8] applications. We chose this pattern because of the separation of concerns it provides and the possibilities for unit testing.

To make the codebase even easier to maintain and improve the quality of life during development, Spring Framework's Inversion of Control container [14] and the Strategy pattern [15] principle were also used.

Spring's dependency injection [16] is a well-known technology in Java [17] enterprise programming. It provides an easy way to declare dependencies, since the API was mostly built following Object-Oriented Programming (OOP) principles. This framework allows us to declare the necessary dependencies for each module. It also provides a BeanFactory interface [18] for advanced configurations. Using these Spring technologies, the object management is Spring's duty.

The Strategy pattern allowed us to have more control over the specific implementation since it follows an interface. Every concrete implementation follows an interface, making it possible to change a class dynamically without changing the code. Spring's dependency injection works very well with this strategy design pattern. This makes unit tests much easier when the concrete implementation is not intended to be tested without changing its code.

4.1 API Architecture

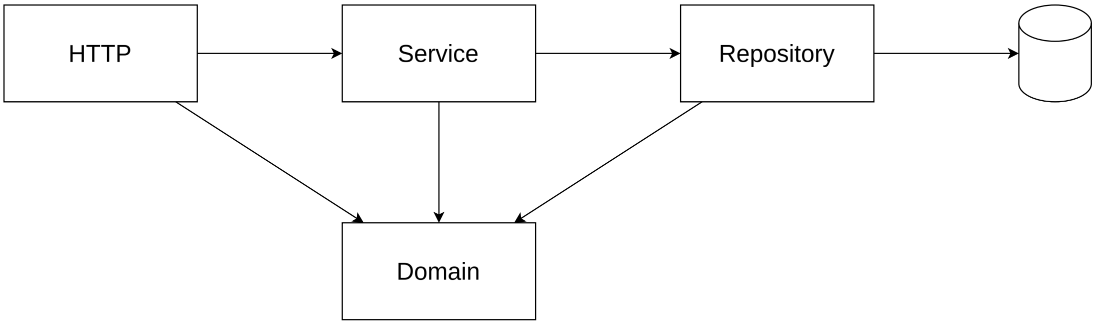


Figure 4.1: API Architecture

Figure 4.1 provides a simple overview of the implemented API. The **HTTP** module (Controller) is responsible for exposing the endpoints and handling the messages. When a request is made, the HTTP module receives the request and hands it to the **Service** module. This is where the logic and verifications are performed. Since it is necessary to fetch and save data, a **Repository** module is needed. The repository module is responsible for communicating with the database.

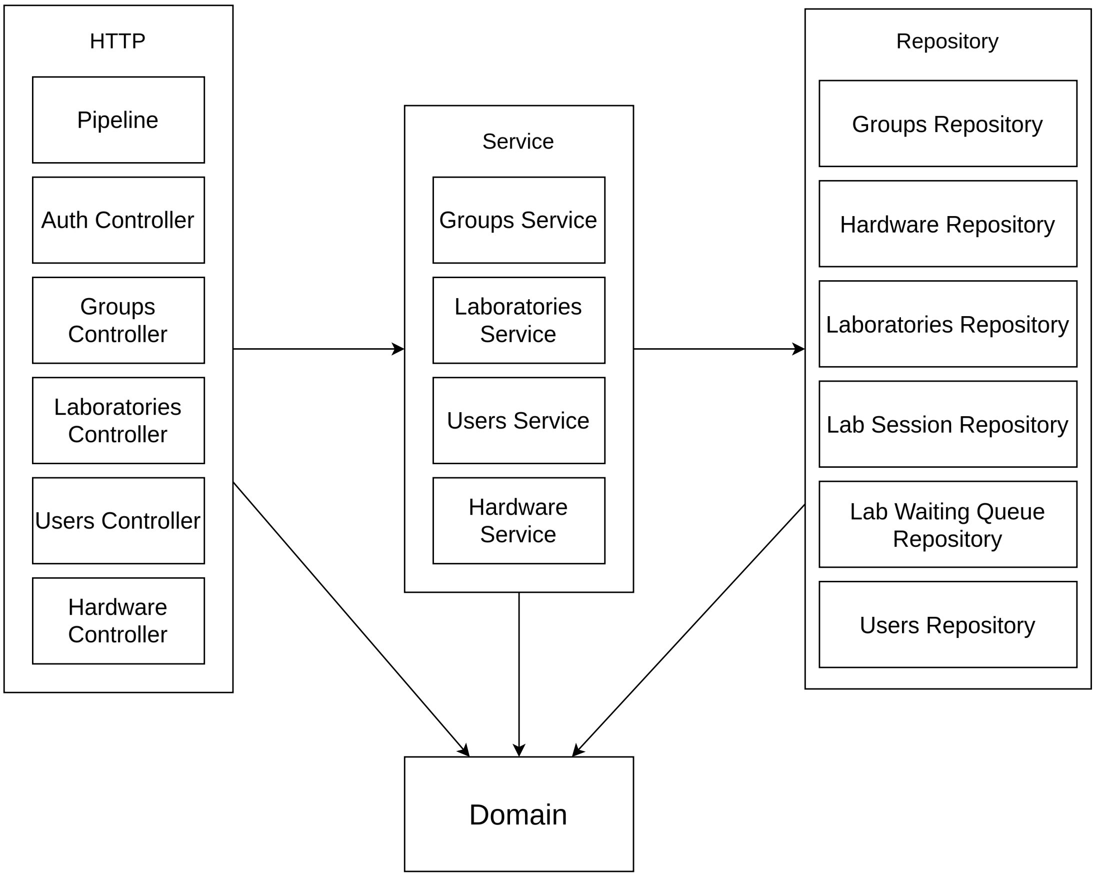


Figure 4.2: API Detailed Architecture

Figure A.1 provides a more detailed overview of how the architecture is composed.

HTTP Module

The HTTP module contains the controllers [19], each one with its functions. The pipeline contains the argument resolvers [20] and interceptors [21]. For the implemented system, only one argument resolver and two interceptors were implemented. The argument resolver is used to provide user information to the controllers. Since the authentication method we used was token-based, this argument resolver extracts user information from the request. Every controller that has a parameter with the type *AuthenticatedUser* will be authenticated.

For the request to contain the needed information about the user, an interceptor is required. This is one of the two interceptors implemented. Every request, before reaching the controller, passes through every configured interceptor. This authentication interceptor checks if the handler parameters contain a parameter of the type *AuthenticatedUser*. If it does, the entire process of getting the token from the request, verifying it, and retrieving the user is performed. If not, normal execution continues.

Listing 4.1: Type AuthenticatedUser verification example

```
if (handler is HandlerMethod &&
    handler.methodParameters.any {
        it.parameterType == AuthenticatedUser::class.java
    }
)
```

The other interceptor is for an API key. It checks if the handler contains a custom annotation. If yes, the API key is validated; if not, an unauthorized response is sent. This interceptor is useful for the login endpoint. This login endpoint is to be performed in the Web Application and is not meant to be used by end users.

Service Module

The service module performs the necessary checks, using domain classes defined in the Domain module. These classes provide configurations and methods for validating certain data. Configurations in domain classes are provided by a JavaScript Object Notation (JSON) file containing domain restrictions.

Listing 4.2: Example of the group entry

```
"group": {
    "groupName": {
      "min": 3,
      "max": 100,
      "optional": false
    },
    "groupDescription": {
      "min": 10,
            "max": 1000,
            "optional": true
        }
    }
```

This JSON file is converted to a class using Kotlin Serialization [22]. This allows an easy way to change specific values without touching the codebase.

Repository Module

The repository module serves as the data access layer, responsible for managing all communication between the application and the database. This functionality is implemented using Java Database Interface (JDBI) [23], a lightweight and efficient database access library that provides a clean, declarative API for database operations.

JDBI offers several advantages for database interaction through its declarative approach. It enables the use of interfaces, annotations, and custom mappers to perform database access operations without requiring extensive boilerplate code. The library is built on top of Java Database Connectivity (JDBC) [24], leveraging the standard Java database connectivity framework while providing a more intuitive and developer-friendly interface. This design philosophy results in a natural idiom that integrates seamlessly with both Java and Kotlin codebases.

The repository module implements the Repository pattern, which provides a consistent interface for data access operations while abstracting the underlying database implementation details.

4.2 Role-Based Access Control System

As the introduction (chapter 1) and database (chapter 3) already introduced what user types the system has as well their possible interaction and roles, an approach is needed to control that interactions to ensure the correct access control to resources. For this a Role-Based Access Control (RBAC) approach was implemented to ensure that users have access only to the resources and actions appropriate for their role, coming from the user attribute role in the database.

As mentioned before, a user, when authenticated, is assigned a role, such as student, professor, or administrator, which determines their permissions within the platform. The service module is responsible in checking the user's role before allowing access to sensitive operations, such as managing laboratory sessions, accessing administrative features, or modifying user data.

4.3 Conclusion

Since the API is expected to be public in future work, the documentation is published in Postman [25]. The public API documentation [26] and the private API documentation [27] are hosted as public websites in postman. This provides an easy-to-use documentation builder. When the

API reaches a stable version and is made public, users will have to generate an authorization token via website.

Chapter 5

Web Application

This chapter serves as the main documentation for the @Web (Next.js) application, providing a comprehensive overview of its architecture, features, authentication mechanisms, and integration strategies. It is intended as the primary reference for understanding the design and implementation choices made in the development of the web application, and should be consulted for any details regarding the frontend of the Remote Lab platform [10].

The web application is the primary interface for users to interact with the Remote Lab platform. Built with Next.js (React), it provides a modern, responsive, and user-friendly experience for students, professors, and administrators [10]. The application enables users to access, schedule, and manage laboratory resources remotely, supporting a wide range of devices and ensuring accessibility for all user roles.

5.1 Overview

The web application serves as the main point of interaction, integrating seamlessly with backend services via RESTful APIs. It supports multiple user roles, each with tailored access and features according to their permissions. The interface dynamically adapts to the user's role, ensuring that each user only sees the features and data relevant to them.

The decision to use Next.js as the framework was driven by its robust feature set and alignment with modern web development best practices. Next.js, built on top of React, offers built-in solutions for routing, server-side rendering, API integration, and more [28, 29, 30, 31]. This allowed the team to leverage existing knowledge while accelerating development and ensuring scalability and maintainability.

A key architectural choice is the retrieval of domain configuration in JSON format from the backend API. This ensures that the frontend's domain-related settings are always synchronized with those defined on the backend, centralizing domain management and reducing the risk of inconsistencies.

5.2 Architecture and Logic Separation

The application leverages Next.js to achieve a clear separation between client-side and server-side logic. All sensitive operations and data exchanges with the backend API are performed server-side, enhancing security and control over data flow [29, 30]. This architecture enables efficient server-side rendering, improved performance, and a better user experience. For more details on using Next.js to perform API requests from the server, see [32].

To optimize performance and user experience, the frontend performs input validation before sending requests to the API. This approach reduces the number of unnecessary API calls by catching invalid data early, providing immediate feedback to users, and minimizing server load. Importantly, the validation rules on the frontend are kept synchronized with the backend by leveraging the domain configuration JSON retrieved from the API. This ensures that both the client and server enforce consistent validation logic, reducing the risk of discrepancies and improving the reliability of the application.

5.3 Main Features

- **Authentication:** Secure login using Microsoft OAuth (NextAuth), supporting university credentials and automatic role assignment [12].

- **Dashboard:** Personalized dashboard displaying relevant information, upcoming sessions, and quick access to key features.

- **Laboratory Management:** Professors and administrators can create, edit, and manage laboratory sessions, equipment, and participant lists.

- **Calendar and Scheduling:** Interactive calendar for booking and managing laboratory sessions, with real-time availability and notifications.

- **Role-Based Access:** The interface adapts to the user's role, showing only the features and data relevant to their permissions. Users with higher roles can view and interact with the platform as if they had a lower role for testing and support purposes (RBAC).

- **Responsive Design:** Optimized for desktops, tablets, and mobile devices, ensuring accessibility and usability across platforms.

- **Loading UI:** The application uses loading states and skeleton screens to provide feedback during data fetching, improving perceived performance [33].

An additional usability feature implemented in the frontend is the ability for users with higher roles to switch their view to that of a lower role. For example, an administrator can choose to view the application as a professor, and a professor can view it as a student. This functionality is particularly useful for testing, support, and understanding the user experience from different perspectives, ensuring that features and permissions are correctly enforced for each role.

To enhance the user experience and make the application feel as much like a single-page application with multiple menus as possible, the frontend leverages Next.js modals using parallel and intercepting routes [28]. This approach allows users to open dialogs, forms, or detailed views as overlays without leaving the current page context, providing smooth navigation and reducing full page reloads. By utilizing these advanced routing features, the application maintains a cohesive and interactive interface, improving usability and workflow efficiency.

5.4 Authentication with NextAuth

Authentication is implemented using NextAuth.js, a flexible authentication solution for Next.js applications. NextAuth is integrated to provide secure, seamless login experiences using Microsoft OAuth, allowing users to authenticate with their university credentials [12].

When a user attempts to log in, they are redirected to the Microsoft login page. Upon successful authentication, NextAuth handles the OAuth flow, retrieves the user's profile information, and establishes a session. The session includes the user's role (such as student, professor, or administrator), which is determined based on information provided by the authentication provider or assigned within the application logic.

NextAuth manages user sessions securely, storing authentication tokens and session data in HTTP-only cookies to prevent unauthorized access from the client side. The authentication state is accessible throughout the application, enabling role-based access control and dynamic adaptation of the user interface according to the user's permissions.

This approach ensures that only authorized users can access protected resources and features, while also providing a familiar and convenient login experience. The integration with Microsoft OAuth leverages the institution's existing identity infrastructure, enhancing security and simplifying user management.

NextAuth.js is highly extensible and supports a wide range of authentication providers through its flexible OAuth configuration [12]. This means that, in addition to Microsoft OAuth, it is possible to integrate other OAuth-based authentication methods commonly used by universities, such as Google, GitHub, or institutional identity providers that support OAuth 2.0 or OpenID Connect. Furthermore, NextAuth allows the implementation of custom OAuth providers, making it feasible to connect to a university's own authentication server if needed. This flexibility ensures that the authentication system can adapt to different institutional requirements and future changes in identity management strategies [12].

5.5 Integration, Security, and Deployment

The web application communicates securely with the backend via RESTful APIs, using authentication tokens to protect sensitive operations. User sessions and data are managed according to best practices, ensuring privacy and integrity. The frontend is designed to prevent unauthorized access and to provide a robust, extensible foundation for future enhancements.

Deployment is streamlined using Next.js's built-in tools and best practices [34]. Environment variables are used to manage configuration securely across different environments [35]. The application is containerized and orchestrated alongside other platform components, ensuring consistency and reliability in both development and production using modern Continuous Integration and Continuous Deployment (CI/CD) practices.

5.6 Summary

The web application leverages the power and flexibility of Next.js to deliver a secure, scalable, and user-centric platform for remote laboratory access. Its architecture ensures clear separation of concerns, robust authentication, and a responsive user experience, while its integration with modern deployment practices guarantees maintainability and future growth. For further details on the features and implementation of the web application, this chapter should be used as the main reference, in conjunction with the official Next.js documentation [10].

Chapter 6

Project Organization and Deployment

6.1 Project Structure

The Remote Lab project is organized into several main directories, most of which are managed as GitHub submodules. This modular approach enables independent development, versioning, and access control for each core component, supporting both scalability and security. Submodules also facilitate collaboration among different teams and ensure that sensitive information is handled appropriately.

The main submodules and directories are:

- **api/** – Contains the backend source code, implemented in Kotlin with Spring Boot. Responsible for business logic, user management, and laboratory session control.

- **db/** – Includes database scripts, supporting the system's persistence layer.

- **docs/** – Stores project documentation, including technical reports, user guides, and architectural diagrams.

- **img/** – Contains project images, such as diagrams, screenshots, and other visual assets.

- **nginx/** – Provides NGINX configuration files for reverse proxying, load balancing, and secure access to backend services [36].

- **private/** – Dedicated to sensitive files and configurations, such as environment variables and secrets. This submodule is not included directly in the main repository, ensuring that only authorized members have access to confidential information like API keys and external service credentials.

- **website/** – Holds the frontend web application, built with Next.js (React) [10]. Provides the user interface for laboratory access, scheduling, and management.

- **wiki/** – Stores the GitHub Wiki pages, including project documentation, deployment instructions, and other relevant information.

This structure allows for independent development, testing, and deployment of each component, while best practices such as containerization and secure secret management ensure the project is robust and ready for collaborative development and future expansion.

6.2 Deployment

The deployment process for the Remote Lab platform is designed to be straightforward, secure, and reproducible, leveraging modern DevOps practices and containerization technologies [37, 38].

6.2.1 Containerization and Orchestration

All major components—including the backend (API), frontend (website), and database—are containerized using Docker [37]. This guarantees consistency across development, testing, and production environments. Docker Compose orchestrates multi-container deployments, manages networking between services, and handles environment-specific configurations [38].

6.2.2 Environment Configuration and Secrets

Sensitive configuration files and environment variables are managed in the **private/** submodule. This submodule contains secrets such as API keys, database credentials, and authentication settings, tailored to the platform's requirements. Access is restricted to authorized team members, ensuring the security of confidential information.

6.2.3 Automation with start.sh

To streamline deployment, the platform provides a `start.sh` script at the repository root. This script automates the initialization of all required services and dependencies with a single command. It builds Docker images, starts containers using Docker Compose, and ensures that environment variables and configuration files are correctly loaded from the `private/` submodule.

The `start.sh` script also supports several flags to customize the deployment process, such as selecting the environment (development or production), starting only the API, enabling Cloudflare tunneling, or switching branches. These options make it easy to adapt deployment to different scenarios with simple command-line arguments.

6.2.4 Cloudflare Tunneling

To facilitate secure remote access to development or demonstration environments, Cloudflare Tunneling is used [39]. This solution exposes local services to the internet without requiring firewall changes, port forwarding, or public IPs. Cloudflare Tunnel creates a secure connection between the local machine and a public endpoint managed by Cloudflare, routing external traffic in an encrypted manner to the internal environment using the HTTP or HyperText Transfer Protocol Secure (HTTPS) protocols. This is especially useful for:

- Allowing team members, professors, or evaluators to access development instances remotely.
- Demonstrating the system without deploying to a production environment.
- Testing external integrations or federated authentication in controlled environments.

In this project, Cloudflare Tunnel is integrated into the automation workflow via `start.sh` and Docker Compose. By using the appropriate flag (`cloudflare` or `c`) in the startup script, the tunnel service is automatically started alongside the other containers, making remote access simple and secure.

This approach reduces operational complexity, increases security, and streamlines the system's development and validation cycle.

6.2.5 Nginx as Reverse Proxy

In the current implementation, NGINX acts as a reverse proxy to route HTTP requests to both the frontend (website) and backend (API) services [36]. This allows the web application and the API to be accessed through a single entry point, even though they run in separate containers and listen on different internal ports. NGINX transparently proxies requests from the frontend to the backend, simplifying client-side configuration and enabling seamless communication between modules.

At this stage, NGINX is not configured for load balancing or native HTTPS termination. All traffic is handled over HTTP within the Docker network, and any external HTTPS is managed by tunneling solutions such as Cloudflare when needed.

The NGINX configuration files are located in the `nginx/` directory and are included in the Docker Compose setup, ensuring that the reverse proxy is automatically started and configured alongside the other services during deployment.

6.2.6 Deployment Steps

1. **Clone the Repository and Submodules:** Clone the main repository and initialize all submodules, including `private/`, to ensure all components and configurations are available.

2. **Configure Environment Variables:** Ensure that all required environment variables and secret files are present in the appropriate locations, as provided by the `private/` submodule.

3. **Build and Start Services:** Use the provided `docker-compose.yml` file to build and start all services with a single command (e.g., `docker compose up --build`), or simply run the `start.sh` script at the project root, which automates the entire process including building images, starting containers, and loading environment variables and secrets.

4. **Access the Platform:** Once all containers are running, the platform can be accessed via the configured web address. NGINX is used as a reverse proxy to route traffic securely to the appropriate services [36].

6.3 Build and CI/CD

- **Gradle:** Used for building and managing backend dependencies.

- **NPM:** Used for frontend dependency management and builds.

- **Dockerfiles:** Multi-stage builds are used for both backend and frontend to optimize image size and security with Docker.

- **GitHub Actions:** (If applicable) Used for continuous integration and automated builds as part of the CI/CD pipeline [40].

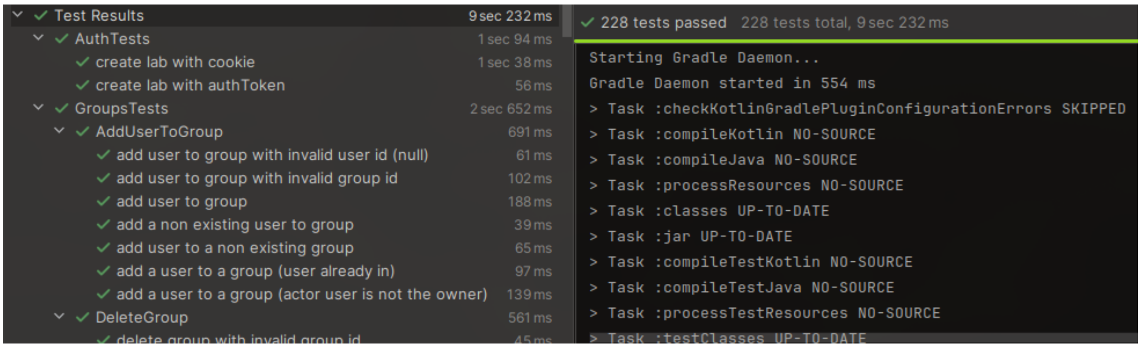
6.4 Summary

The implemented infrastructure leverages modern web technologies, containerization, and modular design to provide a robust, scalable, and maintainable platform for remote laboratory access [10, 37]. The deployment process is automated, secure, and ready for future growth, ensuring that the platform can evolve to meet new requirements and increased demand.

Chapter 7

Experimental Results

To validate the proper behavior of the system, comprehensive unit tests were implemented for the web API. Each module undergoes thorough testing, covering both success and error states to ensure robustness and reliability.



```
⌄ ✓ Test Results                                               9 sec 232 ms
  ⌄ ✓ AuthTests                                                1 sec 94 ms
      ✓ create lab with cookie                                 1 sec 38 ms
      ✓ create lab with authToken                                    56 ms
  ⌄ ✓ GroupsTests                                              2 sec 652 ms
    ⌄ ✓ AddUserToGroup                                              691 ms
        ✓ add user to group with invalid user id (null)              61 ms
        ✓ add user to group with invalid group id                   102 ms
        ✓ add user to group                                         188 ms
        ✓ add a non existing user to group                           39 ms
        ✓ add user to a non existing group                           65 ms
        ✓ add a user to a group (user already in)                    97 ms
        ✓ add a user to a group (actor user is not the owner)       139 ms
    ⌄ ✓ DeleteGroup                                                 561 ms
        ✓ delete group with invalid group id                         45 ms
```

```
✓ 228 tests passed  228 tests total, 9 sec 232 ms
Starting Gradle Daemon...
Gradle Daemon started in 554 ms
> Task :checkKotlinGradlePluginConfigurationErrors SKIPPED
> Task :compileKotlin NO-SOURCE
> Task :compileJava NO-SOURCE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE
> Task :jar UP-TO-DATE
> Task :compileTestKotlin NO-SOURCE
> Task :compileTestJava NO-SOURCE
> Task :processTestResources NO-SOURCE
> Task :testClasses UP-TO-DATE
```

Figure 7.1: Unit Tests Results

In future work, we aim to expand the test coverage significantly and implement load testing to evaluate system performance under various conditions. These load tests will provide valuable insights into response times during peak usage periods, enabling us to identify bottlenecks and optimize system performance accordingly. The results from these comprehensive tests will guide further improvements to enhance the overall system efficiency and user experience.

Chapter 8

Conclusions

This report has presented the design, implementation, and validation of a comprehensive platform for remote laboratory access, addressing the growing need for flexible and accessible laboratory resources in educational and research environments. The solution successfully bridges the gap between traditional hands-on experimentation and the requirements of modern remote learning and collaboration.

The developed platform successfully fulfills all primary objectives established at the project's inception. The implementation of a robust web API provides comprehensive management capabilities for users, laboratories, and hardware resources. The intuitive web application interface ensures accessibility for users with varying technical backgrounds, while the secure authentication and authorization mechanisms, coupled with role-based access control, guarantee system integrity and appropriate resource allocation.

The well-designed database system ensures reliable data persistence, supporting the platform's scalability requirements. Furthermore, the implementation of remote manipulation capabilities of laboratory devices enables real-time interaction with experimental equipment, maintaining the essential hands-on experience that characterizes quality laboratory work.

8.1 Limitations and Future Work

While the current implementation successfully addresses the primary objectives, several areas present opportunities for future enhancement. The expansion of the testing framework to include comprehensive load testing will provide valuable insights into system performance under high-usage conditions. This analysis will inform optimization strategies and guide infrastructure scaling decisions.

Future development efforts should focus on expanding hardware compatibility to support a broader range of laboratory equipment types. Additionally, the implementation of advanced monitoring and analytics capabilities could provide valuable usage insights and facilitate resource optimization. The integration of collaborative features, such as shared experimentation sessions and real-time collaboration tools, would further enhance the platform's educational value.

Furthermore, the development of mobile applications and the enhancement of accessibility features would broaden the platform's reach and usability across diverse user populations.

References

[1] MIT iLab Project. ilab shared architecture (isa). https://icampus.mit.edu/projects/ilabs/index.html, 2025. Accessed: 2025-05-29.

[2] LabShare Project. Labshare: Collaborative remote laboratories. https://dbpedia.org/page/Labshare, 2025. Accessed: 2025-05-29.

[3] WebLab-Deusto. Weblab-deusto: Remote laboratory for electronics. https://www.weblab.deusto.es/, 2025. Accessed: 2025-05-29.

[4] L. F. Zapata Rivera and Larrondo Petrie. The remote laboratory management system (rlms) pattern. https://hillside.net/sugarloafplop/2018/program/papers/GroupA/12.3.pdf, 2018. Accessed: 2025-05-30.

[5] D. Lowe, T. Machet, et al. Sahara labs: Remote laboratory framework. https://github.com/sahara-labs, 2013. University of Technology Sydney. Accessed: 2025-05-30.

[6] VMware Tanzu. Spring framework. https://spring.io/projects/spring-framework. Accessed: 2025-06-01.

[7] Jetbrains. Kotlin programming language. https://kotlinlang.org/. Accessed: 2025-06-01.

[8] VMware Tanzu. Spring. https://spring.io/. Accessed: 2025-06-01.

[9] The PostgreSQL Global Development Group. Postgresql. https://www.postgresql.org/. Accessed: 2025-06-01.

[10] Vercel. Next.js documentation. https://nextjs.org/docs, 2025. Accessed: 2025-05-30.

[11] Meta Open Source. React. https://react.dev/reference/react. Accessed: 2025-06-01.

[12] Vercel. Authentication. https://nextjs.org/docs/app/building-your-application/authentication, 2025. Accessed: 2025-05-30.

[13] Microsoft. Microsoft oauth. https://learn.microsoft.com/en-us/entra/architecture/auth-oauth2. Accessed: 2025-06-01.

[14] VMware Tanzu. The ioc container. https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html. Accessed: 2025-06-01.

[15] Wikipedia. Strategy pattern. https://en.wikipedia.org/wiki/Strategy_pattern. Accessed: 2025-06-01.

[16] VMware Tanzu. Spring depedency injection. https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html. Accessed: 2025-06-01.

[17] Oracle. Java programming language. https://www.java.com/en/. Accessed: 2025-06-01.

[18] VMware Tanzu. Interface beanfactory. https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/BeanFactory.html. Accessed: 2025-06-01.

[19] VMware Tanzu. Annotated controllers. https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller.html. Accessed: 2025-06-01.

[20] VMware Tanzu. Interface handlermethodargumentresolver. https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/messaging/handler/invocation/HandlerMethodArgumentResolver.html. Accessed: 2025-06-01.

[21] VMwaree Tanzu. Interface handlerinterceptor. https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/HandlerInterceptor.html. Accessed: 2025-06-01.

[22] Jetbrains. Kotlin serialization. https://kotlinlang.org/docs/serialization.html. Accessed: 2025-06-01.

[23] Jdbi. Jdbi 3 developer guide. https://jdbi.org/. Accessed: 2025-06-01.

[24] Oracle. Java jdbc api. https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/. Accessed: 2025-06-01.

[25] Postman. Postman. https://www.postman.com/. Accessed: 2025-06-01.

[26] RL Dev Team. Remote lab public api. https://documenter.getpostman.com/view/31383926/2sB2cVgNBA. Accessed: 2025-06-01.

[27] RL Dev Team. Remote lab private api. https://documenter.getpostman.com/view/31383926/2sB2qUm4cA. Accessed: 2025-06-01.

[28] Vercel. App router: Routing, layouts, and pages. https://nextjs.org/docs/app/building-your-application/routing, 2025. Accessed: 2025-05-30.

[29] Vercel. Server and client components. https://nextjs.org/docs/app/building-your-application/rendering/server-and-client-components, 2025. Accessed: 2025-05-30.

[30] Vercel. Data fetching. https://nextjs.org/docs/app/building-your-application/data-fetching, 2025. Accessed: 2025-05-30.

[31] Vercel. Api routes. https://nextjs.org/docs/app/building-your-application/routing/api-routes, 2025. Accessed: 2025-05-30.

[32] Juan Cruz Martinez. Using next.js server actions to call external apis. https://auth0.com/blog/using-nextjs-server-actions-to-call-external-apis/, 2023. Accessed: 2025-05-29.

[33] Vercel. Loading ui and streaming. https://nextjs.org/docs/app/building-your-application/routing/loading-ui-and-streaming, 2025. Accessed: 2025-05-30.

[34] Vercel. Deploying. https://nextjs.org/docs/app/building-your-application/deploying, 2025. Accessed: 2025-05-30.

[35] Vercel. Environment variables. https://nextjs.org/docs/app/building-your-application/configuring/environment-variables, 2025. Accessed: 2025-05-30.

[36] Inc. NGINX. Nginx documentation. https://docs.nginx.com/, 2025. Accessed: 2025-05-30.

[37] Docker Inc. Docker documentation. https://docs.docker.com/, 2025. Accessed: 2025-05-30.

[38] Docker Inc. Docker compose documentation. https://docs.docker.com/compose/, 2025. Accessed: 2025-05-30.

[39] Inc. Cloudflare. Cloudflare tunnel documentation. https://developers.cloudflare.com/cloudflare-one/connections/connect-apps/, 2025. Accessed: 2025-05-30.

[40] Inc. GitHub. Github actions documentation. https://docs.github.com/en/actions, 2025. Accessed: 2025-05-30.

Appendix A

Database Documentation

A.1 Introduction

This document provides a comprehensive overview of the database structure, including its entities, attributes, and relationships. It also details key implementation decisions.

A.2 Database overview

The database has been designed using an Entity-Relationship (ER) approach. This methodology enables a clear understanding of how different entities interact within the system. The following section presents the ER model diagram.

The database is implemented using PostgreSQL and has been tested with sample data in a Docker container.

The next section will present the ER model in detail.

A.3 Entity-Relationship Model

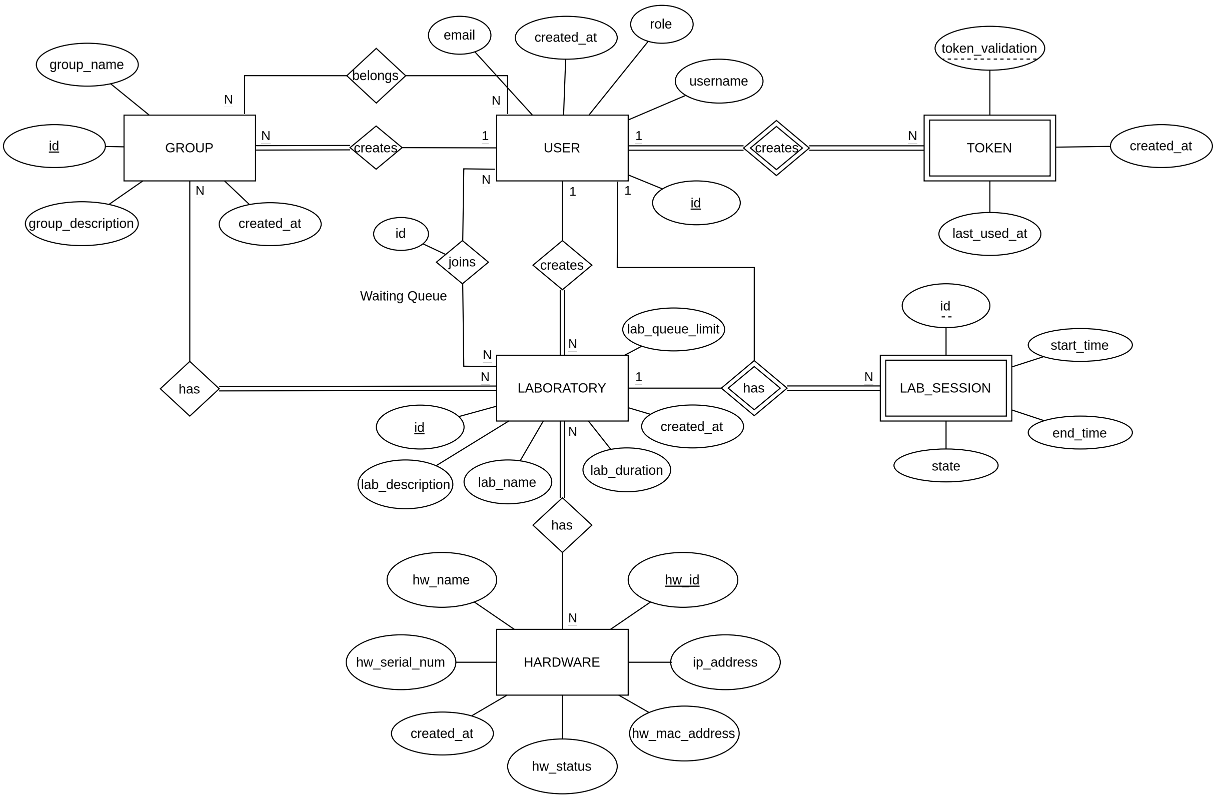


Figure A.1: ER Model

A.4 Entities and Attributes

This section provides a comprehensive description of each entity and their attributes.

A.4.1 User

The **User** is a fundamental entity in the database that represents system users.

It has an **user_id** as its primary key, implemented as an identity column. An identity column automatically generates values from an implicit sequence. Therefore, whenever a new user is created, a unique id is automatically generated. The user_id is an integer data type, chosen to simplify database queries by using a simple numeric identifier instead of email or username.

The entity includes character sequence attributes: **username** and **email**. All these fields are required (not null), and the email must be unique. Initially, the username is automatically set to match the user's username returned by OAuth login. While usernames are used for display purposes and may not be unique, emails serve as unique identifiers for user searches. The specific constraints for these attributes are defined in the application domain.

Additionally, it includes a **student_nr** (student number) as a unique integer field and **created_at** as a required timestamp. The student number is optional, allowing the system to accommodate non-student users such as professors. Notably, there is no discriminator attribute as user type distinctions are handled through the Role-Based Access Control (RBAC) system.

A.4.2 Token

Token is implemented as a weak entity since it cannot exist independently and requires a user_id for identification. Each token is associated with a user and is essential for authentication purposes. Users must have a valid token when interacting with the system.

As a weak entity, it requires a partial key, which is the **token_validation** field. This attribute stores a randomly generated hash created by the system and is used to verify token validity. For instance, when a user's token is included in an authorization header, the system compares it with this stored value during each interaction. It is implemented as a required character sequence.

The entity also includes **created_at** and **last_used_at** timestamps, both required fields. These attributes enable token validity checking based on timing rules defined by the application domain.

A.4.3 Group

The **Group** entity represents various types of user groupings, such as student classes, work groups, or professor groups.

Each group has a **group_id** as its primary key, implemented as an auto-generated integer identity column. This unique identifier facilitates efficient querying and group identification.

The **group_name** is a required character sequence that stores the user-defined name for the group. Group names can be modified after creation, with specific length and format restrictions defined by the application domain.

Groups can include an optional **group_description** stored as a text type. This field allows users to provide detailed information about the group's purpose or characteristics. While technically unlimited in length, practical limitations may be imposed by the application domain.

The entity also includes a **created_at** timestamp, which can be used for analytical and administrative purposes.

A.4.4 Laboratory

The **Laboratory** entity represents physical or virtual laboratory spaces within the system.

Each laboratory has a **lab_id** as its primary key, implemented as an auto-generated integer identity column. This unique identifier simplifies laboratory identification and database queries.

The **lab_name** attribute is a required character sequence that stores the laboratory's designation. Specific naming conventions and restrictions are defined by the application domain.

The entity includes a **lab_duration** attribute, which specifies the standard duration of laboratory sessions in minutes. This integer field is required and helps manage session scheduling.

Additionally, it includes a **created_at** timestamp for tracking laboratory creation and administrative purposes.

A.4.5 Lab Session

Lab Session is implemented as a weak entity that depends on both User and Laboratory entities. A session represents a specific time period during which a user can access and operate laboratory equipment.

The entity uses **session_id** as a partial key, which, combined with user_id and lab_id, uniquely identifies each session. The session_id is an auto-generated integer identity column that facilitates session tracking and queries.

Each session includes required **start_time** and **end_time** timestamps that define the session's temporal boundaries.

The **state** attribute is a required character sequence that indicates the session's current status (e.g., active, inactive, or scheduled). The system can be extended to support additional states as needed, with any such changes to be documented in future updates.

A.4.6 Hardware

The **Hardware** entity represents physical equipment within the laboratory system, such as computers or FPGAs.

Each piece of hardware is identified by a **hw_id** primary key, implemented as an auto-generated integer identity column to facilitate equipment tracking and queries.

The entity includes a required **hw_name** attribute as a character sequence for equipment identification.

A required **hw_serial_num** character sequence stores the hardware's serial number for inventory management.

The **ip_address** and **hw_mac_address** are character sequence fields that may be null depending on the hardware type, as not all equipment requires network connectivity.

A required **hw_status** character sequence tracks the current state of the hardware (e.g., available, occupied).

Like other entities, it includes a **created_at** timestamp for inventory tracking and administrative purposes.

A.5 Associations

This section details the relationships between entities, explaining how they interact within the system and what restrictions they have.

User associations include:

- **Token** - A weak association where each user may have multiple tokens (N:1). Tokens are automatically generated by the system as needed, with the user's primary key serving as part of the token's composite key.

- **App Invite** - A weak association where users can create multiple app invites (N:1). The user's primary key is incorporated into the app invite's key structure. Only a user with Professor's role can create app invites. It is worth noting that an Administrator can also hold the position of Professor due to hierarchical considerations.

- **Group** - Users have two distinct relationships with groups:
  - **creates** - Users can create multiple groups, but each group has exactly one creator (N:1). Only Professor can create groups.
  - **belongs** - Users can belong to multiple groups, and groups can have multiple members (N:N)

- **Laboratory** - Users have two types of laboratory associations:
  - **creates** - Users can create multiple laboratories, but each laboratory has one creator (N:1). Only Professor can create laboratories.
  - **joins** - Users can join multiple laboratories over time, and laboratories can be joined by multiple users (N:N). While the application domain may impose restrictions (e.g., limiting concurrent laboratory access), the database structure maintains this flexibility to support historical tracking.

- **Lab Session** - A weak association where users can have multiple sessions, but each session belongs to exactly one user (N:1)

App Invite maintains an additional association with the Group entity, linking invites to specific groups. Each invite corresponds to one group, while groups can have multiple associated invites (N:1). This enables automatic group assignment upon registration.

Group has an additional association with Laboratory, controlling laboratory access permissions. Groups can be associated with multiple laboratories, and laboratories can be accessible to multiple groups (N:N).

Laboratory maintains these additional associations:

- **Lab Session** - Laboratories can have multiple sessions, but each session is associated with exactly one laboratory (N:1)

- **Hardware** - Laboratories can be assigned multiple pieces of hardware, and hardware can be assigned to multiple laboratories over time (N:N)

Note that additional constraints and business rules are implemented at the web API level. Please refer to the web API documentation for detailed information about these restrictions.