PTGest

Management System for Personal Trainers and Businesses

Daniel Sousa
Pedro Macedo

Supervisor: Filipe Freitas

Final report written for Project and Seminary
BSc in Computer Science and Computer Engineering

July 2024

Instituto Superior de Engenharia de Lisboa

PTGest

Management System
for Personal Trainers and Businesses

48642 Daniel Martins Cabrita de Sousa

49471 Pedro Miguel Afonso Macedo

Orientadores: Filipe Bastos de Freitas

Final report written for Project and Seminary
BSc in Computer Science and Computer Engineering

July 2024

Abstract

This project aims to develop a comprehensive platform to facilitate efficient integration and management among trainees, personal trainers (PTs), and fitness companies, which may include consultancy firms or organizations with their own gyms.

The solution involves creating a management application with a user-friendly interface capable of handling training sessions, training plans, sets, and exercises. The platform will also provide functionalities for companies or independent trainers to manage trainer and trainee information, including the ability to assign, reassign, add, or edit details related to both trainers and trainees.

Additionally, the platform will serve as a centralized location for trainers to store and share reports, session feedback, and trainees' progress using measures such as body mass index (BMI), body fat percentage, and muscle mass percentage. This will allow trainers to track trainees' progress, adjust training plans accordingly, and communicate effectively to share their evolution.

We developed a Web application, in the server side we used Kotlin along with the Spring Boot framework, ensuring a scalable solution for the server. The platform is hosted on the cloud using Render, enabling easy access and management of the application. It will be designed to be scalable, allowing for the addition of new features and functionalities as needed. Data storage will be handled by a PostgreSQL database, providing a reliable and secure solution for managing the data.

For the user interface, the Vue.js framework will be used along with Vite, offering a fast and efficient way to develop the frontend of the application. The platform was designed to be responsive, ensuring easy access and usability on various devices. Additionally, it will feature an intuitive interface that is easy to navigate and use, enhancing overall user experience.

Resumo

Este projeto tem como objetivo desenvolver uma plataforma abrangente para facilitar a integração e gestão eficiente entre trainees, personal trainers (PTs) e empresas de fitness, que podem incluir empresas de consultoria ou organizações com seus próprios ginásio.

A solução envolve a criação de uma aplicação de gestão com uma interface amigável, capaz de gerenciar sessões de treinamento, planos de treino, séries e exercícios. A plataforma fornecerá também funcionalidades para que empresas ou treinadores independentes possam gerenciar informações de treinadores e trainees, incluindo a capacidade de atribuir, realocar, adicionar ou editar detalhes relacionados tanto aos treinadores quanto aos trainees.

Além disso, a plataforma servirá como um local centralizado para que os treinadores armazenem e compartilhem relatórios, feedback das sessões e progresso dos trainees, utilizando medidas como índice de massa corporal (BMI), percentual de gordura corporal e percentual de massa muscular. Isso permitirá que os treinadores acompanhem o progresso dos trainees, ajustem os planos de treino conforme necessário e se comuniquem efetivamente para compartilhar a evolução dos trainees.

Desenvolvemos uma aplicação Web, no lado do servidor, utilizamos Kotlin juntamente com a framework Spring Boot, garantindo uma solução escalável para o servidor. A plataforma está hospedada na nuvem utilizando o Render, permitindo um fácil acesso e gestão da aplicação. Ela será projetada para ser escalável, permitindo a adição de novos recursos e funcionalidades conforme necessário. O armazenamento de dados será feito por meio de um banco de dados PostgreSQL, proporcionando uma solução confiável e segura para a gestão dos dados.

Para a interface do usuário, será utilizado o framework Vue.js junto com o Vite, oferecendo uma forma rápida e eficiente de desenvolver o frontend da aplicação. A plataforma será projetada para ser responsiva, garantindo fácil acesso e usabilidade em vários dispositivos. Além disso, contará com uma interface intuitiva que é fácil de navegar e usar, aprimorando a experiência geral do usuário.

Contents

List of Figures

Listings

List of Acronyms

API Application Programming Interface

BMI Body Mass Index

DBMS Database Management System

PT Personal Trainer

JWT JSON Web Token

PWA Progressive Web App

SQL Structured Query Language

URI Uniform Resource Identifier

HTTP Hypertext Transfer Protocol

DOM Document Object Model

JSX JavaScript XML

HMR Hot Module Replacement

ESM ECMAScript Modules

DAW Web Application Development

HTTPS Hyper Text Transfer Protocol Secure

RBAC Role-based access control

Chapter 1

Introduction

This chapter is dedicated to explaining our motivation for this project and its organization.

1.1 Motivation

The motivation for this project is to provide a platform that allows trainees to have a more personalized and efficient training experience while also providing personal trainers and fitness companies with a tool to manage their clients more effectively. This project addresses the need for a more integrated and efficient fitness management system that can benefit both trainees and fitness professionals.

With the increasing popularity of fitness and the growing demand for personal trainers, there is a need for a more efficient and effective way to manage training sessions, track progress, and share results with trainees seeking to improve their fitness levels. This project aims to provide a solution by creating a platform that offers a personalized and efficient training experience for trainees and a robust client management tool for personal trainers and fitness companies.

The platform will feature sessions that can be either trainer-guided or based on a training plan created by the trainer. It will include notes for the session, feedback mechanisms, progress reports, and a notification system. Additionally, trainers will be able to record trainee measurements, calculate results, and store them in a database to monitor progress. This versatile fitness management application will enhance the interaction between trainees, personal trainers, and fitness companies, ultimately improving the overall fitness experience for everyone involved.

1.2 Functional Requirements

- **User Registration**: Allow the registration of different types of users, including trainees, individual PTs, subcontracted PTs and companies. Each type of user will have different data fields to fill in during registration:

1. **Trainees** can be registered by PTs or companies, with an email generated and sent by the system containing a method for defining a password;
2. **Hired PTs** can only be registered or unregistered through a company that adds them into the system. An email is generated and sent by the system with a method to set a password, while **independent PTs** register without the intervention of another entity;
3. **Companies** register themselves independently of any other entity.

- **Training Session Management**: The system allow PTs to schedule, reschedule and manage training sessions. It also possible to add details such as muscle group, exercises to be performed, training duration, date, time and location of the session;

- **Session Calendar**: The platform provide a calendar for both the trainee and the PT, displaying their scheduled sessions. These sessions could be training sessions based on a specific training plan, supervised training or evaluation sessions;

- **Feedback and Reviews**: The system allows trainees to evaluate the PTs if the session is supervised by them, using a grading system (for example, assigning a grade between 1 and 5). This evaluation can be accessed by both the hiring company and the supervised trainee. Furthermore, each session include feedback sections for individual exercises and for the session overall. These sections can be filled out by either the trainee or the PT and can be reviewed by them at a later time;

- **Trainee Information Registration**: The platform allows PTs to record information related to a trainee, such as weight, height, body mass index (BMI[1]), percentage of lean and fat mass, and dimensions of muscle groups;

- **Progress Reports**: PTs have the possibility to create progress reports for trainee, which can be private (only visible to PT) or public (visible to both the trainee and the PT);

- **Email Notifications**: The system is able to send emails to notify users of sessions and password changes;

- **Creation of Custom Exercises**: PTs are able to create custom exercises and add them to the favorite section for quick access, with the possibility of attaching demonstration videos.

[1]Body mass index

1.3 Non-Functional Requirements

- **Cloud Deployment:** The system is designed for deployment in a cloud environment, allowing for flexibility, scalability and reliability. It supports seamless deployment processes, automatic updates and efficient resource management.
- **Compatibility:** The system is compatible with various devices and browsers.
- **Maintainability:** The system is easy to modify and update, allowing for the addition of new functionalities and the correction of errors efficiently.
- **Scalability:** It is capable of handling an increase in the number of users or data volume without compromising performance.
- **Portability:** The application is able to function on different platforms, being responsive.

1.4 Development Methodology

The development methodology used in this project was *Parallel Development*[1]. This methodology involves separating the frontend and backend components by team members, enabling each member to develop their part simultaneously, as opposed to the traditional sequential development approach where the backend and all its components are developed first before the frontend. This approach aims to reduce development time since team members can work independently without waiting for the other component to be ready.

To implement this methodology, the team members decided which features would be developed in each component, following a sequential order of development. This allowed each member to start developing their respective part. Once both the frontend and backend components were ready, integration of the two components took place. This enabled testing of the application as a whole, allowing for the identification and correction of any errors or potential improvements.

1.5 Report Organization

This report is organized into 6 chapters. In Chapter 2, we discuss the project architecture, including the block diagram, backend organization, and technologies used in both the frontend and backend. Chapter 3 provides a detailed examination of the backend side of the project, covering aspects such as database structure, HTTP API, services, error handling and authentication.

Chapter 3 provides the key aspects of backend development, crucial for understanding the system's architecture and functionality. It starts with an overview of the project structure

and relational database schema, followed by the backend's interaction with the database. The repository methodology used for data access and the service layer for business logic are explained. The HTTP API covers endpoint handlers and routing, while authentication details the token generation and validation. Error handling, trainee data management, and measurement techniques are described. The notification system and processes for creating workouts and sets are covered, along with testing strategies and the backend deployment process.

Chapter 4 addresses the frontend development, starting with client routes. It continues with the discussion on Vue components and properties injection, followed by an overview of the frontend project structure. Client-server communication is explained next, highlighting how the frontend interacts with the backend. The chapter concludes with a quick explantion about deployment process for the frontend system.

Chapter 5 focuses on the user interface, presenting the different views of the application and the user interactions with the system. It includes images of the application's main screens.

The final chapter, Chapter 6, presents the conclusions of the report. It summarizes the completed function requirements and outlines potential future work.

Chapter 2

Architecture

In this section, will be presented the overall architecture of the project including the main components and their interactions, as well as the technologies used and the reasons behind their choice.

2.1 Main Components

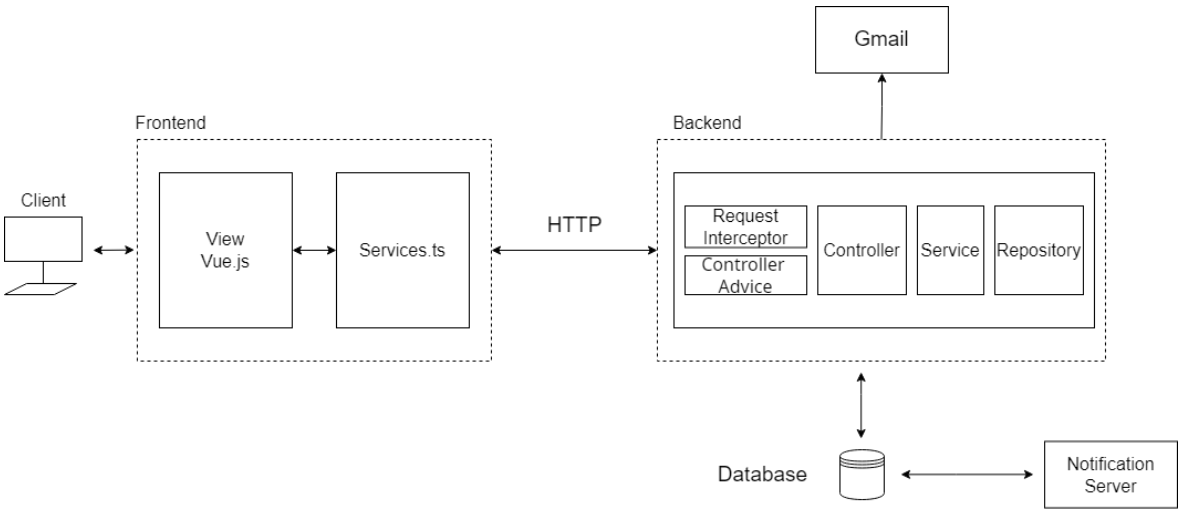

Figure 2.1: Application Main Components

In Figure 2.1, the application architecture is illustrated, highlighting the three main sections: the frontend, the backend and the notification server. The frontend is responsible for presenting the application interface to the user, while the backend manages the business logic and data storage.

The frontend is further divided into two primary components: Views and Services. **Views** are implemented using Vue.js and are responsible for the user interface, including components (reusable parts of the application) and pages (the main views of the application). **Services** handle communication with the backend, processing requests and responses.

The backend consists of five main components: Controllers, Services, Repositories, an

Authentication Interceptor, and an Error Handler. The **Controllers** handle client requests route requests to the appropriate Services and send responses back to the client. **Services** validate incoming data, execute business logic (such as creating users, sessions, or training plans), and interact with Repositories for data storage and retrieval. **Repositories** manage direct interactions with the database, executing queries as needed.

The **Authentication Interceptor** verifies user authentication and permissions for accessing requested resources. The **Error Handler** captures exceptions thrown by the application, returning appropriate responses with corresponding status codes and messages.

Additionally, the system includes a **Gmail** service block, which the application uses to send emails to users.

Finally, the notification server is responsible for sending to the users email notifications about their sessions for the next day.

2.2 Technologies

The backend technologies included **Kotlin**[2] with **Spring Boot**[3], and **PostgreSQL**[4] as the *DBMS* [1]. Additionally, **JDBI**[6] was used as the database access library.

The technologies chosen for this project were based on the group's experience and the project's requirements and goals. The main goal was to achieve fast and reliable development, so the technologies were selected accordingly. The technologies used in the frontend were **Vue.js**[7] together with **Vite**[8], using **TypeScript**[9] as the main language. **Vuex**[10] was also used to manage state, providing context instead of passing all the data through props.

Finally the **Render**[11] was used to deploy the application and the notification server.

2.2.1 Backend

In the backend, the framework used was **Spring Boot** together with the **Kotlin** programming language, chosen because of the group's experience with these technologies and the fast development they provide. Additionally, the **JDBI** library was chosen for database access due to its simplicity and efficiency in interacting with **PostgreSQL**. **PostgreSQL** was selected because a relational database was needed to store the system data, and it is an open-source database management system known for its reliability and robustness.

To complement the backend technologies, the libraries *JJWT*[12], *spring-boot-starter-mail*[13] (*JavaMailSender*) and **javax.mail**[14] were used for the authentication system to generate and validate **JWT**[2] tokens and to send emails.

Finally, the library **Mockito**[15] was used for mocking the database access module in

[1] "*A Database Management System (DBMS) is a software system that is designed to manage and organize data in a structured manner. It allows users to create, modify, and query a database, as well as manage the security and access controls for that database.*"[5]

[2] JSON Web Token

tests, and the **Quartz**[16] library for scheduling database access and email sending from the notification server.

2.2.2 Frontend

To develop the frontend side of this project, we chose the framework **Vue.js**, a progressive JavaScript framework used for building user interfaces and **single-page applications** (SPAs). We also used **Vite**, a modern build tool and development server designed to provide a fast and efficient development experience for modern web projects.

Additionally, we employed a **Vue library** called **Vuex**, which is a state management library for **Vue.js** applications. Vuex works as a store for all the components in an application, with rules that ensure **state** changes occur in a predictable manner.

Vue.Js vs React

Some of the reasons why Vue was chosen instead of React were:

1. **State Management: Vue.js** has an integrated and intuitive state management system using its built-in reactivity and **Vuex**, which is the official state management library. **React.js** provides effective component-level state management with **hooks** and the **Context API** for global state but often relies on additional libraries like **Redux** and **MobX** for more complex state management needs.

2. **Performance:** Both Vue.js and React.js utilize a **virtual DOM** to enhance rendering performance. However, Vue.js offers a more efficient rendering system through its compiler, minimizing the need for manual optimization.

3. **Syntax:** Vue.js uses **HTML templates** and **JSX** to create components, while React.js relies exclusively on JSX, allowing a faster render times.

4. **Data Binding: Vue.js** uses **two-way** data binding by default, which means that **every time** a change is made in the UI it will be **automatically** reflected on the data and **vice-versa**. **React.js** only uses **one-way** data binding, **requiring** the **update** of the UI every time the data changes.

Vite vs Webpack

Vite was also chosen instead of Webpack with some reasons being:

1. **Development Experience:**
    - **Vite**: Fast startup, hot module replacement (**HMR**[3]) out of the box, faster feedback due to native **ESM** [4] support. Ideal for rapid development iterations.
    - **Webpack**: Slower startup due to pre-compilation, potentially longer build times for large projects.
2. **Bundling:**
    - **Vite**: Vue.js often produces smaller bundles thanks to code-splitting and directly serving code, leading to faster loading times.
    - **Webpack**: Larger bundle sizes due to traditional bundling approach.
3. **Optimized Production Builds:**
    - **Vite**: Uses Rollup for smaller production bundles.
    - **Webpack**: Requires additional configuration and plugins to achieve the same level of optimization as Vite.

[3]Hot Module Replacement is a feature that allows developers to update modules in a running application without requiring a full reload.

[4]ESM or ECMAScript Modules allow developers to organize and reuse code across different files and projects

Chapter 3

Backend

In this chapter, some development approaches will be discussed. The main goal is to present the development methodology, the project structure, the relational database, the database access, the services, the WEB API, the authentication system and the error handling.

3.1 Project Structure

The project structure for the backend is divided into two main parts: the source code for the main server and the source code for the notification server. The main server is responsible for handling requests from the frontend, while the notification server is responsible for sending notifications to users.

The main server source code is located in the `code/jvm` folder. This code is organized into the following packages:

- **domain:** contains the domain classes for the application
- **http:** contains the classes responsible for handling HTTP requests
- **service:** contains the classes responsible for the business logic
- **repository:** contains the classes responsible for database access

The notification server source code is located in the `code/jvm-notification-server` folder. This code is organized into the following packages:

- **jobs:** contains the classes responsible for accessing the database and sending notifications
- **model:** contains the classes responsible for the notification data
- **repository:** contains the classes responsible for database access
- **service:** contains the class responsible for sending notifications

3.2 Relational Database

To store the system data, such as user information, authentication data, associations between users (e.g. trainer responsible for the trainee or company where the trainer works), training plans, and their respective sessions, a relational database management system was used. PostgreSQL was chosen, as mentioned in section 2.2. PostgreSQL is an open-source relational database management system known for its reliability and robustness.

3.2.1 User

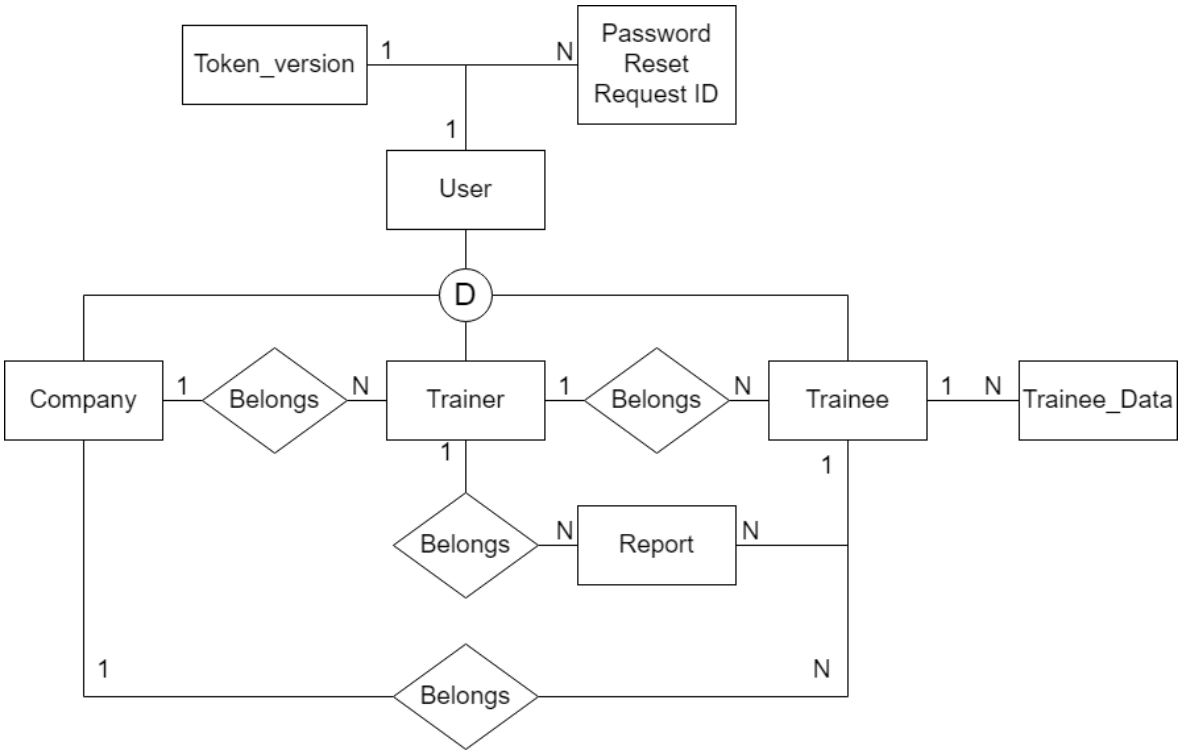

Figure 3.1: User Entity-Relationship Diagram

The model in Figure 3.1 represents the main entities of the system, related to the system users and their respective authentication data. The User entity is divided into three different types, each represented by a different entity with specific attributes (`trainee`, `company` and `trainer`), in addition to the attributes common to all users (e.g. email, password, name, etc.).

These entities are related to each other to represent the relationships between system users, such as the relationship between a trainee and their responsible trainer, or the relationship between a company and the trainer who works for that company. In the case of a trainer hired by a company, the relationship between the company and the trainer not only relates the users through unique identifiers but also stores the training capacity stipulated by the company that hired the trainer.

For the **trainee** entity, in addition to the information common to all users and the information specific to this type of user, it will also be associated with a **trainee_data** entity and a **report** entity. The **trainee_data** entity stores the trainee's data collected by the responsible trainer and the respective date of collection, such as weight, height, percentage of body fat, etc. The **report** entity stores the reports prepared by the responsible trainer, along with the creation date of these reports, which include information about the training and the trainee's progress. Depending on the **visibility** field, these reports can be viewed by the trainee or not.

Finally, there are two additional tables related to the user. The **Token_Version** table stores the version of the `refresh token` and `access token`. This version is incremented whenever the user changes their password, invalidating tokens with the old version on the server. The **Password Reset Request ID** table stores the request ID generated by the forget password method. This ID is used in the method that allows the user to change their password based on the forget password request.

3.2.2 Workout

The model in Figure 3.2 represents the entities related to the training sessions and the possible feedback associated with these sessions and/or respective sets.

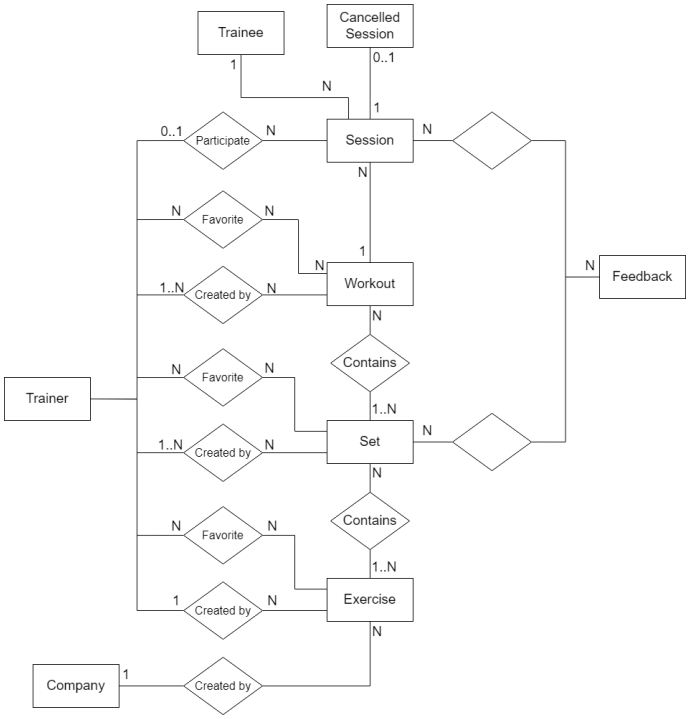

Figure 3.2: Workout Entity-Relationship Diagram

The Figure 3.2 presents the five main entities related to training sessions: **session**, **workout**, **set**, **exercise** and **feedback**. The **session** entity represents a training session, which

includes a training plan (**workout**), the unique identifier of the trainee participating in the session, the identifier of the workout associated with the session, the start and end dates of the session, the type of session (indicating whether it is guided by a trainer) and any notes the trainer wants to leave for the trainee. If the session is `trainer-guided`, the responsible trainer is also associated with the session. If either the trainer or the trainee cannot participate in the session, they can cancel it by creating an entry in the `cancelled session` table.

The **workout** entity represents a training plan, which is composed of a series of **sets**. Each training plan includes the name of the plan for future reference, a possible description of the plan's contents and an array of up to three `muscle_group` elements, indicating the muscle groups the plan targets. This entity is related to the `set` entity, where the unique identifiers of both the workout and the set are stored, along with an integer representing the order of the set within the training plan.

The **set** entity represents a series of exercises to be performed by the trainee. This entity includes a unique name for identification, a field for any notes the trainer wants to convey to the trainee regarding the set and a type indicating whether the set is a **Dropset**, **Superset**, or **Simpleset**. The **set** entity is linked to the **exercise** entity, which represents the exercises that comprise the set. It stores the unique identifiers of the set and the exercise, an integer indicating the order of the exercise within the set, and the exercise execution details in `JSON` format, as these details may vary based on the type of exercise and set.

The **exercise** entity stores the name and description of the exercise, a set of up to three muscle groups that the exercise targets, a modality indicating whether the exercise involves strength with weights, strength without weights, cardio, or other types, and a field to store a possible link to a video demonstrating the exercise.

In addition to these entities, there are entities representing possible favorite exercises, sets, and workouts for the trainer. There is also an entity for the feedback that the trainee and/or trainer can provide regarding the training session or set.

For more detailed diagrams, see Appendix Figure A.1 and Figure A.2.

3.3 Relational Database Access

As mentioned in Section 2.2, the **JDBI** library was used to access the **PostgreSQL** *DBMS*. **JDBI** is a library that facilitates interaction between the database and the server application. It provides a simple and efficient API based on lambda expressions and reflection, allowing for seamless database interaction.

To enable interaction with the database, two interfaces were created: **Transaction** and **TransactionManager**. These interfaces were implemented by the classes **JdbiTransaction** and **JdbiTransactionManager**, respectively, following the principles learned in the Web

Application Development (DAW) course, for database interaction using the `JDBI` library.

```
interface Transaction {
    val authRepo: AuthRepo
    (...)
    fun commit()
    fun rollback()
}
```

Listing 3.1: Data transaction interface

In Listing 3.1 the **Transaction** interface defines the contract for various data repositories that can be accessed, such as the `AuthRepo`. Additionally, this interface specifies two transaction control methods, *commit* and `rollback`, which are implemented in the **JdbiTransaction** class.

```
interface TransactionManager {
    fun <R> run(block: (Transaction) -> R): R

    fun <R> runWithLevel(level: TransactionIsolationLevel, block: (Transaction) -> R):
        R
}
```

Listing 3.2: Data transaction manager interface

The **TransactionManager** interface, showned in Listing 3.2, defines methods for executing blocks of code within a transaction context. These methods include **run** and **runWithLevel**, which are implemented in the **JdbiTransactionManager** class. The **run** method executes a block of code using the default transaction isolation level of the database, which in this case is **Read-committed** for *PostgreSQL*. The **runWithLevel** method, on the other hand, allows for executing a block of code using a specified isolation level.

In addition to the **Transaction** and **TransactionManager** interfaces, several mapper classes were created to parse specific types from the database into Kotlin types or objects that cannot be converted automatically.

```
class DateMapper : ColumnMapper<Date> {
    override fun map(r: ResultSet, columnNumber: Int, ctx: StatementContext?): Date {
        return Date(r.getTimestamp(columnNumber).time)
    }
}
```

Listing 3.3: Data mapper example

In Listing 3.3, the **DateMapper** class is responsible for parsing a Timestamp from the database into a Date object in Kotlin. This class implements the **ColumnMapper** interface, which maps a column from the database to a Kotlin object, specifically a Date object in this case.

3.4 Repository

In the **repository** package, is composed by the implementation of the **Transaction** and **TransactionManager** interfaces, as well as sets of interfaces and classes that represent the repositories of the application. These repositories are responsible for interacting with the database and performing operations such as inserting, updating and deleting data.

This project includes eleven repositories, each dedicated to managing a specific type of data. These repositories are:

- **AuthRepo:** Manages user authentication data.
- **UserRepo:** Manages user data.
- **CompanyRepo:** Manages company data.
- **ExerciseRepo:** Manages exercise data.
- **WorkoutRepo:** Manages workout data.
- **SetRepo:** Manages set data.
- **SessionRepo:** Manages session data.
- **ReportRepo:** Manages report data.
- **TraineeRepo:** Manages trainee data.
- **TraineeDataRepo:** Manages trainee measurement data.
- **TrainerRepo:** Manages trainer data.

```
1  class JdbiTransaction(private val handle: Handle) : Transaction {
2      override val authRepo: AuthRepo = JdbiAuthRepo(handle)
3      (...)
4
5      override fun commit() {
6          handle.commit()
7      }
8
9      override fun rollback() {
10         handle.rollback()
11     }
12 }
```

Listing 3.4: JdbiTransaction Implementation.

```
class JdbiTransactionManager(
    private val jdbi: Jdbi
) : TransactionManager {
    override fun <R> run(block: (Transaction) -> R): R =
        jdbi.inTransaction<R, Exception> { handle ->
            val transaction = JdbiTransaction(handle)
            block(transaction)
        }

    override fun <R> runWithLevel(level: TransactionIsolationLevel, block: (Transaction
        ) -> R): R =
        jdbi.inTransaction<R, Exception>(level) {
            val transaction = JdbiTransaction(it)
            block(transaction)
        }
}
```

Listing 3.5: JdbiTransactionManager Implementation.

The **JdbiTransaction** class, shown in Listing 3.4, implements the **Transaction** interface and provides access to the repositories. This class is responsible for committing or rolling back transactions. The **JdbiTransactionManager** class, shown in Listing 3.5, implements the **TransactionManager** interface and is responsible for managing transactions and executing blocks of code within a transaction context.

```
interface AuthRepo {
    fun createUser(name: String, email: String, passwordHash: String, role: Role): UUID

    fun createCompany(id: UUID)

    (...)
}
```

Listing 3.6: AuthRepo interface.

```kotlin
class JdbiAuthRepo(private val handle: Handle) : AuthRepo {
    override fun createUser(
        name: String,
        email: String,
        passwordHash: String,
        role: Role
    ): UUID =
        handle.createUpdate(
            """
            insert into "user" (name, email, password_hash, role)
            values (:name, :email, :passwordHash, :role::role)
            """.trimIndent()
        )
            .bindMap(
                mapOf(
                    "name" to name,
                    "email" to email,
                    "passwordHash" to passwordHash,
                    "role" to role.name
                )
            )
            .executeAndReturnGeneratedKeys("id")
            .mapTo<UUID>()
            .one()

    override fun createCompany(id: UUID) {
        handle.createUpdate(
            """
            insert into company (id)
            values (:id)
            """.trimIndent()
        )
            .bind("id", id)
            .execute()
    }

    (...)
}
```

Listing 3.7: JdbiAuthRepo implementation.

The **AuthRepo** interface, shown in Listing 3.6, defines the contract for the authentication repository. This interface specifies methods for creating users and companies, among other operations. The **JdbiAuthRepo** class, shown in Listing 3.7, implements the **AuthRepo** interface and is responsible for interacting with the database to perform operations such as creating users and companies, as well as other operations. This class uses the **Handle** object provided by the **JDBI** library to interact with the database, executing SQL queries and binding parameters.

3.5 Services

In this layer of the application, the services are responsible for implementing the business logic. Examples of operations performed by this layer include creating a new user, authenticating a user, recovering a password, creating a new workout, creating a new workout session, etc.

Here, some data received from the control layer undergoes further validation, which could not be fully validated by the deserialization library (`Jakarta`). These validations include checking if the email is in a valid format, if the password meets the minimum size requirement or if the name is not an empty string. To perform these validations, the `javax.validation` library[17] was used, enabling simple and efficient data validation, as illustrated in Listing 3.8

```
data class Company(
    @field:NotEmpty(message = "Name cannot be empty.")
    val name: String,

    @field:NotEmpty(message = "Email cannot be empty.")
    @field:Email(message = "Invalid email address.")
    val email: String,

    @field:Size(min = 8, message = "Password must have at least 8 characters.")
    val password: String
)
```

Listing 3.8: Request body object

In this example, the **Company** class has three attributes: `name`, `email`, and `password`. The `name` attribute cannot be an empty string, the `email` attribute must be in a valid email format and not be empty, and the `password` attribute must have at least 8 characters. Since these checks are already performed during the data deserialization process, there is no need to repeat them in the service layer.

To create the services, the `@Service` annotation from `Spring` was used. This annotation allows the creation of a service that is managed and instantiated automatically, using reflection. It also enables the injection of dependencies required by the service, such as the `TransactionManager` for database interaction, as illustrated in Listing 4.4.

```
@Service
class AuthService(
    private val authDomain: AuthDomain,
    private val jwtService: JwtService,
    private val mailService: MailService,
    private val transactionManager: TransactionManager
) { ... }
```

Listing 3.9: Service example

In the example provided in Listing 4.4, the **AuthService** service has four dependencies: **authDomain**, **jwtService**, **mailService** and **transactionManager**. These dependencies are automatically injected by `Spring`, enabling interaction with the database, generation of JWT tokens, sending emails and executing blocks of code within a transaction, respectively.

```
fun signUpCompany(
    name: String,
    email: String,
    password: String
) {
    transactionManager.run {
        val authRepo = it.authRepo
        val userRepo = it.userRepo

        if (userRepo.userExists(email)) {
            throw AuthError.UserRegistrationError.UserAlreadyExists
        }

        val passwordHash = authDomain.hashPassword(password)

        val userId = authRepo.createUser(name, email, passwordHash, Role.COMPANY)

        authRepo.createCompany(userId)
    }
}
```

Listing 3.10: Service function example

In Listing 3.5, an example of a method from the `AuthService` service is provided, demonstrate how the new company is created in the system. In this method, the **transactionManager** is utilized to execute a block of code within a transaction with a default isolation level. Within this transaction, a new user is created in the system with the role of `company`, and a new company is associated with the created user.

The service layer only needs to ensure that the received data is properly formatted for insertion into the database.

3.6 HTTP API

3.6.1 Endpoint Method Handlers and Routing

The **Spring Boot** framework uses an annotation-based approach to define application endpoints. The `@RestController` annotation designates a class as a controller, responsible for receiving HTTP requests and directing them to methods that process these requests. Additionally, the `@RequestMapping` annotation establishes the base path for all endpoints within the class. The `@GetMapping`, `@PostMapping`, `@PutMapping` and `@DeleteMapping` annotations specify the HTTP methods that the endpoints will handle, as demonstrated in Listing 3.11.

```
@RestController
@RequestMapping("/api")
class AuthController() {
    @PostMapping("/signup")
    fun signup(...): ResponseEntity<*> {...}
}
```

Listing 3.11: Routing example

In Listing 3.11, the `AuthController` class is defined as a controller responsible for receiving HTTP requests that start with the */api* path. Additionally, the *signup* method is defined to process `POST` requests made to the */api/signup* endpoint.

To the methods that need to have a parameter in their *URI*, it is possible to define the parameter in the `@GetMapping`, `@PostMapping`, `@PutMapping` or `@DeleteMapping` annotation. This parameter is then passed as an argument with the same name as the parameter defined in the annotation to the method that will process the request, as shown in Listing 3.12.

```
@RestController
@RequestMapping("/api")
class TrainerController() {
    @GetMapping("/exercise/{exerciseId}")
    fun getExerciseDetails(
        @PathVariable exerciseId: Int,
    ): ResponseEntity<*> { ... }
}
```

Listing 3.12: Routing example with variable parameter

If the client makes a **HTTP GET** request to the endpoint */api/exercise/1*, the request will be directed to the `getExerciseDetails` method of the `TrainerController` class. This method will process the request with the `exerciseId` equal to 1 and then return a response to the client.

3.6.2 Request Body Deserialization

To deserialize the data received in the request body, the `@RequestBody` annotation is used. This annotation is responsible for deserializing the data received in the request body into an object of the type defined in the method parameter, as shown in Listing 3.13.

```
@RestController
@RequestMapping("/api")
class AuthController() {
    @PostMapping("/signup")
    fun signup(
        @Valid @RequestBody
        userInfo: SignupRequest
    ): ResponseEntity<*> { ... }
}
```

Listing 3.13: Example of a method with a request body

In the example above, the `signup` method receives a `SignupRequest` object as a parameter, which is deserialized from the data received in the request body. Additionally, the `@Valid` annotation is used to validate the data received in the request body, as defined in the `SignupRequest` class, as mentioned in Section 3.5.

For this method, the request body could come in two different formats: one for the **independent trainer** and another for **companies**. To handle this, the **SignupRequest** class can be structured with two different inner *data classes*, each with its respective fields, as shown in Listing 3.14.

```kotlin
@JsonTypeInfo(
    use = JsonTypeInfo.Id.NAME,
    include = JsonTypeInfo.As.PROPERTY,
    property = "user_type"
)
@JsonSubTypes(
    Type(value = SignupRequest.Company::class, name = "company"),
    Type(value = SignupRequest.Independenttrainer::class, name = "independent_trainer")
)
sealed class SignupRequest {
    @JsonTypeName("company")
    data class Company( ... ) : SignupRequest()
    @JsonTypeName("independent_trainer")
    data class Independenttrainer( ... ) : SignupRequest()
}
```

Listing 3.14: Example of a deserialization class with polymorphic types

The annotations used in Listing 3.14 were used for deserialization in this method:

- **@JsonTypeInfo**: class/property annotation used to indicate details of what type information is included in serialization;
- **@JsonSubTypes**: class annotation used to indicate sub-types of annotated type; necessary when deserializing polymorphic types using logical type names (and not class names);
- **@JsonTypeName**: class annotation used to define logical type name to use for annotated class.

Deserialization process

When Spring processes an HTTP request containing a JSON body, it employs the Jackson library to convert this body into a Kotlin object. During this conversion process, Jackson utilizes annotations such as **@JsonTypeInfo**, **@JsonSubTypes** and **@JsonTypeName** to determine the appropriate subclass for deserialization. These annotations help Jackson understand how to map the incoming JSON data to the corresponding Kotlin class based on the type information provided.

In the example of the `SignupRequest` class, previously mentioned, the deserialization process uses 3 annotations:

1. **@JsonTypeInfo** tells Jackson to analyze the value of the `user_type` property, present in the body of the request, to determine which subclass should be used in the deserialization process;

2. **@JsonSubTypes** maps the possible values for the `user_type` property to the respective subclasses of the `SignupRequest`;

3. **@JsonTypeName** specifies the name that will be used in the JSON to represent a certain subclass, which is later used in the property specified in the `@JsonTypeInfo` annotation.

When a request is received with a body, Jackson creates an instance of the appropriate subclass by analyzing the `user_type` property and following the mappings specified by the annotations. Afterwards, it populates the fields of this instance with the values present in the JSON payload.

This information was obtained from the Jackson documentation[18].

3.7 Authentication

The application uses **email and password-based authentication**. Upon login, the server creates a pair of **JWT** tokens: an access token and a refresh token. The access token is used for authenticating the user in the application, while the refresh token is used to obtain a new access token once the current one expires. The access token has a short expiration time, ensuring security, while the refresh token has a longer expiration time to allow seamless user experience without frequent re-authentication.

This authentication approach was chosen because is secure and efficient, since the **JWT** tokens are self-sufficient, that is, they do not need to be stored in the server, and the user information that is necessary to send to the server for authentication is stored in the token itself and encoded through the `HS256` algorithm and a secret, making the communication between the client and the server secure. This allows the server to be more scalable because it does not need to store any tokens of each authenticated client.

To improve this authentication system, the server stores a version identifier for the generated tokens in the database. When a user changes their password either due to a forgotten password or a routine update, the version identifier is updated in the database. Consequently, any tokens generated before this update are invalidated. As a result, if a user attempts to make an authenticated request with an outdated token, the request will be denied.

3.7.1 JWT Generation and Validation

For the implementation of JWT authentication, the **io.jsonwebtoken** library was used, which is a Java implementation of the JWT specification. This library handles the generation and validation of JWT tokens, enabling secure user authentication within the application.

Token Generation

The information of the user encapsulated in the *JWT* token is:

- **ID**: Unique identifier of the user;
- **Subject**: User's role;
- **Expiration**: Token's expiration date;
- **Version**: Version of the token.

The token is generated using this information along with a secret key, which is used to sign the token and ensure its authenticity.

For the generation of the token, a `HashMap` with a field named `version` and its respective value is created to store additional information such as the **ID**, **Subject** and **Expiration**. An object of type `JwtBuilder` is then constructed by calling the `Jwts.builder` method, which is responsible for generating the JWT token with the relevant information.

Using this `JwtBuilder` object, the `setClaims` method is called, receiving the previously created `HashMap`. The `setId`, `setSubject`, and `setExpiration` methods are then used to add the user's information to the token. Finally, the `signWith` method is called, which receives the signature algorithm and the secret key, returning the generated JWT token. The `JwtBuilder` object is then compacted to a `String` through the `compact` method.

Integration with Cookies

The approach used for integrating JWT with cookies goes beyond merely storing the JWT token value in the cookie; it also stores the token expiration date (the same date present in the JWT token). To complete this approach, the expiration date is stored in the cookie only to ensure that it is kept on the client side while it is still alive. After its lifetime ends, it is removed. Despite this, it is still necessary to make a token validation request to the API to verify if they are valid. Additionally, the cookie is set to be **HttpOnly**, **Secure**, **same-site: Strict**. This ensures it is only accessible by the server, sent exclusively over HTTPS connections, and protected against Cross-Site Request Forgery (CSRF) attacks.

Token Information Reading

The JWT token information is read by calling the `Jwts.parser.setSigningKey` method, which takes the token signature key and returns an object of type `Claims`. Subsequently, the `parseClaimsJws` method is invoked with the JWT token string, extracting the token information. Finally, the `body` parameter of the `Claims` object is used to retrieve the token information, formatted like a `HashMap`.

3.8 Authentication Request Flow

For each request made by the client to the server, the client must include both the access token and the refresh token in the request header, under the **Authorization** header, preceded by the word "Bearer". Alternatively, for requests made by the browser, the tokens can be sent in a cookie. Upon receiving the request, it is intercepted by the `AuthInterceptor` class, which checks whether the requested method requires authentication. If authentication is required, the class verifies the validity of the access token and determines if it has the appropriate permissions to access the requested resource based on its `role`.

If no access token is provided, the server checks for the existence and validity of the refresh token. If the refresh token is valid, a new pair of tokens is generated and sent to the client in the response.

```
class AuthInterceptor(...) : HandlerInterceptor {
    override fun preHandle(
        request: HttpServletRequest,
        response: HttpServletResponse,
        handler: Any
    ): Boolean {
        if (handler is HandlerMethod && isAuthRequired(handler)) {
            val tokenDetails = try {
                processTokens(request, response)
            } catch (e: Exception) {
                revokeCookies(response)
                throw e
            }
            verifyRole(handler, tokenDetails)
            if (handler.methodParameters.any { it.parameterType == AuthenticatedUser::
                class.java }) {
                AuthenticatedUserResolver.addUserTo(
                    AuthenticatedUser(
                        id = tokenDetails.userId,
                        role = tokenDetails.role
                    ),
                    request
                )
            }
        }
        return true
    }
}
```

Listing 3.15: Authentication Interceptor

When a user makes a request to the server, each request is intercepted by the `AuthInterceptor`. This interceptor checks first if the method is of the type `HandlerMethod`. It then verifies if the method or the class containing the method has the `AuthenticationRequired` annotation.

The token validation involves decoding the token to ensure it has not been altered, has not expired and was signed with the correct key. The `processTokens` function is called to handle this verification. This function retrieves the tokens from the cookies or headers and decodes the access token to process the request. If the `processTokens` method throws any exception, the server revokes any cookies on the client and returns an error to the client.

To verify if the authenticated user has the required role to access the method, the parameters from the `AuthenticationRequired` annotation are accessed. This annotation specifies the roles that have permission to access the method. If the array of `roles` is empty, it means the method accepts requests from any user. This allows to define the necessary roles for accessing all methods of a class or just a specific method. If the user possesses one of the required roles, the request is processed; otherwise, an authorization error is returned.

Finally, if the method requires the `userId` and `role` from the authenticated user, information contained in the JWT, an `AuthenticatedUserResolver` is used to provide this information to the method.

3.9 Error Handling

The error handling system is designed to return a response to the client with a specific status code and message based on the error thrown by the application. This system uses the `@ControllerAdvice` annotation, which enables the creation of a class that intercepts exceptions thrown by the application. The class then returns a response to the client with the appropriate status code and message.

To create the error handling system, a class was created with the `@ControllerAdvice` annotation. This class contains methods that catch specific groups of exceptions and return a response to the client with the respective status.

```
@ControllerAdvice
class ExceptionAdvice {
    @ExceptionHandler(
        value = [
            IllegalArgumentException::class,
            IllegalStateException::class,
        ]
    )
    fun handleBadRequest(e: Exceptions): ResponseEntity<Problem> =
        Problem(
            type = URI.create(PROBLEMS_DOCS_URI + e.toProblemType()),
            title = e.message ?: "Bad Request",
            status = HttpStatus.BAD_REQUEST.value()
        ).toResponse()
}
```

Listing 3.16: Example of an Exception Handler

The `handleBadRequest` method is defined to catch exceptions of the types `IllegalArgumentException` and `IllegalStateException`. This method returns a response to the client in the form of a `Problem` object, which represents an issue that occurred in the application, including the respective status code and message. This response consists of an object with a type, title and status:

- **Type:** A URI representing the type of problem that occurred, pointing to the documentation of the problem.
- **Title:** The message of the exception thrown by the application.
- **Status:** The status code of the response.

3.10 Trainee Data

A body measurement system for trainees was created, where the responsible trainer enters the trainee's measurements, and which are processed by the server and stored in the database as a JSON object. This system allows the trainer to input measurements such as weight, height, body circumferences, body fat percentage and skinfold measurements.

Additionally, the trainer can provide the measurements, and the system will calculate the trainee's body composition, obtaining values for BMI, body fat percentage, fat mass and lean mass.

The following formulas are used to obtain these data:

- **BMI:** weight / (height * height)
- **Fat Mass:** weight * (bodyFatPercentage / 100)
- **Lean Mass:** weight - fat mass

3.10.1 Measurement Techniques

To calculate the trainee's body fat percentage, three measurement techniques were implemented, depending on the number of skinfolds measured. These measurements can be taken from 3, 4 or 7 skinfolds[19].

3-Site Skinfold Technique

The trainer must measure different skinfolds depending on the trainee's gender:

- **Male:** Pectoral, Abdominal, and Thigh
- **Female:** Triceps, Suprailiac, and Thigh

After obtaining these measurements, the body fat percentage is calculated using the following formulas:

- **Male:** 1.1093800 - 0.0008267 * sum_of_measurements + 0.0000016 * (sum_of_measurements^2) - 0.0002574 * age
- **Female:** 1.0994921 - 0.0009929 * sum_of_measurements + 0.0000023 * (sum_of_measurements^2) - 0.0001392 * age

4-Site Skinfold Technique

This technique uses the same skinfold measurements for both genders: Triceps, Thigh, Abdominal and Suprailiac, but with different formulas:

- **Male:** (0.29288 * sum_of_measurements) - (0.0005 * (sum_of_measurements^2)) + (0.15845 * age) - 5.76377
- **Female:** (0.29669 * sum_of_measurements) - (0.00043 * (sum_of_measurements^2)) + (0.02963 * age) + 1.4072

7-Site Skinfold Technique

This technique uses the same measurements for both genders: Pectoral, Abdominal, Thigh, Triceps, Suprailiac, Subscapular and Midaxillary.

The formula applied is: 1.112 - (0.00043499 * sum_of_measurements) + (0.00000055 * sum_of_measurements^2) - (0.00028826 * age)

With this system, trainers can conduct detailed and precise monitoring of trainees' physical progress, facilitating the customization of training and evaluations.

3.11 Notification System

This system is an application developed in **Kotlin** that leverages the **JDBI** library for database access, **javax.mail** for sending emails and **Quartz** for scheduling tasks. The purpose of the system is to access the database and retrieve all training sessions scheduled for the following day, organized by the trainer, for all users. Subsequently, the system sends an email to each user with details about their upcoming training sessions.

The system creates two jobs: one for trainers and one for trainees, with each job having a corresponding trigger. These jobs are scheduled to run daily at 6:00 PM. Each job fetches information from the database and sends an email to the users with their scheduled training sessions for the next day.

```kotlin
val traineesNotificationJob = JobBuilder.newJob(TraineesNotificationJob::class.java)
    .withIdentity("traineesNotificationJob")
    .build()
    .apply {
        jobDataMap["mailSender"] = mailService
        jobDataMap["sessionRepo"] = sessionRepo
    }
val traineesNotificationTrigger = TriggerBuilder.newTrigger()
    .withIdentity("traineesNotificationTrigger")
    .startAt(DateBuilder.todayAt(18, 0, 0)) // Schedule the trigger to start at 6:00 PM
    .withSchedule(SimpleScheduleBuilder.repeatSecondlyForever(60 * 60 * 24)) // Repeat
        every 24 hours
    .forJob(traineesNotificationJob)
    .build()
```

Listing 3.17: Job and Trigger example.

The Listing 3.17 shows the creation of the job and trigger for the trainees' notification. The job is instantiated using the `TraineesNotificationJob` class, and the job data map is utilized to pass dependencies to the job. The trigger is configured to initiate at 6:00 PM and to repeat every 24 hours.

```kotlin
class TraineesNotificationJob: Job {
    override fun execute(context: JobExecutionContext?) {
        val mailSender = context?.mergedJobDataMap?.get("mailSender") as? MailService
        val sessionRepo = context?.mergedJobDataMap?.get("sessionRepo") as? SessionRepo

        if (mailSender == null || sessionRepo == null) {
            throw IllegalStateException("Dependencies not found")
        }

        (...)
    }
}
```

Listing 3.18: Job implemantation example.

The Listing 3.18 shows the implementation of the `TraineesNotificationJob` class, which implements the `Job` interface. The `execute` method is invoked when the job is triggered. It retrieves the dependencies from the job data map and performs the necessary operations to obtain the training sessions and send emails to the users.

3.12 Workout and Set Creation

To enhance the efficiency of workout and set creation, a search functionality was implemented to identify any existing workout or set similar to the one the user is trying to create. This approach helps to avoid the creation of duplicate workouts and sets by associating them with the trainer who is creating the new workout or set.

3.12.1 Workout Creation

For workout creation, the system searches for existing workouts that have the same sets in the same order. When a user attempts to create a new workout, the system queries the

database to find workouts with identical sets and orders.

```
SELECT workout_id
FROM workout_set
GROUP BY workout_id
HAVING array_agg(set_id ORDER BY order_id) = :sets::integer[]
```

Listing 3.19: Finds workouts with the same sets in the same order.

This query groups sets by `workout_id` and checks if the ordered list of `set_ids` matches the sets the user is trying to create. If a match is found and the trainer does not already have the workout associated with them, the system will associate the existing workout with the trainer.

3.12.2 Set Creation

For set creation, the system searches for existing sets that have the same exercises in the same order and with the same details. When a user attempts to create a new set, the system performs two checks: one for the order of exercises and another for the details of each exercise.

The first query checks for the order of exercises:

```
SELECT set_id
FROM set_exercise
GROUP BY set_id
HAVING array_agg(exercise_id ORDER BY order_id) = :exercisesArray::integer[]
```

Listing 3.20: Identifies sets with exercises in the same order.

This query groups exercises by `set_id` and verifies if the ordered list of `exercise_ids` matches the exercises the user is trying to create.

The second query verifies the details of each exercise:

```
SELECT EXISTS(
    SELECT 1
    FROM set_exercise
    WHERE set_id = :setId
      AND exercise_id = :exerciseId
      AND (
        SELECT jsonb_object_agg(key, value)
        FROM jsonb_each_text(details)
      ) = (
        SELECT jsonb_object_agg(key, value)
        FROM jsonb_each_text(:details::jsonb)
      )
)
```

Listing 3.21: Checks for sets with matching exercise details.

This query checks if there is any row in the `set_exercise` table with the specified `set_id` and `exercise_id`, where the details `JSONB` column exactly matches the provided `:details` `JSONB` object.

Key PostgreSQL Expressions Used:

- **SELECT EXISTS(...):** Checks if at least one row matches the conditions inside the `EXISTS` clause. Returns true if such a row exists, otherwise false.
- **SELECT 1 FROM set_exercise:** This inner query selects rows from the `set_exercise` table. It doesn't matter what is selected (here it selects 1), as the `EXISTS` clause only cares about the presence of rows.
- **SELECT jsonb_object_agg(key, value) FROM jsonb_each_text(details):** Converts the details column, a `JSONB` object, into a set of key-value pairs and then re-aggregates it back into a `JSONB` object.
- **SELECT jsonb_object_agg(key, value) FROM jsonb_each_text(:details::jsonb):** Does the same as the previous expression but on the :details parameter to compare the details of the exercise ignoring the order of the keys.

3.13 Testing

Tests were conducted on the services using the **Mockito** library to simulate the database. These tests were designed to verify if the logic in these modules was correct and achieved the expected results.

For testing the remaining modules of the application, such as controllers, repositories, and the pipeline, the Postman app and the frontend application were used to ensure the responses from the server were correct and aligned with the planned outcomes.

3.14 Deployment

The deployment of the **server**, **notification server** and **database** was carried out using Render.

To deploy the **database**, a PostgreSQL instance was created via the Render interface. Subsequently, the necessary schemes and tables were set up using the SQL scripts located in the `code/sql` directory.

For the servers applications deployment, a Docker image was created using a Dockerfile in the respective directory. This image was then deployed as a Web Service on Render, with the required environment variables configured through the Render interface.

The following commands were used to create and deploy the Docker image for the server application:

1. `docker build -t docker_username/server .` - Build the image using the Dockerfile
2. `docker push docker_username/server` - Push the image to Docker Hub

Chapter 4

Frontend

In this chapter, we will discuss various development approaches. The primary goal is to present each client route from the frontend and explore the **Role-Based Access Control** (RBAC) approach for managing user access based on a role system. Additionally, we will explain how Vue components and property injection work, the file organization used and the client-server communication process.

4.1 Client Routes

(/) - Entry Point of the application.

4.1.1 Auth

- **/signup** - Page to sign up the user.
- **/login** - Page to login the user.
- **/forgetPassword** - In case a user forget his password this page requests the user email to send them the reset link.
- **/resetPassword/:token** - This page is used to reset the user password, from the user token received as a path variable.

4.1.2 Users

1. **Companies**
    - **/trainees** - Page that lists all trainees.
    - **/register-trainee/:isTrainee** - Page to register a new trainee
    - **/trainers** - Page that lists all trainers.
    - **/register-trainer/:isTrainer** - Page to register a new trainer.

- **/trainee/:traineeId/:assignTrainer** - Page that list all trainers available to associate to a trainee. The traineeId path variable specifies the trainee to be associated by their token. The assignTrainer path variable is a boolean indicating whether the operation is assigning a trainer (true) or reassigning a trainer (false).

2. **Hired Trainer/Independent Trainer**

- **/register-trainee/:isTrainee** - Page to register a new trainee
- **/trainees** - Page that lists all trainees.
- **/exercises** - Page that list all available exercises and allows to filter them by category, muscle groups or favourites.
- **/exercises/add-exercise** - Page that allows the creation of a custom exercise.
- **/sets** - Page that lists all available sets.
- **/sets/custom-set** - Page that allows the creation of a new set.
- **/sets/setDetails/:setId** - Page that shows the set details.
- **/workouts** - Page that lists all available workouts.
- **/workouts/add-workout** - Page that allows the creation of a new workout.
- **/sessions** - Page that shows all trainee or trainer sessions.
- **/sessions/:traineeId** - Page that shows all trainer sessions of a certain trainee.
- **/sessions/:traineeId/add-session** - Page that allows a trainer to add a new session.
- **/sessions/session/:sessionId** - Page that shows the details of a session.
- **/sessions/session/:sessionId/edit** - Page that allows the trainer to make changes in the session, for example date, workout, etc.
- **/sessions/:sessionId/cancel** - Page that allows a trainer or trainee to cancel a session.
- **/reports/:traineeId** - Page that allows a trainer to get all the trainee reports.
- **/reports/:traineeId/addReport** - Page that allows a trainer to add a new trainee report.
- **/reports/:traineeId/:reportId** - Page that allows a trainer to get the details of a report.
- **/:traineeId/profile** - Page that allows a trainer to see the trainee profile.
- **/trainee/:traineeId/data-history** - Page that lists the trainee body informations.

- **/trainee/:traineeId/data-history/:dataId** - Page that allows a trainer to see details of a previously added trainee body information.
- **/trainee/:traineeId/data-history/add** - Page that allows a trainer to add trainee body data.

3. **Trainees**

- **/sessions** - Page that shows all the trainee sessions.
- **/sessions/session/:sessionId** - Page that shows a trainee session details.
- **/sessions/:sessionId/cancel** - Page that allows a trainee to cancel a session.
- **/reports/:traineeId** - Page that shows all the trainee reports.
- **/reports/:traineeId** - Page that shows all the trainee reports.
- **/reports/:traineeId/:reportId** - Page that shows the report details of a trainee.
- **/:traineeId/profile** - Page that shows a trainee profile.
- **/trainee/:traineeId/data-history** - Page that lists all the trainee body data reports.
- **/trainee/:traineeId/data-history/:dataId** - Page that shows a trainee body data report details.

4. **Utils**

- **/email-sucess** - Page that shows a message to the user to check the email when resetting password.

This routes have been created and managed by the Vue router plugin, which for each route, creates a route object containing the URI, route name, the root component for that view and the meta field, which is an array of roles that have permission to access the view. With the meta field we use our own simple implementation of RBAC Listing 4.1 that verify if the user role stored in Vuex is included inside the meta field array of a Vue route shown in Listing 4.2.

```
import store from "../../../store"
class RBAC {
    static getUserRole(): string {
        return store.getters.userData.role
    }
    static isCompany(): boolean {
        return this.getUserRole() === "COMPANY"
    }
    static isTrainee(): boolean {
        return this.getUserRole() === "TRAINEE"
    }
    static isTrainer(): boolean {
        return this.getUserRole() === "INDEPENDENT_TRAINER"
    }
    static isHiredTrainer(): boolean {
        return this.getUserRole() === "HIRED_TRAINER"
    }
}

export default RBAC
```

Listing 4.1: RBAC

```
//route example
{ path: "/trainers", name: "trainers", component: Trainers, meta: { requiresAuth: true
    , roleNeeded: ["COMPANY"] } },

router.beforeEach((to) => {
    if (to.meta.roleNeeded) {
        const roles = to.meta.roleNeeded as string[]
        const rbac = RBAC
        const userRole = rbac.getUserRole()
        if (!roles.includes(userRole)) {
            return { name: "error" }
        }
    }
})
```

Listing 4.2: Example of RBAC use

Each root component is made with some child components in order to simplify the root component and take advantage of the reuse of those child components.

4.2 Vue Components and Properties Injection

Vue is based on React, where the properties of the component as in React are passed by the parent component. The props object is a parameter of each component that helps to pass all the information needed to be rendered on that component. The properties inside the component are defined by the parent component, which sends the values to these props. Vue has a system called with a name like called storage which is able to be executed for the data used in the app to become common to all the components in the app the same way React context system does. Vuex can be described as a collection of variables that represent the application state, and these variables will be updated only by defined mutations and actions.

The variables that depend on the state might be for instance an identifier(id) of a user, the mutations and actions are the functions that will update the state values, and the getters are the methods one can employ to know this variable current value. In addition, Vuex Persist is a library of Vuex that enables developers to maintain the context data even when a page refreshes, and these data are not erased.

In the Listing 4.3, an example of a Vue component implementation is shown and in Figure 5.7 the component is used to favorite set .

```
<template>
    <div @click="handleLike" class="like-box">
        <font-awesome-icon :class="[isLiked ? 'heart-icon-active' : 'heart-icon']" :
            icon="faHeart" />
    </div>
</template>

<script setup lang="ts">
import { faHeart } from "@fortawesome/free-solid-svg-icons"
import { FontAwesomeIcon } from "@fortawesome/vue-fontawesome"
import { ref } from "vue"
import { unlikeSet, likeSet } from "@/services/TrainerServices/sets/setServices.js"

const props = defineProps<{
    setId: number
    isFavorite: boolean
}>()

const isLiked = ref(props.isFavorite)

const handleLike = () => {
    if (isLiked.value) {
        unlikeSet(props.setId);
    } else {
        likeSet(props.setId);
    }
    isLiked.value = !isLiked.value;
};
</script>

<style scoped>
//Component CSS
</style>
```

Listing 4.3: Favorite box Component Example.

4.3 Project Structure

The Frontend project structure is represented by the following scheme:

The views are located in the src/views/ directory and are divided into auth, home, and user:

- **auth** - auth contains all views related to user authentication.
- **home** - contains all components related to the home page.

- **user** - is further divided into trainee views, trainer views, and company views.

There is a 'models' and 'components' subfolder for each root view and inside are the needed classes and child components correspondingly. The src/components directory is for components that do not belong to any one particular view e.g. the NavBar. In the src/assets directory one finds all the images used throughout the application amongst them being a folder containing examples of profile pictures named userIcon. There can be found the Vue router in src/plugins/router.ts where it has been defined to include routes together with router functions that are required. The services are contained within the src/services directory, which in turn is divided into subfolders, each named after the corresponding service. Inside these subfolders, there are files that contain the methods for communicating with the API. For example, the authService folder contains the authServices.ts file, which includes methods related to user authentication. The icons used on the application are from `FontAwesomeIcons`.

4.4 Client-Server Communication

The communication between the **client** and the **server** occurs through the **HTTP Web API** where requests are made in order to **send** or **get** data. These operations are present in the `src/services` directory as we mention before.

For example, if a trainer wants to favorite a exercise, they need to make an **API request** for the specified **URI** in the **API** as shown on Listing 4.4 with the necessary data, such as the exercise Id they want to favorite.

```
async function likeExercise(exerciseId: string): Promise<void> {
    const uri = `${apiBaseUri}/api/trainer/exercise/${exerciseId}/favorite`
    try {
        await fetchData(uri, "POST", null)
        return
    } catch (error) {
        console.error("Error liking exercise:", error)
        throw error
    }
}
```

Listing 4.4: Services example.

4.5 Deployment

Similar to the backend, the frontend is deployed on Render but it was deployed as a Static Site instead of a WebService. The deployment process is automated, triggering whenever a new commit is pushed to the `main` branch.

Chapter 5

User Interface

Our application is divided in 3 types of users: Trainers, Trainees and Companies so we develop a dynamic user interface based on the user role. On the figure bellow, Figure 5.1 is shown all the available paths a certain user can follow.

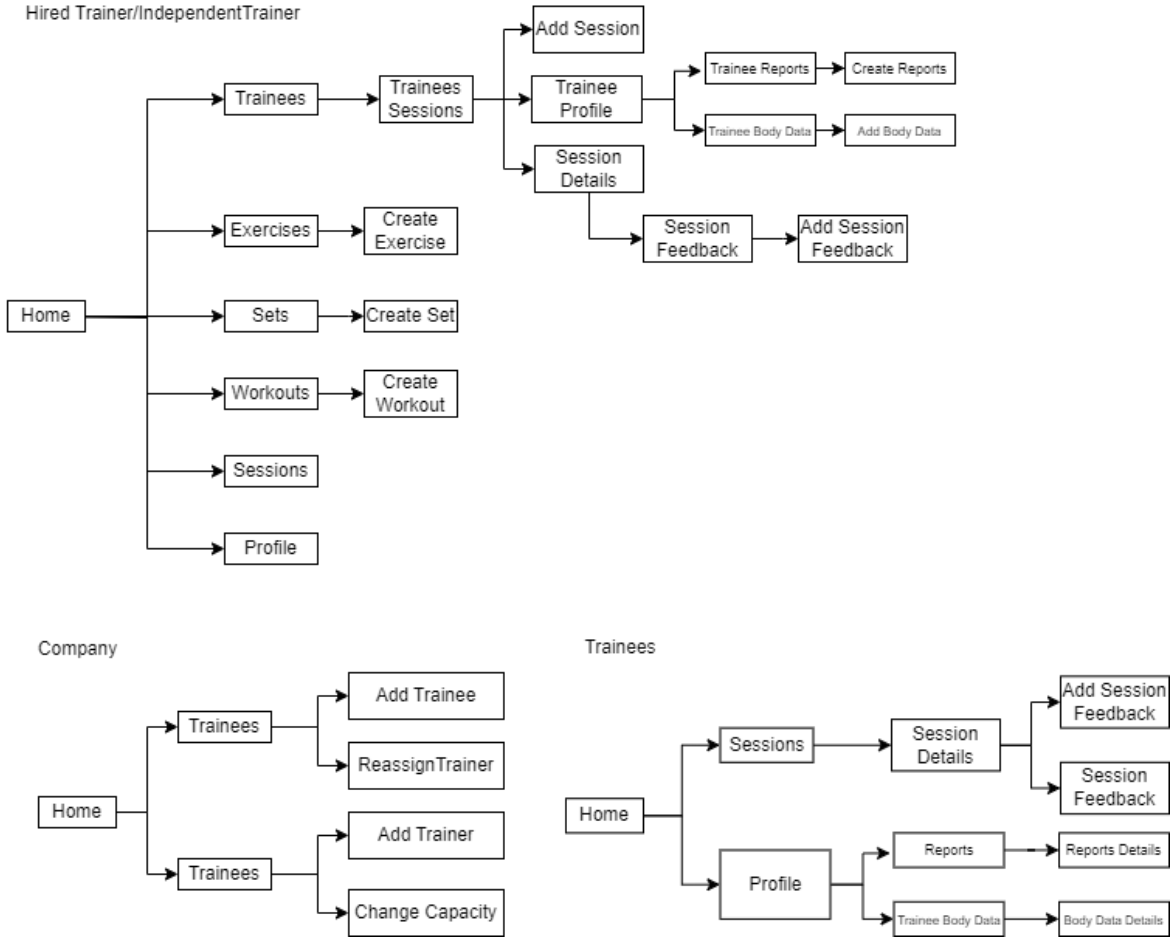

Figure 5.1: User Interface diagram

The main view of our web application is the home view shown in 5.2 where we have a calendar that will show the sessions scheduled with or without the Trainer.

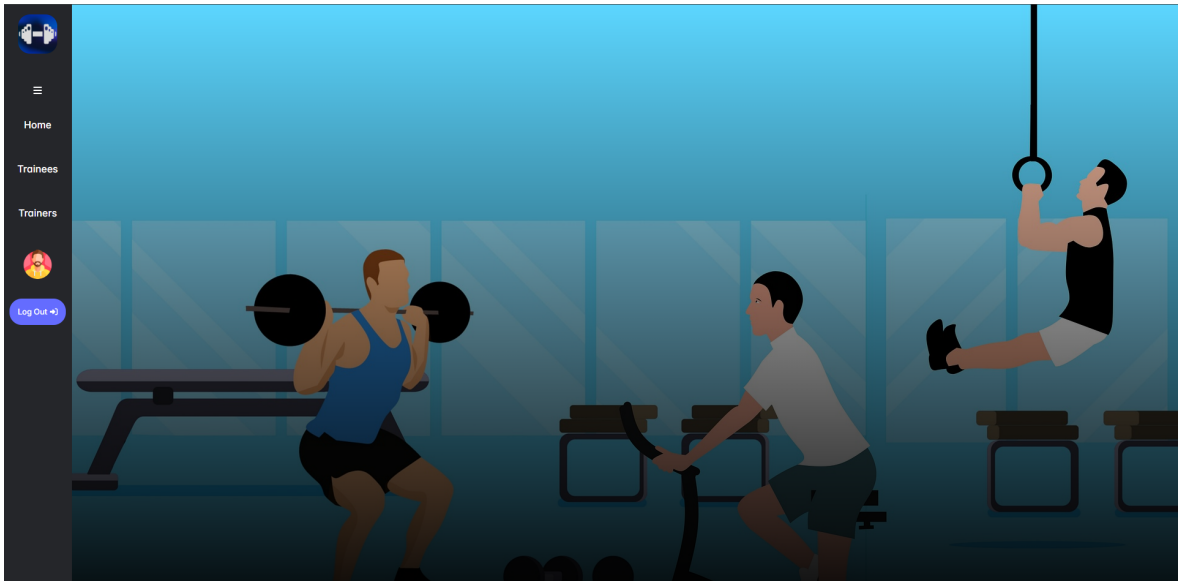

Figure 5.2: Home view page

5.1 Company

For the companies some views are important such as the Trainers and Trainees views shown in Figures 5.3 and 5.4 where a company can list, filter by gender or sort by availability the Trainers or Trainees of the company.

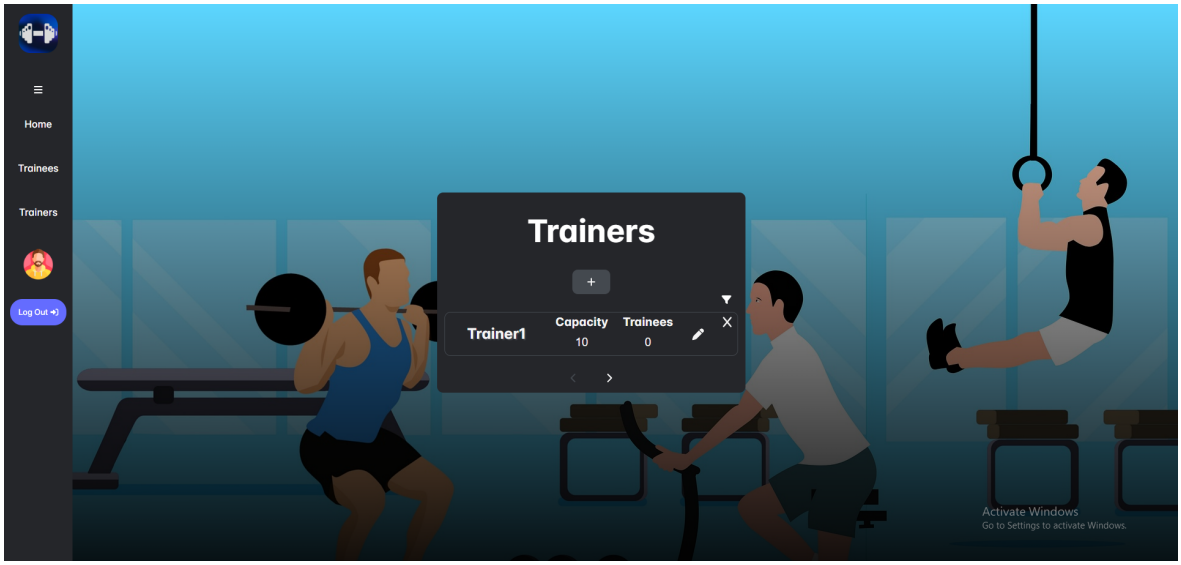

Figure 5.3: Trainees view page

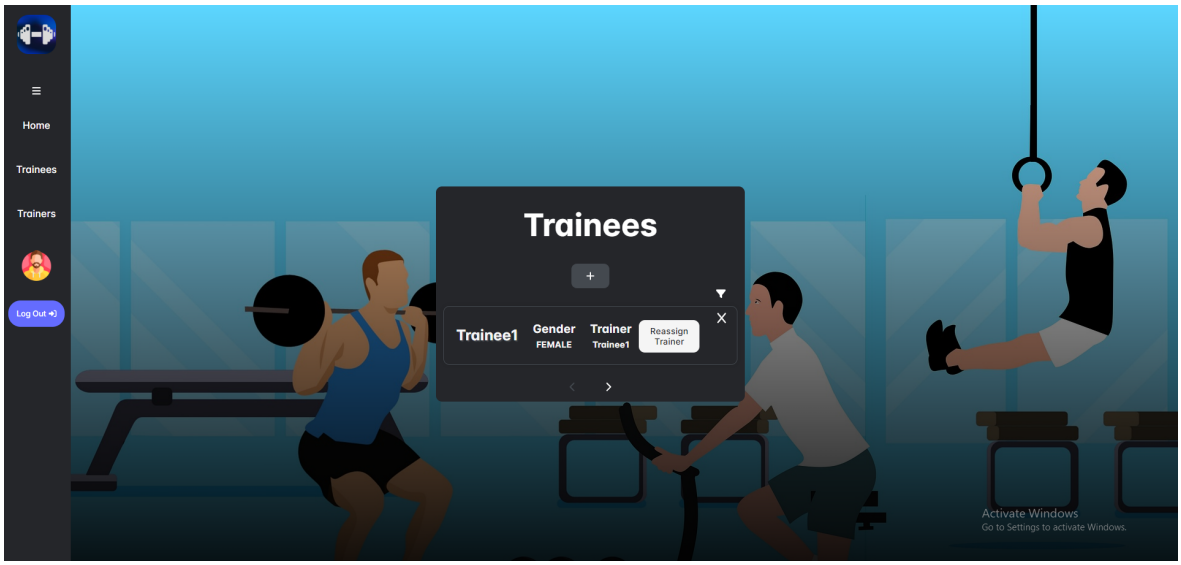

Figure 5.4: Companies view page

Companies can also assign or reassign a trainer, as shown in figure 5.4 which allows them to associate a trainee to a trainer or reassign the trainee to another trainer.

5.2 Hired Trainers, Independent Trainers and Trainees

Both the Hired Trainer and the Independent Trainer will have access to the Trainees view, but the Hired Trainer will not have the permissions to add, remove or assign trainees. Since a Independent company is working for himself, the assign and reassign on this view is also disabled.

They both have access to the exercises view shown in figure 5.5 where they can view all the exercises present on the database, filter them by name, muscle groups or modality and also create new ones that they like and they think are important shown in figure 5.6.

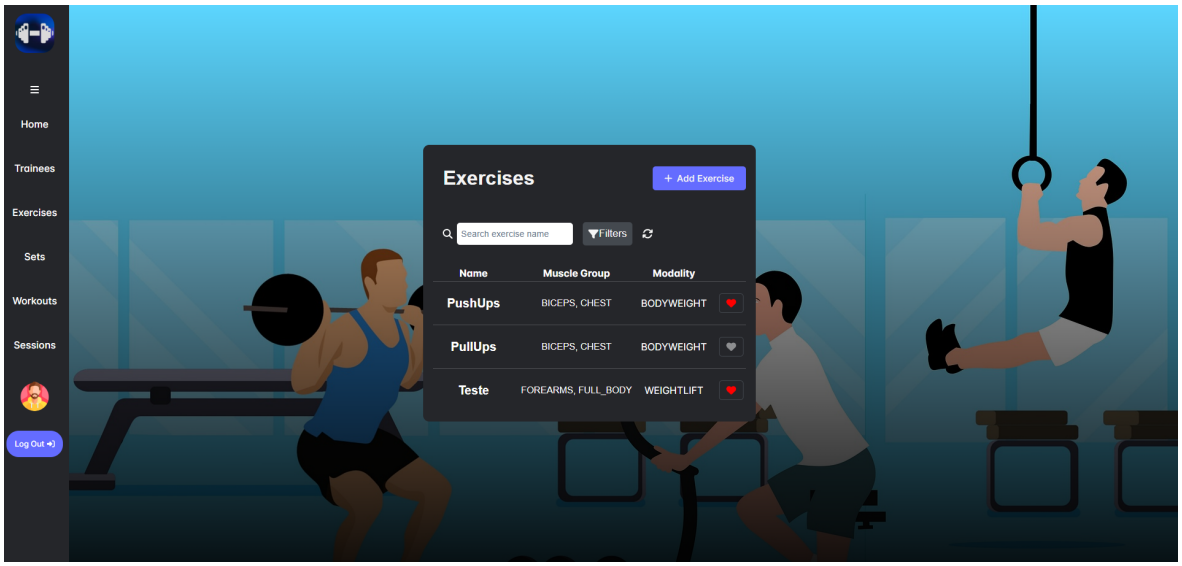

Figure 5.5: Exercises view page

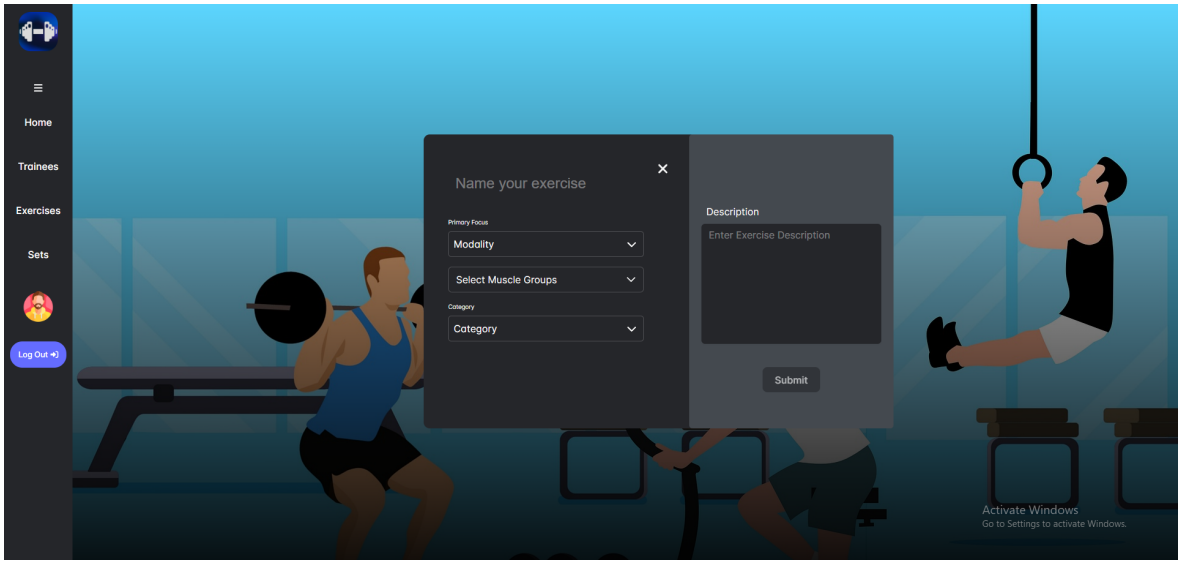

Figure 5.6: Create a new exercise view

The same applies for the Set view shown in figures 5.7, they can view all the sets available, filter them or create new ones shown in figure 5.8. They can also add a set or a workout to the favorites so its easier to find and use them.

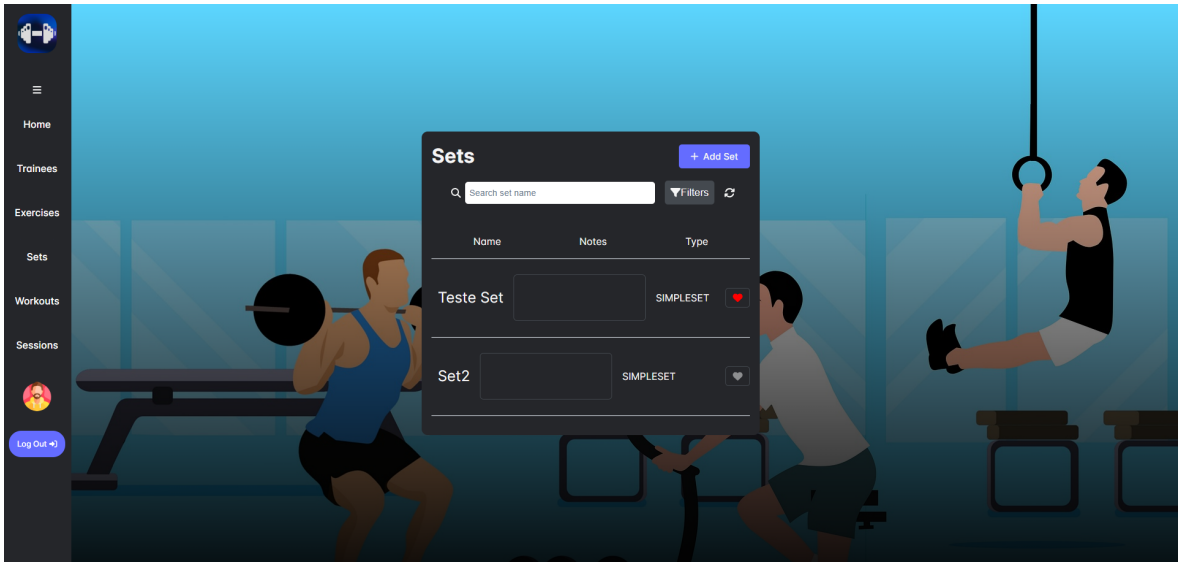

Figure 5.7: Sets view page

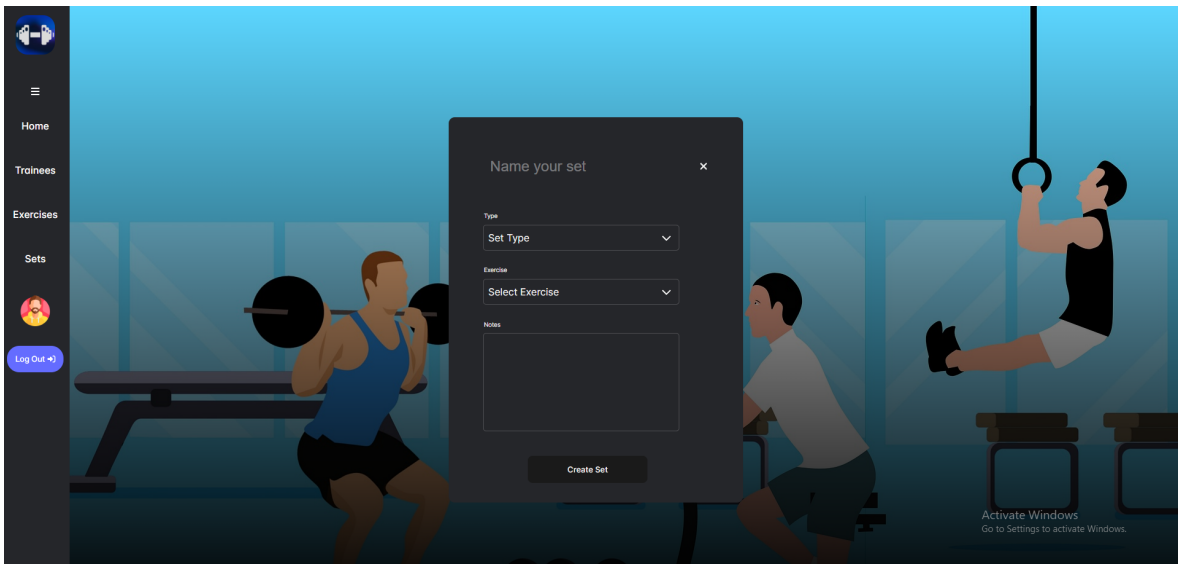

Figure 5.8: Create new set view

The Workouts page lists all the workouts available for the trainers to use on theirs sessions shown in figure 5.9, also on this page is possible to create or filter workouts as shown in figure 5.10 . As shown in figure 5.10 its possible to see that when creating a new workout, if the trainer wants to create a new set he is capable of doing that, or use a existing one instead.

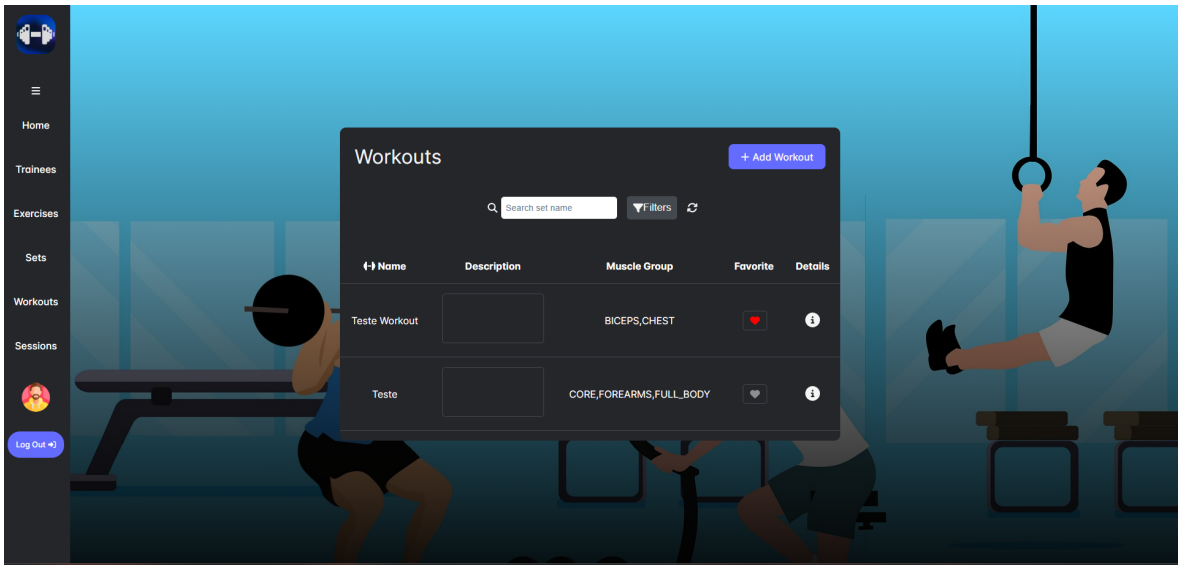

Figure 5.9: workouts view

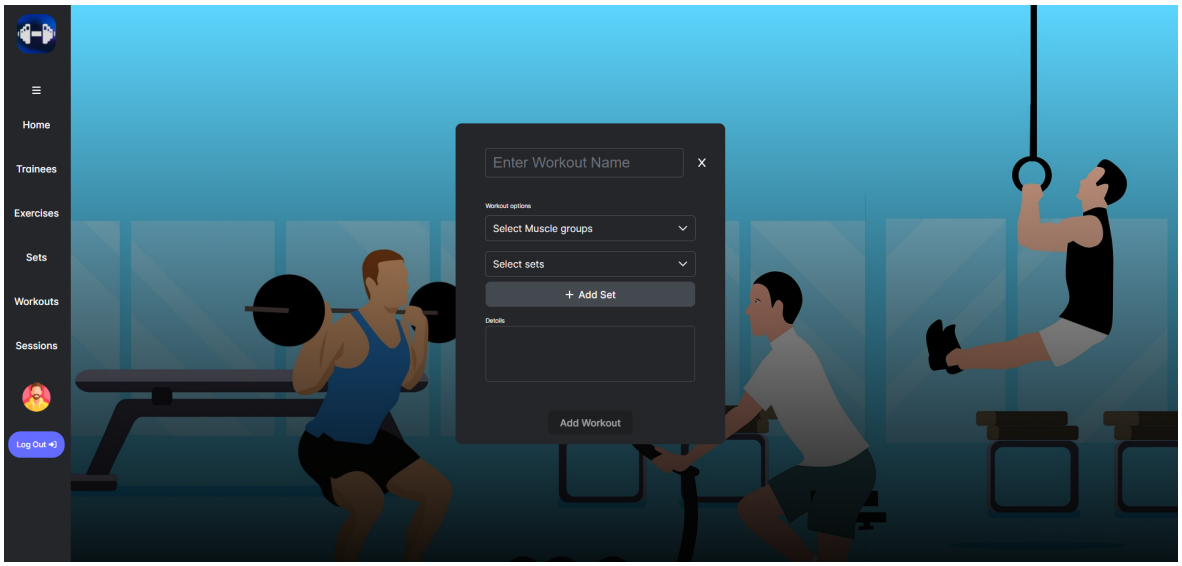

Figure 5.10: Create new workout view

On the Trainees page trainers could go to the Trainee Sessions page by clicking on the trainee they want to see from the trainees list.

On the this page the trainer and the trainee can see a calendar and a container that lists all the sessions they have and a profile box only the trainer see, Figure 5.11 Figure 5.12.

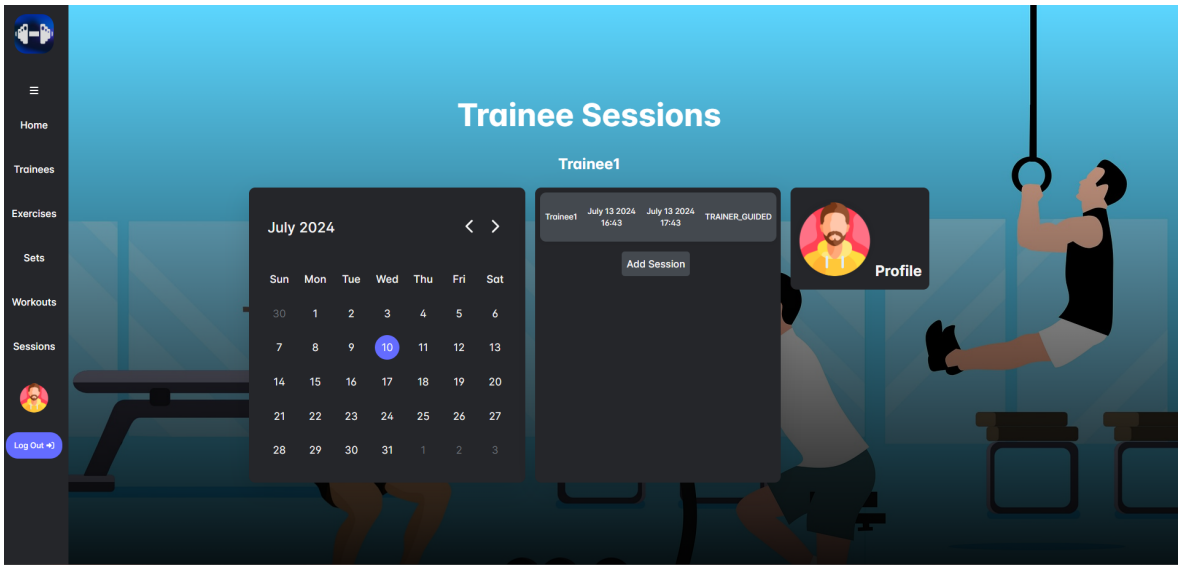

Figure 5.11: Trainee Sessions View- Trainer Perspective

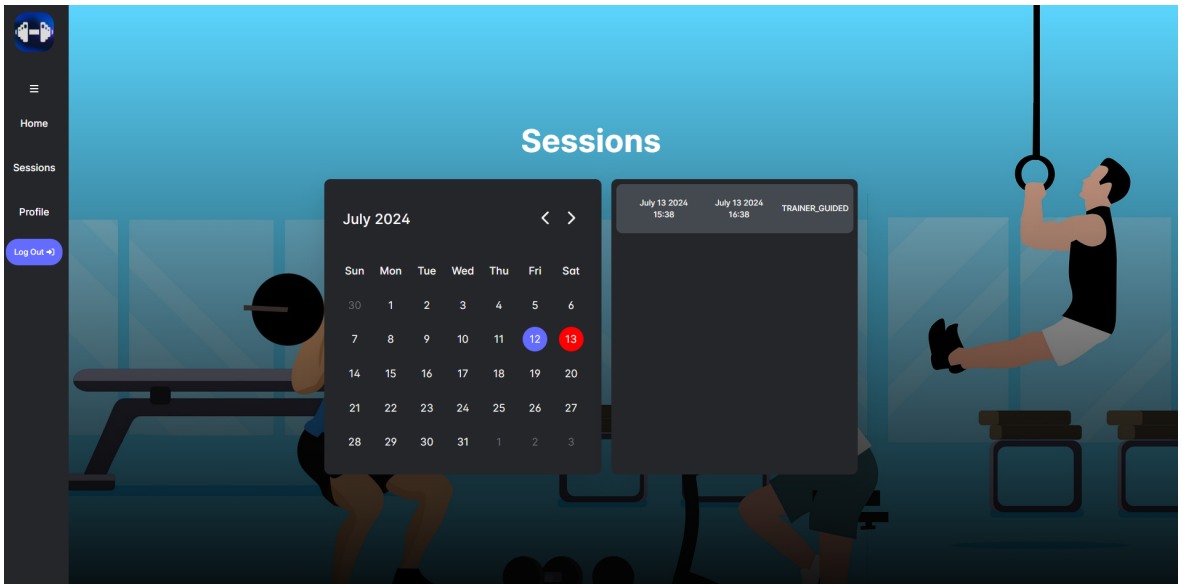

Figure 5.12: Trainee Sessions View- Trainee Perspective

The Trainers could also add a new session by clicking on the add session button on the sessions container and fill the required fields, figure 5.13.

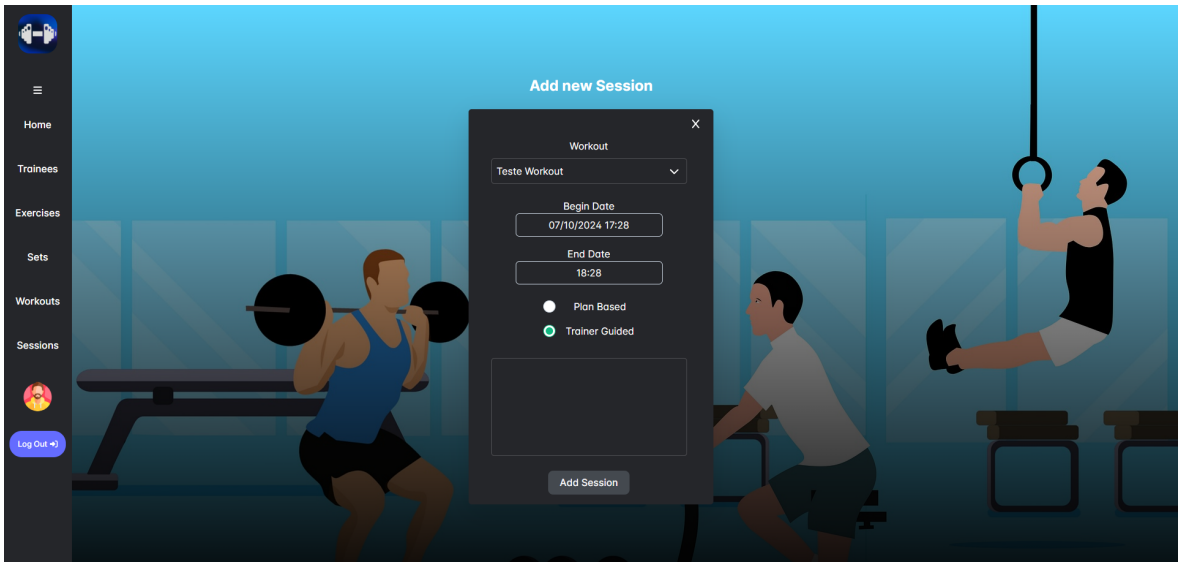

Figure 5.13: Session Details

The Trainer can see the session details by clicking on the session inside the container and edit the session or cancel it if stills 24 hours remaining till the session,figures 5.15 and 5.15, the Trainee could only cancel the session and view the details.

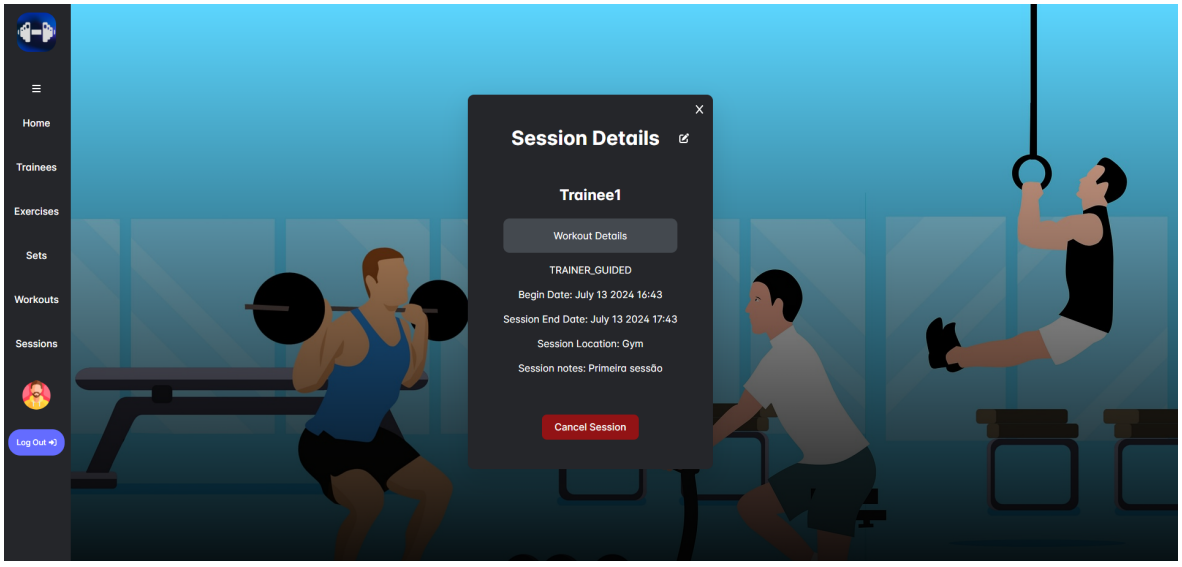

Figure 5.14: Trainee Sessions details

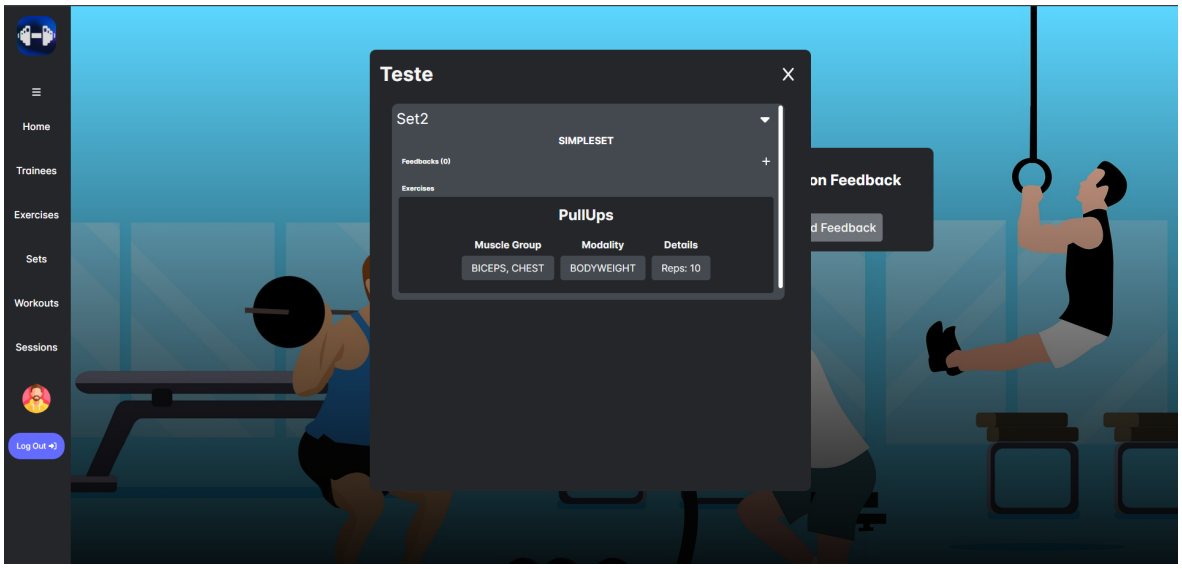

Figure 5.15: Trainee Sessions details

On this page they can also see the Session feedback's where both the trainer and trainee could write their feedback about the session itself or about a specific set, figure 5.15.

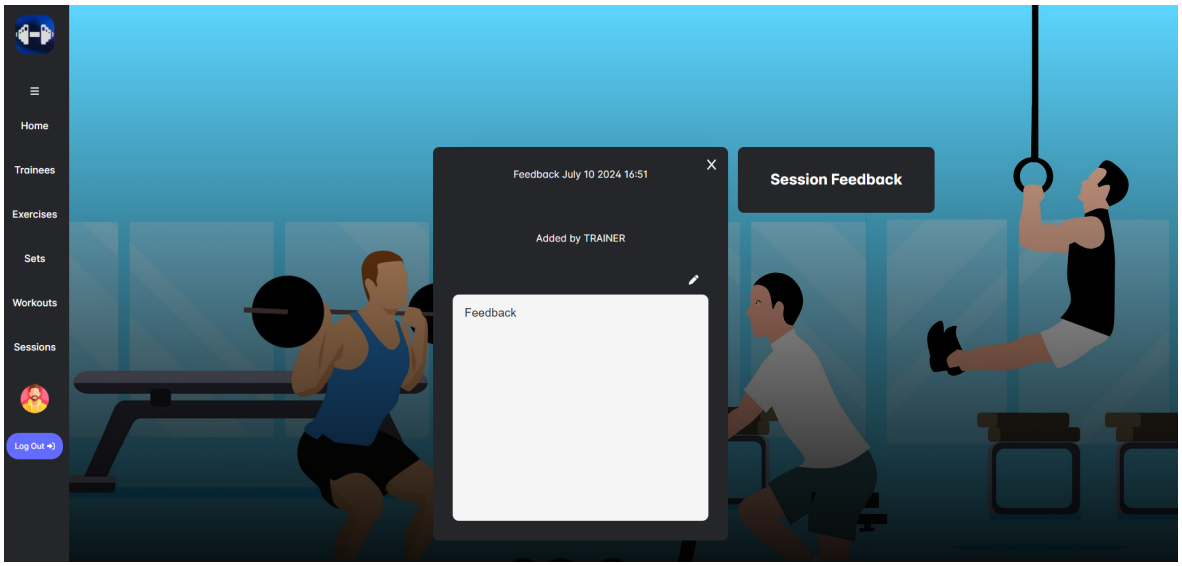

Figure 5.16: Trainee Sessions Feedback

Trainer could also go to the Trainee Profile by clicking on the avatar next to the Session container, where they will see some information about the Trainee and two buttons that navigates him to the Trainee reports and Trainee Data History respectively, figure 5.17.

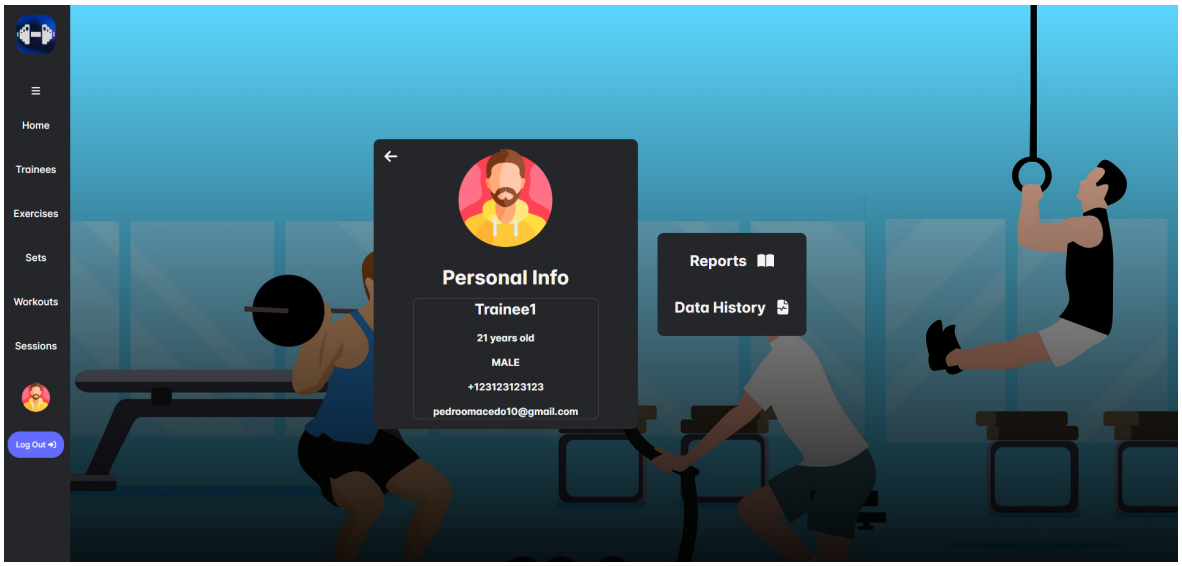

Figure 5.17: Trainee Profile

On the Trainees Reports page both trainees and trainers could see the reports but only the trainer can create new ones, with the possibility of making them private, so him is the only one could see them, figures 5.18 and 5.19.

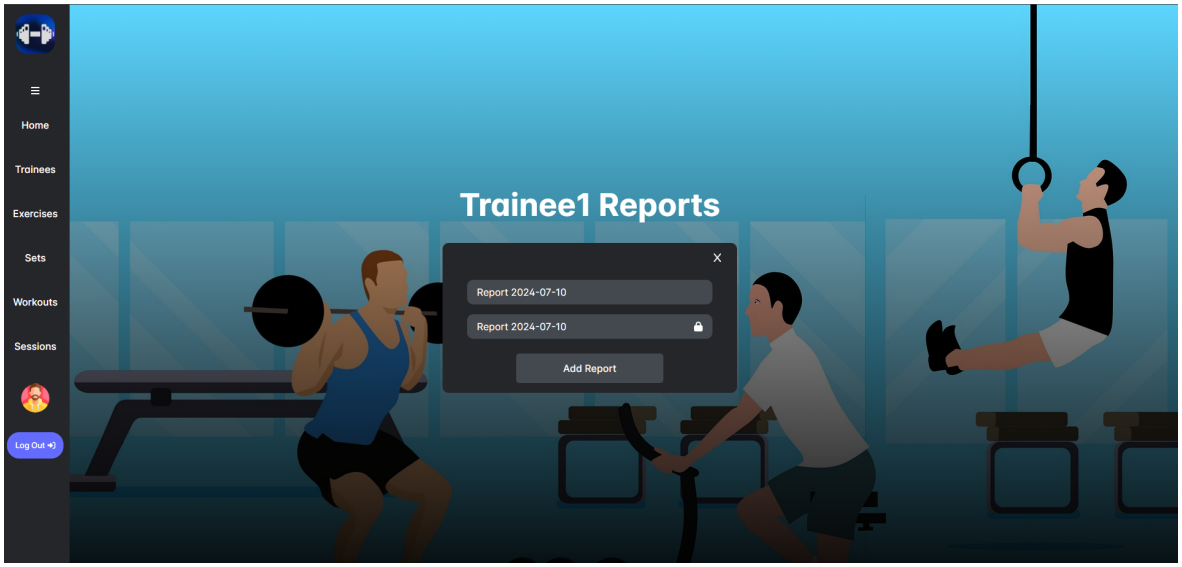

Figure 5.18: Trainee Reports

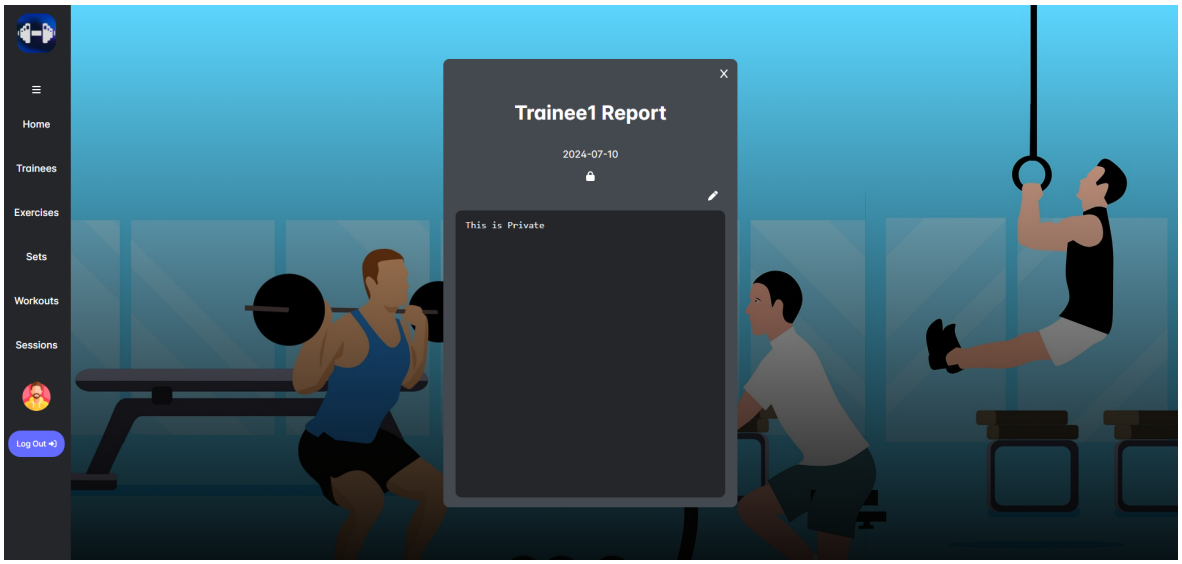

Figure 5.19: Trainee Reports Details

On the Trainee Data History the Trainers and Trainees can see the body data reports made by the trainer figure 5.20. This data reports contains some body measured values and some calculated values like BMI, fat body percentage and lower body fat percentage in kilos, where the trainer measure the trainee muscles, fill the required fields and waits for the result figure 5.21.

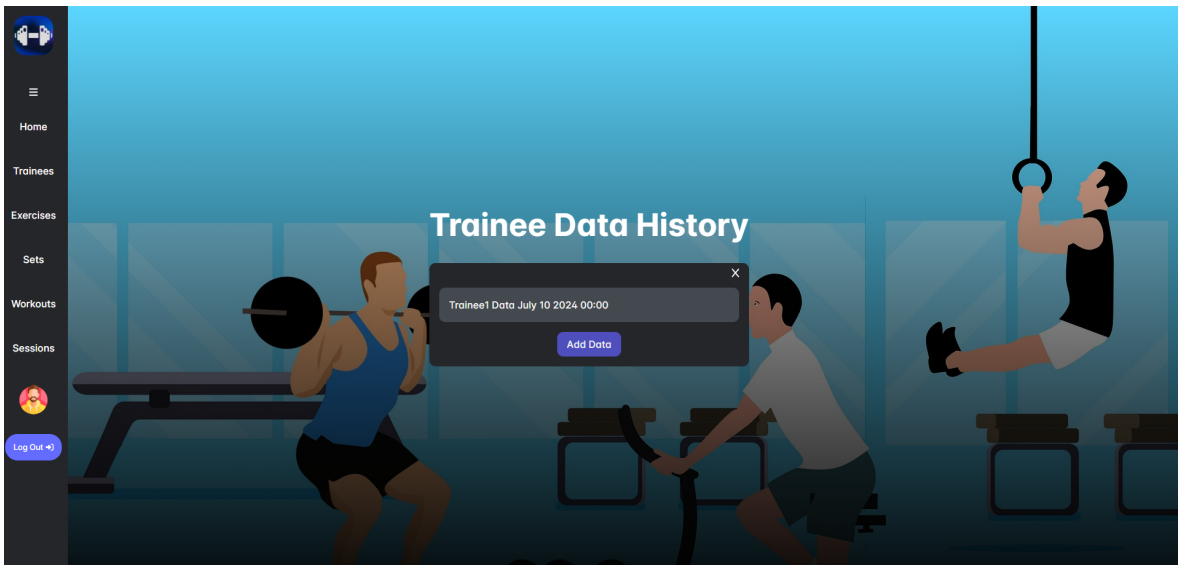

Figure 5.20: Trainee Data History

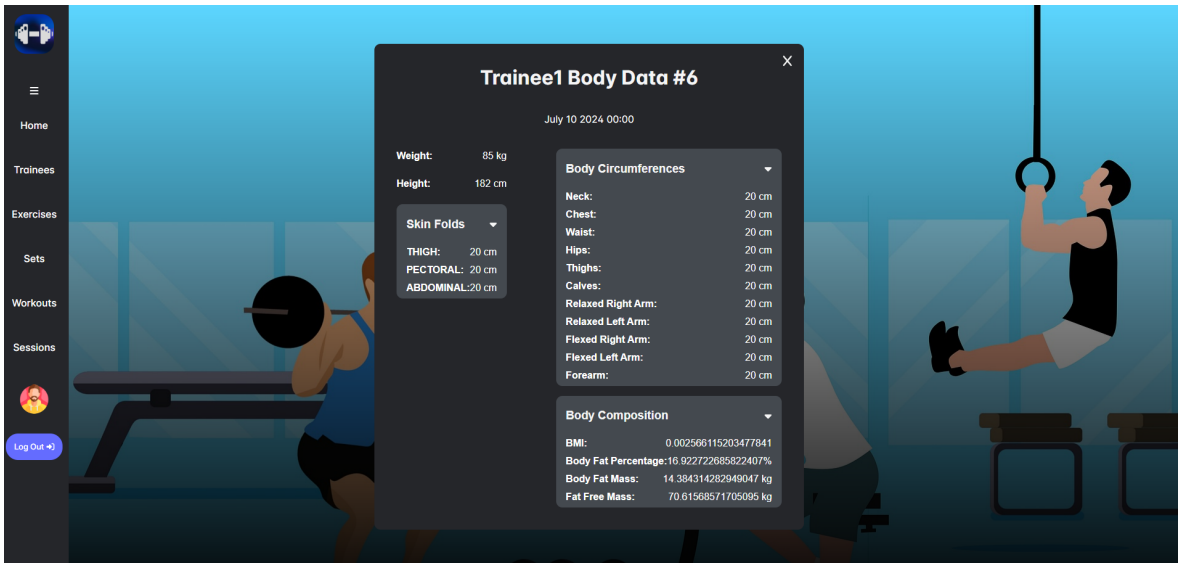

Figure 5.21: Trainee Data History Details

Chapter 6

Conclusions

Over the course of this project, we have implemented features such as user registration, training session management, feedback and reviews, and progress reports, which provide a comprehensive tool for managing personal training activities. Despite the challenges faced, including issues with time zone handling and session location processing, the system's successful developed.

Looking ahead, there is big opportunity to enhance the platform further. Future work should focus on improving the existing features, addressing unresolved issues, and implementing the proposed additional functionalities like membership management, media upload, and video call sessions. These improvements will ensure that PTGest remains a versatile and valuable tool for all its users, supporting their fitness journeys with advanced technology and innovative solutions.

In conclusion, the PTGest Management System presented a significant challenge where we had to meticulously plan and develop a project from scratch, applying all the knowledge we acquired during the course. Additionally, mastering a new frontend framework like Vue.js proved to be a challenging yet rewarding effort.

6.1 Completed Function Requirements

In the course of this project, we have successfully implemented the following functional requirements:

- **User Registration:** A secure and efficient authentication system was implemented, supporting all four types of users and their respective variants.
- **Training Session Management:** The system allows personal trainers (PTs) to schedule, reschedule and manage training sessions, including details such as muscle groups, exercises to be performed, training duration, date and time of the session and the session's location. Some issues remain unresolved, such as the ability for trainers to remove workouts, exercises or sets, problems with time zone handling, and the method

of storing and processing session locations.

- **Session Calendar:** A calendar feature was provided for both trainees and PTs, displaying their scheduled sessions.
- **Feedback and Reviews:** A feedback system for sessions and respective sets was implemented, though the grading system was not made.
- **Trainee Information Registration:** A robust system was developed to store trainee measurements and calculate BMI, body fat percentage, lean mass and fat mass values. This system offers detailed information to help trainers and trainees track progress.
- **Progress Reports:** Progress reports were implemented in a simple version, allowing users to write their reports in a text box and send them to the server for storage.
- **Email Notifications:** Email notifications were implemented, allowing the system to send users notifications for password changes, session updates, and password reset requests.
- **Creation of Custom Exercises:** A feature was added to enable trainers or the company to create custom exercises for use in training sessions.

6.2 Future Work

Improve the features already implemented, addressing all the mentioned issues, and implement the missing features that were not completed by the time of this report.

Additionally, we should enhance the system by implementing the optional features mentioned in the project proposal, which are:

- **Membership Management**: The system should allow the management of membership fees and values per class, both for companies and for individual *PTs*;
- **Implementation of a PWA (Progressive Web App)**: This would allow the platform to be installed and function as a native application on various devices and operating systems;
- **Media Upload**: The platform could allow for the upload of PDF documents, images, and videos, providing greater interactivity and understanding of exercises and training plans;
- **Invoice Generation**: The platform could generate invoices for fees and payments, facilitating financial management for users;
- **Evolution Charts / Progress History**: The platform could present evolution charts to visualize trainees progress over time;

- **Integration with Google Maps**: This functionality could allow for the mapping of *PT's* by gym(s) they can attend or franchise gyms;
- **Video Call Sessions**: The platform could allow for training sessions via video call, providing greater flexibility for users.

Bibliography

[1] Parallel development. https://www.perforce.com/blog/vcs/parallel-development. Accessed: 2024-06-01.

[2] Kotlin. https://kotlinlang.org/. Accessed: 2024-06-01.

[3] Spring. https://spring.io/. Accessed: 2024-06-01.

[4] Postgresql. https://www.postgresql.org/. Accessed: 2024-06-01.

[5] Dbms. https://www.geeksforgeeks.org/introduction-of-dbms-database-management-system-se Accessed: 2024-06-01.

[6] Jdbi. https://jdbi.org/. Accessed: 2024-06-01.

[7] Vue.js. https://vuejs.org/. Accessed: 2024-06-01.

[8] Vite. https://vitejs.dev/. Accessed: 2024-06-01.

[9] Typescript. https://www.typescriptlang.org/. Accessed: 2024-06-01.

[10] Vuex. https://vuex.vuejs.org/. Accessed: 2024-06-01.

[11] Render. https://render.com/. Accessed: 2024-07-10.

[12] Jjwt. https://github.com/jwtk/jjwt. Accessed: 2024-06-01.

[13] Java mail sender for spring. https://docs.spring.io/spring-boot/reference/io/email.html. Accessed: 2024-06-01.

[14] javax.mail. https://docs.oracle.com/javaee%2F7%2Fapi%2F%2F/javax/mail/package-summary.html. Accessed: 2024-07-10.

[15] Mockito. https://site.mockito.org/. Accessed: 2024-07-10.

[16] Quartz. https://mvnrepository.com/artifact/org.quartz-scheduler/quartz. Accessed: 2024-07-10.

[17] Java validation. https://www.baeldung.com/java-validation. Accessed: 2024-06-01.

[18] Jackson annotations. https://github.com/FasterXML/jackson-annotations/wiki/Jackson-Annotations. Accessed: 2024-06-01.

[19] Measurement techniques. https://blog.sanny.com.br/como-calcular-percentual-gordura-adipometro. Accessed: 2024-07-01.

Appendix A

Database

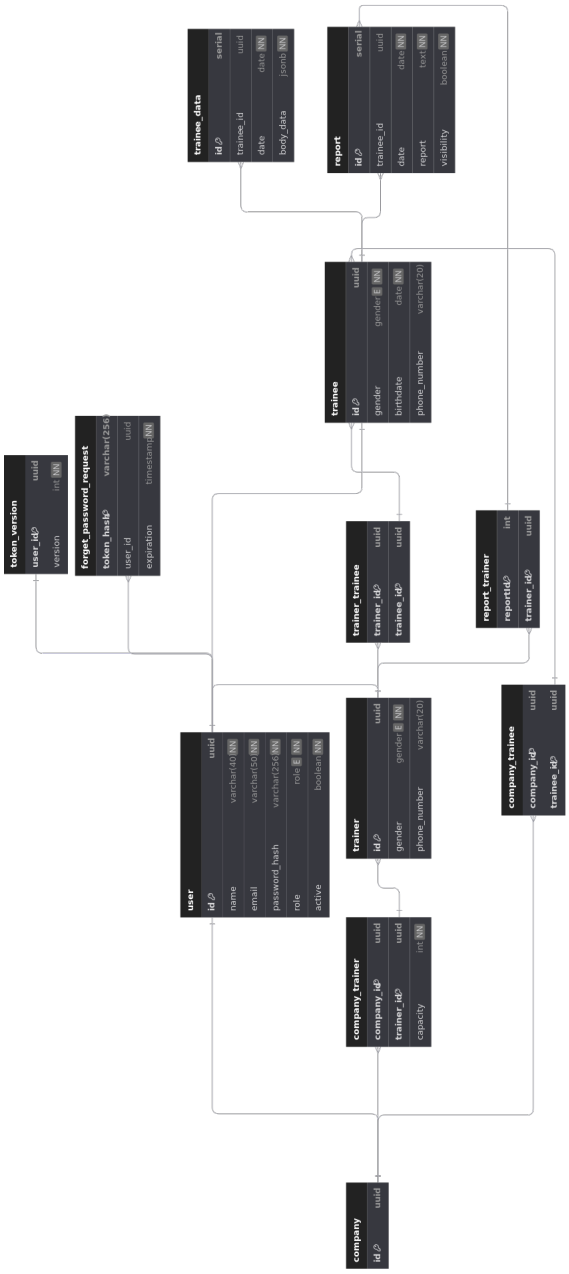

Figure A.1: User Diagram

Figure A.2: Workout Diagram