SVEUČILIŠTE U ZAGREBU

**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

FINAL WORK br. xxxx

**DEVELOPING A WORKOUT PLANNER WEB APPLICATION**

Dalijo Vorkapić

Zagreb, mjesec 2022.

[Introduction 1](#_Toc103114473)

[1. Requirement specification 3](#_Toc103114474)

[1.1 Functional requirements 3](#_Toc103114475)

[1.1.1 Actors and their functional requirements 4](#_Toc103114476)

[1.1.2 Use cases 5](#_Toc103114477)

[1.1.3 Use case diagram 16](#_Toc103114478)

[1.2 Non-functional requirements 17](#_Toc103114479)

[2. System architecture and design 18](#_Toc103114480)

[2.1 Server side 19](#_Toc103114481)

[2.1.1 Server architecture 21](#_Toc103114482)

[2.2 Client-side 21](#_Toc103114483)

[2.2.1 Client architecture 22](#_Toc103114484)

[2.3 Database 22](#_Toc103114485)

[2.3.1 Entity tables description 23](#_Toc103114486)

[2.4 Class diagram 27](#_Toc103114487)

[3. Implementation 27](#_Toc103114488)

[4. Conclusion 27](#_Toc103114489)

# Introduction

In the last few years, there has been a great increasing interest of people in physical activities, such as sports, exercising and training, and they have all become an important part of their lives. With the initial appearance of the coronavirus, people suddenly had much more time and comfort for themselves, and many of them started with the so-called self-care routines. The main part of most of these routines, the one that has been preserved and grown to this day, is individual training. It is popular because first of all, it is very healthy and beneficial for human body and mind, 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, and the person can come up with its own plan. By that approach, people have to put together their own exercise and workouts details, plans and schedule, as well as monitor their progress, usually through tracking the number of repetitions and sets of specific exercise, or by tracking the required training execution time. It takes a lot of time, thought and energy, and the final product, usually turns out to be disorganized, incomplete and difficult to update and keep the track of, taking into account that planning, writing and tracking is 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 progress and organization of the workouts because it reduces the complexity of performing the mentioned actions and 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 can be applied to any type of personal planning and progress tracking application.

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

Requirement specification will be carried out using the defined functional requirements of the application and by combining them with the use cases. Architecture of the application will be divided into the 3 groups: database, backend (“server-side”) development and frontend (“client-side”) development.

As for the necessary tools for application development and documentation, the Astah UML tool will be used to create UML diagrams for requirements specification and application architecture. Regarding the architecture and implementation, Java Spring Boot framework will be used for backend programming, Angular TypeScript framework for frontend programming and PostgreSQL for database. For application deployment, Heroku Server, Git for the implementation versioning and Github for project repository.

The aim of this paper is to present the full process of software development and to deliver a complete and stable user web application.

# Topic evaluation

….

# Technologies

With the aim of developing a full-stack web application, a wider set of technologies to use are needed. Firstly, Java Spring Boot framework is used for application’s server-side programming. Developing, version controlling, testing, and executing Java server-side source code is done through the IntelliJ IDEA community edition development environment. Secondly, Angular TypeScript framework is used for developing application’s client-side. Its source code is developed, version controlled and executed through the Visual Studio Code source code editor. Thirdly, the database of the application is created through the PostgreSQL relational database management system. And, last but not least, Heroku cloud platform is used to deploy the application on the real-time server.

## 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 comprehensive facilities to developers for software developing. An IDE normally consists of a source code editor, build automation tools, debuggers, compilers, interpreters, etc. IntelliJ IDEA is designed by JetBrains, a Czech software development company which makes tools and development environments for software developing and project managing. IntelliJ IDEA is available in two editions, as an Apache licensed community edition, and in 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”, and one of the commonly most used IDEs for server-side and Java programming.

IntelliJ IDEA offers wide area of features, such as good developing ergonomics, easy and simple 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 “Example of IntelliJ IDEA user interface (*Figure 2.1*)”. One of the key features is the deep code insight. IntelliJ IDEA creates a virtual map of the project 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.

Text

Description automatically generated

Figure 2.1:IntelliJ IDEA user interface

## 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, and it is one of the best choices in terms of web application client-side development, for which it is used in this project. Visual Studio Code helps the programmer in code writing, debugging, and it corrects the code using the intelli-sense feature, which can detect if any snippet of code is left incomplete or undeclared. It has built-in support for multiple programming languages and has a rich ecosystem of extensions for other languages and runtimes. It provides interactable and simple user interface, repository support, web-support, 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 his robust architecture, it is light-weight, fast, responsive, and interactive, and most importantly, completely free of cost.

“Example of Visual Studio Code user interface (*Figure 2.2*)”

Text

Description automatically generated

Figure 2.2: Visual Studio Code user interface

## Java Spring Boot framework

Java Spring Boot is a tool that makes the developing of the web application and microservices faster and easier, by extending the capabilities of the original Spring Framework, on which it is based. Spring Framework is an open-source Java-based framework used to create a micro service, which is an architecture that allows the developers to develop and deploy services independently, meaning each service running has its own 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 application, Spring Boot Framework mitigates this effort, by making the developing of web application and microservices with Spring Framework faster and easier. It achieves it with: autoconfiguration, meaning that 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.

Spring Boot follows a layered hierarchical architecture in which each layer communicates with the layer directly below or above. “Diagram of layered architecture (*Figure 2.3*)”. There are four abstract layers in Spring Boot, presentation layer, business layer, persistence layer, and database layer.

Graphical user interface, application

Description automatically generated

Figure 2.3: Spring Boot layers hierarchy

The presentation layer consists of controllers. Each controller handles its own HTTP requests, such as POST, GET, UPDATE, DELETE, through the specified paths. “Example of controller (*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 (*Code 2.1*)”.

‘{“name”:”John”, “age”:30, “interests”:[“hiking”, “sports”, “cars”]}’

Code 2.1: JSON example

The presentation layer, with the intention of processing the data, 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 as well as authorization and validation. In terms of application logic, it consists of service classes that handle the functions defined through the persistence layer, that are linked with the database communication. “Example of the service class (*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 it translates the objects received from the business layer or from the database layer into the models which are then sent further deeper or higher into the hierarchy. “Example of the data access service (*Code 2.4*)”. The database layer is database itself, in which 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

## 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, client-side communication, as well as a suite of developer tools to help in developing, building, testing, and updating the source code.

Angular apps are modular, which means each Module defines a set of associated Components. A Component controls a patch of the screen called a View, defined through the HTML template. Angular creates, updates, and destroys Components as the use moves through the application. “Example of Component class declaration (*Code 2.1*)”.

@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 to exist in the separation of concerns. Their purpose is to help in dividing the application into small, different logical reusable units. “Service declaration example (*Code 2.1*)”.

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 larger subjects of Angular is Dependency Injection, a design pattern in which one or more dependencies are injected into a dependent object. Such pattern is applied when it is wanted to separate actions of what-to-do part of the when-to-do part. Angular uses his its own Dependency Injection framework which is 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 (*Code 2.3*)”.

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 key 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 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 (*Code 2.4*)”.

<h1>{{source\_object\_string}}</h1>

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

Code 2.8: Data binding example

In order to perform actions when different moments occur, Angular provides lifecycle hooks as interfaces that can be implemented by any component. For example, moments like component initialization (OnInit), exit (OnDestroy), update (OnChanges), and more. “Example of declaring initialization lifecycle hook (*Code 2.5*)”.

export class MyComponent implements OnInit {

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

Code 2.9: Initialization lifecycle hook example

Alongside with components, Angular contains two more kinds of directives. Structural directives that change the structure of the view (NgFor, NgIf). “Example of structural directives (*Code 2.6*)”. NgIf presents / hides the html component, based on the outcome of the attached expression (true or false). NgFor iterates over array of items, and for each item it creates the additional HTML element to which it is attached. The other type are the attribute directives that are 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 usable 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, usually as a subject or the observable. Through that stream, 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. “Example of a subscription action (*Code 2.7*)”. Received data through the subscription can come in three variants, as a passed value, an error or as 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 forms, and template driven forms, alongside with their validation options. And last but not least, without mentioning many other possibilities and features Angular contains, for navigating between the component views, Angular uses the routing module. “Example of component routes (*Code 2.8*)”.

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

# Requirement specification

Requirement specification is the first phase of software development. It is a collection of all requirements that are to be imposed on the design and verification of the application. A requirement is a thing that product must do (“system shall do”) or a 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 requirement. “Differences between functional and non-functional requirements (*Table 1.1*);

Table 1.1: 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. |

## 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. Initiator is the type of actor that directly interacts with the system, while participant does not interact, and it only gets passively affected by system.

### Actors and their functional requirements

**Actors:**

* Initiators
  + Administrator
  + User
* Participants
  + Database

**User can:**

* Login to the application
* View the data of:

1. Workouts
2. Exercises
3. Schedule
4. Statistics

* Create, update, and delete data of:

1. Workouts
2. Exercises
3. Scheduled workouts

**Administrator can:**

* Everything that user can
* Manage users
  1. Register new users
  2. Remove users

**Database:**

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

### Use cases

A use case is a written description of how users will perform tasks on the website. It outlines, from a user's point of view, a system's behavior as it responds to a request. Each use case is represented as a sequence of simple steps, beginning with the user's goal, and ending when the goal is fulfilled. Use cases add value because they help explain how the system should behave, and in the process, they also help brainstorm what could go wrong.  They provide a list of goals, and this list can be used to establish the cost and complexity of the system.

In this work, use cases describe a combination of the following elements: primary actor, goal, precondition, basic flow (main success scenario) and alternative paths (variations to the main success scenario).

Description of application use cases:

**UC1 – Registration**

* Primary actor: Administrator
* Goal: Registration of the new user
* Participants: Database
* Precondition: Administrator logged in
* Basic flow:
  1. Administrator clicks on the account icon
  2. Administrator selects “Register new user” from the menu
  3. Registration form pop-up window opens
  4. Administrator inserts the new user data
  5. Administrator submits the input
  6. New user added to the database
  7. Pop-up window closes
  8. Account menu closes
* Alternative paths:

3. Administrator clicks outside the window

1. Registration pop-up window closes

4. Administrator cancels the registration

1. Registration pop-up window closes

**UC2 – Login**

* Primary actor: User
* Goal: User login
* Participants: Database
* Precondition: User registered by administrator
* Basic flow:
  1. User clicks selects the Login option
  2. User is being redirected to the Login page
  3. Login form is displayed
  4. User inserts email and password
  5. User submits the input
  6. Database approves the credentials
  7. User is being redirected to the home page
* Alternative paths:

4. Improper input

1. User is not able to submit the input

5. Given credentials are wrong

1. Login page reopens

**UC3 – Logout**

* Primary actor
  1. User
  2. Administrator
* Goal: User login
* Participants: Database
* Precondition: User / Administrator logged in
* Basic flow:
  1. User clicks on the account icon
  2. User selects “Logout” from the menu
  3. User is redirected to the Login page

**UC4 – Exercises overview**

* Main Actor: User
* Goal: Displaying a list of currently stored exercises
* Participants: Database
* Precondition: Login
* Basic Flow:
  1. On the home screen, User selects the Exercises option
  2. User is being redirected to the Exercise page
  3. List of Exercises is displayed

**UC5 – Filtering exercises**

* Main Actor: User
* Goal: Displaying filtered exercises
* Participants: Database
* Precondition: Login
* Basic Flow:
  1. On the home screen, user selects the “Exercises” option
  2. User is being redirected to the Exercises page
  3. List of exercises is displayed on the page
  4. User inserts filter in the filter input
  5. Filtered exercises are displayed on the page
* Alternative path:

5. No exercise shown for given filter

**UC6 – Creating new exercise**

* Main Actor: User
* Goal: Adding new exercise
* Participants: Database
* Precondition: Login
* Basic Flow:
  1. On the home screen, User selects the Exercises option
  2. User is being redirected to the Exercises page
  3. User selects “Create new exercise” option
  4. A new window with the create form pops up
  5. User inserts the data for the new exercise
  6. User submits the input
  7. New exercise created in the database
  8. Pop-up window closes
  9. Exercises page refreshes with the changes
* Alternative path

4. User clicks outside the window

1. Pop-up window closes

6. User cancels the input

1. Pop-up window closes

6. Exercise “Name” input is left empty

1. User is not able to submit the input

7. Exercise name already exists

1. Submit request is rejected

**UC7 – Editing exercise**

* Main Actor: User
* Goal: Editing chosen exercise
* Participants: Database
* Precondition:
  1. User logged in
  2. Exercise existing
* Basic Flow:
  1. On the home screen, User selects the Exercises option
  2. User is being redirected to the Exercises page
  3. User selects the “Edit” icon of the chosen exercise
  4. The change form window pops up
  5. User changes the desired exercise data
  6. User submits the changes
  7. Exercise is updated in the database
  8. Pop-up window closes
  9. Exercises page refreshes with the changes
* Alternative path

4. Exercise “Name” input is changed to empty

1. User is not able to submit the input

6. Exercise name already exists

1. Update request is rejected

6. User cancels the changes

1. Pop-up window closes

**UC8 – Deleting exercise**

* Main Actor: User
* Goal: Deleting chosen exercise
* Participants: Database
* Precondition:
  1. User logged in
  2. Exercise existing
* Basic Flow:
  1. On the home screen, User selects the Exercises option
  2. User is being redirected to the Exercise page
  3. User selects the “Delete” icon of the chosen exercise
  4. An alert window pops up
  5. User confirms the deletion
  6. Exercise is deleted from the database
  7. Pop-up window closes
  8. Exercises page refreshes with the changes
* Alternative path

4. User clicks outside of the pop-up window

1. Pop-up window is closed

5. User cancels the deletion

1. Pop-up window is closed

**UC9 – Workout overview**

* Main Actor: User
* Goal: Displaying a list of currently stored workouts
* Participants: Database
* Precondition: User logged in
* Basic Flow:
  1. On the home screen, User selects the Workouts option
  2. User is being redirected to the Workouts page
  3. Workout list is displayed

**UC10 – Filtering workouts**

* Main Actor: User
* Goal: Displaying filtered exercises
* Participants: Database
* Precondition: Login
* Basic Flow:
  1. On the home screen, user selects the “Exercises” option
  2. User is being redirected to the Exercises page
  3. List of exercises is displayed on the page
  4. User inserts filter in the filter input
  5. Filtered exercises are displayed on the page
* Alternative path:

5. No exercise shown for given filter

**UC11 – Workouts details overview**

* Main Actor: User
* Goal: Displaying details for selected workout
* Participants: Database
* Precondition:
  1. User logged in
  2. Workout existing
* Basic Flow:
  1. On the home screen, User selects the Workouts option
  2. User is being redirected to the Workouts page
  3. Workout list is displayed
  4. User clicks on the desired workout
  5. Workouts details are shown on the page

**UC12 – Creating new workout**

* Main Actor: User
* Goal: Adding new workout
* Participants: Database
* Precondition: User logged in
* Basic Flow:
  1. On the home screen, User selects the Workouts option
  2. User is being redirected to the Workouts page
  3. User selects “Create new workout” option
  4. A new window with the create form pops up
  5. User inserts the data for the new workout
  6. User submits the input
  7. New workout created in the database
  8. Pop-up window closes
  9. Workouts page refreshes with the changes
* Alternative path

4. User clicks outside the window

1. Pop-up window closes

6. Workout “Name” input is left empty

1. User is not able to submit the input

6. User cancels the insert

1. Pop-up window closes

7. Workout name already exists

1. Submit request is rejected

**UC13 – Editing workout**

* Main Actor: User
* Goal: Editing chosen workout
* Participants: Database
* Precondition:
  1. User logged in
  2. Workout exists
* Basic Flow:
  1. On the home screen, User selects the Workouts option
  2. User is being redirected to the Workouts page
  3. User selects the “Edit” icon of the chosen workout
  4. The change form window pops up
  5. User changes the desired workout data
  6. User submits the changes
  7. Workout is updated in the database
  8. Pop-up window closes
  9. Workouts page refreshes with the changes
* Alternative path

4. User clicks outside the window

1. Pop-up window closes

5. Exercise “Name” input is changed to empty

1. User is not able to submit the input

6. Exercise name already exists

1. Update request is rejected

6. User cancels the changes

1. Pop-up window closes

**UC14 – Deleting workout**

* Main Actor: User
* Goal: Deleting chosen workout
* Participants: Database
* Precondition:
  1. User logged in
  2. Workout exists
* Basic Flow:
  1. On the home screen, User selects the Workouts option
  2. User is being redirected to the Workouts page
  3. User selects the “Delete” icon of the chosen workout
  4. An alert window pops up
  5. User confirms the deletion
  6. Workout is deleted from the database
  7. Pop-up window closes
  8. Workouts page refreshes with the changes
* Alternative path

4. User clicks outside of the pop-up window

1. Pop-up window is removed

5. User cancels the deletion

1. Pop-up window is removed

**UC15 – Schedule overview**

* Main Actor: User
* Goal: Displaying the current schedule
* Participants: Database
* Precondition: User logged in
* Basic Flow:
  1. On the home screen, User selects the Schedule option
  2. User is being redirected to the Schedule page
  3. User schedule is displayed

**UC16 – Adding workout to schedule**

* Main Actor: User
* Goal: Adding workout to schedule
* Participants: Database
* Precondition:
  1. User logged in
  2. Workout exists
* Basic Flow:
  1. On the home screen, User selects the Schedule option
  2. User is being redirected to the Schedule page
  3. User marks the wanted period on the schedule
  4. Workout selection pop-up window opens
  5. User selects which workout to put in the schedule
  6. User submits the changes
  7. Pop-up window closes
  8. Scheduled workout is added to database
  9. Schedule display updates
* Alternative path:

6. User cancels the changes

1. Pop-up window closes

**UC17 – Workout period change**

* Main Actor: User
* Goal: Changing the period of workout
* Participants: Database
* Precondition:
  1. User logged in
  2. Scheduled workout exists
* Basic Flow:
  1. On the home screen, User selects the Schedule option
  2. User is being redirected to the Schedule page
  3. User resizes the workout period box he wants to change
  4. Scheduled workout is updated in the database
  5. Schedule display updates

**UC18 – Deleting scheduled workout**

* Main Actor: User
* Goal: Deleting workout from schedule
* Participants: Database
* Precondition:
  1. User logged in
  2. Scheduled workout exists
* Basic Flow:
  1. On the home screen, User selects the Schedule option
  2. User is being redirected to the Schedule page
  3. User selects the workout period box he wants to delete
  4. User selects the “Delete” icon
  5. Alert pop-up window opens
  6. User confirms deletion of the scheduled workout period
  7. Pop-up window closes
  8. Scheduled workout is removed from the database
  9. Schedule display updates
* Alternative path:

5. User clicks outside of the window

1. Pop-up window closes

6. User cancels the changes

1. Pop-up window closes

**UC19 – Statistics overview**

* Main Actor: User
* Goal: Displaying user workout statistics
* Participants: Database
* Precondition: User logged in
* Basic Flow:
  1. On the home screen, User selects the Statistics option
  2. User is being redirected to the Statistics page
  3. User statistics are displayed
* Alternative path:

3. No statistics available

### 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 (*Figure 1.1*).”

Diagram

Description automatically generated

Figure 3.1: Application's use case diagram

## 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 application
  + Fast communication with database
* Scalability
  + Application can stand high workload
* Availability
  + Application is always online
  + Multiple users possible in real time
* Reliability
  + No unexpected crashes
* Security
  + Authentication
  + Password hashing
  + Input validation
* Usability
  + Minimalistic and simple user interface

# System architecture and design

The architecture of this application can be divided into the following components “Architecture Layout (*Figure 2.1*).”:

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

Diagram

Description automatically generated

Figure 4.1: Architecture layout

## Server side

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

Server side communicates in two directions. In one direction, it communicates with the database, while in 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, etc. that 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:

1. Uniform interface – 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

Application’s server side is divided into four layers:

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

Client side communicates with server side, through the HTTP request made to the URL defined by the controllers. Controller than 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, in order 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 Data access object interface. In terms of communication with the database, it uses JDBC (Java Database Connectivity) API, which connects and executes queries with the database. “Server-side layers (*Figure 2.2*).”

Diagram

Description automatically generated

Figure 4.2: Server-side layers

### Server architecture

Application’s server-side structure is organized into packages. 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. 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 packages structure is shown in (*Figure 4.3*)”.

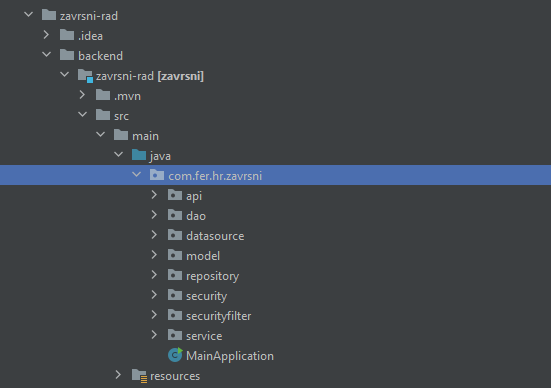


Figure 4.3: Server-side directory structure

Api package represents controllers, dao and repository contain the persistence layer handlers, content inside service handles the business layer of the application, models represent the overall data structures, while security and securityfilter handle authentication and authorization processes of application.

## Client-side

Client-side refers to everything in web application that is displayed or takes place on the client (end user device). This includes what the user sees (User Interface), along with any actions that 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 to dynamically add 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 of 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. It is a TypeScript-based free and open-source web application framework, used for building single-page client applications. It is a complete rewrite from the AngularJS, and got improved in segments like:

* It uses a hierarchy of components as its primary architectural characteristic
* Modularity
* TypeScript language
* Dynamic loading
* Iterative callbacks provided by RxJS

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, who 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. Components use services, which 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, etc.

### Client architecture

…

## Database

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 the database is a relational database. The data within it is typically modeled in rows and columns in a series of 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 access control of the 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.

Application’s database consists of the next entities:

* AppUser
* Workout
* Exercise
* ExerciseWorkout
* Schedule

Since entities need to connect with each other, it is necessary to determine the relations. “Entity Relationship Diagram (*Figure 2.3*).”

Diagram

Description automatically generated

Figure 4.4: Entity relationship diagram

### Entity tables description

**AppUser**

A table in which user access data and user roles in the system are stored. Every user has his uniquely generated ID, unique email address, name, hashed password, and his role. User role can be administrator or the regular user. (*Table 2.1*)

Table 2.1: appuser

|  |  |  |
| --- | --- | --- |
| appuser | | |
| Attribute | Type | Description |
| user\_id | INT | User id |
| email | VARCHAR (100) | User’s email address |
| name | VARCHAR (100) | User’ name |
| password | VARCHAR (200) | User’s hashed password |
| role | VARCHAR (50) | User role (administrator/user) |

**Workout**

A table in which are stored workouts and their information. It is defined by its own uniquely generated ID, unique workout name, duration, and complexity. Each workout (created by the user) has the constraint that it can be accessible by that user only. (*Table 2.2*)

Table 2.2: workout

|  |  |  |
| --- | --- | --- |
| workout | | |
| Attribute | Type | Description |
| workout\_id | INT | Workout id |
| name | VARCHAR (100) | Workout name |
| duration | INT | Workout duration |
| complexity | VARCHAR (100) | Workout complexity |
| user\_id | INT | User id that created this workout |

**Exercise**

A table that defines the exercises and their basic information. Every exercise has its own uniquely generated ID, unique exercise name, image URL (image name), and its description. (*Table 2.3*)

Table 2.3: exercise

|  |  |  |
| --- | --- | --- |
| exercise | | |
| Attribute | Type | Description |
| exercise\_id | INT | Exercise id |
| name | VARCHAR (100) | Exercise name |
| description | VARCHAR (400) | Exercise description |
| imageurl | VARCHAR (300) | URL to the exercise image |

**ExerciseWorkout**

This table is a connection between the Workout and Exercise entities. It is defining which exercise belongs to which workout, and in that workout, how much of repetitions and sets each exercise takes. It is generally an n:n connection between the workouts and the exercises. (*Table 2.4*)

Table 2.4: exerciseworkout

|  |  |  |
| --- | --- | --- |
| exerciseworkout | | |
| Attribute | Type | Description |
| workout\_id | INT | Workout id for which this is created |
| exercise\_id | INT | Exercise id for which this is created |
| sets | INT | Number of exercise sets |
| reps | INT | Number of exercise repetitions |

**Schedule**

A table that is used to display the scheduled workouts. It is storing the workout through its ID, and also contains the start time and end time attributes, that define in which period the workout schedule is saved. By the users ID attribute we define that every user can have up to 1 schedule. (*Table 2.5*)

Table 2.5: schedule

|  |  |  |
| --- | --- | --- |
| schedule | | |
| Attribute | Type | Description |
| schedule\_id | INT | Schedule id |
| starttime | TIMESTAMP | Timestamp when the workout is scheduled |
| endtime | TIMESTAMP | Timestamp till when the workout is scheduled |
| workout\_id | INT | Workout id which is added to the schedule |
| user\_id | INT | User id for which this schedule is created |

## Class diagram

…

# Implementation

Missing…Code, testing, user interface, etc.

# Conclusion