Zahvala

Zahvaljujem svojoj mentorici doc. dr. sc. Marini Bagić Babac na podršci, pomoći te korisnim savjetima i sugestijama tijekom pisanja ovog završnog rada.

Zahvaljujem svojim bliskim prijateljima i kolegama uz koje je studiranje bilo znatno zabavnije i lakše.

Posebno se zahvaljujem svojim roditeljima koji su me podupirali tijekom mog cjelokupnog obrazovanja.

Table of Contents

Introduction

In the last few years, people have been increasingly interested in physical activities, such as sports, exercise, and training, and they have become an essential part of their lives. With the initial appearance of the coronavirus, people suddenly had much more time and comfort for themselves, and many started with the so-called self-care routines. The central part of most of these routines, which has been preserved and grown to this day, is individual training. It is popular because it is very healthy and beneficial for the human body and mind. Furthermore, it requires little to no equipment, it can be performed anywhere, at home, outside, or in the gym, and it does not necessarily require a personal coach; since all the guides can be found online, the person can come up with its plan. By that approach, people need to put together their exercise and workouts details, plans, and schedule, as well as monitor their progress, usually through tracking the number of repetitions and sets of specific exercises or by tracking the required training execution time. It takes much time, though, and energy, and the final product, usually turns out to be disorganized, incomplete, and difficult to update and keep track of, taking into the account that planning, writing, and tracking are done with a classic pen and paper, note application or with an excel spreadsheet.

Since we live in the age of the Internet, where various services are available to the users through web applications, The Web Workout Planner application will serve as a solution to the mentioned problem of monitoring the progress and organization of the workouts because it reduces the complexity of performing the mentioned actions. Additionally, it offers insight into the user statistics of workouts to help track the progress better.

This work presents the process of developing a Web Workout Planner application, although its template applies to any planning and progress tracking application.

The Web Workout Planner application development process is presented in stages. It is based on the Software development lifecycle process, which consists of a detailed plan describing requirements analysis, requirements specification, architecture designing, product implementation, and product testing and deployment.

Requirement specifications are carried out using the application's defined functional requirements and combining them with the use cases. The architecture of the application is divided into the 3 groups: database, backend ("server-side"), and frontend ("client-side").

This paper aims to present the entire software development process and the delivery of a complete and stable user web application.

1. Related work

Many excellent examples of applications are similar in theme and structure to this application. Such applications include data grouping, scheduling events, progression tracking, and more. Most examples come from the same domain, the domain

of recreational exercising. For the start, My Workout Plan mobile application [17]. It is the application for managing exercise, creating routines, playing the workouts, and recording and showing the process. The application views and options are shown in Figure *1.1.*

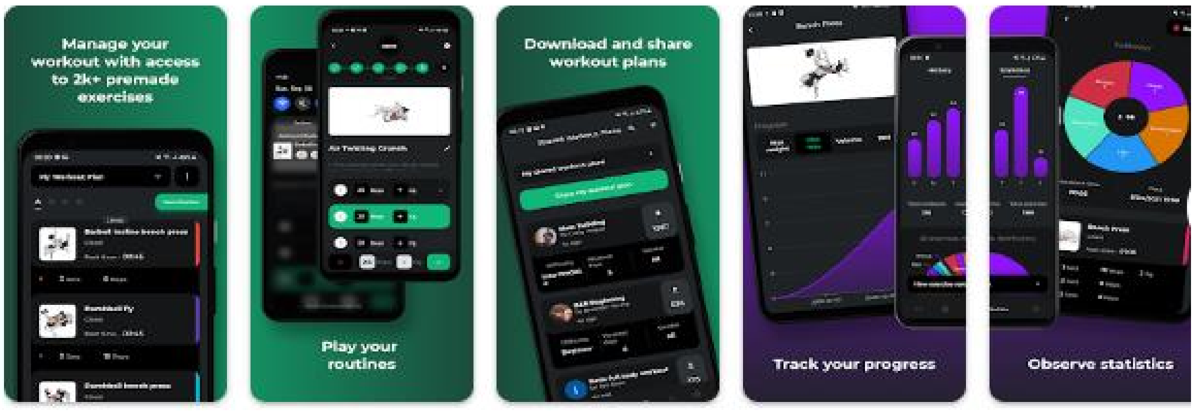

Manage your workout with access to 2k+ premade exercises
Play your routines
Download and share workout plans
Track your progress
Observe statistics

Figure 1.1: My Workout Plan mobile application

The next, more domain-general application for scheduling events is Google Calendar. It is a time-management web application that allows the user to view and organize the schedule and keep track of events [18]. The Google Calendar interface is shown in *Figure 1.2.*

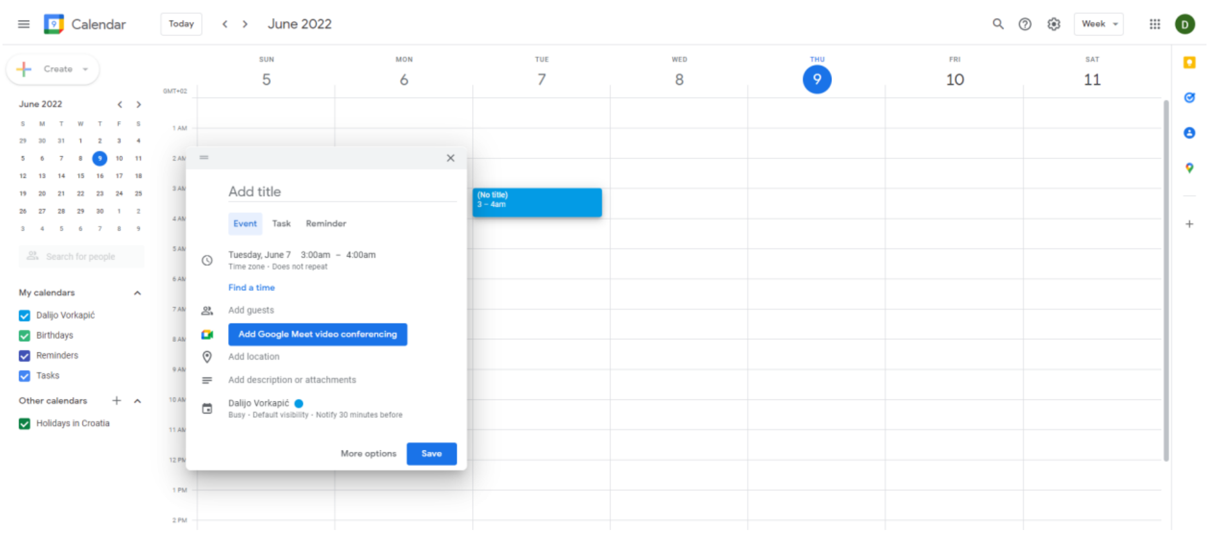


Figure 1.2: Google Calendar interface

2. Technologies

A broader set of technologies to use are needed to develop a full-stack web application. Full-stack development refers to the development of a complete web application. [1] Firstly, the Java Spring Boot framework is used for the application's server-side programming. The developing, version controlling, testing, and executing of Java server-side source code is done through the IntelliJ IDEA community edition development environment. Secondly, the Angular TypeScript framework is used for developing the application's client-side. Its source code is developed, version-controlled, and executed through the Visual Studio Code source code editor. Thirdly, the application's database is created through the PostgreSQL relational database management system. Lastly, the Heroku cloud platform is used to deploy the application on a real-time server.

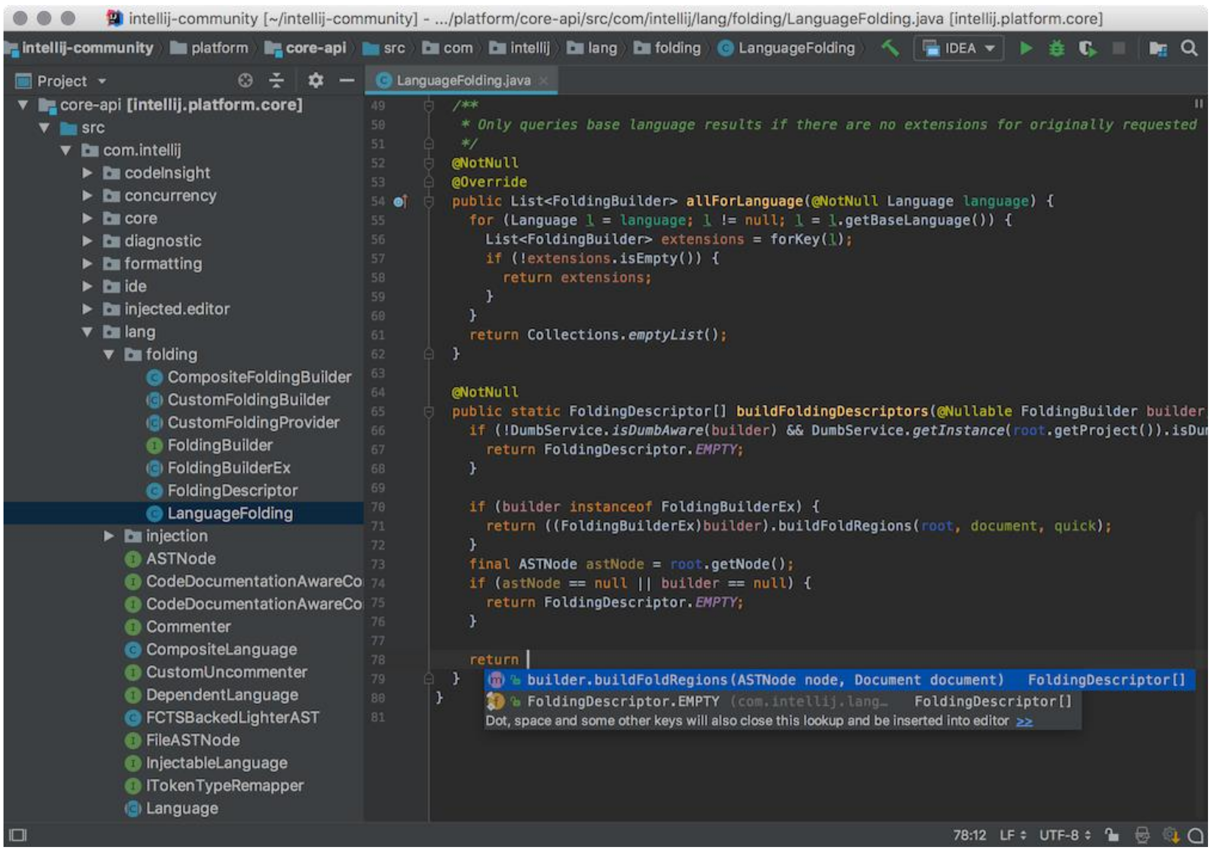
2.1 Development environment

A development environment is a workspace consisting of processes and programming tools for developing, testing, and debugging an application or program [2]. It provides developers with a user interface for tracking the development process. Their main goal is to facilitate and make the development in a particular language more effortless and more understandable.

2.1.1 IntelliJ IDEA

IntelliJ IDEA is an integrated development environment (IDE) written in Java for developing computer software written in JAR (Java Archive) based languages, like Java, Kotlin, and Groovy. An integrated development environment is a software application that provides developers with comprehensive facilities for software development. An IDE typically consists of a source code editor, build automation tools, debuggers, compilers, interpreters, and more. IntelliJ IDEA is designed by JetBrains, a Czech software development company that makes tools and development environments for software development and project management. [4] IntelliJ IDEA is available in two editions: an Apache-licensed community edition and a proprietary commercial edition. It is on the top spots of the most popular developer environment tool ranking in the Stack Overflow's "IDE of choice 2021" [3] and one of the commonly most used IDEs for server-side and Java programming.

IntelliJ IDEA offers a wide area of features, such as good developing ergonomics, straightforward project start-up, keyboard shortcuts for everything, standard and custom user interface themes, accessibility, instant navigation and search, source code versioning, supplementing the core functionality with additional plugins. An example of the IntelliJ IDEA user interface is shown in *Figure 2.1.* One of the key features is deep code insight. IntelliJ IDEA creates a virtual project map by indexing the initial source code. Using the information from the virtual map, it can detect errors, suggest code completion variants with precise context-awareness, perform refactoring, and more. Additionally, it comes with a powerful toolset for running, testing, and debugging the source code.

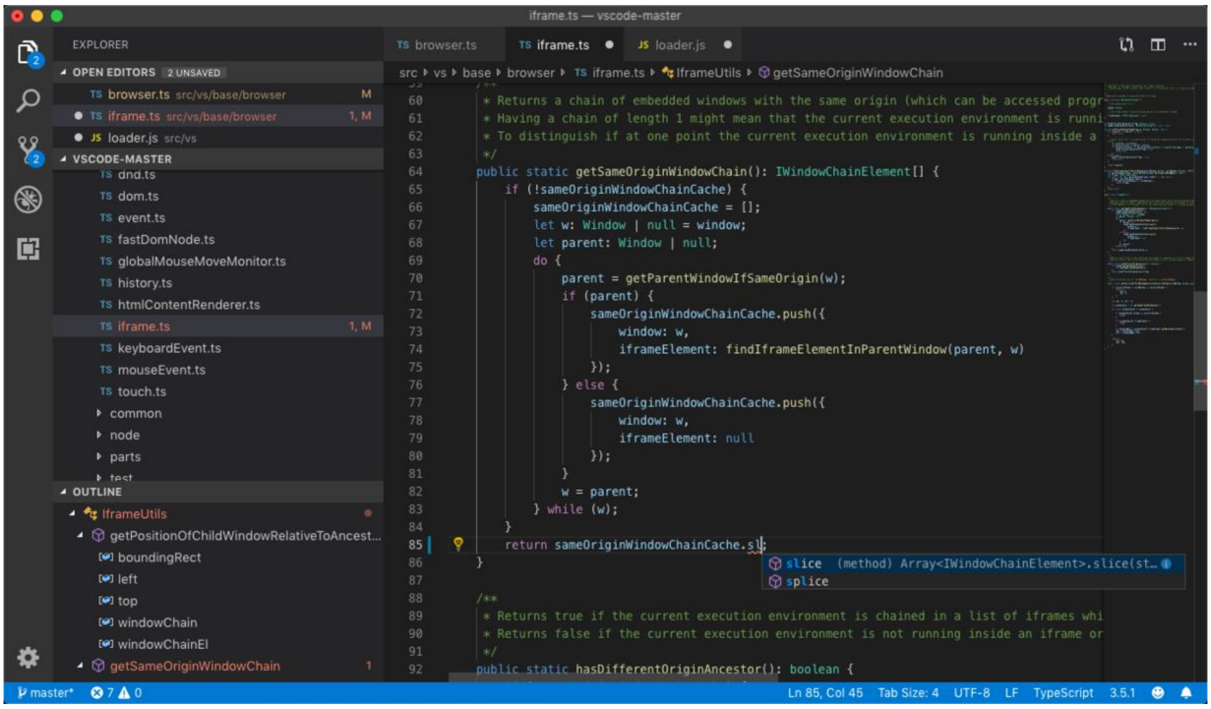


```
/**
 * Only queries base language results if there are no extensions for originally requested
 */
@NotNull
@Override
public List<FoldingBuilder> allForLanguage(@NotNull Language language) {
  for (Language l = language; l != null; l = l.getBaseLanguage()) {
    List<FoldingBuilder> extensions = forKey(l);
    if (!extensions.isEmpty()) {
      return extensions;
    }
  }
  return Collections.emptyList();
}

@NotNull
public static FoldingDescriptor[] buildFoldingDescriptors(@Nullable FoldingBuilder builder
  if (!DumbService.isDumbAware(builder) && DumbService.getInstance(root.getProject()).isDu
    return FoldingDescriptor.EMPTY;
  }

  if (builder instanceof FoldingBuilderEx) {
    return ((FoldingBuilderEx)builder).buildFoldRegions(root, document, quick);
  }
  final ASTNode astNode = root.getNode();
  if (astNode == null || builder == null) {
    return FoldingDescriptor.EMPTY;
  }

  return
}      builder.buildFoldRegions(ASTNode node, Document document)   FoldingDescriptor[]
}      FoldingDescriptor.EMPTY (com.intellij.lang…    FoldingDescriptor[]
       Dot, space and some other keys will also close this lookup and be inserted into editor >>
```

Figure 2.1:IntelliJ IDEA user interface

2.1.2 Visual Studio Code

Visual Studio Code is a free source code editor developed by Microsoft. It was ranked as the most popular developer environment tool in the "IDE of choice" Stack Overflow 2021 Developer Survey [3], and it is one of the best choices for web application client-side development, for which it is used in this project. Visual Studio Code helps the programmer in code writing and debugging. It corrects the code using the intelli-sense feature, which can detect if any code snippet is left incomplete or undeclared [5]. It has built-in support for multiple programming languages and has a rich ecosystem of extensions for other languages and runtimes. It provides an interactable and straightforward user interface, repository support, and web support, and it offers hierarchy file structure and terminal, git, and multi-projects support. The advantage of Visual Studio Code over other source code editors and IDEs is its robust architecture. It is lightweight, fast, responsive, interactive, and, most importantly, free of cost. An example of a Visual Studio Code user interface is shown in *Figure 2.2*.


iframe.ts — vscode-master
EXPLORER
OPEN EDITORS 2 UNSAVED
browser.ts src/vs/base/browser M
iframe.ts src/vs/base/browser 1, M
loader.js src/vs
VSCODE-MASTER
dnd.ts
dom.ts
event.ts
fastDomNode.ts
globalMouseMoveMonitor.ts
history.ts
htmlContentRenderer.ts
iframe.ts 1, M
keyboardEvent.ts
mouseEvent.ts
touch.ts
common
node
parts
test
OUTLINE
IframeUtils
getPositionOfChildWindowRelativeToAncest...
boundingRect
left
top
windowChain
windowChainEl
getSameOriginWindowChain 1
browser.ts
iframe.ts
loader.js
src › vs › base › browser › iframe.ts › IframeUtils › getSameOriginWindowChain
master* ⊗ 7 ⚠ 0
Ln 85, Col 45 Tab Size: 4 UTF-8 LF TypeScript 3.5.1

```
     * Returns a chain of embedded windows with the same origin (which can be accessed progr
     * Having a chain of length 1 might mean that the current execution environment is runni
     * To distinguish if at one point the current execution environment is running inside a
     */
    public static getSameOriginWindowChain(): IWindowChainElement[] {
        if (!sameOriginWindowChainCache) {
            sameOriginWindowChainCache = [];
            let w: Window | null = window;
            let parent: Window | null;
            do {
                parent = getParentWindowIfSameOrigin(w);
                if (parent) {
                    sameOriginWindowChainCache.push({
                        window: w,
                        iframeElement: findIframeElementInParentWindow(parent, w)
                    });
                } else {
                    sameOriginWindowChainCache.push({
                        window: w,
                        iframeElement: null
                    });
                }
                w = parent;
            } while (w);
        }
        return sameOriginWindowChainCache.sl;
    }                                          slice  (method) Array<IWindowChainElement>.slice(st…
                                               splice
    /**
     * Returns true if the current execution environment is chained in a list of iframes whi
     * Returns false if the current execution environment is not running inside an iframe or
     */
    public static hasDifferentOriginAncestor(): boolean {
```

Figure 2.2: Visual Studio Code user interface

2.2 Development framework

A development framework is a software structure consisting of sets of tools and libraries that provide a fundamental structure to support the application development for a specific environment.

2.2.1 Java Spring Boot Framework

Java Spring Boot is a development framework based on the Spring framework. Spring framework is an open-source Java-based framework used to create a microservice, an architecture allowing developers to develop and deploy services independently, meaning each service running has its process. Spring Framework offers built-in support for typical tasks an application needs to perform, such as data binding, type conversion, validation, exception handling, resource and event manager, and more. Since that regular Spring Framework requires significant time and knowledge to configure, set up, and deploy Spring backend applications, Spring Boot Framework mitigates this effort by making developing web applications and microservices with Spring Framework faster and easier. It achieves it with: autoconfiguration, meaning that the application is initialized with pre-set dependencies that do not need to be configured manually, opinionated approach, meaning it adds and configures starter dependencies based on the project needs, without requiring the developer to make all those decisions and set up everything manually, and thirdly, it helps developers to create standalone applications that run on their own by embedding a web server during the initialization process, without the need of an external web server, making the whole application launchable on any platform with ease [*6*]

Spring Boot follows a layered hierarchical architecture in which each layer communicates with the layer directly below or above [7]. A diagram of layered architecture is shown in *Figure 2.3.* There are four abstract layers in Spring Boot, presentation layer, business layer, persistence layer, and database layer.

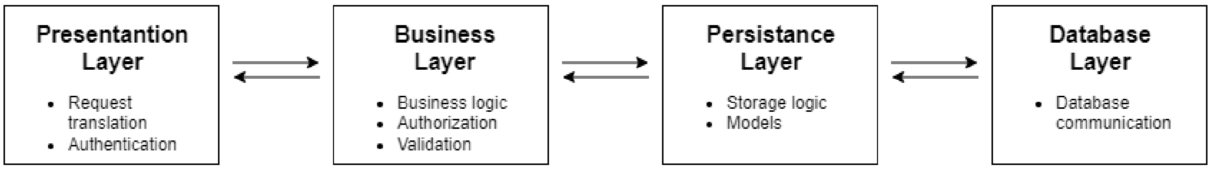


Figure 2.3: Spring Boot layers hierarchy

The presentation layer consists of controllers. Each controller handles its HTTP requests, such as POST, GET, UPDATE, and DELETE, through the specified paths. An example of a controller is shown in Code *2.2*. The data is sent and received in JSON format. JSON (JavaScript Object Notation) is a lightweight, language-independent text format for storing and transporting data. It consists of objects that have their set properties with their included values. JSON example is shown in *Code 2.1.*

```
'{"name":"John", "age":30, "interests":["hiking", "sports", "cars"]}'
```

Code 2.1: JSON example

In order to process the data, the presentation layer translates the JSON parameters into Java objects, which are then transferred to the deeper layer of logic, the business layer.

```
@RestController
class EmployeeController {

      private final EmployeeService service; //business layer

. . .

@GetMapping("/employees")
List<Employee> getAllEmployees() {
      return service.getEmployees();
}

@PostMapping("/employees")
Employee addNewEmployee(@RequestBody Employee newEmployee) { //mapping JSON
      return service.saveEmployee(newEmployee);              //object to the
                                                             //Employee model
}
```

Code 2.2: Controller class example

The business layer handles all the business logic, authorization, and validation. In application logic, it consists of service classes that handle the functions defined through the persistence layer linked with the database communication. An example of the service class is shown in *Code 2.3.*

```
@Service
public class EmployeeService {

      private EmployeeDataAccessObject employeeDAO; //reference to the
                                                    //persistence layer

      . . .

      public List<Employee> getEmployees() {
             return employeeDAO.fetchEmployees();

      }
      . . .
}
```

Code 2.3: Service class example

The persistence layer contains all the storage logic and data access services. It directly communicates with the database layer and translates the objects received from the business layer or the database layer into the models, which are then sent deeper or higher into the hierarchy. "Example of the data access service is shown in *Code 2.4*. Finally, the database layer is the database itself, where all CRUD (Create, Retrieve, Update, Delete) operations can be performed.

```
@Repository()
public class EmployeeDataAccessService implements EmployeeDataAccessObject {

      private final JdbcTemplate jdbc;  //reference to the database layer
                                        //accessor
. . .

      @Override
      public List<Employeee> fetchEmployees() {
              final String sql="SELECT * FROM emplooyes";
              List<Employee> employees = jdbc.query(sql,…). . .

              Return employees;
      }
. . .
}
```

Code 2.4: Data access service class example

2.2.2 Angular framework

Angular is a component-based platform and a framework built on TypeScript, used for building single-page scalable client web applications using HTML and TypeScript language. It is a collection of well-integrated libraries that cover a wide variety of features, including routing, forms management, and client-side communication, as well as a suite of developer tools to help develop, build, test, and update the source code.

Angular apps are modular, which means each Module defines a set of associated Components. [8] A Component controls a patch of the screen called a View, defined through the HTML template. Angular creates, updates, and destroys Components as the user moves through the application. An example of Component class declaration is shown in *Code 2.5.*

```
@Component ({
      selector: 'my-component'
      templateUrl: './my-component.component.html'
})
export class MyComponent implements OnInit {
      // Component code
}
```

Code 2.5: Component declaration

A Service class is a part of Angular that handles and defines things that can not fit into a Component and find their reason for existing in the separation of concerns. Their purpose is to help divide the application into small, logical, reusable units. The service declaration example is shown in *Code 2.6.*

```
export class TicketService {

    getTickets()   { return tickets; }
    addTicket(ticket: any) { tickets.push(ticket); }
    deleteTicket(ticket: any))  { tickets.pop(ticket); }
}
```

Code 2.6: Service declaration

One of the more significant subjects of Angular is Dependency Injection, a design pattern in which one or more dependencies are injected into a dependent object. Such a pattern is applied when it is wanted to separate actions of the what-to-do part from the when-to-do part. Angular uses its Dependency Injection framework, divided into three parts: the injector, the provider, and the dependency. For example, injecting different required services into the different components or registering a provider of any service being used, thus creating dependency. Dependency Injection between service and a component example is shown in *Code 2.7.*

```
import { Injectable } from '@angular/core';
import { TicketService } from './ticket-service';

@Injectable()
export class TicketService {

    getTickets()   { return tickets; }
    addTicket(ticket: any) { tickets.push(ticket); }
    deleteTicket(ticket: any))  { tickets.pop(ticket); }
}

. . .
```

```
import { Component } from '@angular/core';
import { TicketService } from './ticket-service';

@Component({
    selector:    'my-component',
    templateUrl: './my-component.component.html',
    providers:  [ TicketService ]
})


export class HeroListComponent implements OnInit {
    constructor(private ticketService: TicketService {}

    // Component code
}
```

Code 2.7: Dependency injection example

Data Binding is another crucial feature of Angular, which allows properties of two objects to be linked so that a change in one causes a change in the other, like establishing a connection between the user interface (HTML) and the application. It defines the relationship between a source object that provides the data and a target object that will use the data from the source object. Alongside regular data binding, Angular also provides event binding, for example, executing a component function on a button click event. Examples of data binding are shown in *Code 2.8*.

```
<h1>{{source_object_string}}</h1>

<button (click)="doSomething()">Do something</button>
```

Code 2.8: Data binding example

Angular provides lifecycle hooks as interfaces that any component can implement to perform actions when different moments occur. For example, moments like component initialization (OnInit), exit (OnDestroy), update (OnChanges), and more. An example of declaring an initialization lifecycle hook is shown in *Code 2.9*.

```
export class MyComponent implements OnInit {

      ngOnInit() {
              // Some code to execute from the start
```

```
        }
}
```

Code 2.9: Initialization lifecycle hook example

Alongside components, Angular contains two more kinds of directives. Structural directives change the view's structure (NgFor, NgIf). An example of structural directives is shown in Code *2.10.* NgIf presents/hides the HTML component based on the outcome of the attached expression (true or false). NgFor iterates over an array of items, and for each item, it creates the additional HTML element to which it is attached. The other type is the attribute directives used as attributes of the HTML elements, for example, NgClass and NgStyle.

```
 <div *ngIf="character" class="name">{{character.name}}</div>

<ul>
      <li *ngFor="let character of characters">{{character.name}}</li>
</ul>
```

Code 2.10: Structural directives example

One of the Angular most essential features are observables, provided through the RxJS library, used for reactive programming where the data is in asynchronous streams. Observables provide support for passing the data between the publisher and the subscribers in the application. Firstly, on the publisher side, an observable data stream is defined as a subject or the observable. Through that stream, the publisher can send asynchronous or regular data. On the other end, components that require that data will subscribe to the subject and "wait" for data to be sent through the stream. Once it is sent, the subscribers receive it, and they can store it in their component variable. An example of a subscription action is shown in *Code 2.11.* Received data through the subscription can come in three variants: a passed value, an error, or a completed subscription. Various functions in regular return an observable to which components can subscribe, for example, HTTP operation methods.

```
import { Observable } from 'rxjs/Rx'
import { Injectable } from '@angular/core'
import { Http } from '@angular/http'

@Injectable()
```

```
export class TicketService  {

      constructor(public http: Http) {}

      public getTickets() {
              return this.http.get('/api/tickets') //returns an Observable
      }

. . .

export class MyComponent {

      constructor(public ticketService: TicketService) {}


      ngOnInit() {
              this.ticketService.getTickets().subscribe({ //three outcomes
                      next(response){},
                      error(err) {},
                      complete() {}
              })
      }
}
```

Code 2.11: Observables action example

Angular also offers two types of forms, reactive and template-driven forms, and their validation options. Lastly, without mentioning many other possibilities and features, Angular contains, for navigating between the component views, Angular uses the routing module. An example of component routes is shown in *Code 2.12*.

```
import { RouterModule, Routes } from '@angular/router';

const appRoutes: Routes = [
    { path: 'characters',         component: CharactersComponent },
    { path: 'character/:id',      component: CharacterDetailComponent },
    { path: '', redirectTo: '/characters', pathMatch: 'full' },
    { path: '**',                 component: PageNotFoundComponent }
];
@NgModule({
    imports: [
        RouterModule.forRoot(appRoutes)
    ]
})
export class AppModule { }
```

Code 2.12: Routes example

3. Requirement specification

The requirement specification is the first phase of software development. It is a collection of all requirements to be imposed on the design and verification of the application. A requirement is a thing that product must do ("system shall do") or quality it must have ("system shall be"). It is determined by the requirements engineering process, which analyzes, structures, documents, and verifies user-required system services and usage constraints. Based on the content, each requirement specification can be divided into function and non-function requirements. Differences between functional and non-functional requirements are shown in *Table 3.1.*

Table 3.1: Functional vs. Non-functional requirements

	Functional requirements	**Non-functional requirements**
Objective	Describe what the product does	Describe how the product works
End result	Define product features	Define product properties
Focus	Focus on user requirements	Focus on user expectations
Documentation	Captured in use case	Captured as a quality attribute
Essentiality	Mandatory	Not mandatory, but desirable
Origin type	Usually defined by user	Usually defined by developers
Testing	Component, API, UI testing	Performance, usability, security testing
Types	Interface, authentication, authorization levels, etc.	Usability, scalability, reliability, Performance, etc.

3.1 Functional requirements

A functional requirement is a description of the service that the software must offer to the stakeholder (actor). It describes a software system or its component. A function is nothing but inputs to the software system, its behavior, and outputs.

For this application, we define actors with their corresponding functional requirements. For example, an initiator is the type of actor that directly interacts with the system, while the participant does not interact and only gets passively affected by the system.

Actors:

- Initiators
  - Administrator
  - User
- Participants
  - Database

User can:

- Log in to the application
- View the data of:
  a. Workouts
  b. Exercises
  c. Schedule
  d. Statistics
- Create, update, and delete data of:
  a. Workouts
  b. Exercises
  c. Scheduled workouts
- Track exercise progress

Administrator can:

- Everything that users can

- Manage users
    a. Register new users
    b. Remove users

Database:

- Saves all data from the application
- Deletes selected data
- Updates selected data

3.1.1 Use cases

A use case is a written description of how users will perform tasks on the website. From a user's point of view, it outlines a system's behavior as it responds to a request. It is described through the use case diagram. A use case diagram is part of UML behavioral diagrams, which depict the elements of a system that are dependent on time, and that convey the dynamic concepts of the system and how they relate to each other, such as graphical depiction of a user's possible interactions with a system, which was previously mentioned through the use cases. Application's use cases UML diagram is shown in *Figure 3.1.*

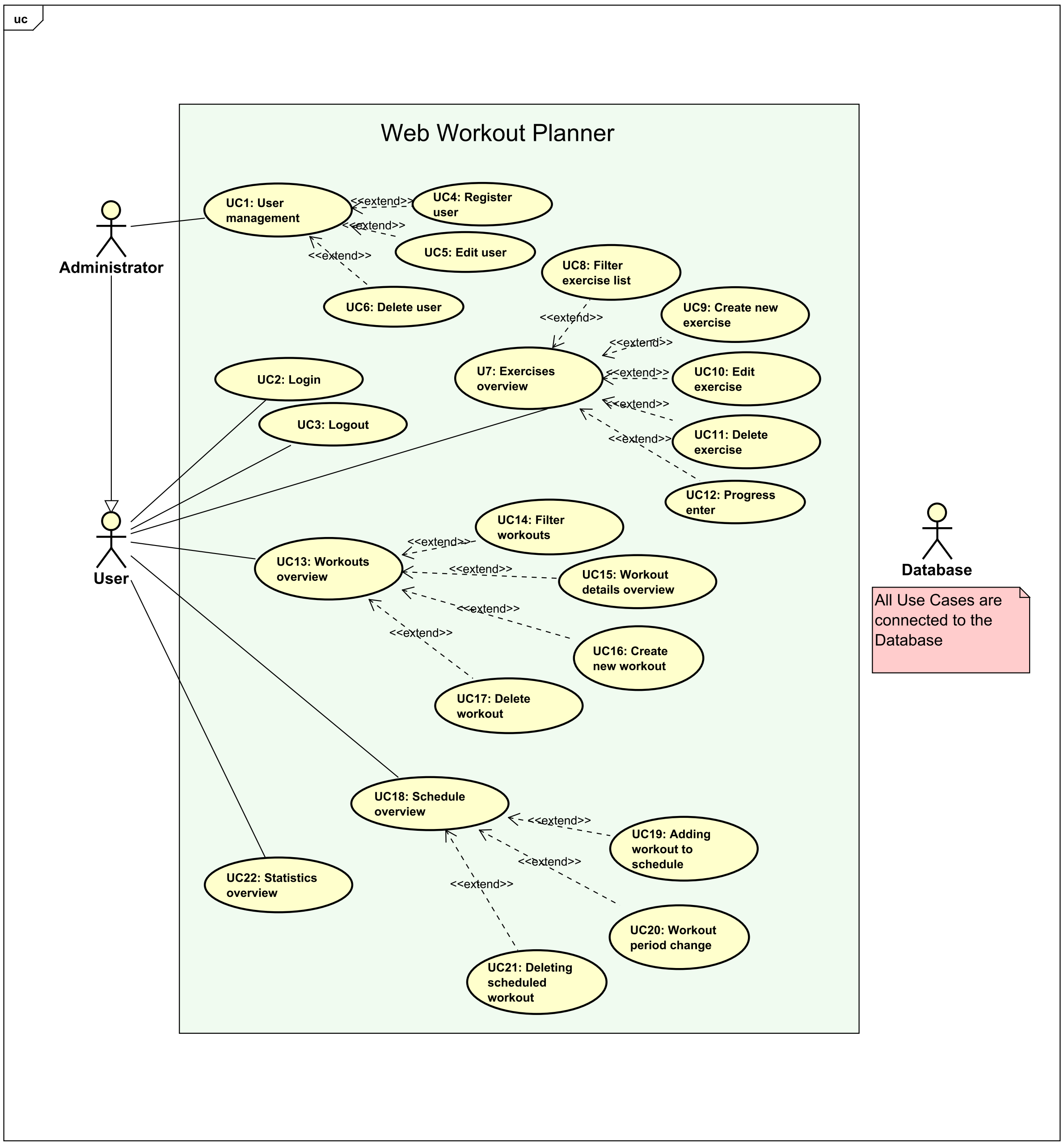


Figure 3.1: Application's use case diagram

3.2 Non-functional requirements

A non-functional requirement is a specification that describes the software's operation capabilities and constraints that enhance its functionality, such as performance, security, reliability, usability, scalability, and maintainability.

Application's non-functional requirements:

- Performance
  - Single page responsive application
- Scalability
  - Application can stand high workload
- Availability
  - Multiple users possible in real-time
- Reliability
  - No unexpected crashes
- Security
  - Authentication
  - Password hashing
  - Input validation
- Usability
  - Minimalistic and easy-to-use user interface

4. System architecture

The architecture of this application is divided into the following components, shown in the architecture layout in *Figure 4.1*.

- Server side (backend)
- Client-side (frontend)
- Database

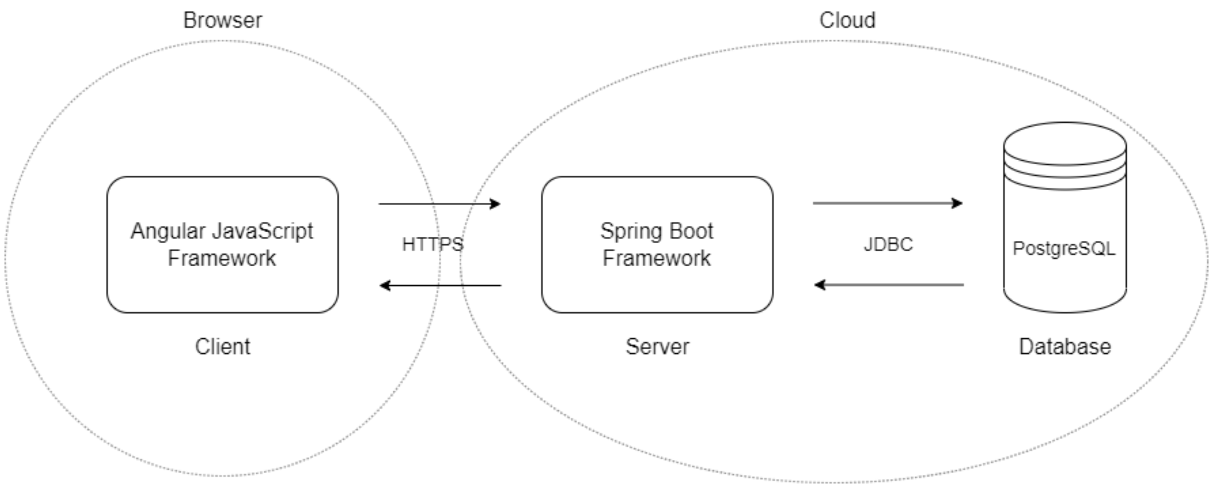


Figure 4.1: Architecture layout

Application architecture is based on the client/server architecture. The server is the provider, combined with the database, while the client acts as a customer. Such architecture's main feature is providing division of responsibilities, meaning that each side does its duty independently of the other, offering a better generalization of the application structure [10].

A database is an organized collection of structured information or data. It is controlled by a DBMS (Database Management System). The most used type of database is a relational database. The data within it is typically modeled in rows and columns in tables to make processing and data querying efficient. The data can be easily accessed, managed, modified, updated, controlled, and organized. Most relational databases use structured query language (SQL) for writing and querying data. SQL provides querying, manipulation, data defining, and database access control [12].

The server side is the main component of this application. In it lies the program logic to deliver to the user what he requested. The complete backend program is built with Java Spring Boot Framework.

The server side communicates in two directions. In one direction, it communicates with the database, while in the other, it communicates with the client. Such communications are achieved by using APIs. The API (Application Programming Interface) is a mechanism that enables two software components to communicate with each other using a set of definitions and protocols. There are four different ways that APIs can work, like the most popular one, REST API.

Regarding this application, Spring Boot is RESTful, meaning it can be based on the REST API approach. REST (Representational State Transfer) is an architectural style that defines a set of functions like GET, POST, PUT DELETE, and more, which web clients use to access the server data. They exchange data using HTTP.

With the aim of Spring boot server-side application to be RESTful, it needs to follow six guiding principles (constraints) of the REST architecture [11]:

1. Uniform interface – the generality of the system architecture, visibility of interactions
2. Client-Server design pattern – enforces the separation of concerns, can be evolved independently
3. Statelessness – server does not save client data between requests
4. Cacheable – response should label itself as cacheable or non-cacheable
5. Layered System – architecture composing of hierarchical layers by constraining component behavior
6. Code on Demand – allows client functionality to extend

Client-side refers to everything in the web application that is displayed on the client (end-user device). It includes what the user sees (User Interface) and any actions an application performs within the user's browser. Any client-side application is built on HTML (Hypertext Markup Language) markup language, which builds a website's structure and renders a website in a browser. For designing the HTML, the default design language is CSS (Cascading Style Sheets), which adds visual design elements to a website. With aiming to make websites dynamic, responsive, and interactive, websites use JavaScript scripting language. It allows for dynamically adding HTML contents to the DOM (The Document Object Model), which is a top-to-down representation of all the elements that make up a web page. It is the interface through which scripts interact with the HTML. Most the webpages nowadays are not coded using JavaScript but with its superset, TypeScript.

TypeScript is a better solution than JavaScript because TypeScript additionally offers:

- Type checking the code, generics
- It provides highly productive development tools
- Simplifies JavaScript code
- Structural

The complete client-side application is built using Angular, a TypeScript-based framework. Angular implements core and optional functionality as a set of TypeScript libraries than can be imported into the application. The architecture of an Angular application relies on certain fundamental concepts. The basic building blocks of the Angular framework are the components that are organized into modules, which collect related code into functional sets. Components define views, which are sets of screen elements that Angular can choose among and modify according to the program logic and data. In addition, components use services that provide specific functionality not directly related to views. Service providers can be injected into components as dependencies, making the code modular, reusable, and efficient. Angular also offers two-way data binding for synchronization between the model and the view, dependency injection, routing, route protection, and more.

5. Implementation

5.1 Database

The database used for this application is PostgreSQL. It is a free and open-source relation database management system (RDBMS) emphasizing extensibility and SQL compliance.

The application's database consists of the following entities:

- AppUser
- Workout
- Exercise
- Progress
- Schedule

Since entities need to connect, it is necessary to determine the relations. The entity Relationship Diagram is shown in *Figure 5.1.*

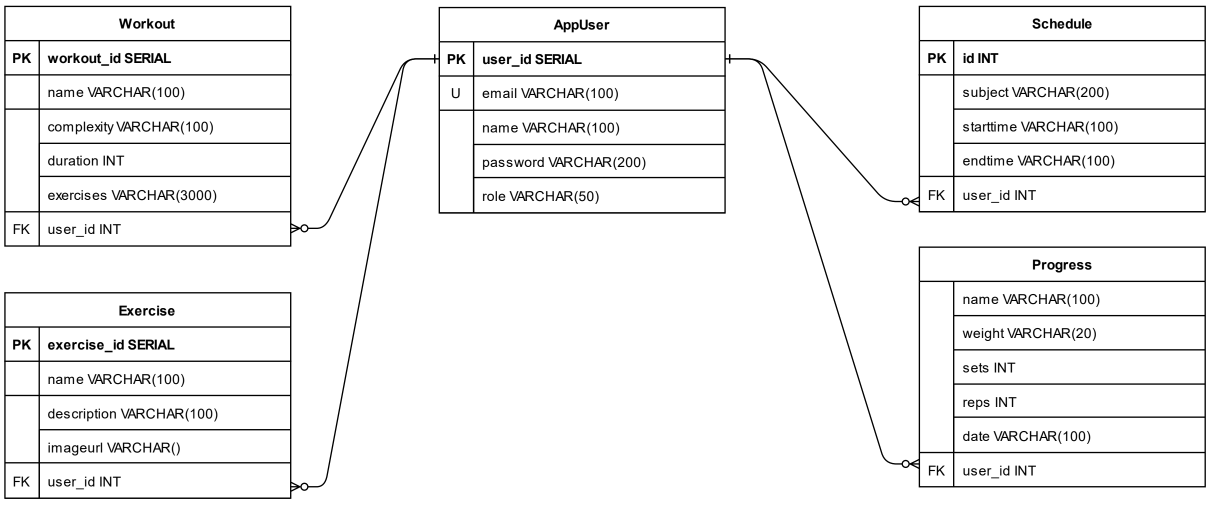


Figure 5.1: Entity relationship diagram

5.2 Server-side

The application's server-side is divided into four layers: Controllers

1. Services
2. Data access services - repository
3. Data access objects

The client-side communicates with the server-side through the HTTP request made to the URL defined by the controllers. The Controller then proceeds the data from the request to the Service, which implements the logic for handling the request. Service then uses the Data access service layer to get defined methods to process the request. Data Access Service creates the methods to communicate with the database based on the methods defined in the Data access object interface. Regarding communication with the database, it uses JDBC (Java Database Connectivity) API, which connects and executes queries with the database. Server-side layers are shown in *Figure 5.2.*

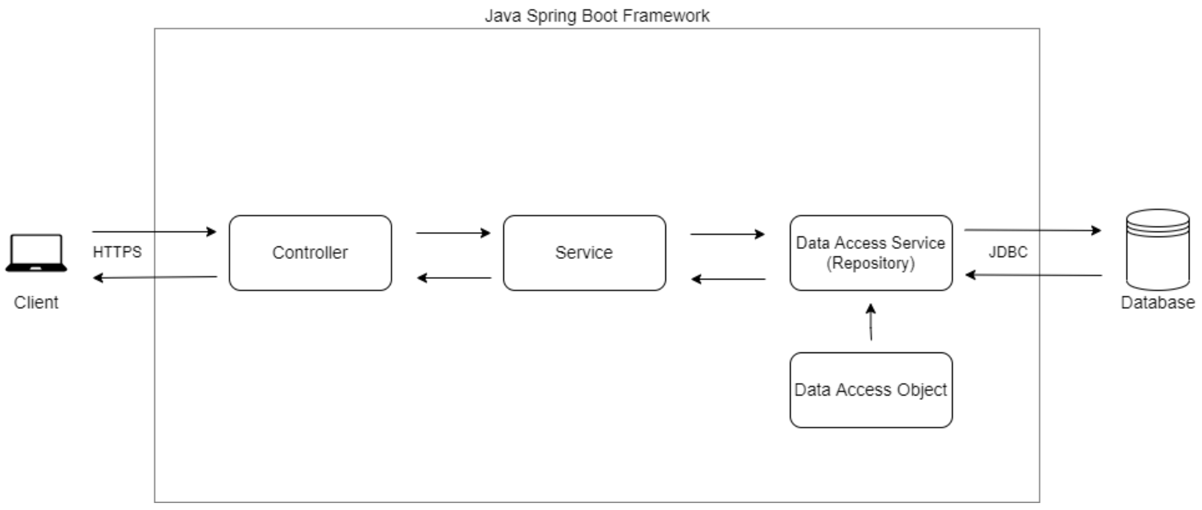
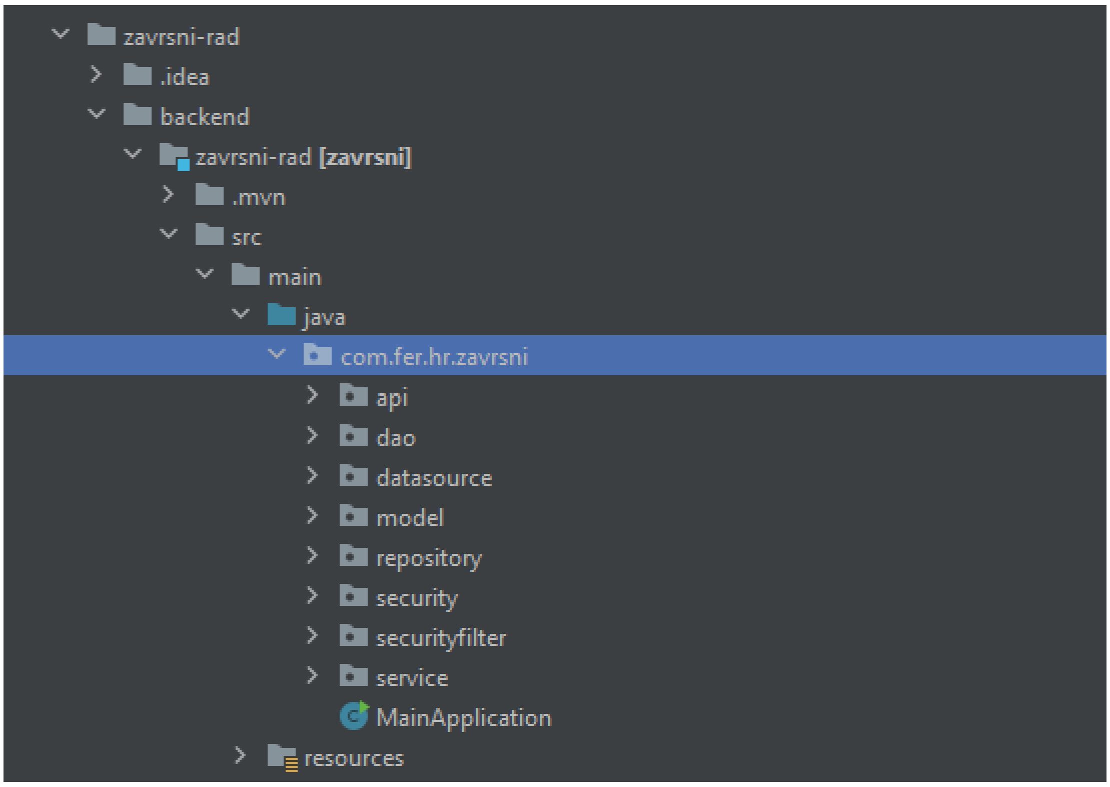


Figure 5.2: Server-side layers

The application's server-side structure is organized into packages. [9] A Java package is a namespace group of similar types of classes, interfaces, and sub-packages. In this application, they group classes and interfaces that belong to the same layer of data handling. The Server-side consists of the main com.fer.hr.zavrsni package, which contains sub-packages: api, dao, datasource, model, repository, security, securityfilter, service, and the main application. Directories and package structure is shown in *Figure 5.3.* api package represents controllers, dao and repository contain the persistence layer handlers, content inside service handles the application's business layer, models represent the overall data structures, while security and securityfilter handle authentication and authorization processes of the application.



```
zavrsni-rad
  .idea
  backend
    zavrsni-rad [zavrsni]
      .mvn
      src
        main
          java
            com.fer.hr.zavrsni
              api
              dao
              datasource
              model
              repository
              security
              securityfilter
              service
              MainApplication
          resources
```

Figure 5.3: Server-side directory structure

5.2.1 Models

A data representation is needed to understand, manipulate, and transfer the received data. A model class is a representation of a data object which can be used for transferring data through the application. It encapsulates direct access to data in an object and ensures that all data in an object is accessible through the getter methods. Server-side models are shown in *Figure 5.4*.

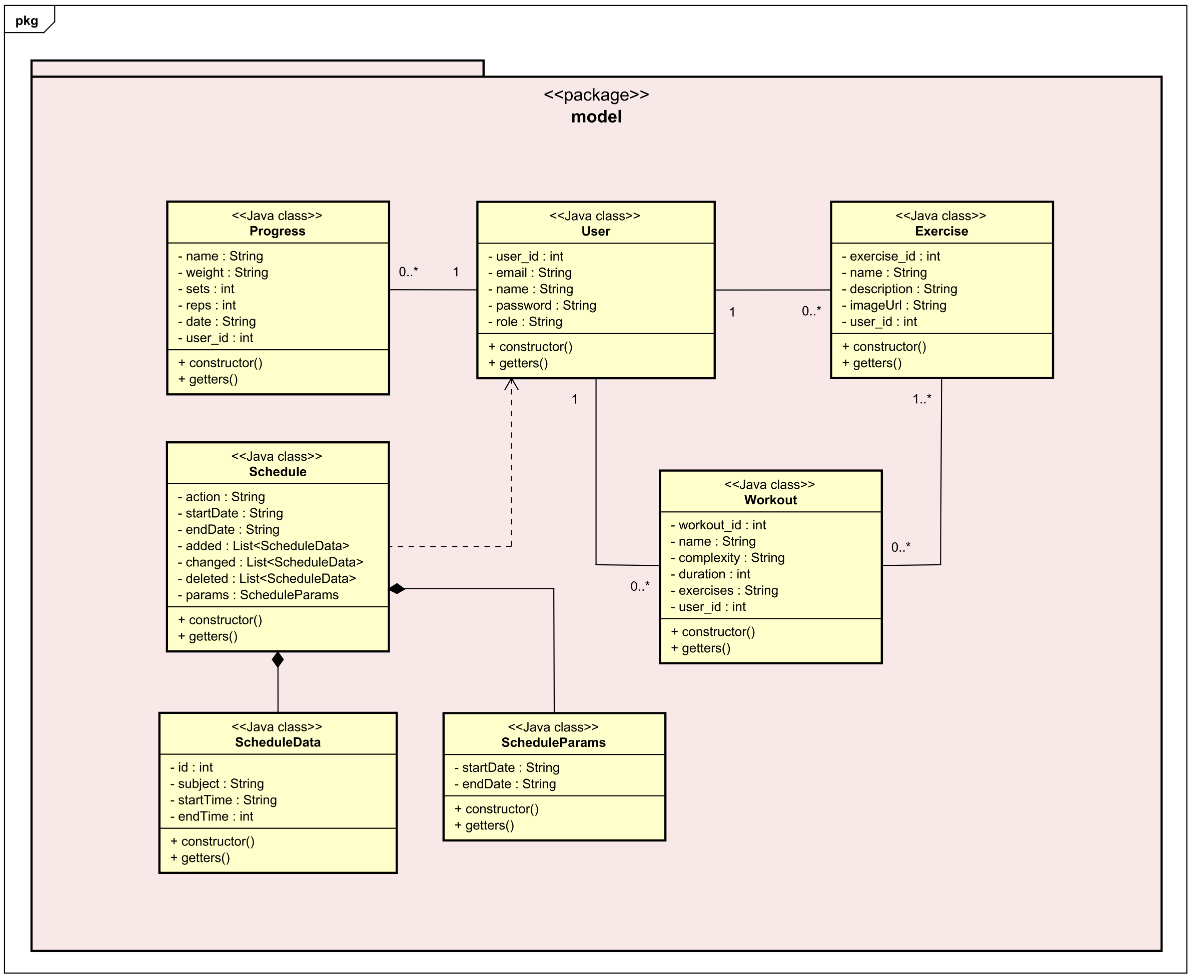


Figure 5.4: Server-side models UML class diagram

The User model represents the registered user in the application. Each user contains a unique id, email, password, and one of two possible roles: ROLE_USER or ROLE_ADMIN. Depending on the role, the user has different privileges through the application.

The Exercise model represents the exercise created by a specific user. It contains a unique id, exercise name, description, related image code, and the user's id that created it. The connection between exercise and user model is represented with one or zero to many relations, meaning one specific user can create from zero to many exercises, while one user can create one specific exercise.

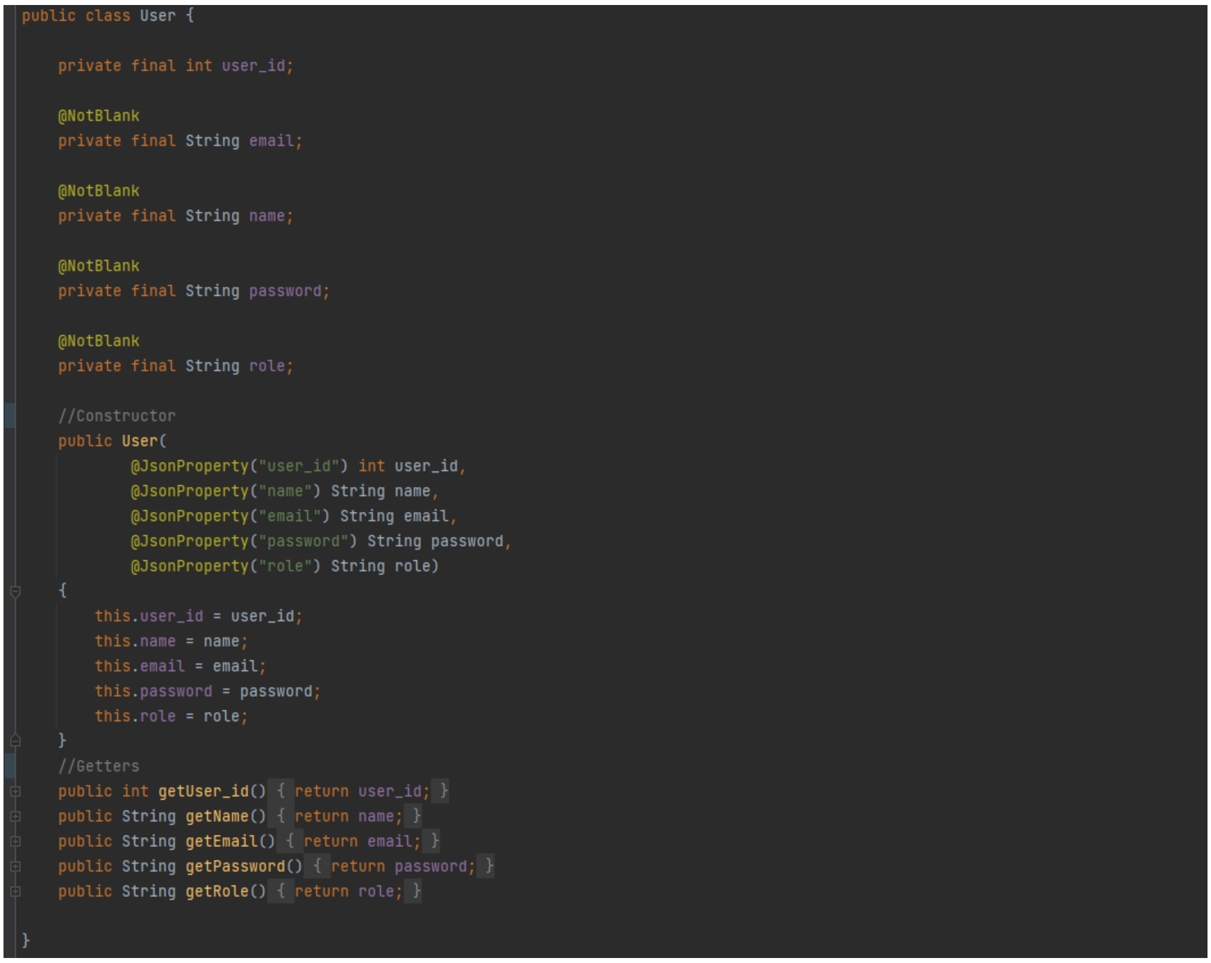
The Workout model represents the workout created by a specific user. Regarding attributes, it contains a unique id, workout name, complexity, duration in minutes, list of attached exercises with additional attributes specific to the workout, and those are weight, sets and reps, and the user's id that created the workout. The connection between the user model and workout model is represented with one – zero to many relations, meaning that one specific user can create from zero to many workouts, while only one user can create one specific workout. The Workout model is also connected with the exercise model through zero to one - one to many relations, which means that one specific workout can contain one to many exercises, and one specific exercise can belong to zero to many workouts.

The Progress model represents the model for handling each inserted progress for a specific exercise. In terms of attributes, it contains a unique exercise name, weight, exercise sets and reps, and the user's id that created the exercise progress. In addition, the Progress model is connected through zero to many – one relation with the user model, meaning that one user can create between zero and many progresses, and only one specific user can create one progress model.

The Schedule model represents the model for handling scheduled events, such as getting events, storing, deleting, and updating. It contains request action attribute, parameters, schedule view start and end dates, and three list attributes that depend on the schedule action, and those are: added, deleted, and changed.

The ScheduleData model represents the actual individual event data from the schedule, which is sent through the list of events. That is why it is connected to the schedule model through the composition, meaning its object's existence depends on the existence of the schedule model's object. ScheduleData model contains event id, subject, event name, and start and the end of the event. The Schedule model is also in composition with the ScheduleParams module since ScheduleParams is needed to define the params attribute of the Schedule model. It contains parameter start and end date.

Each model, alongside the attributes, contains its constructor function and the corresponding getter functions for public retrieval of the values from its private attributes. An example of a model is shown in *Figure 5.5.*



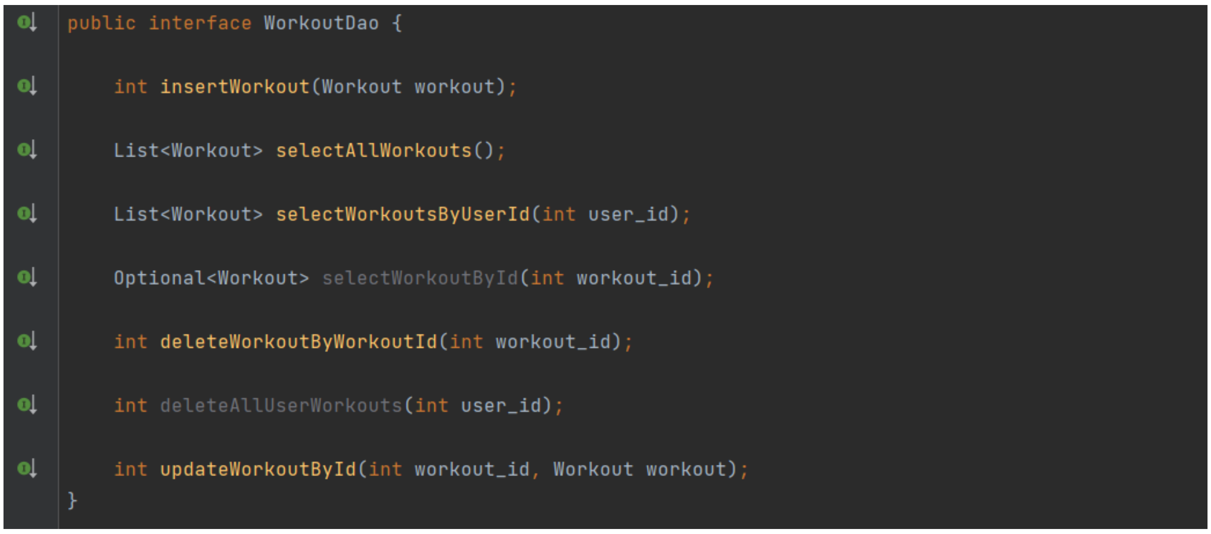
```java
public class User {

    private final int user_id;

    @NotBlank
    private final String email;

    @NotBlank
    private final String name;

    @NotBlank
    private final String password;

    @NotBlank
    private final String role;

    //Constructor
    public User(
            @JsonProperty("user_id") int user_id,
            @JsonProperty("name") String name,
            @JsonProperty("email") String email,
            @JsonProperty("password") String password,
            @JsonProperty("role") String role)
    {
        this.user_id = user_id;
        this.name = name;
        this.email = email;
        this.password = password;
        this.role = role;
    }
    //Getters
    public int getUser_id() { return user_id; }
    public String getName() { return name; }
    public String getEmail() { return email; }
    public String getPassword() { return password; }
    public String getRole() { return role; }

}
```

Figure 5.5: User model class

5.2.2 Data access

After defining the models for representing the data objects, the next step is to define the logic to transfer that data, that is, to receive and send the data.

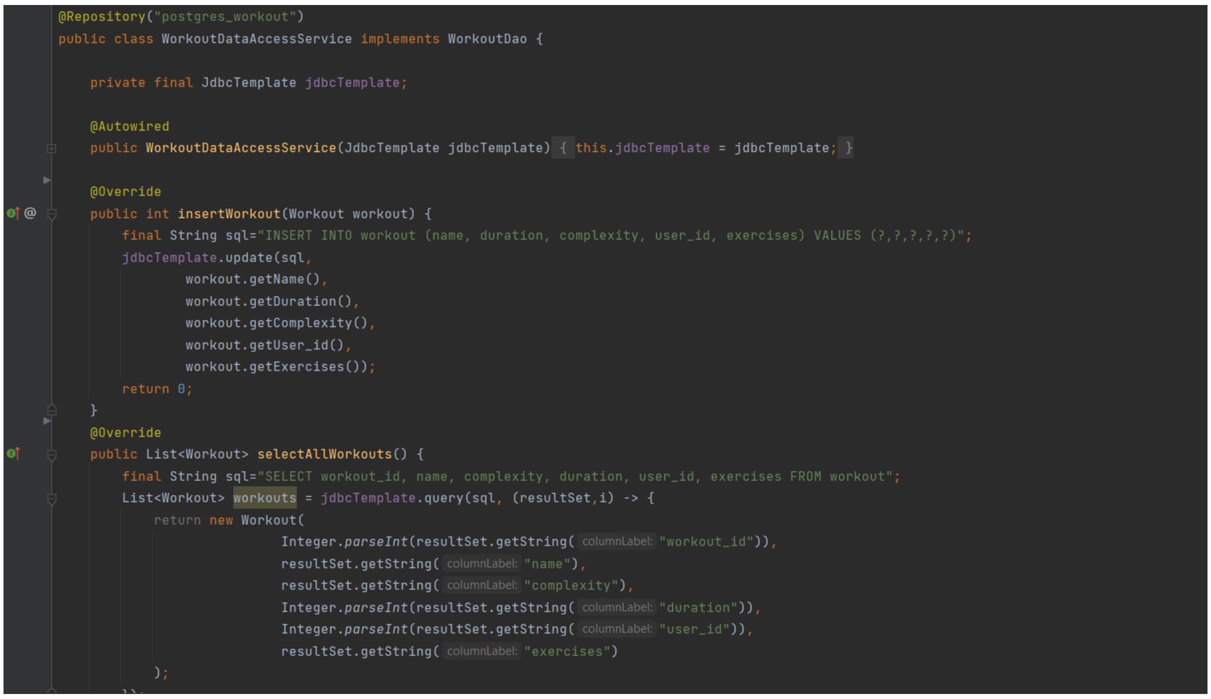
The first thing to implement is the data access objects (DAO). It is an interface responsible for encapsulating the persistence layer's details and providing a data manipulation interface (create, update, and delete operations) for each entity. Application's data access object interfaces are stored inside the DAO package. An example of DAO interface is shown in *Figure 5.6.*



```java
public interface WorkoutDao {

    int insertWorkout(Workout workout);

    List<Workout> selectAllWorkouts();

    List<Workout> selectWorkoutsByUserId(int user_id);

    Optional<Workout> selectWorkoutById(int workout_id);

    int deleteWorkoutByWorkoutId(int workout_id);

    int deleteAllUserWorkouts(int user_id);

    int updateWorkoutById(int workout_id, Workout workout);
}
```

Figure 5.6: Workout DAO interface

The next step is defining classes that will be an actual implementation of the DAO interface. Such classes are called data access services because they provide the service logic for data accessing and handling. Regarding the application, they are stored inside the repository package. Data access services or repository classes communicate with the database through overriding pre-defined DAO interface functions. The communication is established through the Java JDBC API. It stands for Java Database Connectivity, a Java API for connecting and executing queries with the database. It connects to the database through its JDBC driver. JDBC API can update database statements and retrieve the received results from the database. Data access service stores received database data inside the pre-defined models. An example of data access service with JDBC is shown in *Figure 5.7*.

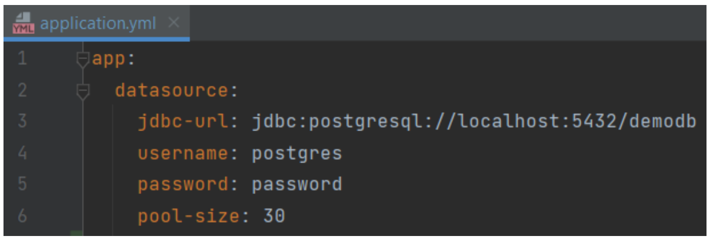


```java
@Repository("postgres_workout")
public class WorkoutDataAccessService implements WorkoutDao {

    private final JdbcTemplate jdbcTemplate;

    @Autowired
    public WorkoutDataAccessService(JdbcTemplate jdbcTemplate) { this.jdbcTemplate = jdbcTemplate; }

    @Override
    public int insertWorkout(Workout workout) {
        final String sql="INSERT INTO workout (name, duration, complexity, user_id, exercises) VALUES (?,?,?,?,?)";
        jdbcTemplate.update(sql,
                workout.getName(),
                workout.getDuration(),
                workout.getComplexity(),
                workout.getUser_id(),
                workout.getExercises());
        return 0;
    }
    @Override
    public List<Workout> selectAllWorkouts() {
        final String sql="SELECT workout_id, name, complexity, duration, user_id, exercises FROM workout";
        List<Workout> workouts = jdbcTemplate.query(sql, (resultSet,i) -> {
            return new Workout(
                            Integer.parseInt(resultSet.getString( columnLabel: "workout_id")),
                            resultSet.getString( columnLabel: "name"),
                            resultSet.getString( columnLabel: "complexity"),
                            Integer.parseInt(resultSet.getString( columnLabel: "duration")),
                            Integer.parseInt(resultSet.getString( columnLabel: "user_id")),
                            resultSet.getString( columnLabel: "exercises")
            );
```

Figure 5.7: Workout data access service class

Given the figure above, @Repository Spring annotation indicates that the class contains logic for storage data manipulation so that the Spring framework can autodetect it through classpath scanning. @Override annotation denotes that the child class method overrides the base class method. In this case, the child is WorkoutDataAccessService and the base class WorkoutDao.

Regarding the database connectivity, Spring Boot provides Hikari. This JDBC data source implementation provides a connection pooling mechanism with the PostgreSQL driver for a successful connection with the PostgreSQL application database. An example of database connection properties is shown in *Figure 5.8*.



application.yml

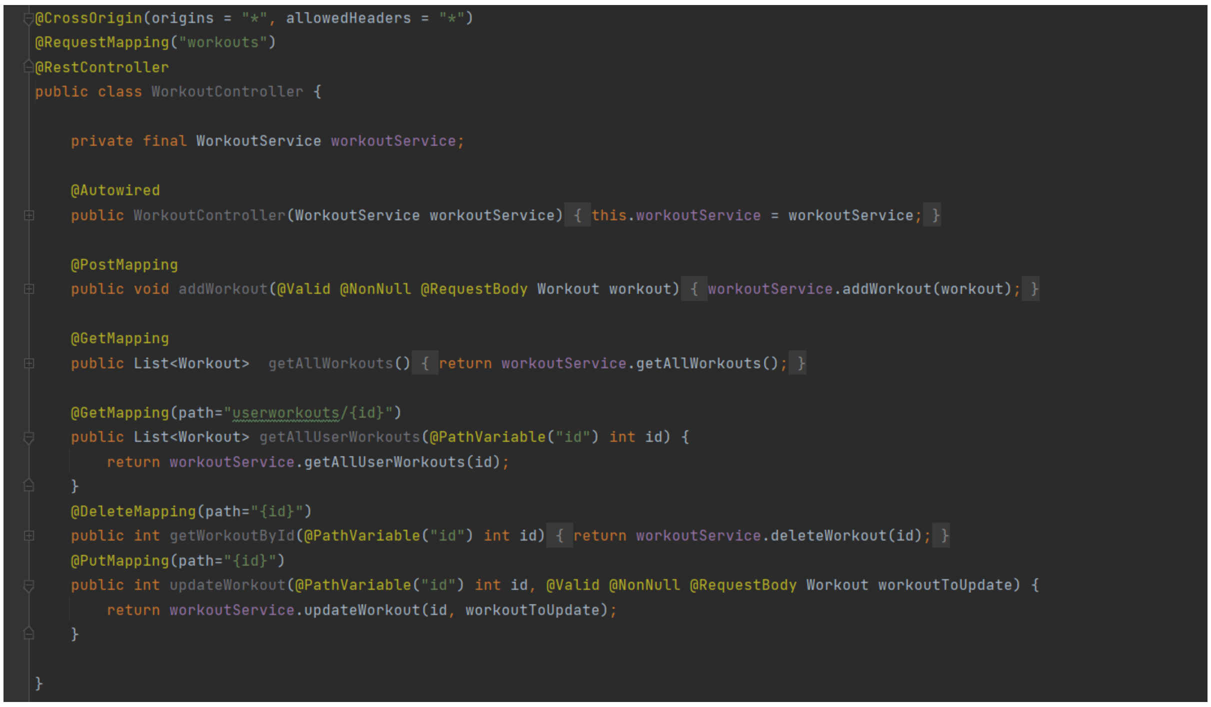
```yaml
app:
  datasource:
    jdbc-url: jdbc:postgresql://localhost:5432/demodb
    username: postgres
    password: password
    pool-size: 30
```

Figure 5.8: Example of database connection properties

Given the figure above, @Configuration annotation indicates that a class declares one or more @Bean annotation methods and may be processed by the Spring container to generate bean definitions and service requests for those beans. @Bean marks the methods for Spring to add to the context for us.

5.2.3 Request handling

After successfully defining the models and data access services with their logic, the next step is to use them through the request handling process. The request handling process is the central part of the presentation layer of the overall Spring Boot application structure. The specific class is called a controller. In terms of the request, it comes from the client-side, in the form of an HTTP request. The controller contains functions to handle different HTTP requests, like POST, PUT, DELETE, and GET, on different URL paths. Inside the function, the controller maps the request or response payload to the model object, passing it further, either to the business layer's services or back to the client-side. This application has its controllers stored inside the api package. An example of controller is shown in *Figure 5.9.*

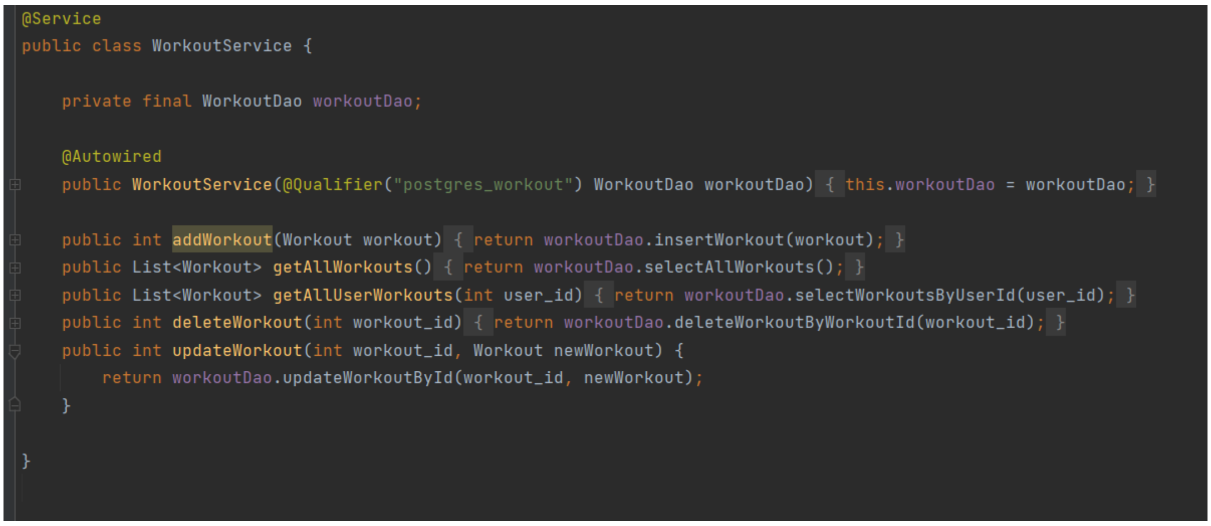


```java
@CrossOrigin(origins = "*", allowedHeaders = "*")
@RequestMapping("workouts")
@RestController
public class WorkoutController {

    private final WorkoutService workoutService;

    @Autowired
    public WorkoutController(WorkoutService workoutService) { this.workoutService = workoutService; }

    @PostMapping
    public void addWorkout(@Valid @NonNull @RequestBody Workout workout) { workoutService.addWorkout(workout); }

    @GetMapping
    public List<Workout>  getAllWorkouts() { return workoutService.getAllWorkouts(); }

    @GetMapping(path="userworkouts/{id}")
    public List<Workout> getAllUserWorkouts(@PathVariable("id") int id) {
        return workoutService.getAllUserWorkouts(id);
    }
    @DeleteMapping(path="{id}")
    public int getWorkoutById(@PathVariable("id") int id) { return workoutService.deleteWorkout(id); }
    @PutMapping(path="{id}")
    public int updateWorkout(@PathVariable("id") int id, @Valid @NonNull @RequestBody Workout workoutToUpdate) {
        return workoutService.updateWorkout(id, workoutToUpdate);
    }

}
```

Figure 5.9: Workout controller class

Regarding the figure above, the @RestController annotation indicates to Spring that this class handles the requests made by the client. @RequestMapping annotation is used to map web request path to Spring controller methods. Regarding the functions, @PostMapping indicates that the function will handle HTTP POST requests. If no path is specified, it will use the path given through the @RequestMapping. The post function's parameter is annotated with @RequestBody, meaning it will map the JSON object from the request to the model object. If an active part of the path is specified, for example, id, inside the mapping annotation, the path variable is accessible through the @PathVariable annotation inside the function parameters.

As mentioned, inside the business layer, the controller-sent requests are processed through the service classes.- They contain the logic for handling the data, for example, setting and inserting the data, data manipulation, and more. The service class handles most of the functionalities through the data access object reference, which can send the received controller request to the database and back. An example of service class is shown in *Figure 5.10.*



```java
@Service
public class WorkoutService {

    private final WorkoutDao workoutDao;

    @Autowired
    public WorkoutService(@Qualifier("postgres_workout") WorkoutDao workoutDao) { this.workoutDao = workoutDao; }

    public int addWorkout(Workout workout) { return workoutDao.insertWorkout(workout); }
    public List<Workout> getAllWorkouts() { return workoutDao.selectAllWorkouts(); }
    public List<Workout> getAllUserWorkouts(int user_id) { return workoutDao.selectWorkoutsByUserId(user_id); }
    public int deleteWorkout(int workout_id) { return workoutDao.deleteWorkoutByWorkoutId(workout_id); }
    public int updateWorkout(int workout_id, Workout newWorkout) {
        return workoutDao.updateWorkoutById(workout_id, newWorkout);
    }

}
```

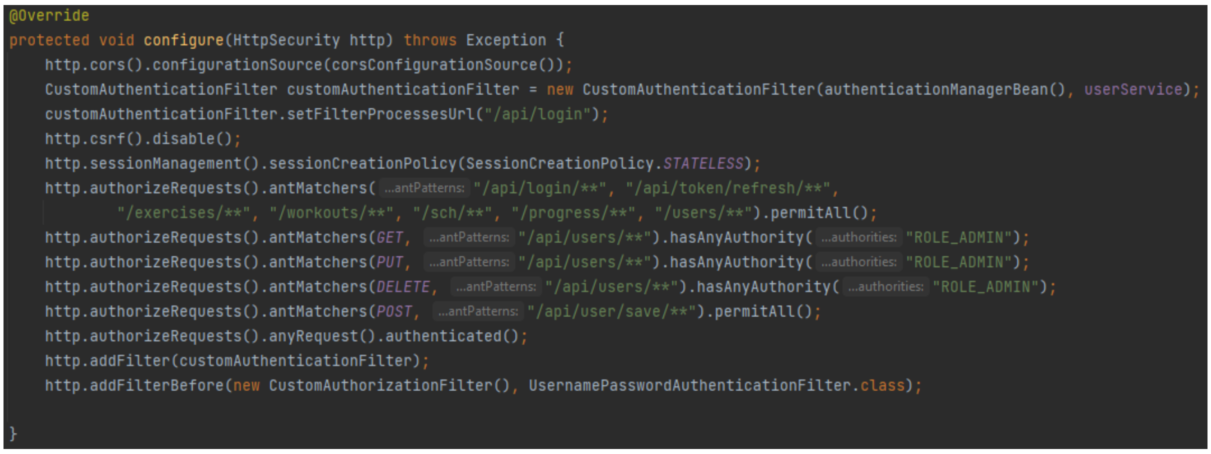
Figure 5.10: Workout service class

Given the figure above, the @Service annotation lets Spring know that this class contains the business logic functionality. Through the @Qualifier annotation, it is specified which bean needs to be injected based on the given name. In our application, the service @Qualifier annotation is always linked to the @Repository annotation.

5.2.4 Security

The complete access-control authorization and authentication of the application are handled through Spring Security. It is a highly customizable framework standard for securing Spring-based applications.

Regarding the application, the whole application security is handled inside of the SecurityConfig class through the configure method, which is used to set up the URL paths security alongside custom authentication and authorization filters, as shown in Figure *5.11*.



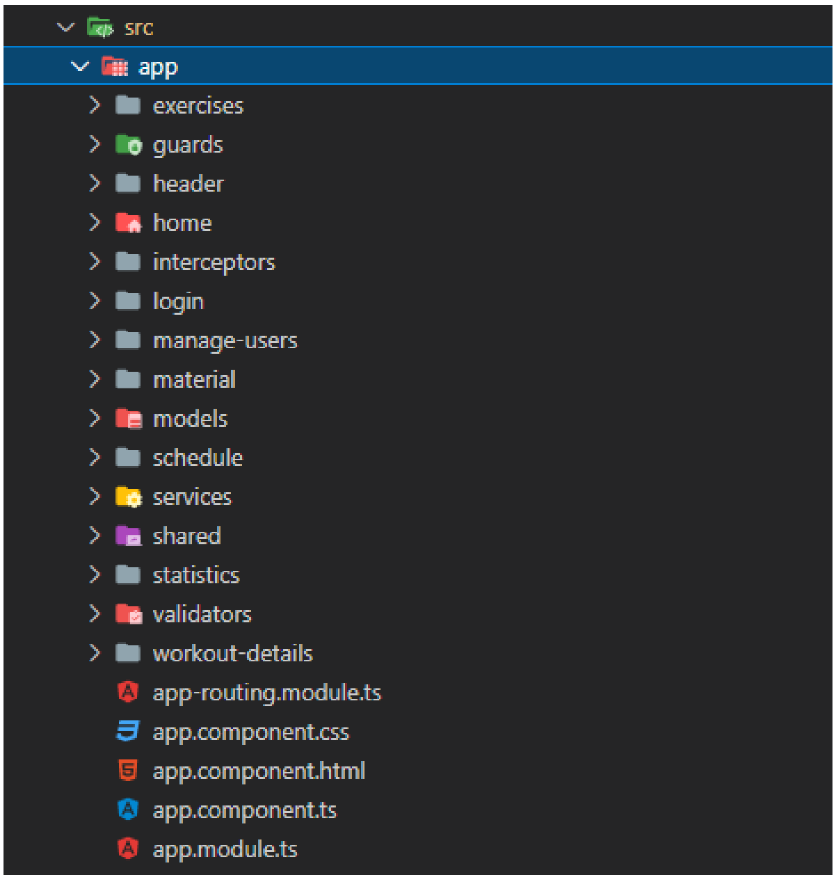
```java
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.cors().configurationSource(corsConfigurationSource());
    CustomAuthenticationFilter customAuthenticationFilter = new CustomAuthenticationFilter(authenticationManagerBean(), userService);
    customAuthenticationFilter.setFilterProcessesUrl("/api/login");
    http.csrf().disable();
    http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    http.authorizeRequests().antMatchers( ...antPatterns: "/api/login/**", "/api/token/refresh/**",
            "/exercises/**", "/workouts/**", "/sch/**", "/progress/**", "/users/**").permitAll();
    http.authorizeRequests().antMatchers(GET, ...antPatterns: "/api/users/**").hasAnyAuthority( ...authorities: "ROLE_ADMIN");
    http.authorizeRequests().antMatchers(PUT, ...antPatterns: "/api/users/**").hasAnyAuthority( ...authorities: "ROLE_ADMIN");
    http.authorizeRequests().antMatchers(DELETE, ...antPatterns: "/api/users/**").hasAnyAuthority( ...authorities: "ROLE_ADMIN");
    http.authorizeRequests().antMatchers(POST, ...antPatterns: "/api/user/save/**").permitAll();
    http.authorizeRequests().anyRequest().authenticated();
    http.addFilter(customAuthenticationFilter);
    http.addFilterBefore(new CustomAuthorizationFilter(), UsernamePasswordAuthenticationFilter.class);

}
```

Figure 5.11: Security configuration

As for user login, the application uses JSON web token (JWT) for authentication to authenticate the user making the request from the client-side. It is an open standard that defines a way for securely transmitting information and representing the sender's identity as a JSON object. It is signed with an HMAC-SHA256 cryptographic hash function. It divides into two forms: the access token, which is used as a proof of identity, and the refresh token, which is used to renew the access token once it is expired. The server-side of the application generates a new JWT for each successful user login and sends it back to the user on the client side. When the logged-in user sends any request to the server, each request must include a valid JWT access token in the header to be authenticated and receive a wanted response. Based on the token, Spring Security handles the user authorization, meaning that with each request, the user is checked to see if he has the right of access to the requested source.

5.3 Client-side

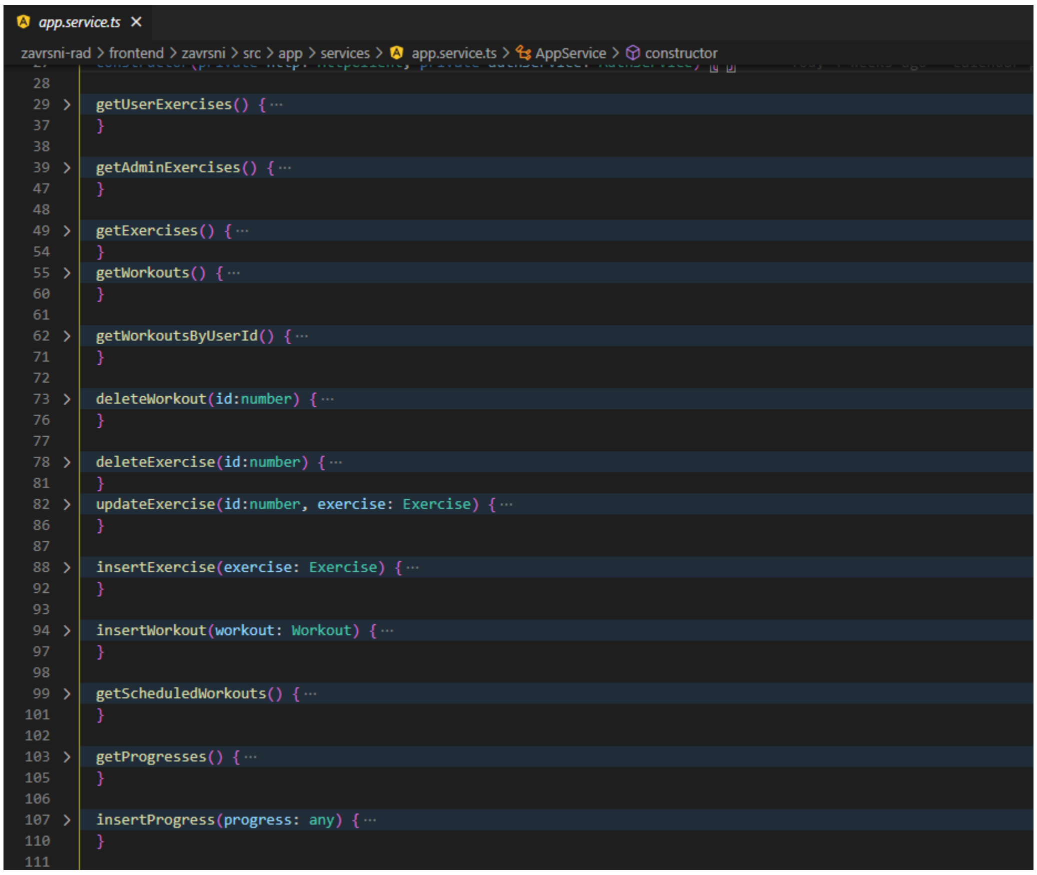
The client-side of the application is built modularly. The organization of the client-side code is shown in Figure *5.12*.



```
src
  app
    exercises
    guards
    header
    home
    interceptors
    login
    manage-users
    material
    models
    schedule
    services
    shared
    statistics
    validators
    workout-details
    app-routing.module.ts
    app.component.css
    app.component.html
    app.component.ts
    app.module.ts
```

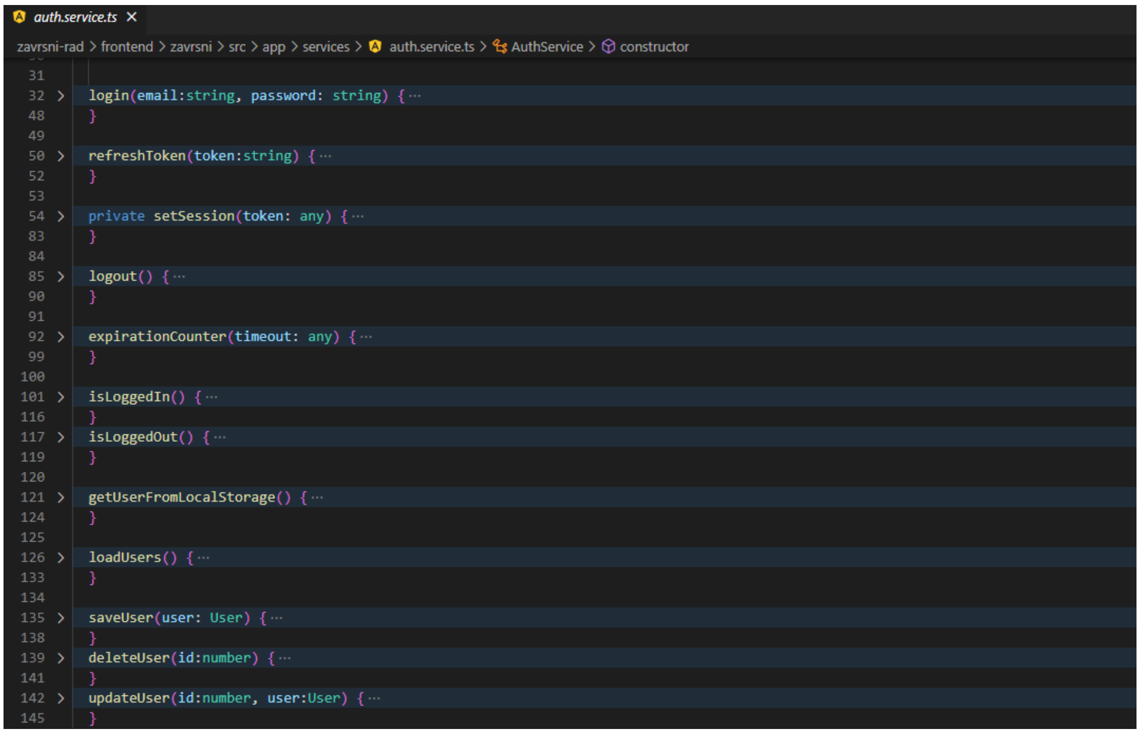
Figure 5.12: Frontend organization

The first step of client-side development is establishing the general data management needs throughout the application. For such use cases, Angular provides services. They are used to maintain the data throughout the lifetime of the application. For this application's needs, two services are implemented. First, since all the data is received through communication with the server, the application has the app service implemented for that purpose. If any of the components need the server data to display, they will fetch it through the app service's functions, as shown in Figure *5.13*. Secondly, the application contains the authentication service, which is used for handling log-in, registration, and overall authentication and authorization security of the user. Its functionalities are shown in Figure *5.14*.



```
app.service.ts
zavrsni-rad > frontend > zavrsni > src > app > services > app.service.ts > AppService > constructor

getUserExercises() {…
}

getAdminExercises() {…
}

getExercises() {…
}
getWorkouts() {…
}

getWorkoutsByUserId() {…
}

deleteWorkout(id:number) {…
}

deleteExercise(id:number) {…
}
updateExercise(id:number, exercise: Exercise) {…
}

insertExercise(exercise: Exercise) {…
}

insertWorkout(workout: Workout) {…
}

getScheduledWorkouts() {…
}

getProgresses() {…
}

insertProgress(progress: any) {…
}
```

Figure 5.13: App service



```
auth.service.ts
zavrsni-rad > frontend > zavrsni > src > app > services > auth.service.ts > AuthService > constructor

login(email:string, password: string) {…
}

refreshToken(token:string) {…
}

private setSession(token: any) {…
}

logout() {…
}

expirationCounter(timeout: any) {…
}

isLoggedIn() {…
}
isLoggedOut() {…
}

getUserFromLocalStorage() {…
}

loadUsers() {…
}

saveUser(user: User) {…
}
deleteUser(id:number) {…
}
updateUser(id:number, user:User) {…
}
```

Figure 5.14: Authentication service

After defining data and users handling functionalities, the next step is to create views through which the data will be shown. In Angular, that is achieved through the components. As shown in Figure *5.12*, the view components of this application are divided into the next modules: home, header, exercises, workout-details, schedule, statistics, login, and manage-users. The home component is the root component of the user interface since it contains the paths to all the others. The general layout of the home component is shown in Figure *5.15*.

For the views, the Angular community offers a wide variety of prebuilt styles and components. Regarding the application, it is worth mentioning the Angular Calendar component, provided by the Syncfusion [15], for schedule component, and Angular material components for the general styling and structure of the views [16].

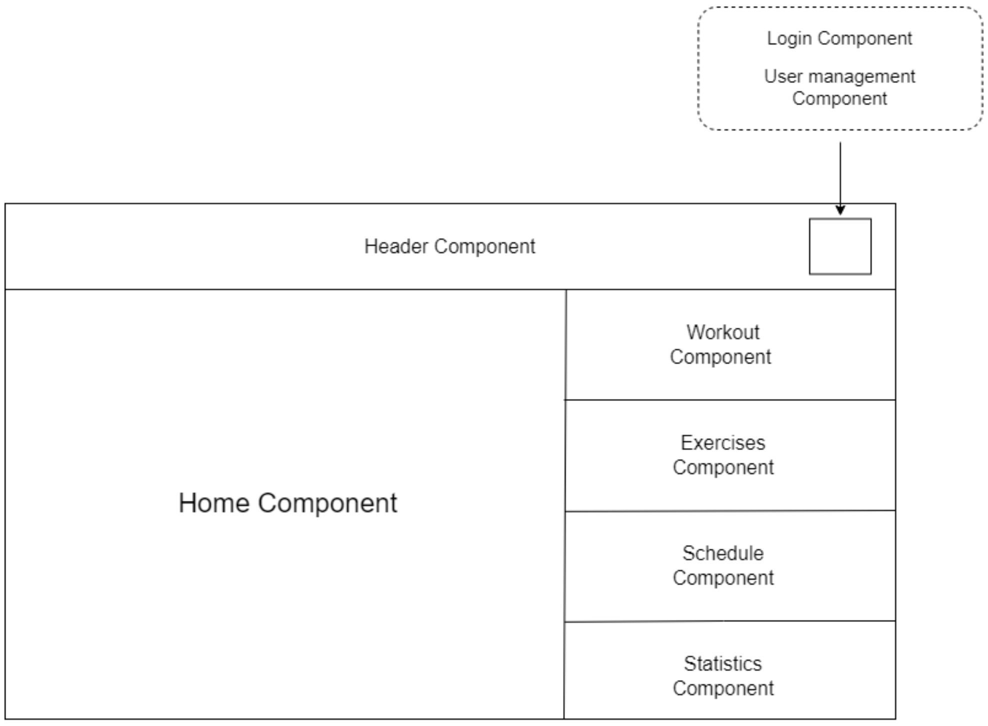
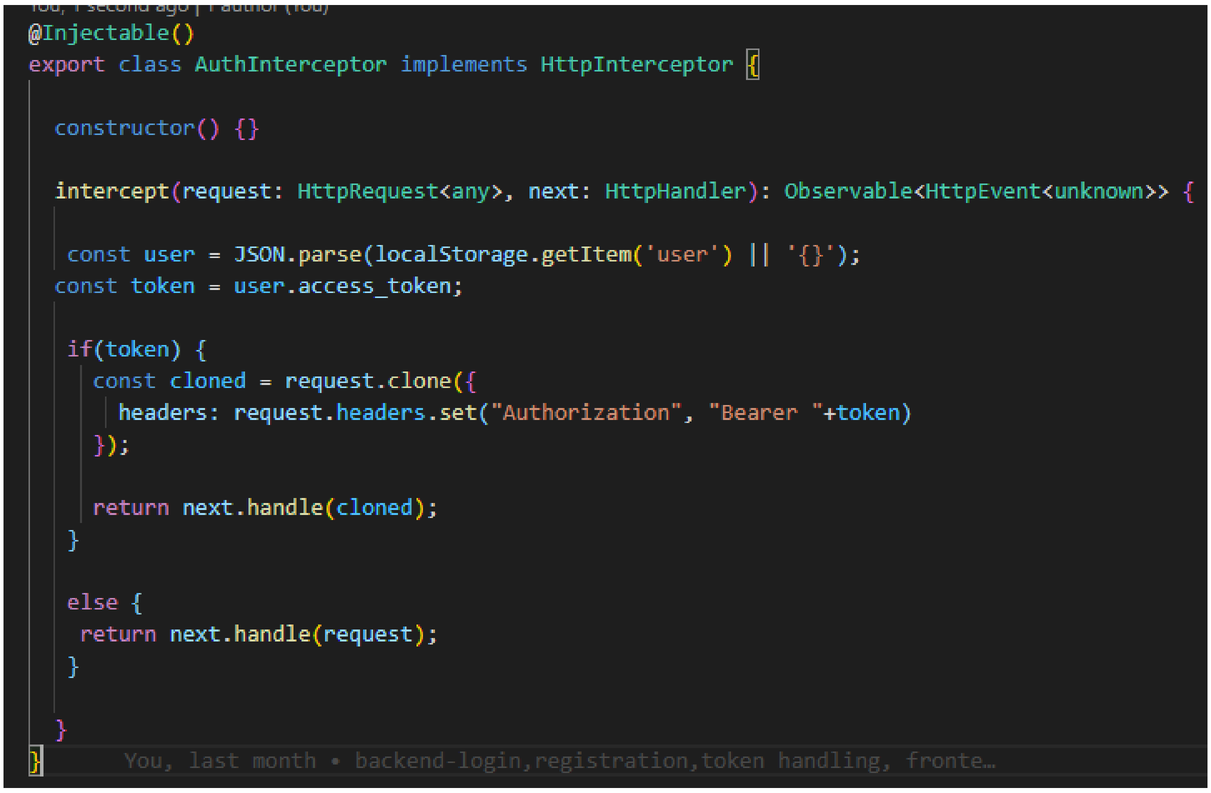


Figure 5.15: Home layout

With the aim of protecting the views and data from unauthorized entities, the application uses the login component. The user is verified on the server by providing an email and password. Since the server-side has the Spring Security JWT implementation, it will send the JWT tokens to the client-side if the login is successful. With those tokens, the client-side can verify that it is authorized to access specifically requested sources. The server requires that each request contains the access token, and for that, Angular uses the HTTP interceptor directive. Its intercept function defines how the request body and header parameters are edited, meaning each request sent from the client-side will have a customized request format. In this case, it appends the access token to the header of each request, as shown in Figure *5.16*.



```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor() {}

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<unknown>> {

   const user = JSON.parse(localStorage.getItem('user') || '{}');
  const token = user.access_token;

   if(token) {
     const cloned = request.clone({
       headers: request.headers.set("Authorization", "Bearer "+token)
     });

     return next.handle(cloned);
   }

   else {
    return next.handle(request);
   }

  }
}       You, last month • backend-login,registration,token handling, fronte…
```

Figure 5.16: Authentication interceptor

Each token has its expiration, and once it is expired, the application will automatically log out the user from the application.

In terms of the authorization, the application provides the URL guards, precisely, Angular guard directives with the canActivate function. It is a function that for each route navigation checks the access condition for that route. The auth guard for this application routes is shown in Figure *5.17*.

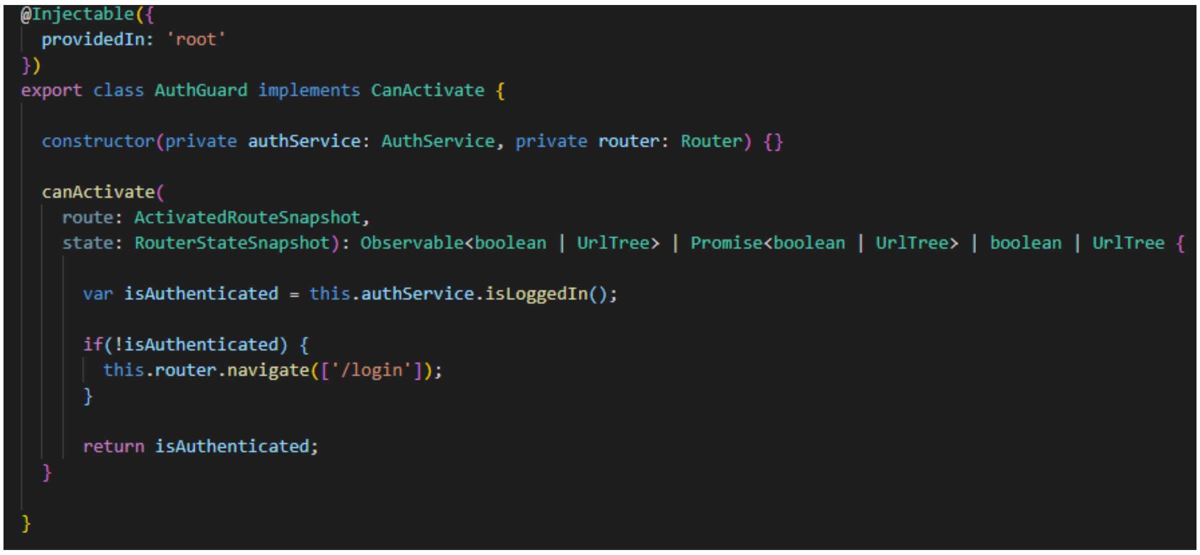

```
@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {

  constructor(private authService: AuthService, private router: Router) {}

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {

      var isAuthenticated = this.authService.isLoggedIn();

      if(!isAuthenticated) {
        this.router.navigate(['/login']);
      }

      return isAuthenticated;
  }

}
```

Figure 5.17: Auth guard

5.3.1 Application interface

This section presents the final user interface of the application. Figure *5.18**Error! Reference source not found.*** shows the login view of the page. It is the first stage for any non-authenticated user since the application has authorization guards. Next, the login form fields are validated with the email and required validators. The successful login redirects the user to the home page shown in *Figure 5.19.* If the login is unsuccessful, the user is kept on the login page with the proper error message, shown in *Figure 5.20.* On the home, page user is prompted with a starting selection section. It provides four main modules of the application: workouts, exercises, schedule, and statistics. Their main views are shown in *Figure 5.21, Figure 5.23, Figure 5.28,* and *Figure 5.32.*

Regarding the exercises view, the user can select between his or the base exercises created by the administrator. Most importantly, the user can create a new exercise through the dialog window triggered by a button click. The create window is shown in *Figure 5.24.* The form's name field is validated with the unique name custom validator and required validator. Users can edit or delete their custom exercises through the exercise card header buttons, which will trigger the dialog windows shown in Figure *5.26* and *Figure 5.27.* The edit dialog form has the unique name custom validator and required validator. Lastly, the user can enter the exercise progression with a button click, opening the dialog window shown in *Figure 5.25.* For progress, the user can select between his exercises combined with the default application exercises. If the exercise already has the progress tracked by the same user, it will be shown through the latest progress section of the window. Additionally, the user can filter the exercises. If the user does not have exercise saved yet, he will be advised with the message to create the first one.

On the workouts page load, a workouts table is shown to the user if the currently logged-in user has created workouts till that moment. Otherwise, a message advises the user to create his first workout. By clicking on the specific workout row of the table, a piece of comprehensive information will show, containing exercises appended to the selected workout, just like in *Figure 5.21.* In the extended window, the user also has the option to delete the selected workout, which with its trigger opens the alert dialog window of the same type as exercise one, shown in Figure *5.27.* Users can create a new workout by clicking on the related button, which will open the workout creation dialog window, shown in Figure *5.22.* Through it, the user can provide basic workout information and append existing exercises to it in proper order. Users can change the order of the exercises as well as delete a specific one from the list.

Next on the list is the schedule. Users can add a workout event by selecting the time frame of their workout and inserting the event name, as shown in Figure *5.29.* Additionally, if the user wants to add an already existing workout, he can choose the more details options shown in *Figure 5.30.* Users can also update and delete the event, which is shown as options in Figure *5.31*.

The user has the Statistics page option, shown in *Figure 5.32*. By selecting the exercise, the user can get the progression insight, shown in Figure *5.33.* The progression graph is shown through three separate attributes, the exercise sets, repetitions, and weight.

Every user has the logout option available by clicking on his avatar. If the user has an administrator role, he also has the manage user option, shown in *Figure 5.34.* By selecting it, the administrator is redirected to the manage users page, shown in Figure *5.35,* where he can register, update, and delete the users. The dialog windows for those actions are almost identical to the exercises page ones.

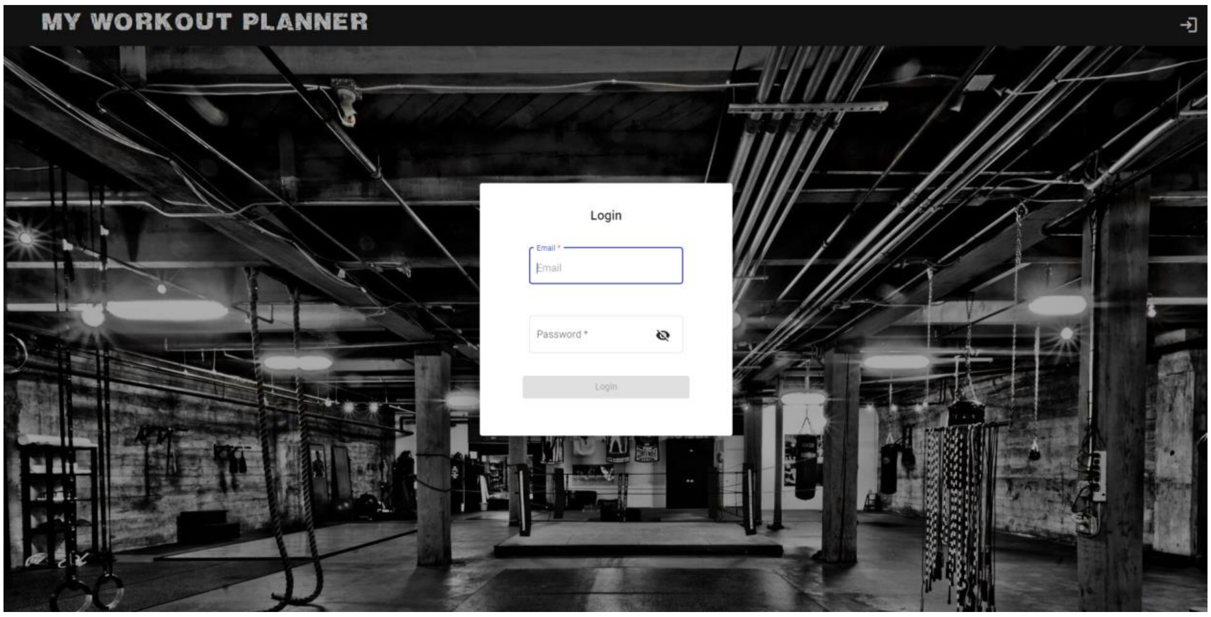


Figure 5.18: Login page

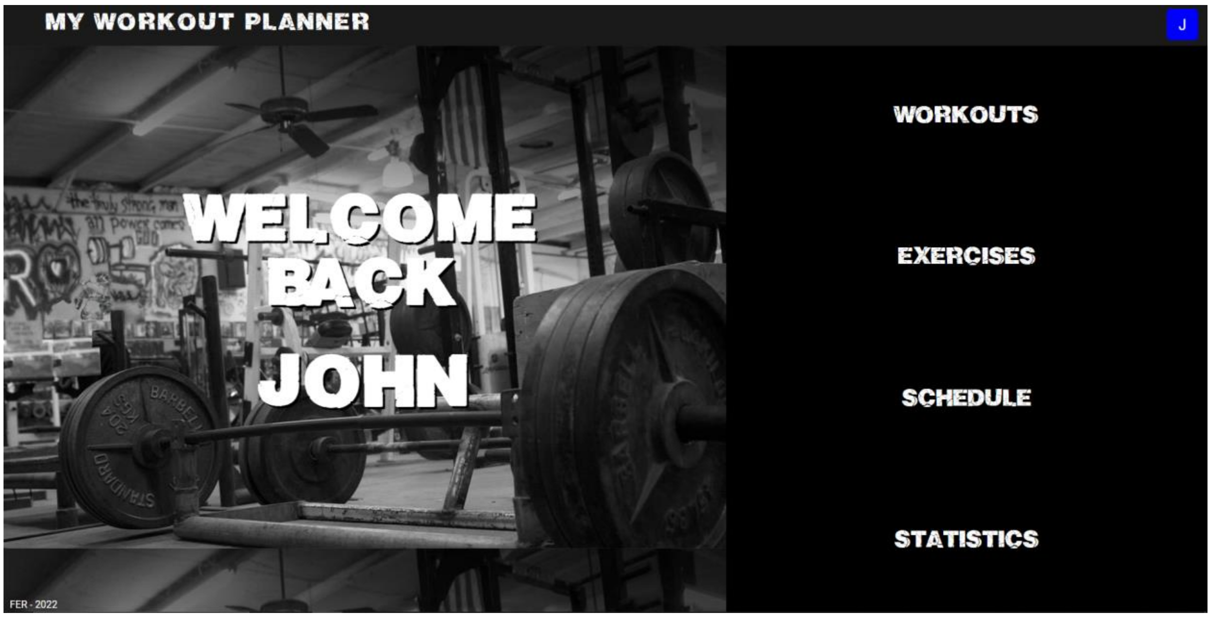
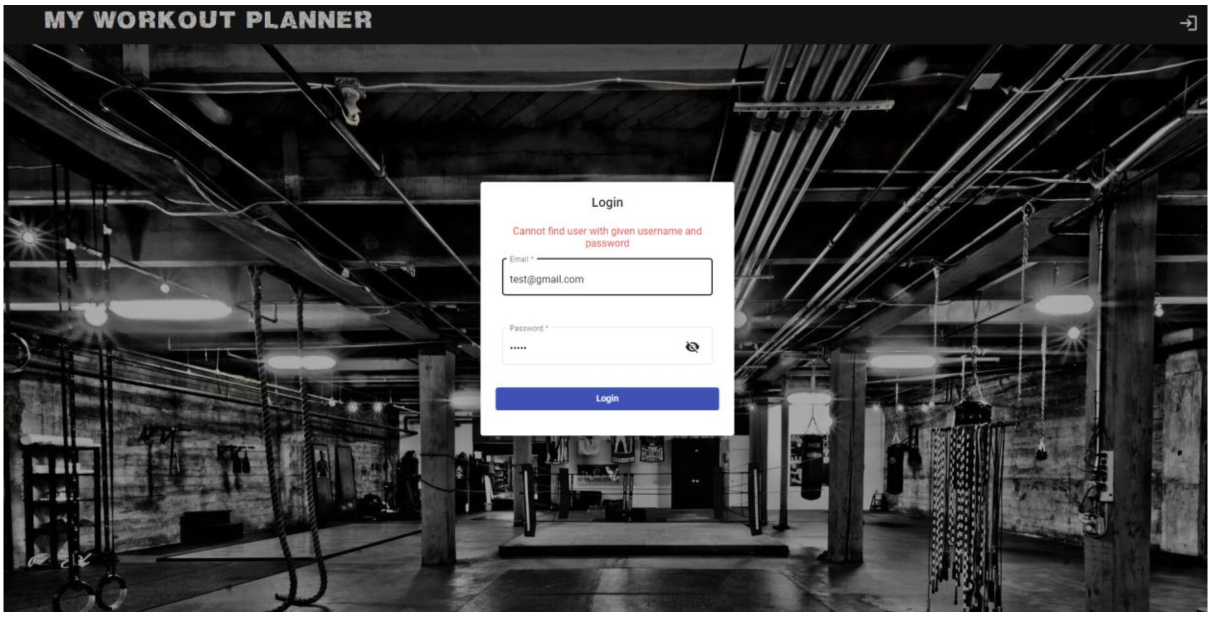


Figure 5.19: Home page

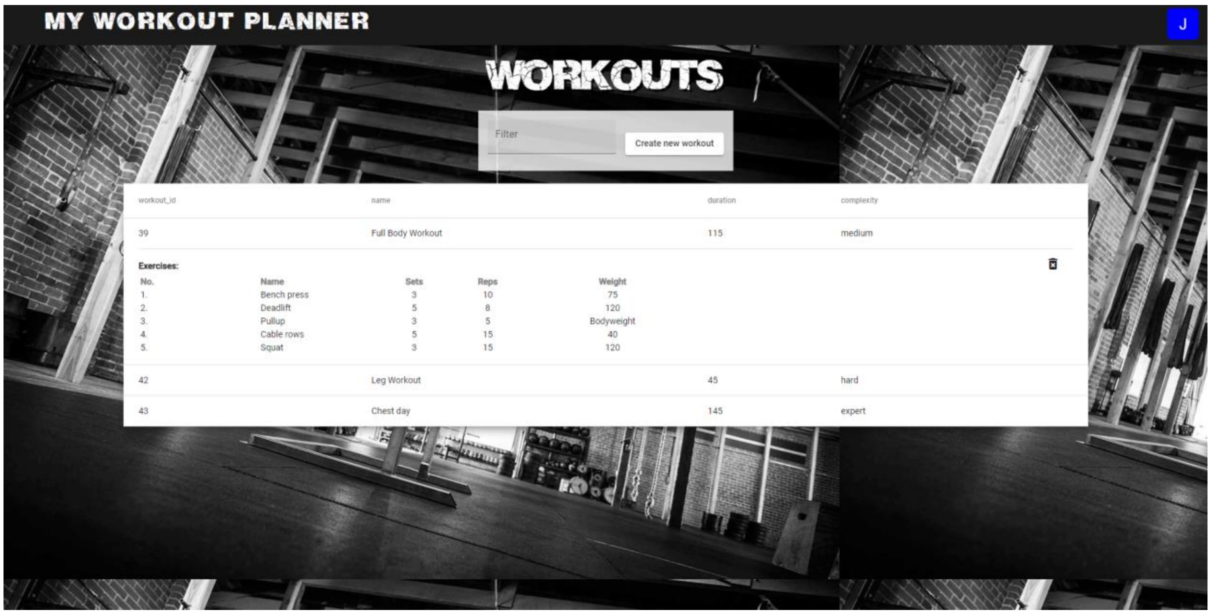


Cannot find user with given username and password

Email: test@gmail.com

Password: •••••

Figure 5.20: Login error

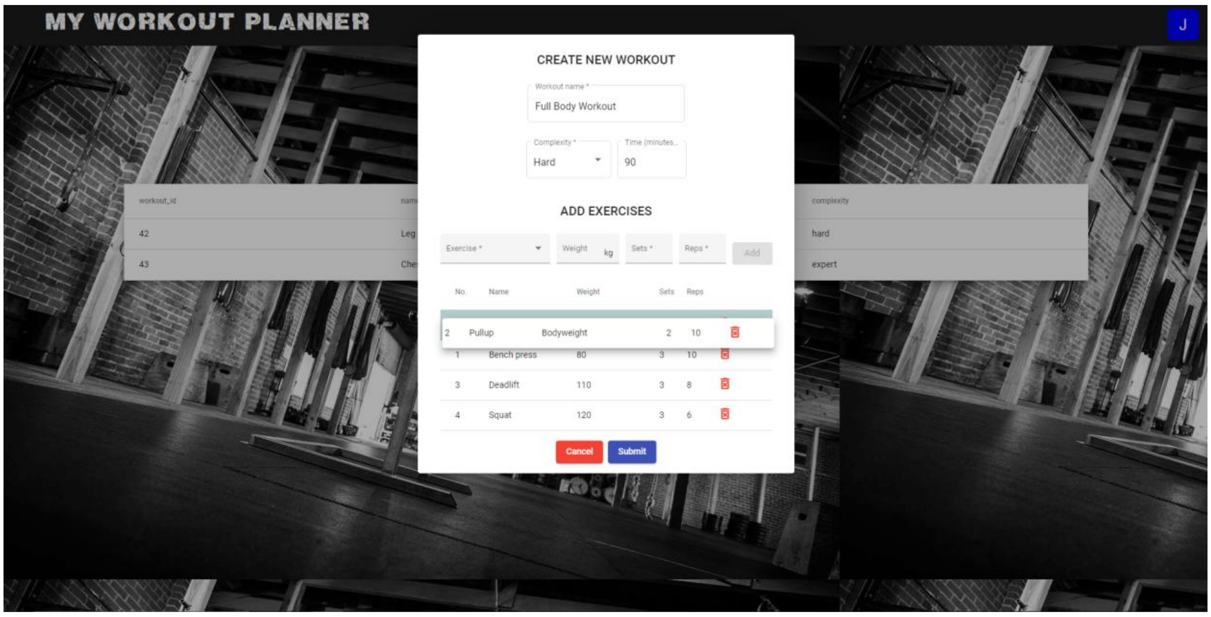


workout_id	name	duration	complexity
39	Full Body Workout	115	medium
42	Leg Workout	45	hard
43	Chest day	145	expert

Exercises:

No.	Name	Sets	Reps	Weight
1.	Bench press	3	10	75
2.	Deadlift	5	8	120
3.	Pullup	3	5	Bodyweight
4.	Cable rows	5	15	40
5.	Squat	3	15	120

Figure 5.21: Workout page



CREATE NEW WORKOUT

Workout name *: Full Body Workout

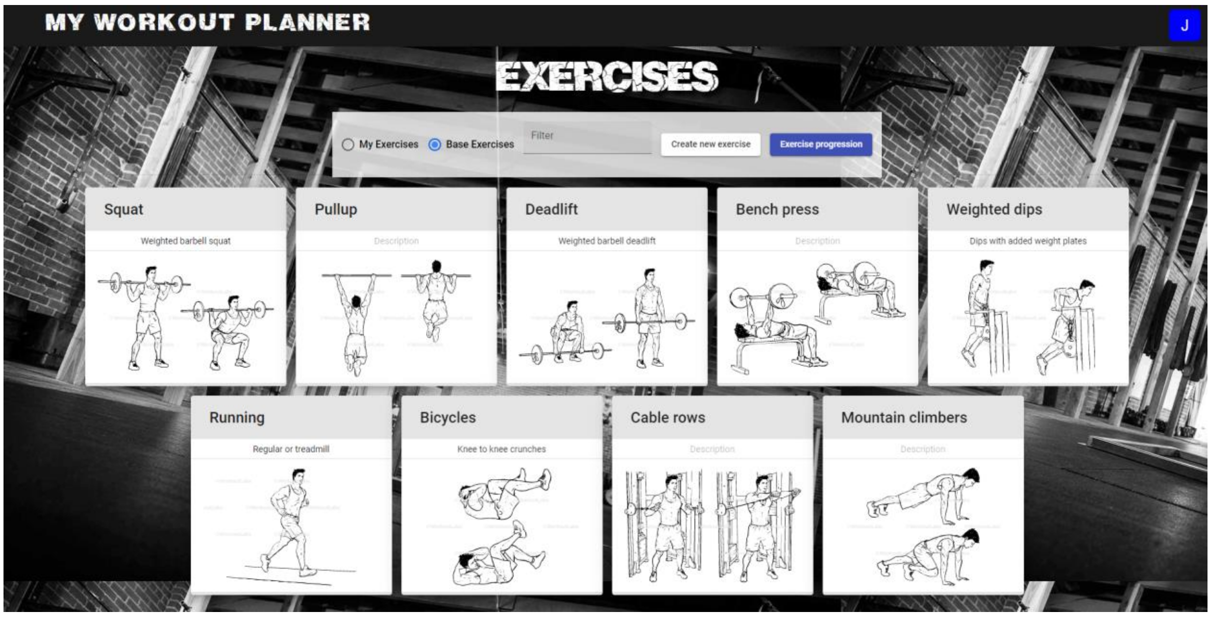
Complexity *: Hard

Time (minutes...: 90

ADD EXERCISES

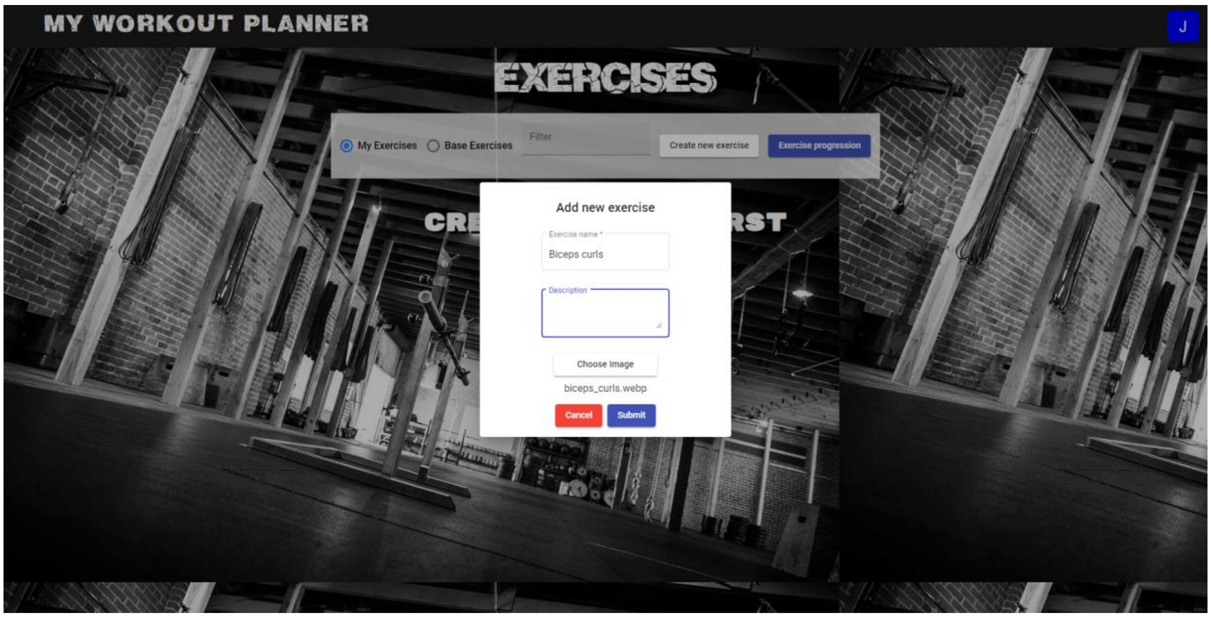
No.	Name	Weight	Sets	Reps
2	Pullup	Bodyweight	2	10
1	Bench press	80	3	10
3	Deadlift	110	3	8
4	Squat	120	3	6

Figure 5.22: Workout create



Squat	Weighted barbell squat
Pullup	Description
Deadlift	Weighted barbell deadlift
Bench press	Description
Weighted dips	Dips with added weight plates
Running	Regular or treadmill
Bicycles	Knee to knee crunches
Cable rows	Description
Mountain climbers	Description

Figure 5.23: Exercise page



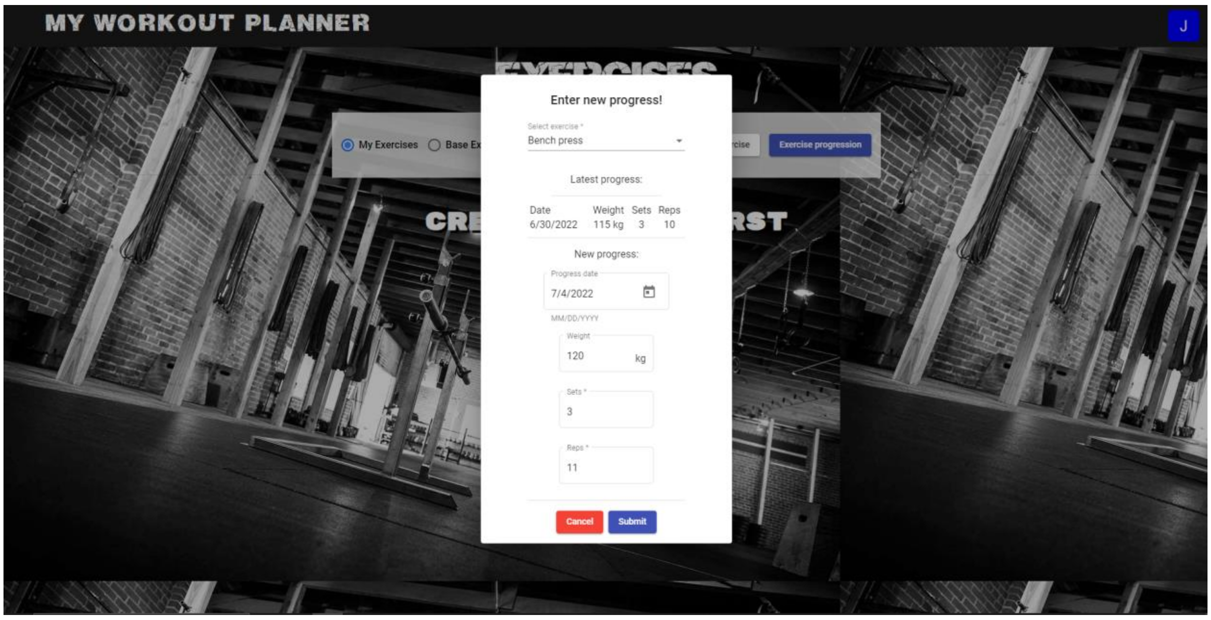
Add new exercise

Exercise name *: Biceps curls

Description:

biceps_curls.webp

Figure 5.24: Exercise create



Enter new progress!

Select exercise *: Bench press

Latest progress:

Date	Weight	Sets	Reps
6/30/2022	115 kg	3	10

New progress:

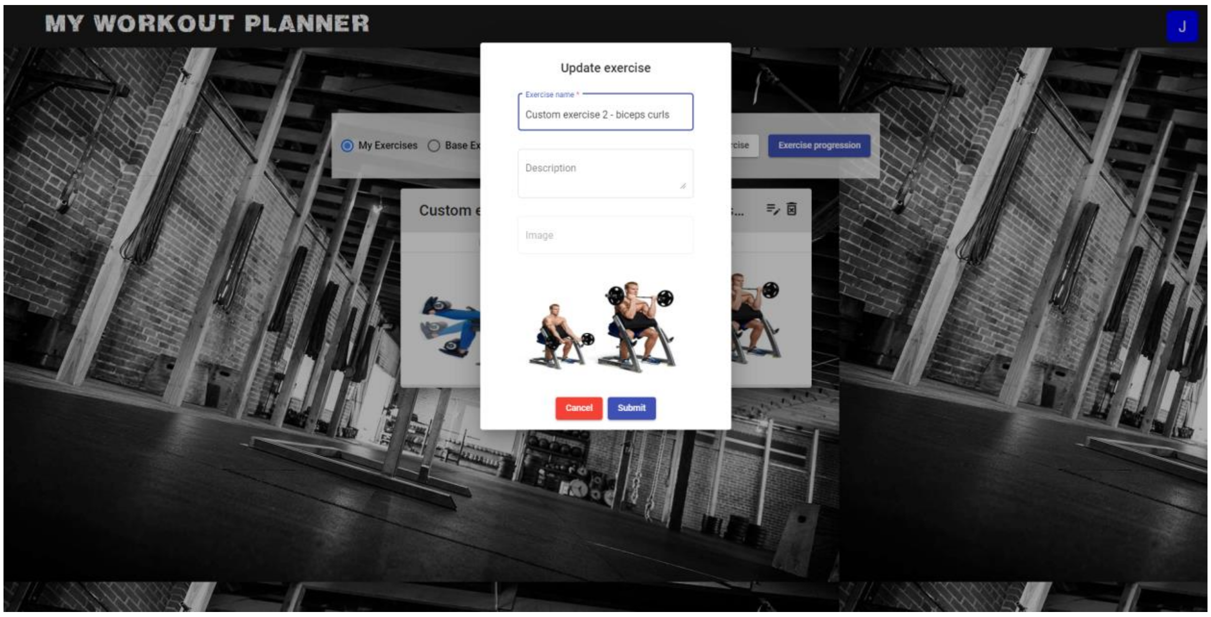
Progress date: 7/4/2022
MM/DD/YYYY

Weight: 120 kg

Sets *: 3

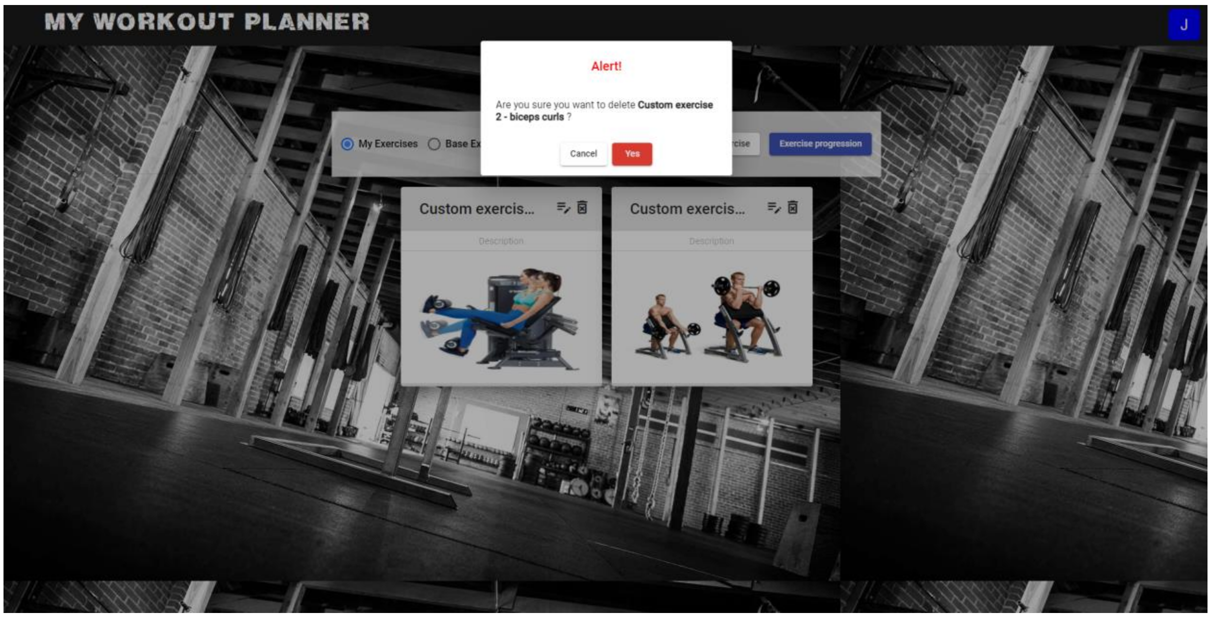
Reps *: 11

Figure 5.25: Exercise progress



Exercise name *
Custom exercise 2 - biceps curls

Figure 5.26: Exercise edit



Alert!

Are you sure you want to delete **Custom exercise 2 - biceps curls** ?

Figure 5.27: Exercise delete

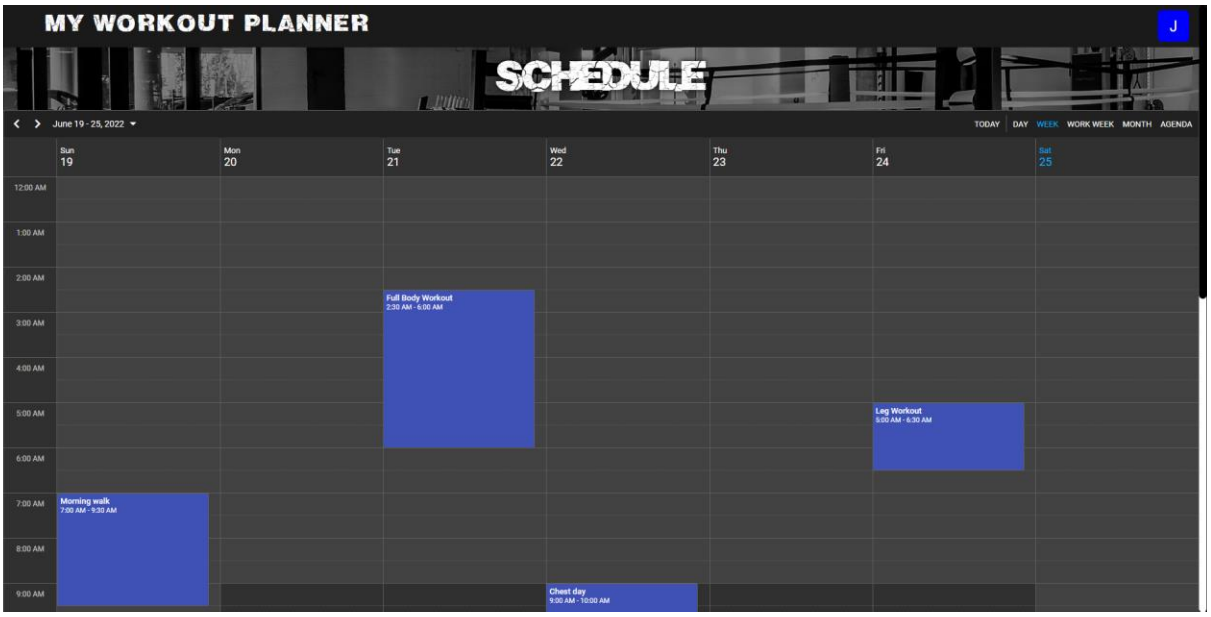


Figure 5.28 Schedule page

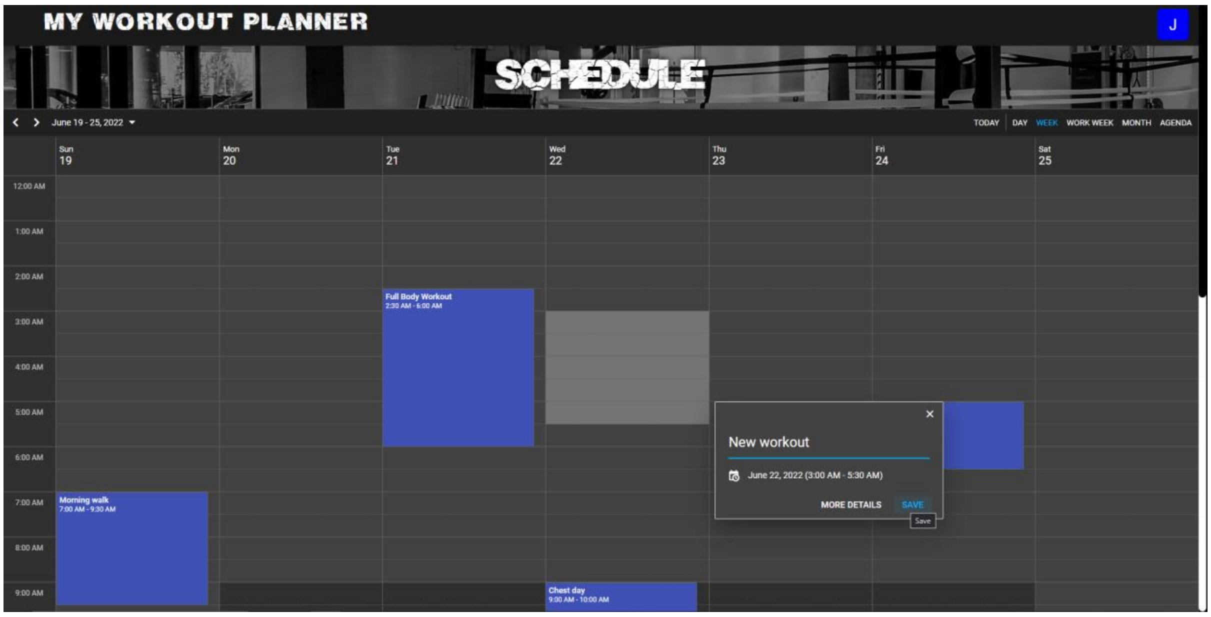
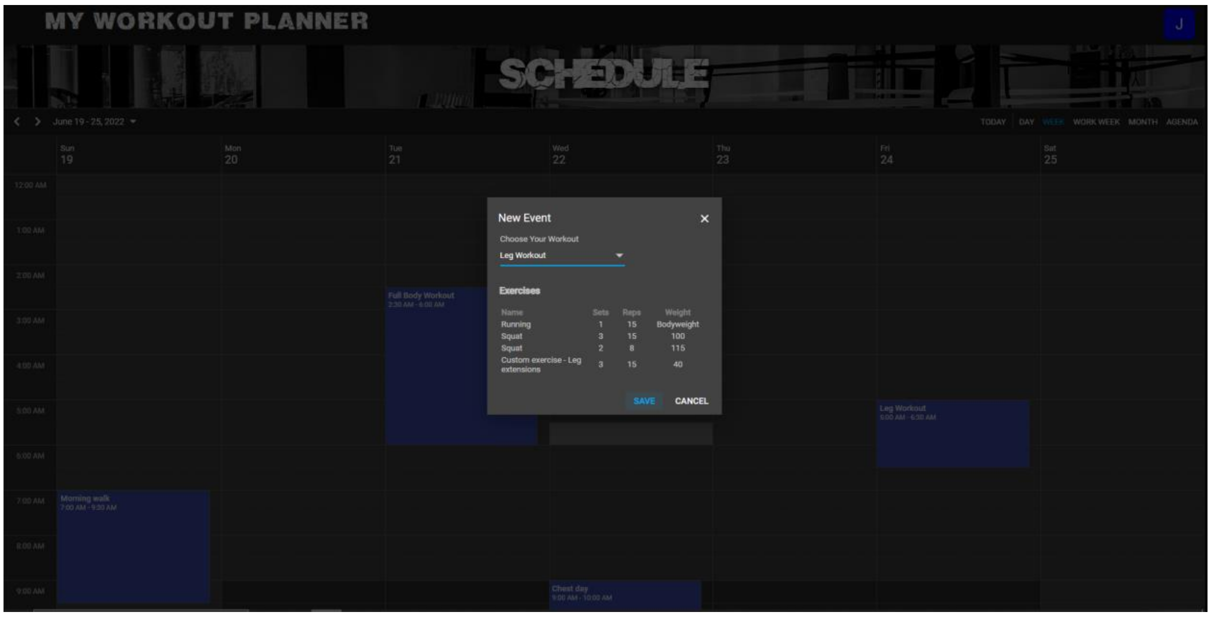


Figure 5.29: Adding scheduled workout


MY WORKOUT PLANNER
J
SCHEDULE
June 19 - 25, 2022
TODAY
DAY
WEEK
WORK WEEK
MONTH
AGENDA
Sun 19
Mon 20
Tue 21
Wed 22
Thu 23
Fri 24
Sat 25
Full Body Workout 2:30 AM - 6:00 AM
Leg Workout 5:00 AM - 6:30 AM
Morning walk 7:00 AM - 9:30 AM
Chest day 9:00 AM - 10:00 AM

New Event

Choose Your Workout

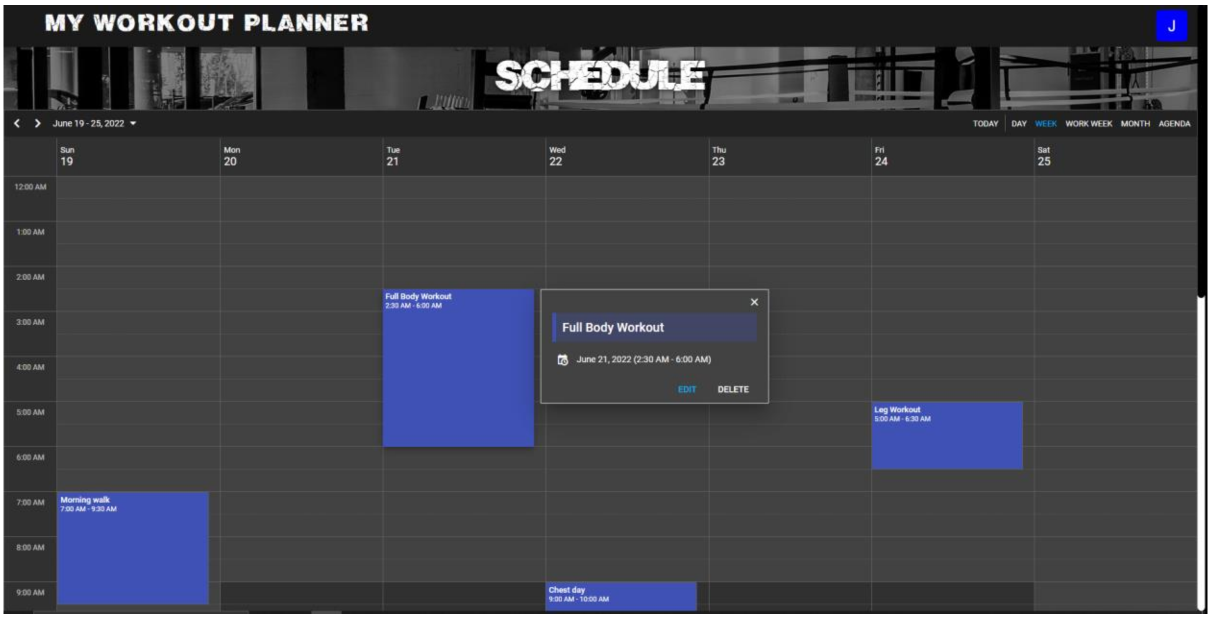
Leg Workout

Exercises

Name	Sets	Reps	Weight
Running	1	15	Bodyweight
Squat	3	15	100
Squat	2	8	115
Custom exercise - Leg extensions	3	15	40

SAVE CANCEL

Figure 5.30: Custom schedule add window


MY WORKOUT PLANNER
J
SCHEDULE
June 19 - 25, 2022
TODAY
DAY
WEEK
WORK WEEK
MONTH
AGENDA
Sun 19
Mon 20
Tue 21
Wed 22
Thu 23
Fri 24
Sat 25
Full Body Workout 2:30 AM - 6:00 AM
Leg Workout 5:00 AM - 6:30 AM
Morning walk 7:00 AM - 9:30 AM
Chest day 9:00 AM - 10:00 AM

Full Body Workout

June 21, 2022 (2:30 AM - 6:00 AM)

EDIT DELETE

Figure 5.31: Schedule edit and delete options

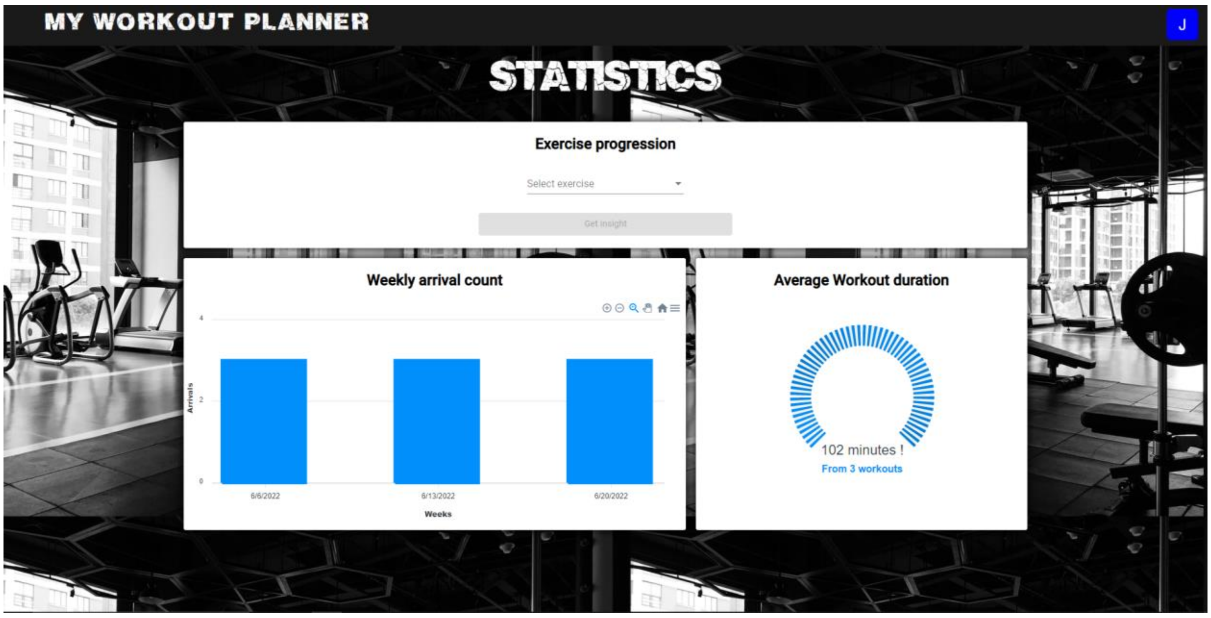


Figure 5.32: Statistics page

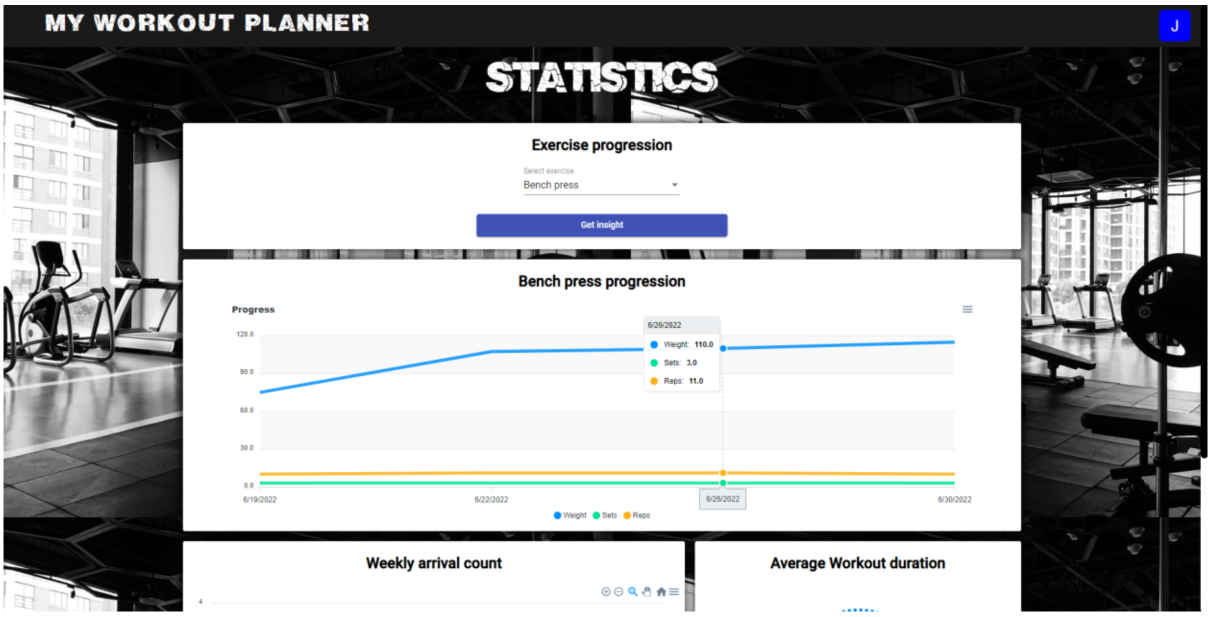


Figure 5.33: Statistics page with exercise insight

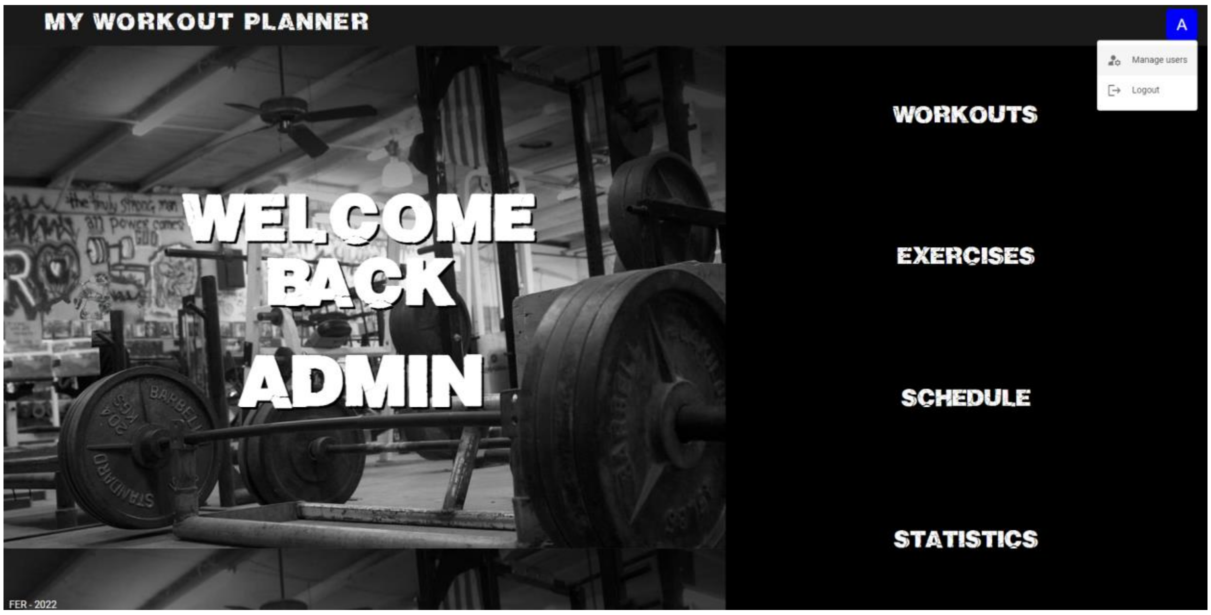
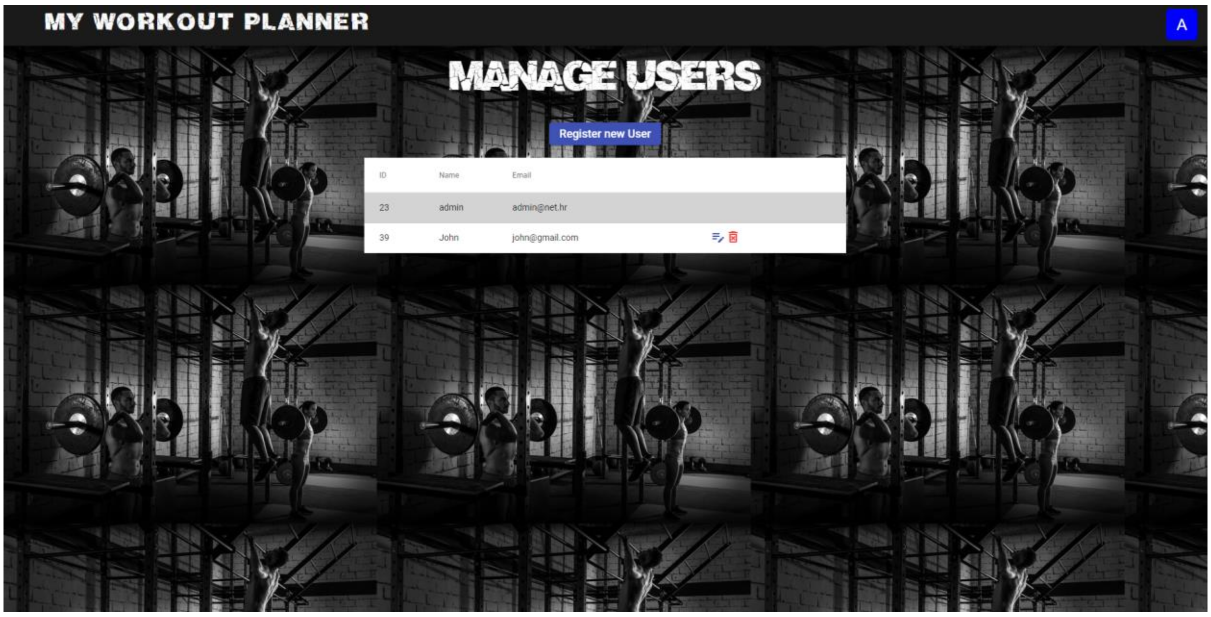


Figure 5.34: Admin manage user option



ID	Name	Email	
23	admin	admin@net.hr	
39	John	john@gmail.com	✎ 🗑

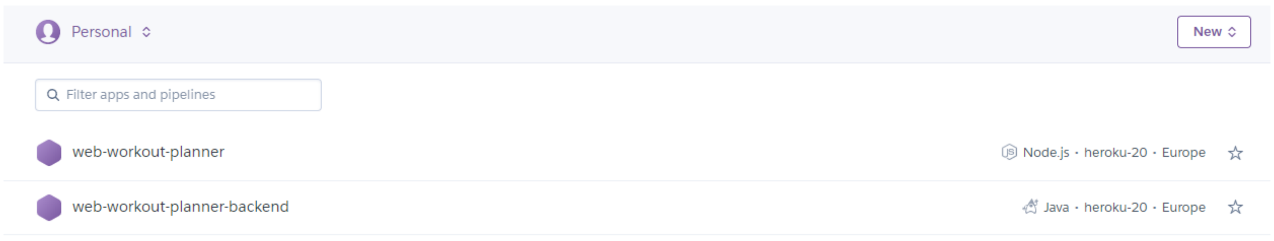
Figure 5.35: Manage user page

5.4 Application deployment

The final step of web application development is application deployment. It is a process of installing, configuring, updating, and enabling a specific application or set of applications to a specific URL on a server. After the deployment process, the application becomes publicly accessible through the URL [13].

Regarding this application, the Heroku platform is chosen as a web server for the deployment. It is a container-based cloud platform service that can run customer apps in virtual containers (Dynos), which execute on a reliable runtime environment. Dynos offer support for multiple codes and programming languages, like Node, Java, Python, PHP, Ruby, and more. The application's database is also stored on Heroku through its managed cloud database service paired with the PostgreSQL driver.

The application is stored as two separate applications on Heroku, the server-side application with the connected database and the client-side application, as shown in Figure *5.36*. The web-workout-planner application contains the final page that the users will see, and it is accessible through the following link: https://web-workout-planner.herokuapp.com/.



web-workout-planner — Node.js · heroku-20 · Europe

web-workout-planner-backend — Java · heroku-20 · Europe

Figure 5.36: Heroku applications

6. Conclusion

Through this paper, the whole workout planner web application development process was shown. In addition to learning the developing process of web applications, other goal of this paper was exploring and learning new technologies from the world of web application developing. Since the architectural goal was to use the client-server architecture, the following technologies were chosen. Spring Boot framework for server-side development, and Angular framework for client-side development. With those two powerful tools, a complete, functional, and aesthetic application was created.

The overall application functionality goals are achieved. All the requirements are fulfilled through the implementation and design segments. The application is fast, lightweight, multiuser, and responsive, with simple user interface that provides easy navigation and insight. Application can be used by individuals for their workout managing, or by a trainer, thanks to the administrator interface for managing and registering users.

In terms of possible application improvements, the application has a lot of room to grow and improve. The first and most important one would be the complete scalability for mobile devices since the application could be accessed immediately during the workout or any type of recreation. The other improvements would be adding more functionalities, like workout timer, sending scheduled workout notification to the user, connection with social media platforms, more detailed exercise guides and videos, a point-based rewarding system for users, more detailed statistics, and more.

Bibliography

[1] *What is full-stack development,* https://www.geeksforgeeks.org/what-is-full-stack-development/, *18.01.2022.*

[2] *Development environment explanation,* https://www.techopedia.com/definition/16376/development_environment, *11.11.2016*

[3] *Developer Survey*, https://insights.stackoverflow.com/survey/2021#overview, *2021.*

[4] *About IntelliJ,* https://www.tutorialspoint.com/intellij_idea/intellij_idea_introduction.htm, *2022.*

[5] *What is Visual Studio Code,* https://www.educba.com/what-is-visual-studio-code/, *2022.*

[6] *Spring Boot explained,* https://www.ibm.com/cloud/learn/java-spring-boot, *25.03. 2020.*

[7] *Spring Boot architecture,* https://www.tutorialandexample.com/spring-boot-architecture, *12.02.2020.*

[8] *Angular architecture,* https://medium.com/@bhavikagarg8/angular-architecture-overview-1e7cc7483a0, *26.01.2016.*

[9] *Spring Boot application example,* https://www.baeldung.com/jsf-spring-boot-controller-service-dao, *07.06.2022.*

[10] *Client/Server architecture,* https://www.techopedia.com/definition/438/clientserver-architecture, *25.08.2020.*

[11] *What is REST*, https://restfulapi.net/, *07.04.2022.*

[12] *What is a Database,* https://www.oracle.com/database/what-is-database/, *2022.*

[13] *What is Application Deployment,* https://www.vmware.com/topics/glossary/content/ap plication-deployment.html, *2022.*

[14] *Heroku,* https://www.heroku.com/what, *2022.*

[15] *Angular Calendar component,* https://ej2.syncfusion.com/angular/documentation/calendar/getting-started/, *2022.*

[16] *Angular Material components,* https://material.angular.io/, *2022.*

[17] *My Workout Plan appliciation,* https://play.google.com/store/apps/details?id=com.myworkoutplan.myworkoutplan&hl=en_US&gl=US, *2022.*

[18] *Google Calendar,* https://calendar.google.com/, *2022.*

Summary

Title: Developing a workout planner web application

The goal of this paper is to show the process of creating a complete workout planner web application. The process consists of requirements specification, system architecture, and implementation. The requirements specification defines all the functionalities that the application must provide to the user. The architecture defines the structure of the entire application. By dividing the application architecture into server and client side, a better division of labor and modularity of the application architecture is ensured. Java Spring Boot framework is used to create the server side, which with its simplicity, auto-configuration and better code structuring complements the Spring framework, which is the basis for creating the server-side of the application. Spring exchanges data with the client side. Angular framework is used to create the client-side part. It provides a structural and modular approach to the development of the user interface, dynamic aspects and data transmission and exchange. It achieves it by combining HTML, CSS, and TypeScript languages.

Keywords: Requirements, architecture, implementation, server, client, Spring, Angular, web application

Sažetak

Naslov: Izrada web aplikacije za planiranje treninga

Zadatak završnog rada je pokazati proces izrade cjelovite web aplikacije za planiranje treninga. Proces se sastoji od specifikacije zahtjeva, arhitekture sustava i implementacije. Kroz specifikaciju zahtjeva se jasno definiraju sve funkcionalnosti koje aplikacija mora pružati korisniku. Arhitektura definira strukturu cijele aplikacije. Podjelom arhitekture aplikacije na poslužiteljsku i klijentsku stranu, osigurava se bolja podjela rada i modularnost same arhitekture aplikacije. Pri izradi poslužiteljske strane korišten je razvojni okvir Java Spring Boot, koji sa svojom jednostavnošću, auto-konfiguracijom i boljim strukturiranjem koda, nadopunjuje Spring razvojni okvir, koji je temelj izrade tog dijela web aplikacije. Spring razmjenjuje podatke sa klijentskom stranom. Za izradu klijentske strane korišten je Angular razvojni okvir, koji omogućava strukturalan i modularan pristup izradi korisničkog sučelja, dinamičkih aspekata i prijenosa i razmjene podataka. To postiže kombinacijom HTML, CSS i TypeScript jezika.

Ključne riječi: zahtjevi, arhitektura, implementacija, poslužitelj, klijent, Spring, Angular, web aplikacija