SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 7290

WEB APLIKACIJA ZA INDIVIDUALNO I TIMSKO DOGOVARANJE SPORTSKIH TRENINGA U DVORANAMA

Ivan Sesar

Zagreb, lipanj 2023.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 7290

WEB APLIKACIJA ZA INDIVIDUALNO I TIMSKO DOGOVARANJE SPORTSKIH TRENINGA U DVORANAMA

Ivan Sesar

Zagreb, lipanj 2023.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 10. ožujka 2023.

ZAVRŠNI ZADATAK br. 7290

Pristupnik:	**Ivan Sesar (0036512063)**
Studij:	Računarstvo
Modul:	Računarska znanost
Mentor:	izv. prof. dr. sc. Alan Jović

Zadatak: **Web aplikacija za individualno i timsko dogovaranje sportskih treninga u dvoranama**

Opis zadatka:

Cilj ovog završnog rada je izrada web aplikacije koja bi olakšala individualno planiranje i grupne dogovore ekipa (timova) za trening. Aplikacija omogućava korisnicima da kreiraju timove čime korisnik koji kreira tim automatski postaje administrator (admin). Ovlast admina je ta da može uređivati podatke o timu i dodavati/brisati članove tima. Ukoliko želi, admin tima može ovlasti admina predati i drugim članovima tima. Za dodavanje korisnika u tim, šalju im se pozivnice. Korisnik u kalendar može dodati individualni trening te može postaviti timski trening ukoliko je admin tima. Postavljanjem timskog treninga automatski se svim članovima u kalendar postavlja timski trening. Ukoliko član grupe ne može prihvatiti trening, ima mogućnost odbijanja treninga čime se briše njegova pristunost na treningu i uklanja iz kaledara. Za svaki individualni i timski trening je vezana dvorana u kojoj se održava tako da se prije postavljanja bilo koje vrste treninga provjerava dostupnost dvorane. Osim dvorane dodjeljuje se i vrsta sporta koji se trenira, što će korisniku dodatno biti naznačeno u kalendaru za svaki trening. Početno, aplikacija prikazuje prijavu/registraciju korisnika. Nakon uspješne ovjere, korisnika se preusmjerava na izbornik gdje je vidljiv korisnikov kalendar. Osim kalendara na izbornoj traci korisniku su dostupni sljedeći izbornici: Grupe, Pozivnice i Profil. Aplikaciju je potrebno ostvariti u programskom jeziku po vlastitom izboru.

Rok za predaju rada: 9. lipnja 2023.

Zahvaljujem mentoru izv.prof.dr.sc. Alanu Joviću na suradnji tijekom izrade završnog rada.

SADRŽAJ

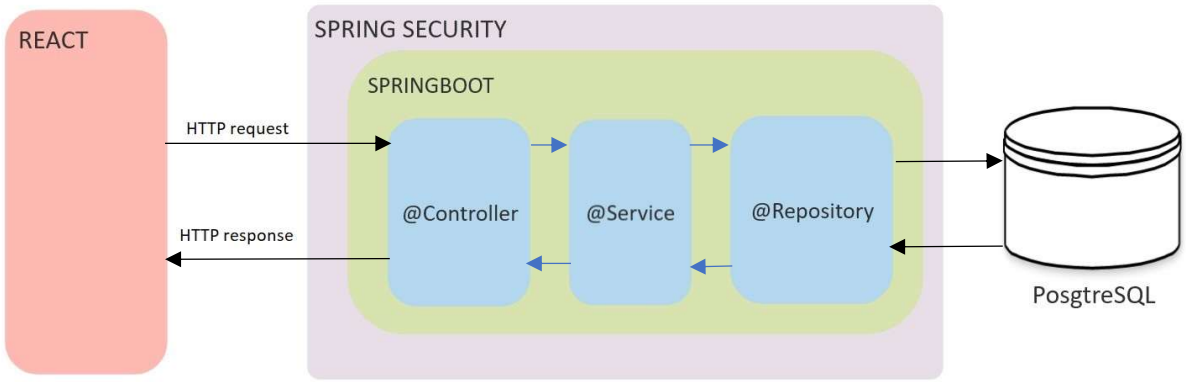
Uvod

Prilikom rekreativnog bavljenja sportom došlo je do ideje izgradnje web aplikacije za individualno i timsko dogovaranje treninga u dvoranama. Ideja je da se grupa sportaša formira u tim u kojemu će administratori tima upravljati samim timovima i timskim treninzima. Svaki timski trening obavlja se u nekoj od sportskih dvorana dok za individualni trening korisnik postavlja mjesto odvijanja. Pri postavljanju timskog treninga provjerava se dostupnost dvorane. Svaki korisnik ima mogućnost izrade tima čime automatski postaje član i administrator tog tima. Korisnici tima imaju ovlast dodavati i ukloniti korisnike iz tima, a administratori dodati ili ukloniti administratora unutar tima. Dodavanje korisnika u tim se odvija preko pozivnice kojom se poziva korisnika u tim. Svi treninzi su vidljivi korisnicima u kalendaru. Za timski trening vidljiv je i broj korisnika koji dolaze kako bi se unaprijed znalo hoće li biti dovoljno korisnika za održavanje treninga. Ukoliko administrator tima procijeni da je broj odazvanih premali, ima mogućnost uklanjanja treninga iz dvorane kako bi se oslobodio termin dvorane za druge timove. Korisnik u svakom trenutku može napustiti tim. Korisniku je vidljiv i njegov profil s osobnim podacima koje ima mogućnost mijenjati ili ukloniti svoj račun. Također uz ulogu korisnika u aplikaciji, postoji i administrator aplikacije koji ima posebne ovlasti: brisanje korisnika, brisanje tima, kreiranje i ažuriranje i brisanje sportske dvorane i dodavanje korisniku ovlast administratora aplikacije. Na aplikaciju je primjenjena sigurnost.

U drugom poglavlju ovog rada opisani su skica rada sustava tj. način rada aplikacije. U trećem poglavlju opisani su pregled funkcionalnih zahtjeva aplikacije i izrada relacijske baze podataka iz ER modela baze podataka. U četvrtom poglavlju opisane su tehnologije i način rada poslužiteljske strane (engl. *backend*) uz primjer prikazan kao kôd. U petom poglavlju opisane su tehnologije i klijentska strana aplikacije (engl. *frontend*) uz primjere koda.

2. Skica rada apikacije

U ovom poglavlju prikazan je način rada aplikacije kao komunikacija između klijentske i poslužiteljske strane. Skica rada aplikacije prikazana je prema slici 2.1. Klijentska strana je implementirana pomoću biblioteke React s dodanim pripadajućim bibliotekama, npr. axios za slanje zahtjeva prema poslužitelju. Poslužiteljska strana je implementirana u radnom okviru SpringBoot i osigurana je pomoću radnog okvira Spring Security. Iako SpringBoot nudi ugrađenu bazu podataka u ovom završnom radu koristi se baza podataka PostgreSQL. Komunikacija se između dvije strane odvija pomoću protokola HTTP.

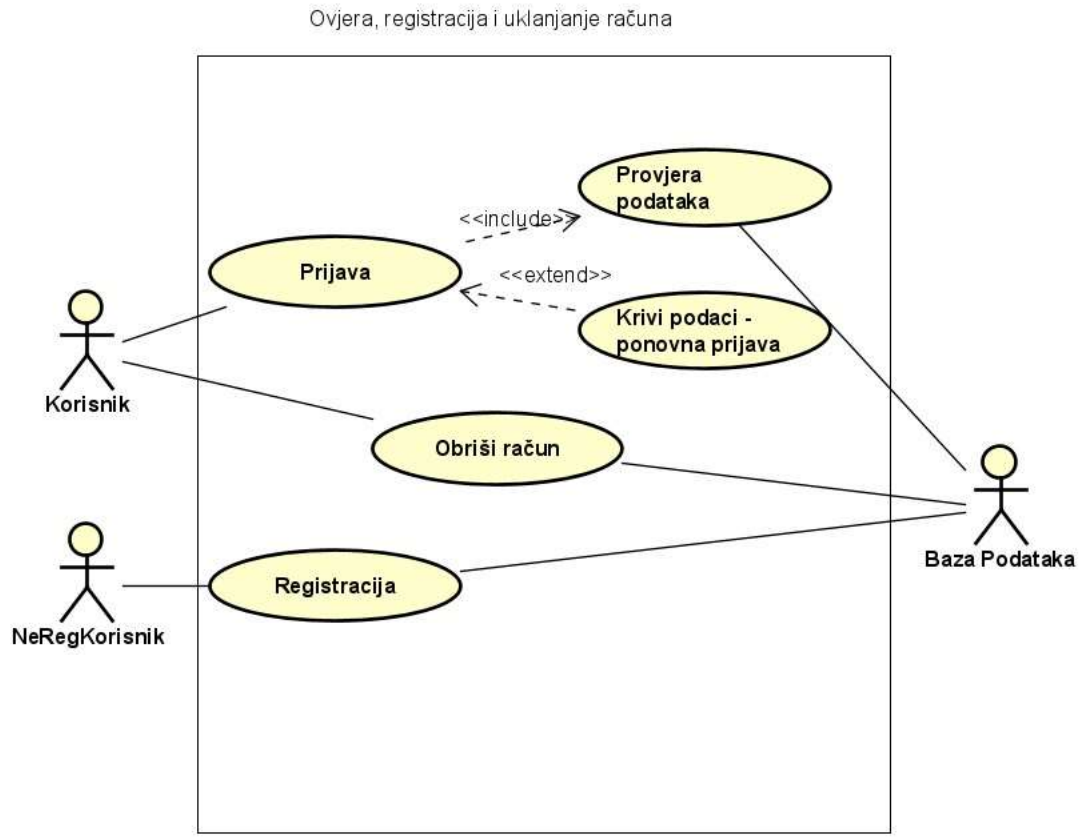


Slika 2.1 Komunikacija između klijentske i poslužiteljske strane

3. Funkcionalni zahtjevi aplikacije i baza podataka

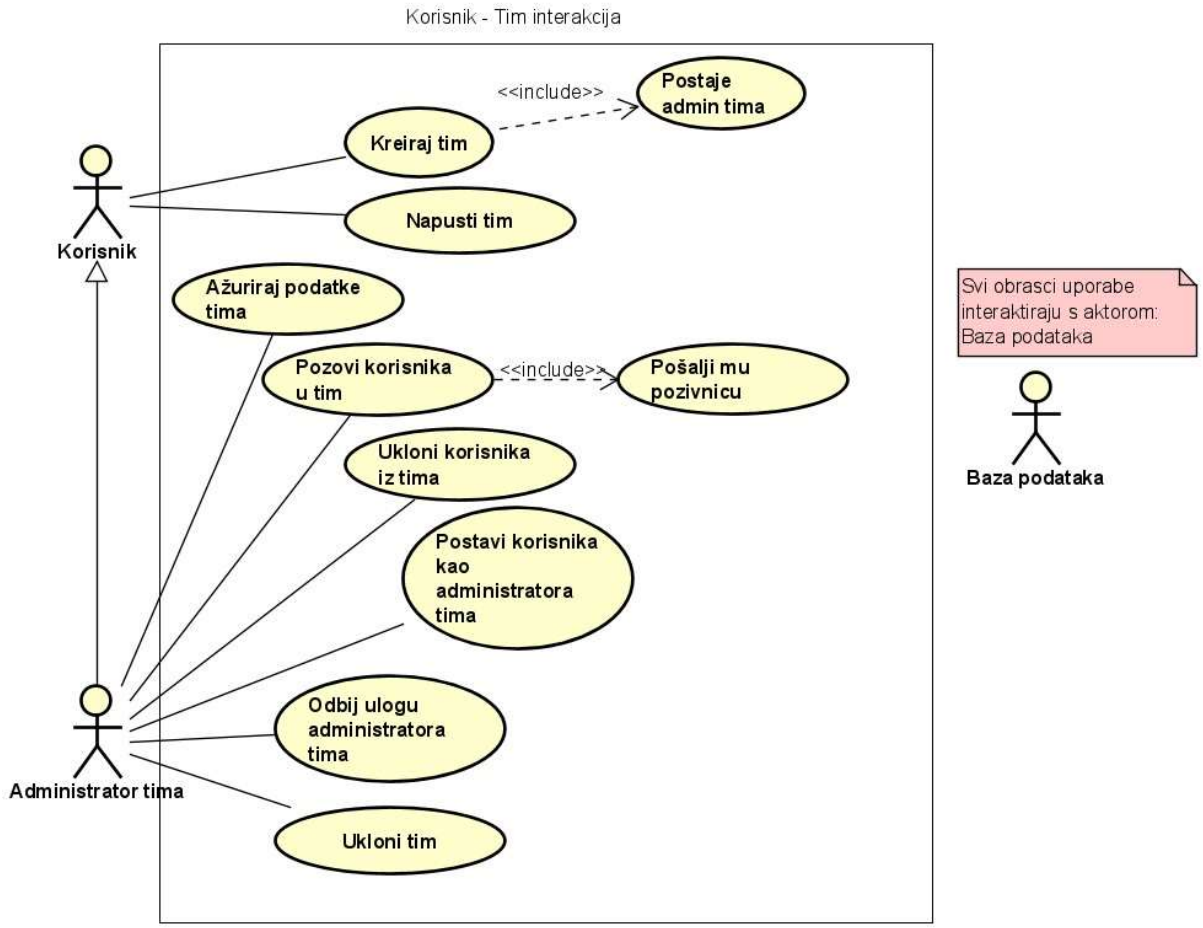
U ovom poglavlju su definirani svi funkcionalni zahtjevi aplikacije opisani pomoću UML dijagrama obrazaca uporabe te je iz teksta zadatka kreiran ER model baze podataka i po njemu izgrađen relacijski model baze podataka.

3.1 Dijagrami obrazaca uporabe



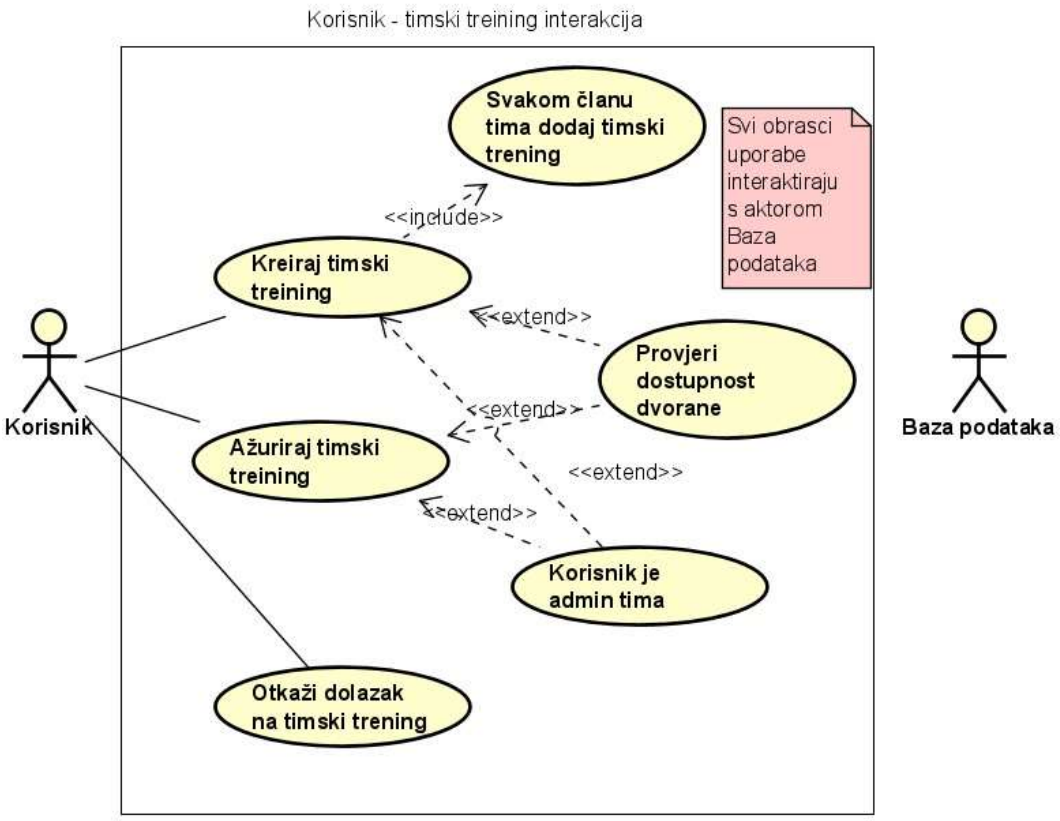
Slika 3.1 Ovjera, registracija i uklanjanje računa

Slika 3.1 opisuje interakciju između neregistriranog korisnika i baze podataka i korisnika koji već ima račun s bazom podataka. Ne registrirani korisnik obavlja jednostavnu registraciju čime postaje korisnik aplikacije. Korisnik, koji ima račun, se prijavljuje u aplikaciju tako da se provjerava postojanost korisnika i valjanost podataka preko baze podataka. Ako je korisnik unio krive podatke, primit će povratnu poruku kako su mu podaci nevaljani za prijavu, nakon čega se korisnik ponovno prijavljuje u aplikaciju. Svi obrasci uporabe interagiraju s aktorom Baza podataka.



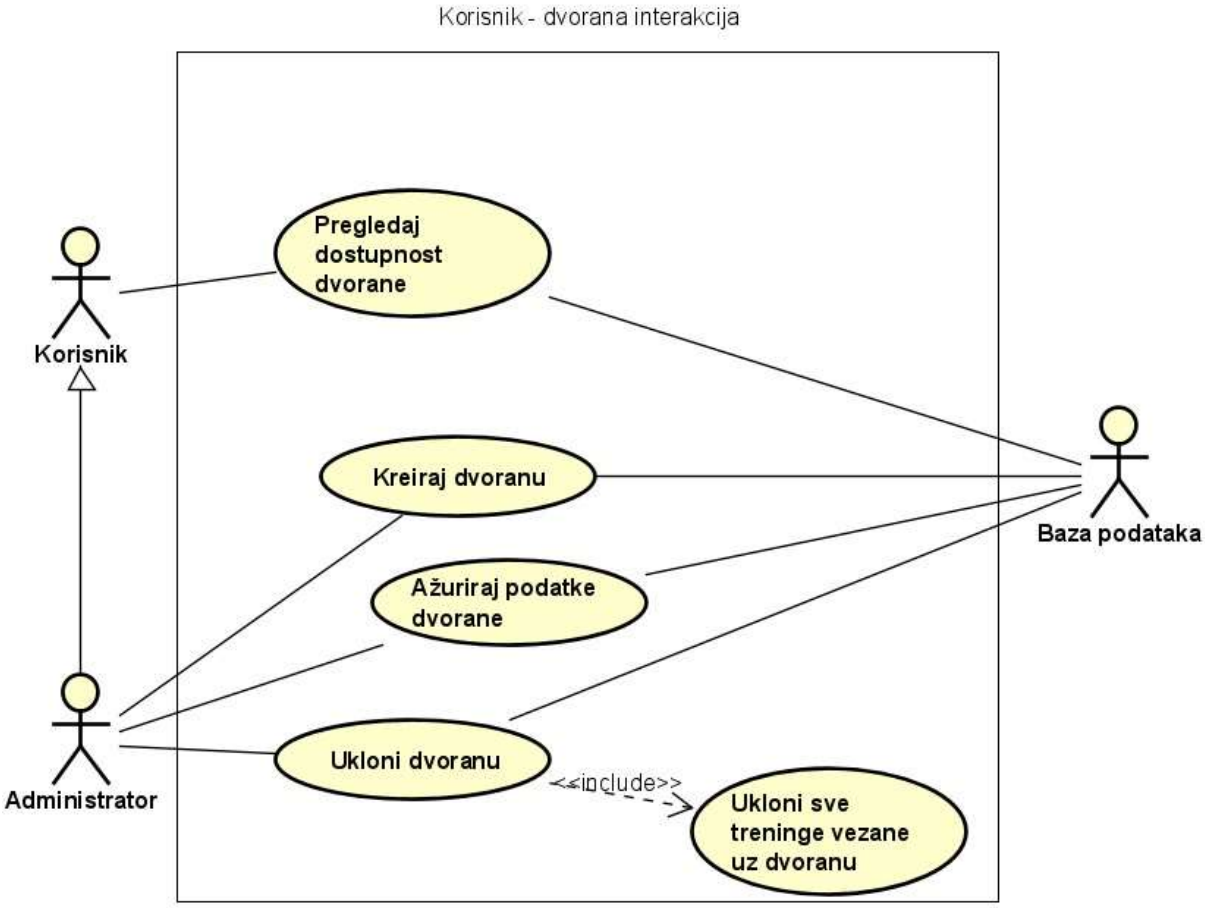
Slika 3.2 Interakcija Korisnik – Tim

Slika 3.2 prikazuje interakciju između korisnika i tima. Korisnik može kreirati i napustiti tim. Pri stvaranju tima, korisnik postaje automatski administrator tima. Aktor administrator tima ima mogućnost pozivanja korisnika u tim, što uključuje slanje pozivnice korisniku, ima mogućnost uklanjanja tima i korisnika iz tima, mogućnost ažuriranja tima, dodavanja korisniku ulogu administartora tima, te odbiti ulogu administratora tima. Svi obrasci uporabe interagiraju s aktorom Baza podataka.



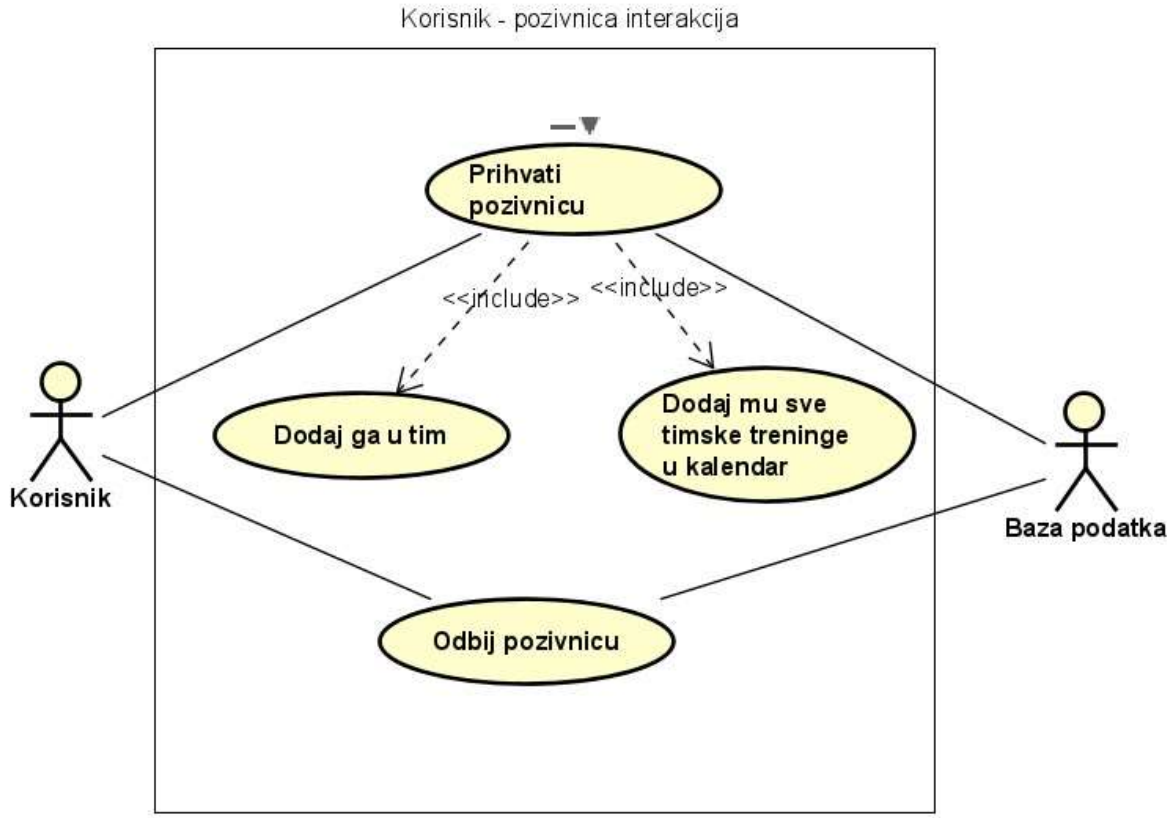
Slika 3.3 Interakcija Korisnik – Timski trening

Slika 3.3 opisuje interakciju između korisnika i timskog treninga. Aktor Korisnik ima mogućnost kreiranja timskog treninga čime se automatski dodaje svim korisnicima tima timski trening u kalendar. Ako postoje promjene vezane uz timski trening npr. promjena lokacije održavanja treninga, korisnik može ažurirati timski trening. Za obje navedene akcije provjerava se dostupnost dvorane. Ako korisnik ne može doći na trening može ga otkazati. Za svaki timski trening na klijentskoj strani vidljiv je broj odazvanih korisnika. Svi obrasci uporabe interagiraju s aktorom Baza podataka.



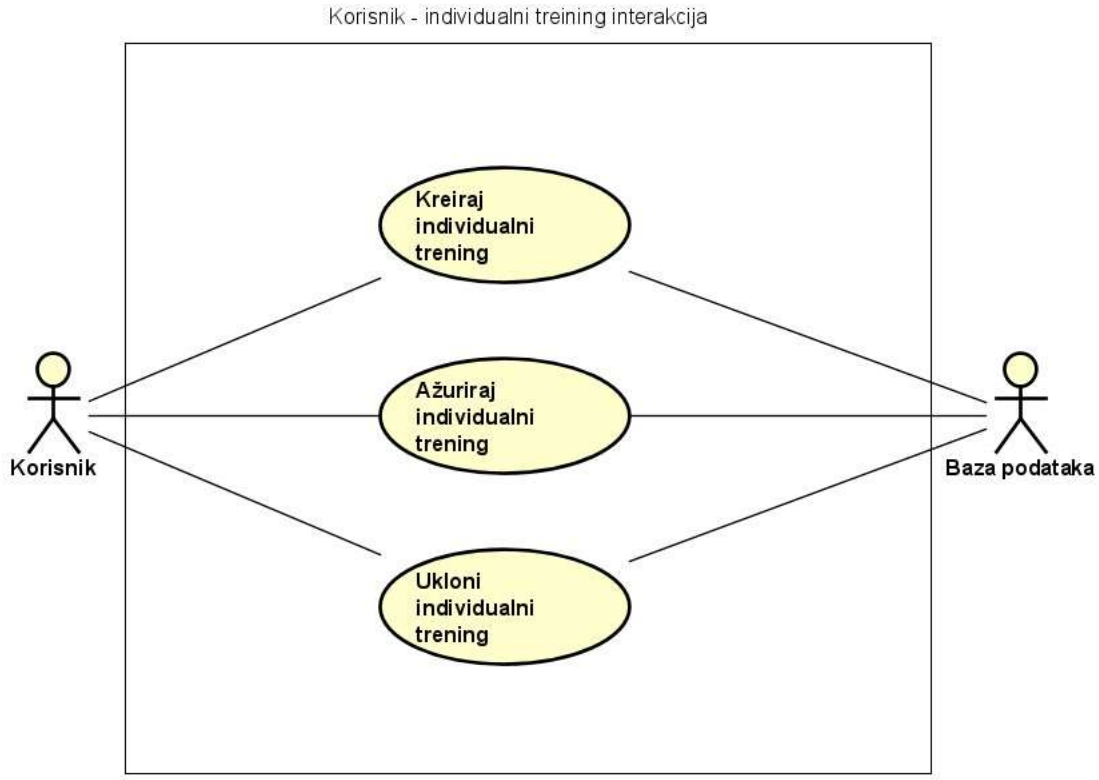
Slika 3.4. Interakcija Korisnik – Dvorana

Slika 3.4. opisuje interakciju između korisnika i dvorane. Aktor Korisnik može dodavajući timske ili individualne treninge pregledati dostupnost dvorane, dok aktor Administrator (odnosi se na administratora aplikacije) može kreirati dvoranu, ažurirati ju i ukloniti uz popratnu akciju uklanjanja svih treninga koji su vezani uz tu dvoranu. Svi obrasci uporabe interagiraju s Bazom podataka



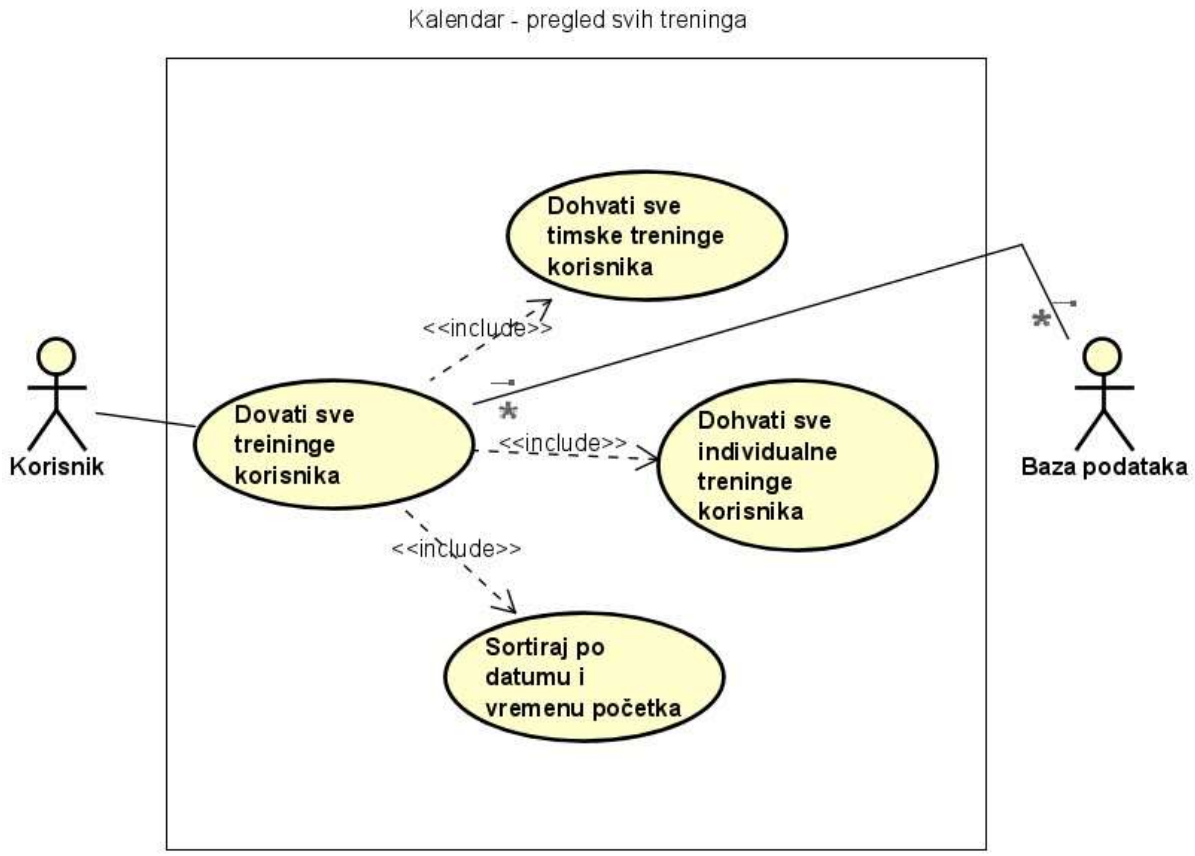
Slika 3.5. Interakcija Korisnik – Pozivnica

Slika 3.5. prikazuje interakciju između korisnika i pozivnice. Kada je korisnik pozvan u tim onda dobiva pozivnicu. Korisnik pozivnicu može prihvatiti čime postaje član tima i automatski dobiva sve timske treninga tima kojeg je prihvatio ili može odbiti pozivnicu u tim. Svi obrasci uporabe interagiraju s aktorom Baza podataka.



Slika 3.6. Interakcija Korisnik – Individualni trening

Slika 3.6. prikazuje interakciju između korisnika i individualnog treninga. Korisnik može kreirati i po potrebi ažurirati individualni trening. Ukoliko korisnik ne može doći na individualni trening jednostavno ga može ukloniti iz kalendara. Svi obrasci uporabe interagiraju s aktorom Baza podataka.



Slika 3.7. Interakcija Korisnik – Kalendar

Slika 3.7. prikazuje interakciju između korisnika i kalendara. Korisnik na klijentskoj strani ima posebnu komponentu za kalendar u kojoj ima pregled svih individualnih i timskih treninga. U samoj komponenti kalendara korisniku je prema prethodnim obrascima uporabe grafički omogućeno da ukloni individualni ili odbije timski trening. Pri dohvatu svih treninga važno je da su oni sortirani prema datumu i vremenu održavanja od najranijeg do najzadnjeg. Obrazac uporabe interagira s aktorom Baza podataka.

3.2 Izrada relacijskog modela baze podataka

U zadatku je navedeno da je potrebno izgraditi bazu podataka kako bi bili definirani svi entiteti kojima upravlja aplikacija. Iz proširenog teksta zadatka završnog rada napravljen je ER model baze podataka koji je pogodan za takve slučajeve. Temeljem ER modela baze podataka kreira se relacijski model baze podataka gdje će biti definirane sve relacije u bazi podataka kako bi rad aplikacije bio funkcionalan prema svim funkcijskim zahtjevima aplikacije.

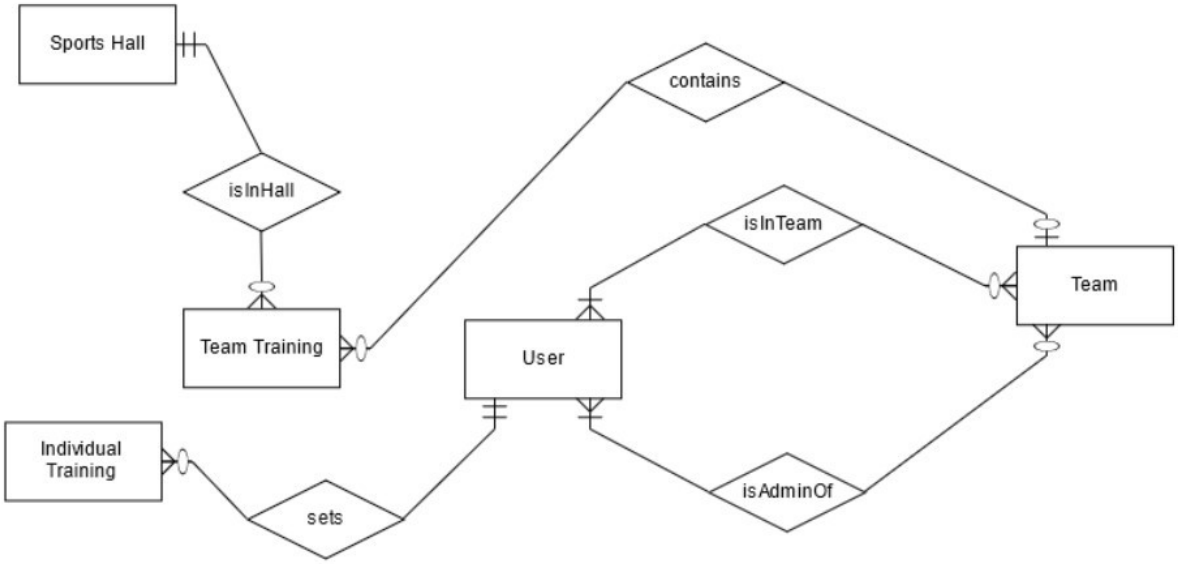
3.2.1 Preoblikovani tekst završnog rada

Preoblikovani tekst zadatka je u obliku koji je pogodan za kreiranje ER dijagrama tj. takav da se mogu iz njega iščitati svi entiteti, veze između entiteta i svi atributi entiteta. Da bi se lakše razabralo što je entitet, a što veza i koji su atributi entiteta i veza, u tekstu je entitetima pozadina svijetlo siva, veze su označene podebljano, a atributi su podvučeni crtom.

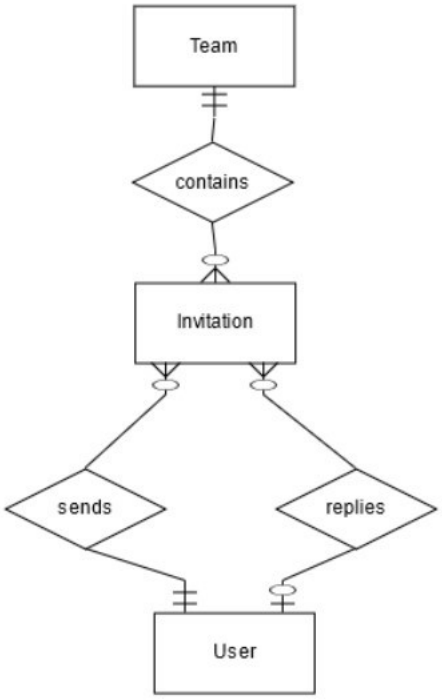
"Aplikacija omogućava prijavu i registraciju korisnika. Korisnik je opisan pomoću atributa: id (primarni ključ), username (jedinstven), firstName, lastName, dateOfBirth,profilePictureUrl. Korisnik može kreirati tim opisan atributima: id (primarni ključ), name, logoUrl, dateOfFoundation. Kreiranjem tima korisnik **je u timu** i korisnik **je admin tima**. Ovlast administratora tima može dodati drugim članovima tima samo administrator tima. Za dodavanje korisnika u tim **šalju se** pozivnice opisane atributima: id(primarni ključ), text i dateOfFoundation koje **sadrže tim**". Korisnik **odgovara** na pozivnicu. Korisnik u kalendar **može dodati** individualni trening opisan atributima: id (primarni ključ), trainingBegin, trainingEnd, sportType, remark, description i place, te **može postaviti** timski trening ako je admin tima. Timski trening je opisan atributima: id (primarni ključ), trainingBegin, trainingEnd, sportType, description i remark. Postavljanjem timskog treninga automatski se svim članovima u kalendar postavlja timski trening. Ako član tima ne može prihvatiti trening, ima mogućnost odbijanja treninga čime se briše njegova prisutnost na treningu i uklanja iz kalendara. Za svaki timski trening **vezana je** dvorana u kojoj se održava tako da se prije postavljanja treninga provjerava dosupnost dvorane. Dvorana je opisana sljedećim atributima: id (primarni ključ), name, pictureUrl, locationUrl, capacity. Osim dvorane dodjeljuje se i vrsta sporta koji se trenira, što će korisniku dodatno biti naznačeno u kalendaru za svaki trening. Početno, aplikacija prikazuje prijavu/registraciju korisnika. Nakon uspješne ovjere, korisnika se preusmjerava na izbornik gdje je vidljiv korisnikov kalendar. Osim kalendara na izbornoj traci korisniku su dostupni sljedeći izbornici: Grupe, Pozivnice i Profil. Aplikaciju je potrebno ostvariti u programskom jeziku po vlastitom izboru.

3.2.2 ER model baze podataka

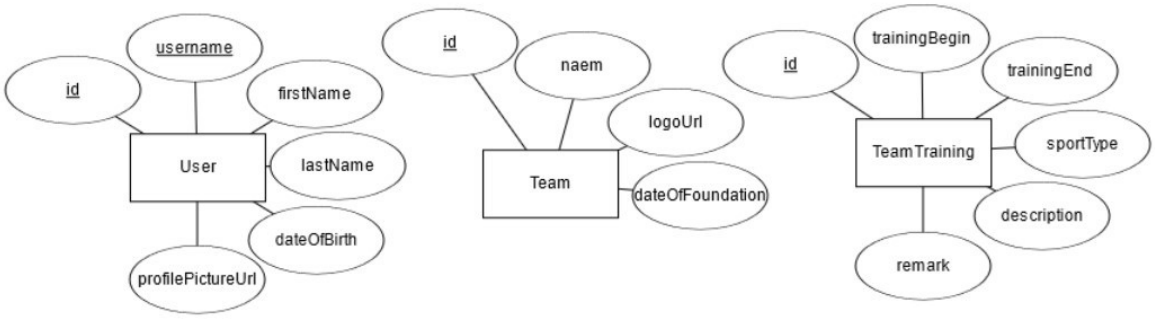
U ovom potpoglavlju prikazan je ER model baze podataka prema podacima iz prethodnog potpoglavlja. Zbog bolje preglednosti ER dijagrama neki entiteti su više puta upotrebljeni pri skiciranju modela te se njihovi atributi ne nalaze odmah na dijagramu zbog gužve koju bi to prouzročilo, već su stavljeni ispod u slike 3.10 i 3.11. Nazivi entiteta i veza su promijenjeni u engleski jezik. Slika 3.8. i 3.9. prikazuju ER dijagram.



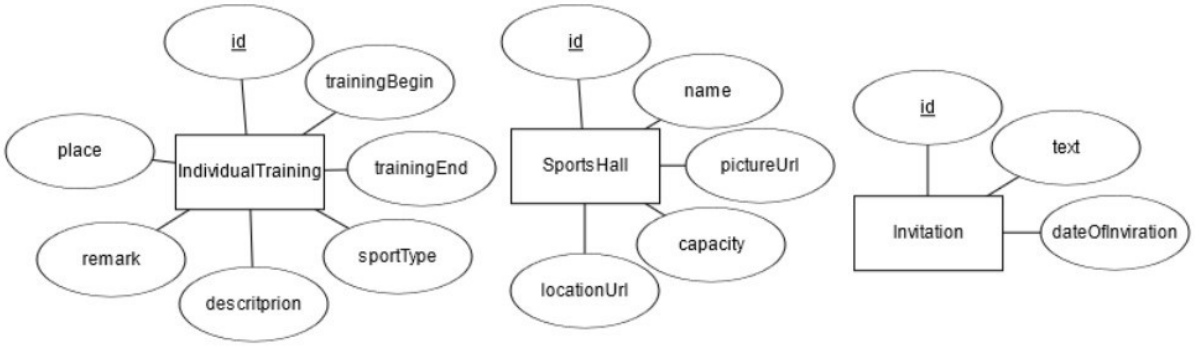
Slika. 3.8 ER diagram - prvi dio



Slika. 3.9 ER diagram - drugi dio



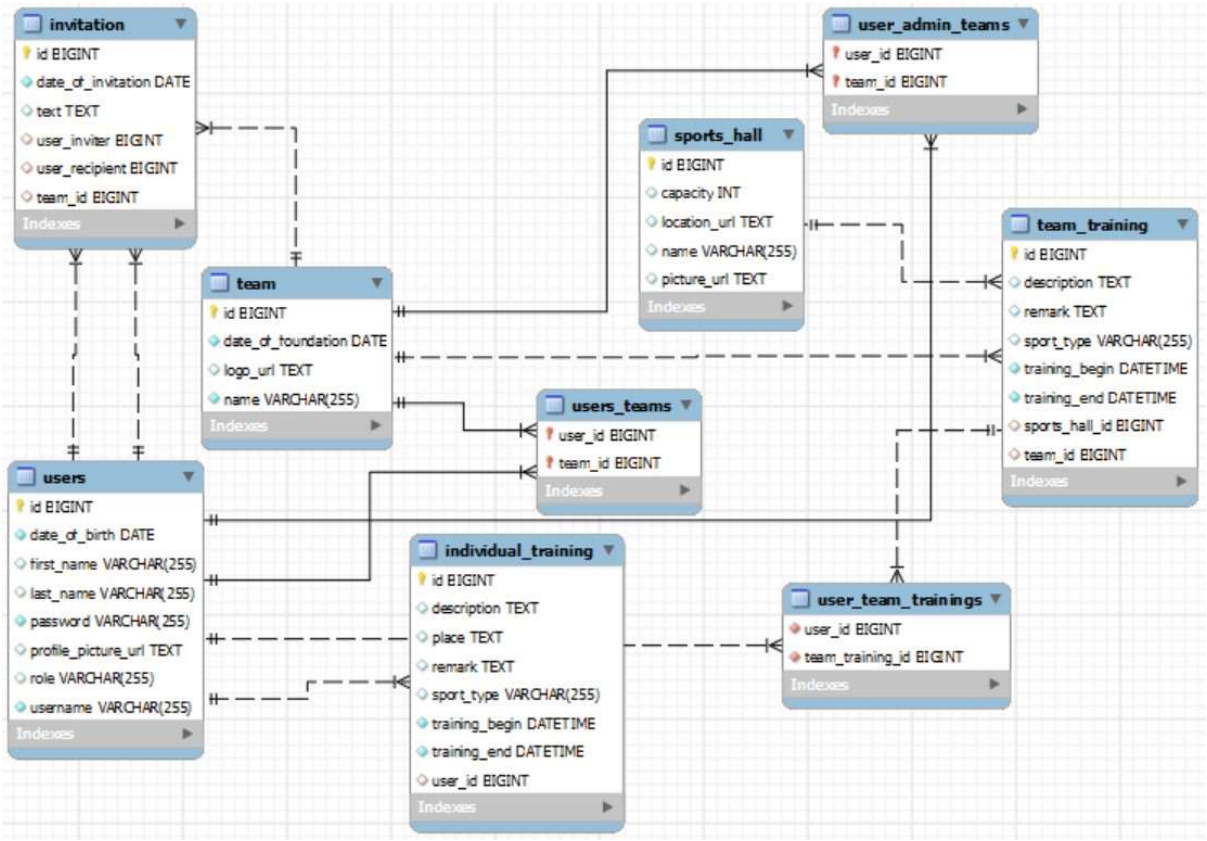
Slika. 3.10 Atributi entiteta - prvi dio



Slika. 3.11 Atributi entiteta - drugi dio

3.2.3 Relacijski model baze podataka

Relacijski model baze podataka je automatski generiran pri uporabi SpringBoota. Slika 3.12 prikazuje relacijski model baze podataka.



Slika 3.12 Relacijski model baze podataka

4. Poslužiteljska strana – opis i primjer izgradnje

Poslužiteljska strana (engl. *back-end*) je implementirana pomoću četiri Springova radna okvira. To su: SpringBoot, Spring Security, Spring Data i Spring. U ovom poglavlju su opisani: 1) princip rada Spring radnog okvira – što su ubacivanje ovisnosti (engl. *Depenency Injection*, kraće DI) i inverzija upravljanja (engl. *Inversion of Control*, kraće: IoC), 2) zašto se koristi SpringBoot, 3) kako najlakše inicijalizirati SpringBootov projekt, 4) opis koraka izgradnje REST API-ja pomoću Spring Boota te 5) kako je osigurana aplikacija (poslužitelj) pomoću Spring Securitya. Za sva navedena potpoglavlja navedeni su primjeri programskog koda koji su korišteni u izradi ovog završnog rada.

4.1 Osnove radnog okvira Spring

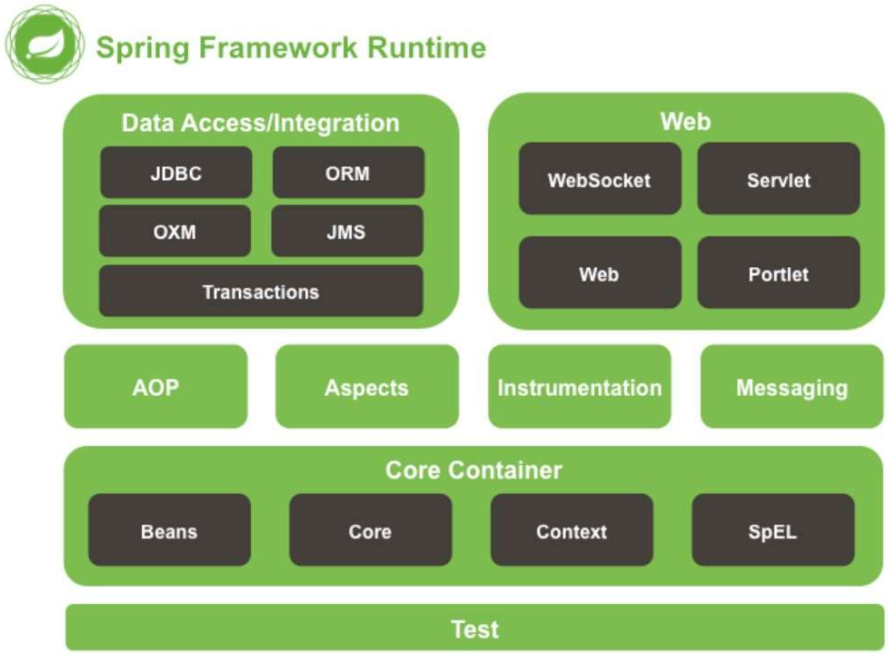
Radni okvir Spring karakteriziraju pojmovi kao što su: Bean, DI, IoC, aplikacijski kontekst i Springov IoC kontejner.

- IoC / DI – je temeljna značajka Springovog radnog okvira koja obavlja nekoliko zadataka: kreiranje objekata (Beanova), konfiguriranje i sastavljanje njihovih zavisosti i upravljanje njihovim životnim ciklusima. IoC kontejner koristi DI za upravljanje komponentama koje tvore aplikaciju. S obzirom na to da upravljanje Springovim komponentama radi radni okvir, a ne razvojni programer, otuda naziv inverzija upravljanja (engl. *Inversion of Control*).
- Spring Bean – je objekt ili komponenta kojom upravlja IoC kontejner, a nalazi se u aplikacijskom kontekstu.

4.2 Spring vs. Spring Boot

U ovom potpoglavlju je objašnjena razlika između radnih okvira Spring i SpringBoot.

Spring je temeljni projekt cijelog Springovog eko-sustava koji nudi niz modula za izradu Springove aplikacije. Moduli su prikazani na slici 4.1.



Slika 4.1 Moduli Springa

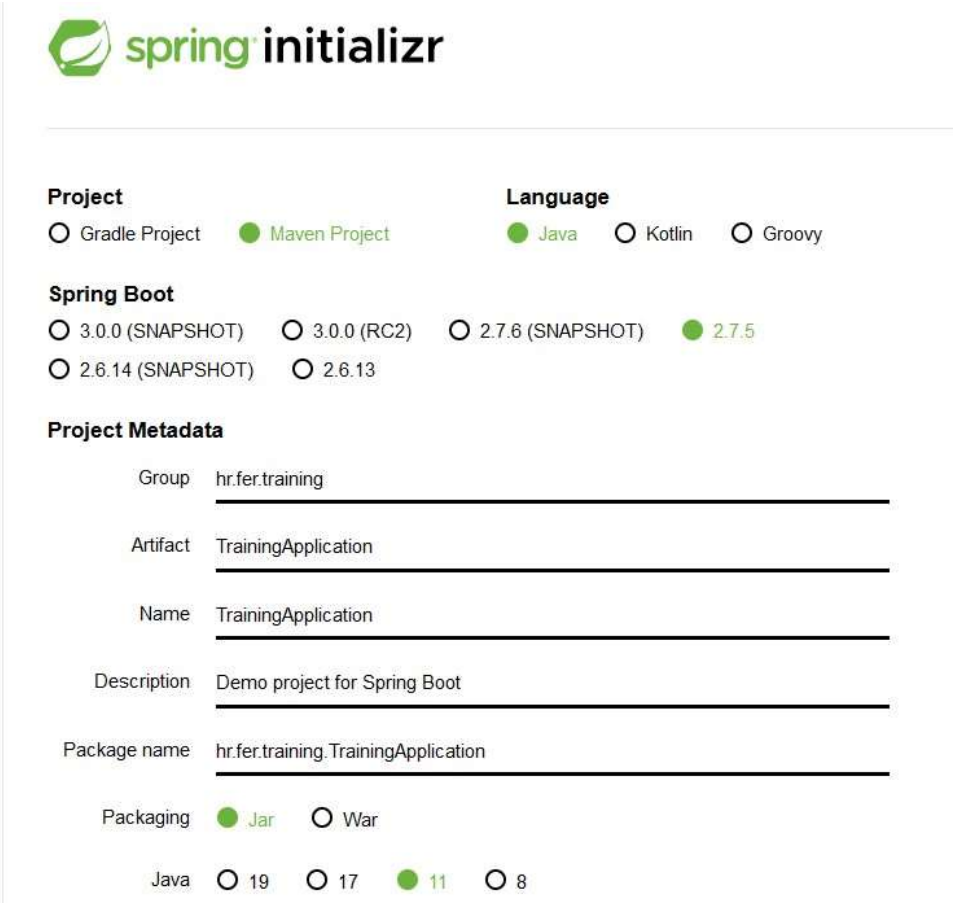
Spring Boot je napravljen kao proširenje Springa s ciljem bržeg početka izgradnje aplikacije. On ne mijenja Spring nego ga koristi u pozadini. Temeljna karakteristika Spring Boota u odnosu na projekt Spring je da pojednostavljuje izradu Springovih aplikacija. Za izgradnju aplikacije u radnom okviru Spring potrebno je mnogo konfiguracije: spajanje s poslužiteljem, spajanje s bazom podataka, konfiguracija za izgradnju MVC-aplikacija... Spring Boot nudi mogućnost izrade aplikacija s minimumom konfiguracije i dolazi s ugrađenim HTTP poslužiteljem(Tomcat, Jetty, Undertow) i ugrađenom bazom podataka. Također, sve verzije zavisnosti s kojima radi Spring Boot su automatski konfigurirane.

4.3 Generiranje Spring Bootovog projekta

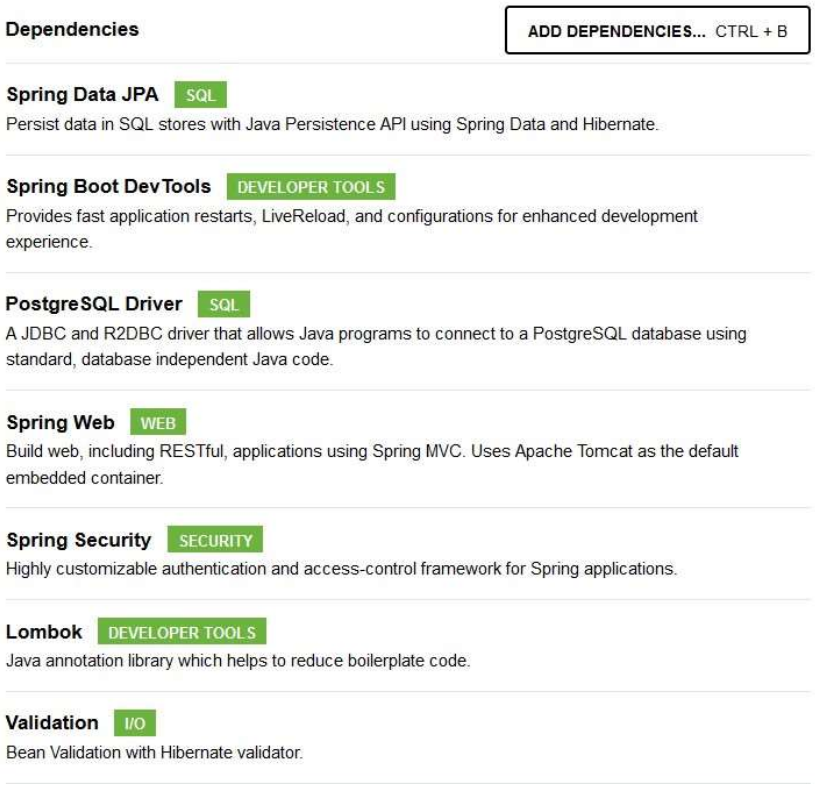
Spring Boot aplikaciju je jednostavno generirati preko web stranice https://start.spring.io/. Primjer inicijalizacije slijedi na slikama 4.2 i 4.3. Navedene ovisnosti su prikazane u tablici 4.1.

Naziv zavisnosti	Opis zavisnosti
Spring Boot Dev Tools	Omogućuje osvježavanje poslužitelja na promjene u programskom kodu
Spring Web	Za izgradnju RESTful aplikacija koristeći Spring MVC, dolazi s ugrađenim poslužiteljem Tomcat.
PostgreSQL Driver	JDBC-driver za korištenje baze podataka PostgreSQL
Spring Data JPA	Specifikacija za komunikaciju s bazom podataka, koristi Spring Data i Hibernate
Spring Security	Za provjeru ovjere i ovlasti u aplikaciji tj. pristup resursima
Validation	Nudi niz anotacijskih provjera i ograničenja nad entitetima aplikacije
Lombok	Anotacijska biblioteka koja reducira „boilerplate“ kod (pisanje gettera, settera, metoda equals, hashcode...)

Tablica 4.1. Ovisnosti aplikacije radnog okvira Spring Boot

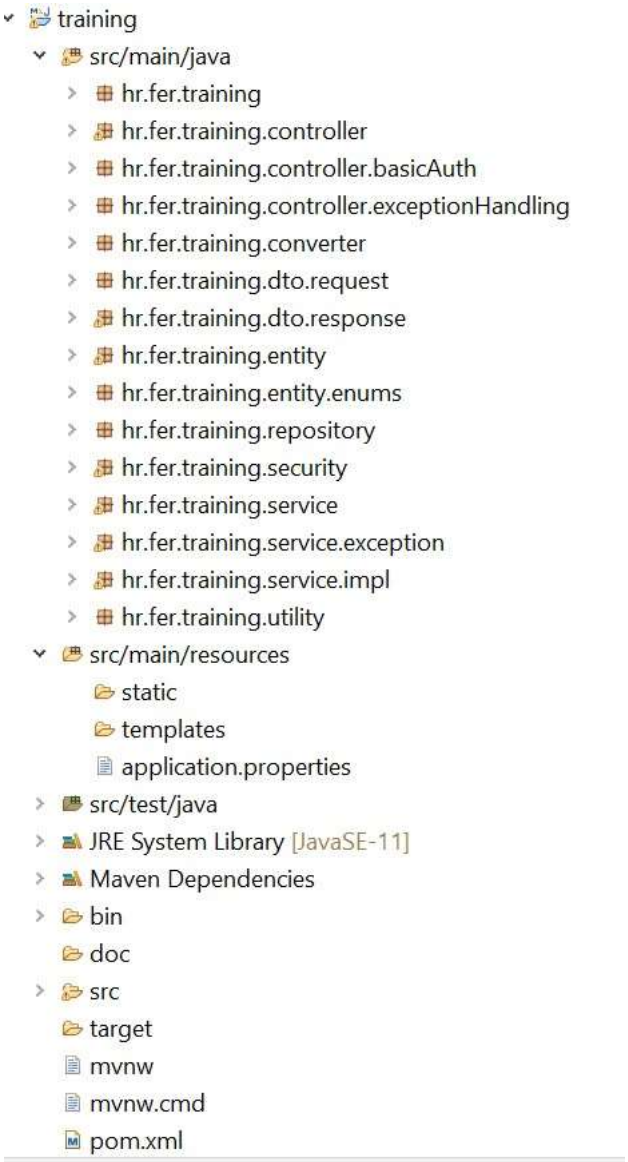


Slika 4.2 Generiranje(Inicijalizacija) projekta u mavenu – prvi dio



Slika 4.3 Generiranje (inicijalizacija) projekta u Mavenu– drugi dio

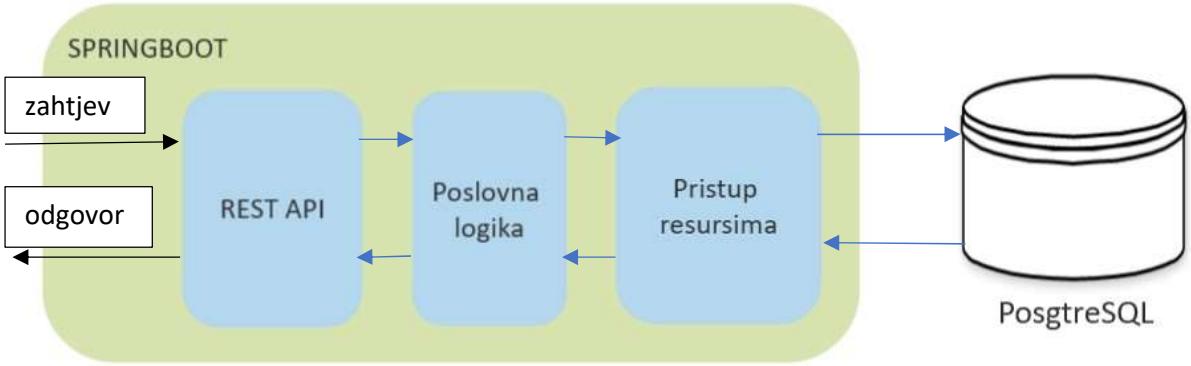
Slika 4.4 prikazuje raspakirani generirani Mavenov projekt. Na slici su prikazani svi paketi u src/main/java koje koristi gotova aplikacija, dok neposredno nakon raspakiranja projekta postoji samo paket hr.fer.training. Svi paketi će biti poslije objašnjeni.



Slika 4.4 Raspakirani Mavenov projekt.

4.4 Izgradnja poslužitelja

U ovom potpoglavlju opisana je izrada poslužitelja kroz slojeve aplikacije s pridodanim slikama programskog koda. Aplikacija ima tri sloja: *controller*, *service* i *repository*. Slika 4.5 prikazuje konceptualnu shemu poslužitelja.



Slika 4.5 Konceptuala shema poslužitelja

4.4.1 Entiteti aplikacije

U ovom potpoglavlju su definirani entiteti aplikacije. Prema prethodnom ER modelu baze podataka iščitavamo koji su entiteti u aplikaciji, koji su njihovi atributi i koje su veze između njih i prema tome kreiramo entitete s pripadajućim varijablama u klasi. U ER modelu baze podataka ovog završnog rada postoje dvije vrste veze, to su „0..1 : 0..N“ i „0..N : 0..N“. Primjer je prikazan kao izgradnja entiteta „User“. Hibernate će u pozadini ostvariti sve naredbe prema bazi podataka (kreiranje relacijskog modela, definiranje ograničenja, povezanost s drugim entitetima itd.). Kod pretvorbi ER modela u relacijski model baze podataka sve veze postaju entiteti i ako je moguće stapaju se s entitetom koji dijeli s njime primarni ključ. Primjer između entiteta „User“ i „Invitation“ i veza „sends“: jedna pozivnica ima točno jednog pošiljatelja, a jedan korisnik može slati više pozivnica. Veza „sends“ ima dva atributa, to su primarni ključevi relacije „User“ i „Invitation“. Odabiremo entitet „Invitation“ kao „owner side“ i stopimo vezu „sends“ u relaciju „Invitation“. Sada enitet „Invitation“ ima još jedan stupac, a to je primarni ključ entiteta „User“ koji je pošiljatelj pozivnice. Time je primarni ključ entiteta „User“ ušao u tablicu entiteta „Invitation“ pod nazivom „user_inviter“ koji definiramo u programskom kodu. Kod relacija „0..N“ : „0..N“, npr. između entiteta „User“, entiteta „Team“ i veze „isInTeam“ odabiremo jednu stranu kao „owner side“, u ovom slučaju to je entitet „User“ i u njegovom programskom kodu definiramo relaciju „Many to many“. U ovom slučaju nastat će relacija „isInTeam“ te se neće spojiti niti s jednim od navedenih entiteta, nego će se kreirati zasebna tablica u relacijskom modelu baze podataka pod nazivom „users_teams“. Sve veze između entiteta su dvosmjerne, tako da ćemo pored klase koja je „owner-side“ imati i u drugoj klasi „referenced-side“ čime ostvarujemo u programu poslužitelja pristup podacima preko obje klase entiteta. Na slici 4.6 i 4.7 prikazani su isječci programskog koda koji pokazuju članske varijable entiteta i veze s drugim entitetima. Svi entiteti su isprogramirani u paketu „hr.fer.training.entity“. Neki od njih koriste konstante koje se nalaze u paketu „hr.fer.training.entity.enums“. Koriste se enumeracije za tip sporta (FOOTBALL, HANDBALL...) i za ulogu u sigurnosti aplikacije (USER, ADMIN). Svaka klasa entiteta ima anotaciju @Entity.

```java
@Entity
@Table(name="users")
@Getter @Setter
@NoArgsConstructor
@AllArgsConstructor
public class User {

    //Fields
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable=false, unique=true)
    private String username;

    @Column(nullable=false)
    @Size(min = 4, message = "Password length must be 4+ chars")
    private String password;

    @Column
    @NotBlank
    private String firstName;

    @Column
    @NotBlank
    private String lastName;

    @Column(nullable = false)
    @Past
    private LocalDate dateOfBirth;

    @Column(columnDefinition = "TEXT")
    private String profilePictureUrl;

    @Enumerated(EnumType.STRING)
    @Column
    private Role role = Role.USER;
```

Slika 4.6 Članske varijable entiteta User

```java
    //Realtions
    @ManyToMany
    @JoinTable(
            name="users_teams",
            joinColumns = {@JoinColumn(name="user_id")},
            inverseJoinColumns = {@JoinColumn(name="team_id")})
    private Set<Team> teams = new LinkedHashSet<>();


    @ManyToMany
    @JoinTable(name="user_admin_teams",
            joinColumns = {@JoinColumn(name="user_id",referencedColumnName = "id")},
            inverseJoinColumns = {@JoinColumn(name="team_id",referencedColumnName = "id")})
    private Set<Team> userAdminTeams = new LinkedHashSet<>();

    @ManyToMany
    @JoinTable(name="user_team_trainings",
            joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
            inverseJoinColumns= {@JoinColumn(name="team_training_id",referencedColumnName="id")})
    private List<TeamTraining> teamTrainings = new ArrayList<>();

    @OneToMany(mappedBy="individualUser", cascade = CascadeType.REMOVE)
    private List<IndividualTraining> individualTrainings = new ArrayList<>();

    @OneToMany(mappedBy="inviter", cascade = CascadeType.REMOVE)
    private List<Invitation> sendedInvitations = new ArrayList<>();

    @OneToMany(mappedBy="recipient", cascade = CascadeType.REMOVE)
    private List<Invitation> recievedInvitations = new ArrayList<>();
```

Slika 4.7 Relacije entiteta User s drugim entitetima

4.4.2 Repozitoriji

Preko repozitorija se pristupa bazi podataka i radi kreiranje, ažuriranje, brisanje i ostale operacije prema bazi podataka. Svi repozitoriji klasa imaju anotaciju „@Repository“ kojom se naglašava uloga beana da ima pristup i komunikaciju s bazom podataka. Svi repozitoriji su kreirani u paketu „hr.fer.training.repository“. Svi repozitoriji su sučelja koja nasljeđuju JpaRepository<Tip entiteta,Tip idja>. JPA (Jakarta Persistence API) je specifikacija koja definira skup sučelja za koje traži implementaciju. Hibernate je taj koji nudi implementaciju. Preko repozitorija JPA postoji automatski nekoliko metoda: count, existsById, findAll, findAllById, findById, getOne, delete, deleteAll, deleteAllInBatch, deleteById, flush, save, saveAll, saveAndFlush. Ako su nam potrebne druge metode, jednostavno ih prema konvenciji napišemo u sučelju, a Hibernate će se pobrinuti za implementaciju. Slika 4.8 prikazuje dodatne usluge koje su potrebne u sustavu u sučelju UserRepository. Ako su potrebni kompliciraniji SQL upiti koristi se JPQL (Jakarta Persistence Query Language).

```
11@Repository
12public interface UserRepository extends JpaRepository<User, Long> {
13
14    boolean existsByUsername(String username);
15
16    Optional<User> findByUsername(String username);
17
18    void deleteByUsername(String username);
19}
```

Slika 4.8 Repozitorij User

4.4.3 Servisi

Svi servisi na poslužitelju imaju anotaciju „@Service“ kojom se naznačuje da je riječ o poslovnoj logici unutar klase. Sučelja sadrže metode koje je potrebno implementirati i nalaze se u paketu „hr.fer.training.service“, dok je ponuđena implementacija sučelja kreirana u paketu „hr.fer.training.service.impl“. uz navedena dva paketa izgrađen je još jedan paket koji se odnosi na iznimke u servisnom sloju. One su definirane u paketu „hr.fer.training.exception“. Svako sučelje ima definirane CRUD metode (**C**reate, **R**ead, **U**pdate, **D**elete) i dodatne pripadajuće metode koje su potrebne u aplikaciji. Primjer sučelja je prikazan na slici 4.9.

```
15 @Service
16 public interface UserService {
17
18     //CRUD
19     UserCreateUpdateRS createUser(User user);
20     UserRS getUser(String username);
21     List<UserRS> getAllUsers();
22     UserCreateUpdateRS updateUser(Long id, User user);
23     void deleteUser(Long id);
24
25     //others
26     void deleteAccount(String username);
27     Set<GeneralTrainingRS> getAllUserTrainings(String username);
28     void cancelTeamTraining(String username, Long teamTrainingId);
29     List<TeamRS> getAllUserTeams(String username);
30     List<InvitationRS> getAllUserInvitations(String username);
31     void leaveTeam(Long teamId, String username);
32
33 }
```

Slika 4.9 Primjer sučelja servisa

Na slici 4.9 je vidljivo kako metode servisa ne vraćaju direktno podatke od entiteta „User“. Razlog tomu je što ponekad podaci na klijentskoj strani ne moraju biti svi iz određenog entiteta ili nisu svi potrebni za neki servis, nego samo dio njih. Npr. kada bi metoda „createUser“ vraćala entitet „User“ onda bi se na klijentsku stranu poslali svi podaci vezani za entitet „User“ kao što su npr. liste timova, individualnih i timskih treninga koji bi također dohvaćali iz njihovih entiteta sve podatke i liste podataka, što je nepotrebno, a i završilo bi se u beskonačnoj petlji nizanja podataka. Beskonačni niz podataka tada bi mogao proizaći npr. iz entiteta „User“ kad bi bila dohvaćena lista timskih treninga, povukla bi se lista korisnika timskog treninga čime bi se opet vratilo na entitet User“ i pozvalo bi se opet listu timskih treninga, itd. Iz tog razloga napravljeni su modeli ili DTO (**D**ata **T**ransfer **O**bject) koji su pojašnjeni u sljedećem potpoglavlju zajedno s konverterima. Unutar implementacije servisa izvršava se validacija pomoću razreda „Assert“ koje nudi Java te unutar njih pomoću konvertera radimo pretvorbu u modele koje vraćamo na klijentsku stranu. Ako se dogodi iznimka nju će uhvatiti „exception handler“ koji je pojašnjen u poglavlju 4.4.6.

Na slici 4.10 je prikazan primjer za implementaciju metode kreiranja korisnika.

```
57    @Override
58    public UserCreateUpdateRS createUser(User user) {
59        Assert.isNull(user.getId(),
60            "[Layer: Service, Class: UserServiceImpl, Method: createUser, Text: Id of user at create must be null]");
61        Assert.notNull(user.getUsername(),
62            "[Layer: Service, Class: UserServiceImpl, Method: createUser, Text: username cannot be null]");
63        Assert.notNull(user.getPassword(),
64            "[Layer: Service, Class: UserServiceImpl, Method: createUser, Text: password cannot be null]");
65        Assert.notNull(user.getFirstName(),
66            "[Layer: Service, Class: UserServiceImpl, Method: createUser, Text: firstName cannot be null]");
67        Assert.notNull(user.getLastName(),
68            "[Layer: Service, Class: UserServiceImpl, Method: createUser, Text: lastName cannot be null]");
69        Assert.notNull(user.getDateOfBirth(),
70            "[Layer: Service, Class: UserServiceImpl, Method: createUser, Text: dateOfBirth cannot be null]");
71        Assert.notNull(user.getProfilePictureUrl(),
72            "[Layer: Service, Class: UserServiceImpl, Method: createUser, Text: profilePictureUrl cannot be null]");
73        if(userRepository.existsByUsername(user.getUsername())) {
74            throw new EntityExistsException("[Layer: Service, Class: UserServiceImpl, Method: createUser]"
75                + " Text: User with username:" + user.getUsername() + " already exists" );
76        }
77
78        user.setPassword(passwordEncoder.encode(user.getPassword()));
79        User u = userRepository.save(user);
80        return UserConverter.toCreateUpdateUserRS(u);
81    }
```

Slika 4.10 Kreiranje korisnika u servisnom sloju

4.4.4 Modeli i konverteri

U prethodnom potpoglavlju (4.4.3) opisana je potreba za modelima. Modeli kojima se koristi poslužitelj su podijeljeni u dva paketa. U „hr.fer.training.dto.request“ nalaze se modeli koje očekujemo primiti s klijentske strane, zajedno s paketom „hr.fer.training.dto.response“ koji sadrži modele koje šaljemo na klijentsku stranu. Na slici 4.11 prikazan je model vezan uz metodu korisnika koji napušta tim.

```
5 @Data
6 //@Data = @Getter, @Setter, @RequiredArgsConstructor, @EqualsAndHashCode, @ToString
7 public class TeamLeaveTeamDto {
8
9     private Long teamId;
10    private String username;
11
12 }
```

Slika 4.11 Prikaz modela za korisnikovo napuštanje tima

Iz modela je vidljivo kako su za metodu napuštanja tima potrebne samo dvije varijable – „id“ tima i „username“ korisnika.

Konverteri su klase u kojima su implementirane pretvorbe između entiteta u modele i obrnuto. Svi konverteri se nalaze unutar paketa „hr.fer.training.converter“, a na slici 4.12 je prikazan konverter za entitet sportske dvorane (SportsHall).

```
7  public class SportsHallConverter {
8
9      // Entity to SportsHall DTO
10     public static SportsHallRS toSportsHallRS(SportsHall sh) {
11         return new SportsHallRS(sh.getId(), sh.getName(), sh.getPictureUrl(),
12                 sh.getCapacity(), sh.getLocationUrl());
13     }
14
15     // SportsHallDto to Entity
16     public static SportsHall toSportsHall(SportsHallDto shd) {
17         return new SportsHall(null,shd.getName(),shd.getPictureUrl(),
18                 shd.getCapacity(),shd.getLocationUrl());
19     }
20
21 }
```

Slika 4.12 Prikaz konvertera za entitet sportske dvorane

4.4.5 Kontroleri

U ovom potpoglavlju opisan je sloj kontroler (nadzornik). Svi kontroleri su označeni anotacijom „@RestController“ koja je spoj dviju anotacija – „@Controller“ i „@ResponseBody“. Anotacija „@Controller“ naznačuje da se radi o razredu sloja kontroler sloju odnosno o razredu u kojem definiramo upravljanje nadolazećim HTTP zahtjevima na poslužitelj. Anotacija „@Controller“ anotacija je blisko povezana s anotacijom „@RequestMapping“, koja govori na koju metodu ovisno o zahtjevu povezujemo metodu u razredu kontrolera, jer ovisno u nadolazećem URLu na kontroler poziva se odgovarajuća metoda. Anotacija „@RequestMapping“ može biti specifičnija odnosno iz nje izlaze: „@PostMapping“, „@GetMapping“, „@PutMapping“, „@DeleteMapping“... koje naznačuju koja HTTP-metoda mora postojati da bi se pozvala konkretna metoda u kontroleru. Anotacija „@ResponseBody“ govori da svaka metoda u kontroleru vraća podatke prema klijentu. Format podataka u komunikaciji kontrolera i klijenta je JSON (**J**ava**S**cript **O**bject **N**otation). Na kontrolerske beanove postavljena je i anotacija „@CrossOrigin“ koja je vezana uz sigurnost na poslužitelju. Ona je objašnjena u daljnjem poglavlju 4.4.7. Slike 4.13 i 4.14 prikazuju isječak programskog koda TeamControllera. Svaka metoda vraća tipizirani objekt „ResponseEntity<T>“. ResponseEntity<T> je klasa u kojoj su sažeti podaci za HTTP tijelo i HTTP status.

```
24@RestController
25@RequestMapping("/api/teams")
26@CrossOrigin(origins = "http://localhost:3000")
27public class TeamController {
28
29    @Autowired
30    private TeamService teamService;
31
32    //CRUD - delete is under ADMIN authority
33    @PostMapping
34    public ResponseEntity<TeamRS> createTeam(@RequestBody TeamCreateDto team){
35        return new ResponseEntity<>(teamService.createTeam(team), HttpStatus.CREATED);
36    }
37    @GetMapping
38    public ResponseEntity<List<TeamRS>> getAllTeams(){
39        return new ResponseEntity<>(teamService.getAllTeams(),HttpStatus.OK);
40    }
41
42    @GetMapping("/{id}")
43    public ResponseEntity<TeamRS> getTeam(@PathVariable Long id){
44        return new ResponseEntity<>(teamService.getTeam(id), HttpStatus.OK);
45    }
46
47    @PutMapping("/{id}")
48    public ResponseEntity<TeamRS> updateTeam(@PathVariable Long id, @RequestBody Team team){
49        if(id != team.getId()) {
50            throw new IllegalArgumentException(
51                "[Layer: Controller, Class: TeamController, Method: updateTeam, Text: idjevi se ne poklapaju]");
52        }
53        return new ResponseEntity<TeamRS>(teamService.updateTeam(team),HttpStatus.OK);
54    }
```

Slika 4.13 Isječak programskog koda kontrolera Team

```
56      //OTHERS
57      @PutMapping("/addUser")
58      public ResponseEntity<TeamRS> addUser(@RequestBody TeamChangeDto tcd){
59          return new ResponseEntity<TeamRS>(teamService.addUser(tcd), HttpStatus.OK);
60      }
61
62      @PutMapping("/removeUser")
63      public ResponseEntity<TeamRS> removeUser(@RequestBody TeamChangeDto tcd){
64          return new ResponseEntity<TeamRS>(teamService.removeUser(tcd), HttpStatus.OK);
65      }
66
67      @PutMapping("/addAdmin")
68      public ResponseEntity<TeamRS> addAdmin(@RequestBody TeamChangeDto tcd){
69          return new ResponseEntity<TeamRS>(teamService.addAdmin(tcd), HttpStatus.OK);
70      }
71
72      @PutMapping("/removeAdmin")
73      public ResponseEntity<TeamRS> removeAdmin(@RequestBody TeamChangeDto tcd){
74          return new ResponseEntity<TeamRS>(teamService.removeAdmin(tcd), HttpStatus.OK);
75      }
76  }
```

Slika 4.14 Isječak programskog koda kontrolera Team

4.4.6 Iznimke

U programskom kodu mogu se dogoditi iznimke na više mjesta. Primjerice u servisu ako se postavi trening unutar zauzetog termina ili se dogodi iznimka pri provjeri s razredom Assert itd., u entitetu tako da je narušeno neko od ograničenja iz paketa anotacija javax.validation (@Size, @Past, @Future...) ili se iznimka dogodi vezano uz sigurnost aplikacije npr. korisnik nema ovlast za određenu operaciju. Stoga hvatanje iznimki je postavljeno u podpaket „hr.fer.training.controller.exceptionHandling“. Sve iznimke hvataju se u jednoj globalnoj klasi anotiranoj s „@ControllerAdvice“. Pomoću anotacije „@Order“ i vrijednošću Ordered.HIGHEST_PRECEDENCE postavljamo najviši prioritet među komponentama. Podaci o iznimci predstavljeni su u objektu ResponseError, a iznimka se hvata pomoću anotacije „@ExceptionHandler“. Slika 4.15 prikazuje isječak iz programskog koda GlobalExceptionHandlera.

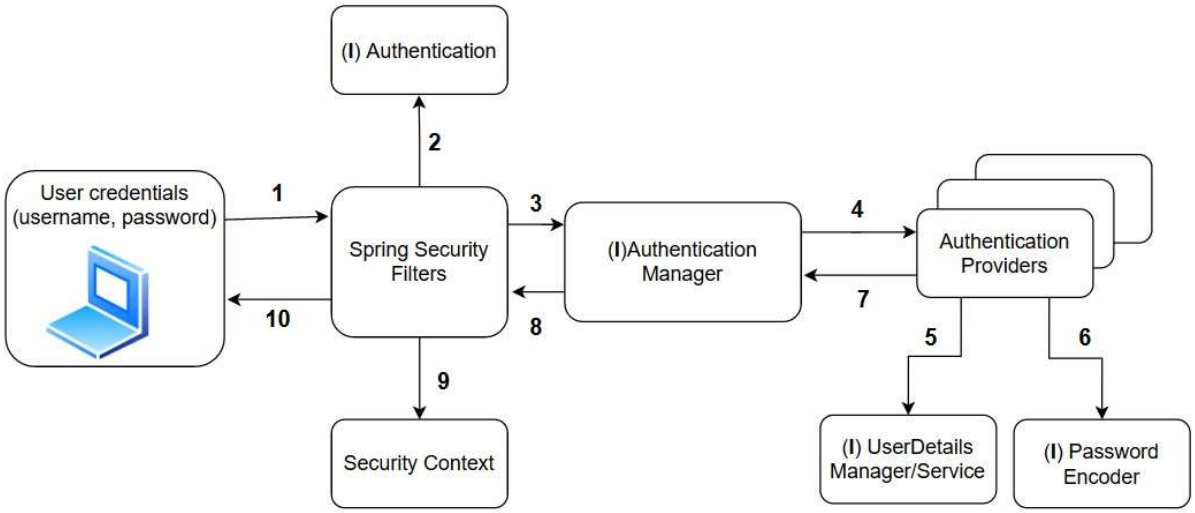
```
17  @Order(Ordered.HIGHEST_PRECEDENCE)
18  @ControllerAdvice
19  public class GlobalExceptionHandler {
20
21      @ExceptionHandler(SportsHallAlreadyReservedException.class)
22      public ResponseEntity<ResponseError> handleSportsHallAlreadyReservedExceptions(Exception exc){
23          ResponseError re = new ResponseError();
24          re.setMessage(exc.getMessage() + "~~~~~~~" + exc.getCause() + "~~~~~~~" + exc.getStackTrace());
25          re.setLocalDateTime(LocalDateTime.now());
26          re.setStatus(HttpStatus.CONFLICT.value());
27          return new ResponseEntity<ResponseError>(re, HttpStatus.CONFLICT);
28      }
29
30      @ExceptionHandler(EntityNotFoundException.class)
31      public ResponseEntity<ResponseError> handleEntityNotFoundExceptions(Exception exc){
32          ResponseError re = new ResponseError();
33          re.setMessage(exc.getMessage() + "~~~~~~~" + exc.getCause() + "~~~~~~~" + exc.getStackTrace());
34          re.setLocalDateTime(LocalDateTime.now());
35          re.setStatus(HttpStatus.NOT_FOUND.value());
36          return new ResponseEntity<ResponseError>(re, HttpStatus.NOT_FOUND);
37      }
```

Slika 4.15 Isječak iz programskog koda GlobalExceptionHandlera

4.4.7 Sigurnost

Sigurnost na poslužiteljskoj strani je implementirana u radnom okviru Spring Security. U ovom završnom radu podrška sigurnosti u aplikaciji je ostvarena s verzijom Spring Security 5. Sigurnost aplikacije je važna jer osigurava korisniku zaštitu privatnih podataka i manipuliranje s istima te osigurava poslovnu logiku unutar aplikacije. Putem Spring Securitya, osim zadane ovjere implementacije preko DaoAuthenticationProvidera, moguća je ovjera „method level security“, što je ovjera putem JWT(**JSON W**eb **T**oken) tokena, ovjera kroz OAuth2 ili vlastita implementacija ovjere. U sigurnosti je potrebno implementirati ovjeru korisnika i ovlasti koje ima. Također, implementirana je i pretvorba osjetljivih podataka (npr. lozinka) kako bi aplikacija bila maksimalno osigurana kao i zaštita od hakerskih napada CSRF (**C**ross **S**ite **R**equest **F**orgery) te CORS(**C**ross **O**rigin **R**esource **S**haring).

Slika 4.16 opisuje arhitekturu odnosno koncept ovjere korisnika. Ispod nje su pojašnjeni razredi i sučelja kojima se Spring Secuitry koristi i tok ovjere.

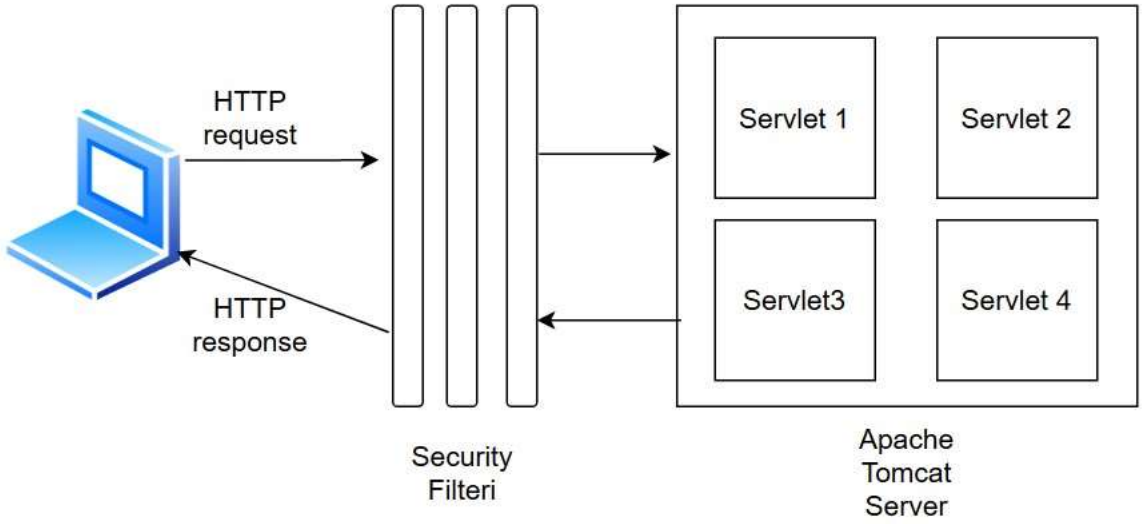


Slika 4.16 Koncept ovjere korisnika

- Spring Secutiry Filters je niz razreda kroz koje svaki nadolazeći zahtjev prolazi. Ukoliko je potrebna ovjera, filter UsernamePasswordAuthenticationFilter od nadolazećih podataka (username, password) kreira objekt Authentication. Zadano, filter DefaultLoginPageGeneratingFilter neovjerenog korisnika preusmjerava na stranicu login i generira korisničko ime „user“ i lozinku.
- Authentication je sučelje koje predstavlja podatke o korisniku unutar konteksta Spring Security i ono se kreira kada dobijemo podatke od korisnika (username i password) na poziv ovjere (login). Authentication sadrži nekoliko metoda, to su: getAuthorities(), getCredentials(), getDetails(), getPrincipal(), isAuthenticated() i setAuthenticated(). Zadana implementacija sučelja je UsernamePasswordAuthenticationToken.
- Authetication Manager je sučelje s jednom metodom – authenticate() unutar čije implementacije ProviderManager (zadana implementacija Spring Securitya) provjerava listu svih Authentication Providera. Ukoliko dobijemo pozitivan odgovor od strane nekog od AuthentcationProvidera vraća se objekt Authentication u filtere koji ga prosljeđuju u Security Context. AuthenticationManager radi pretvorbu iz UserDetails u objekt tipa Authentication.

- AuthenticationProvider je sučelje koje obavlja logiku ovjere i ima dvije metode, to su authenticate() i supports() kojom označava koji tip objekta Authentication se koristi u ovjeri s tim AuthenticationProviderom. Zadani Authentication Provider je DaoAuthenticationProvider koji radi ovjeru uz pomoć implementacija sučelja UserDetailsService/UserDetailsManager i PasswordEncodera. Ukoliko je ovjera neuspješna za svaki AuthenticationProvider aplikacija će vratiti HTTP status 401.
- UserDetailsService je sučelje u kojoj je potrebno implementirati jedinu metodu loadUserByUsername(String username) čija je uloga obaviti logiku ovjere. Metoda vraća objekt tipa UserDetails.
- UserDetailsManager je proširenje UserDetailsServicea koje nudi dodatan niz metoda ukoliko želimo implementirati više mogućnosti. One su: createUser(UserDetails user), updateUser(UserDetails user), deleteUser(UserDetails user), changePassword(String oldPws, String newPwd), userExists(String username). Osim vlastite, Spring Security nudi tri implementacije UserDetailsManagera, to su: InMemoryUserDetailsManager, JdbcUserDetailsManager, LdapUserDetailsManger. U ovom završnom radu radimo samo s UserDetailsService jer metode iz UserDetailsManagera ne želimo implementirati unutar sigurnosti aplikacije nego unutar kontrolera.
- UserDetails je sučelje čijom implemetacijom prikazujemo podatke o korisniku za vrijeme ovjere u AuthenticationProviderima. UserDetails sadrži nekoliko metoda: getAuthorities(), getPassword(), getUsername(), isAccountNonExpired(), isAccountNonLocked(), isEnabled(), isCredentialsNonExpired(), eraseCredentials().
- PasswordEncoder je sučelje koje koristi metodu matches(CharSequence rawPassword, String encodedPassword) za usporedbu valjanosti lozinke i metodu encode(CharSequence rawPassword) kojom kodiramo ili *hashiramo* lozinku kako ne bi bila čitljiva ni lako pogodljiva za sigurnosne napade.
- SecurityContext je kontejner koji sadrži ovjerene zahtjeve.

Komunikacija između klijenta i poslužitelja je ostvarena preko protokola HTTP. Korisnik šalje HTTP-zahtjev koji prolazi kroz filtere za provjeru ovlasti i ovjere. Unutar te zadaće Spring Securitya, poslužitelj se koristi HTTP-Servletima. HttpServleti su Javine klase unutar poslužitelja (u projekt je automatski ugrađen poslužitelj Apache Tomcat) koje parsiraju nadolazeći zahtjev, vade podatke iz zaglavlja i tijela zahtjeva, dok pri slanju odgovora kreiraju HTTP-odgovor. Slika 4.17 prikazuje komunikaciju između filtara i servleta.



Slika 4.17 Komunikacija između filtara i servleta

Glavno sučelje kroz koje su implementirane ovlasti i ovjera korisnika je apstraktna klasa WebSecurityConfigurerAdapter. Nasljeđujući navedeno sučelje nadjačavaju se dvije metode: configure(HttpSecurity http) za ovlasti i configure(AuthenticationManagerBuilder auth) za ovjeru korisnika. Za ovjeru se koristi DaoAuthenticationProvider kojemu proslijedimo bean PasswordEncoder i implementaciju sučelja UserDetailsService. Unutar implementacije UserDetailsService kroz metodu loadUserByUsername(String username) komuniciramo s bazom podataka koristeći repozitorij UserRepository. Odgovor metode loadUserByUsername() je objekt tipa UserDetails koji je također implementiran. PasswordEncoder koji se koristi je BCryptPasswordEncoder. Slika 4.18 prikazuje implementaciju ovlasti, a slika 4.19 prikazuje implementaciju ovjere.

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter{

    //ovlasti
    @Override
    protected void configure(HttpSecurity http) throws Exception {

        //csrf
        http.csrf().disable();

        //ovlasti
        http.authorizeRequests()
            //za preflight requestove
            .antMatchers(HttpMethod.OPTIONS,"/**").permitAll()
            .antMatchers(HttpMethod.POST, "/api/users").permitAll()
            .antMatchers("/api/**").authenticated()
            .antMatchers("/admin/**").hasRole("ADMIN")
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .and()
            .httpBasic();
    }
```

Slika 4.18 Implementacija ovlasti

```
//ovjera
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.authenticationProvider(authenticationProvider());
}

@Autowired
private UserDetailsService userDetailsService;

@Bean
DaoAuthenticationProvider authenticationProvider() {
    DaoAuthenticationProvider daoAuthProvider = new DaoAuthenticationProvider();
    daoAuthProvider.setPasswordEncoder(passwordEncoder());
    daoAuthProvider.setUserDetailsService(this.userDetailsService);
    return daoAuthProvider;
}

@Bean
PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Slika 4.19 Implementacija ovjere

Osim ovjere i ovlasti potrebno je pružiti podršku CORS za web preglednike i spriječiti neželjene napade hakera kao, npr. CSRF. Zadani web preglednici (Mozilla, Chrome...) ne obavljaju komunikaciju prema drugim izvorima (engl. *origin*). Izvor je URL koji je kombinacija HTTP-protokola, adrese poslužitelja/klijenta i vrata (port) na kojem je poslužena aplikacija. Omogućavanje CORS-a možemo napraviti tako da postavimo na svaki kontroler anotaciju @CrossOrigin(origins = „http://localhost:3000) ili globalno definiramo u razredu koji implementira sučelje WebMvcConfigurer. Slika 4.20 prikazuje globalnu konfiguraciju CORS-a kojom dozvoljavamo da klijent s izvora http://localhost:3000 pristupa svim putanjama aplikacije i može obaviti sve metode. U samoj komunikaciji web preglednici prije nego što pošalju HTTP-zahtjev, šalju „pre-flight“ zahtjev kojim provjeravaju ima li klijentska aplikacija pravo na poziv prema resursu na poslužitelju.

```
@Configuration
@EnableWebMvc
public class CorsConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**").allowedOrigins("http://localhost:3000").allowedMethods("*");
    }
}
```

Slika 4.20 konfiguracija CORSa

CSRF je čest napad neovlaštenih i neovjerenih korisnika. Kada se korisnik prijavi u klijentskoj aplikaciji, Spring Security generira kolačić JSESSIONID i vraća ga na web preglednik kako bi isti, umjesto slanja za svaki zahtjev prenosio osjetljive podatke (korisničko ime, lozinku), slao kolačić koji je poznat samo njegovom web pregledniku. No haker može postaviti u preglednik zlonamjeran link ili sliku iza kojeg stoji akcija prema poslužitelju. Npr. u bankovnoj aplikaciji haker uputi zahtjev s prebacivanjem novca drugih klijenata na svoj račun. Ovakav tip napada može se spriječiti slanjem tokena CSRF ili XSRF koji je skriven i haker ga ne može znati ni time upotrijebiti za zlonamjernu akciju, a poslužitelj u nadolazećim zahtjevima provjerava preko tokena CSRF je li zahtjev poslao korisnik ili haker. Za potrebe ove aplikacije onemogućena je sigurnost preko CSRF-a što je vidljivo sa slike 4.18.

4.4.8 Datoteka application.properties

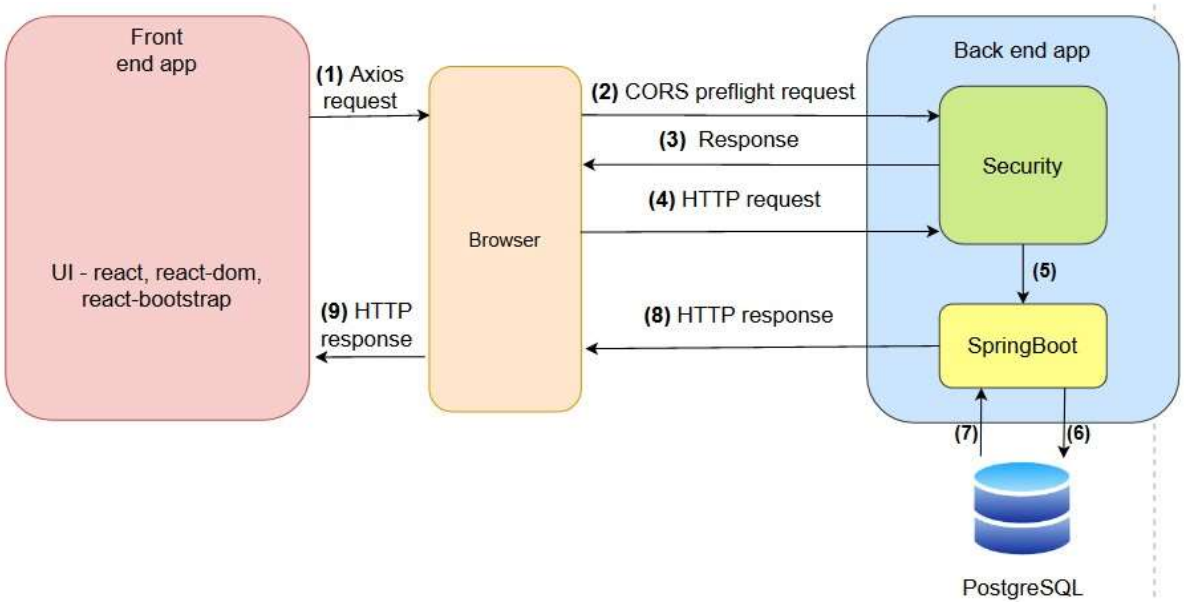
Generiranjem Mavenovog projekta, generira se datoteka application.properties koja služi kao konfiguracijska datoteka. Sve vezano uz konfiguraciju definiramo unutar te datoteke. Slika 4.21 prikazuje isječak iz datoteke vezan za konfiguraciju aplikacije, baze podataka i sigurnosti.

```
### SPRING APLIKACIJA
spring.application.name=TrainingApplication

### POSTGRESQL
spring.datasource.url=jdbc:postgresql://localhost:5433/TrainingAppDatabase
spring.datasource.username=postgres
spring.datasource.password=bazepodataka
spring.jpa.hibernate.ddl-auto=update
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
spring.jpa.show-sql=true
    #omogucava se punjenje baze podataka iz resursa .sql filea pri porketanju aplikacije
spring.sql.init.mode=always

### SECURITY
spring.security.user.name=ivan
spring.security.user.password=ivan
spring.security.user.roles=USER
```

Slika 4.21. Isječak iz konfiguracijske datoteke application properies

5. Klijentska strana – opis i primjer izgradnje

Klijentska strana – (engl *front-end*) je implementirana pomoću biblioteke React. Osim biblioteke React koriste se i druge biblioteke kako bi zadovoljili funkciju aplikacije (react-router-dom, axios...) . U ovom poglavlju opisane su osnove biblioteke React te opis i uloga ostalih biblioteka, a dan je i primjer izgradnje jednostavne web aplikacije koja komunicira prema poslužitelju. Slika 5.1 prikazuje skicu kako Reactova aplikacija komunicira sposlužiteljem na kojem su podaci za prikaz, dok je ispod nje prikazan princip rada.



Slika 5.1 Skica komunikacije klijentske s poslužiteljskom stranom

Klijentska strana – aplikacija komunicira pomoću preglednika (engl. browser), npr. Mozilla Firefox, prema poslužiteljskoj strani.

1. Tijekom rada klijentske aplikacije, klijent npr. klikom gumba pokrene određenu metodu servisa (servisi, axios... su pojašnjeni u poglavlju 5.3.2) koja pomoću biblioteke axios šalje zahtjev pregledniku.
2. Preglednik prvo šalje „pre-flight“ zahtjev kojim porvjerava ima li dozvoljen pristup poslužitelju.
3. Poslužiteljska strana vraća odgovor pregledniku. Ukoliko je on negativan, preglednik u konzoli vraća iznimku, a ukoliko je odgovor pozitivan, postupak se nastavlja.
4. Preglednik šalje HTTP-zahtjev temeljen prema pozivu iz klijentske aplikacije (postavljanje zaglavlja, postavljanje podataka koje šaljemo na poslužitelj... Ukoliko zahtjev zadovoljava konfiguraciju sigurnosti na poslužiteljskoj strani, tijek radnje se nastavlja.
5. SpringBoot pomoću servleta parsira zahtjev i izvršava radnju prema dobivenom zahtjevu.
6. Poslužitelj pristupa bazi podataka.
7. Baza podataka vraća podatke prema Spring Bootovoj aplikaciji.
8. Poslužitelj pomoću „servleta“ kreira HTTP-odgovor i šalje ga prema pregledniku.
9. Klijentska strana prihvaća odgovor i pomoću biblioteka kreira prikaz podataka u komponentama.

5.1 Osnove biblioteke React

React je popularna JavaScriptova biblioteka koja se koristi za izgradnju korisničkih suučelja. U pozadini koristi vlastiti virtualni DOM (**D**ocument **O**bject **M**odel), objekt koji predstavlja strukturu svih čvorova HTML-dokumenta). Pomoću virtualnog DOM-a, kada React vidi gdje je promjena u virutalnom DOM-u u odnosu na RealDOM, koristi biblioteku za renderiranje ReactDom koja osigurava da će se izmijeniti samo one komponente nad kojima se dogodila promjena. React umjesto ponovnog prikaza cijele stranice mijenja samo one komponente (element ili grupu elemenata) na kojima se dogodi promjena. Takav proces se zove „reconciliation".

5.1.1 JSX

JSX (**J**ava**S**cript **X**ML) je sintaksno proširenje JavaScripta koje omogučuje pisanje „Markup languagea" i logike (bilo koji JavaScriptov izraz) u istoj funkciji (komponenti). Ukoliko unutar JSX-a želimo pozivati JavaScriptov izraz, koristimo vitičaste zagrade. Slika 5.2 prikazuje primjer JSX-ovog koda.

```
let text = <h1 className="title">HelloWorld</h1>;
```

Slika 5.2 Primjer JSX-a

JSX se u pozadini prevodi u pozive React.createElement(). Slika 5.3 prikazuje primjer pretvorbe porgramskog koda sa slike 5.2.

```
let text = React.createElement("h1", { className: "title" }, "Hello World");
```

Slika 5.2 Pretvorba JSX-a u pozive React.createElement()

Nužno je da JSX ima samo jedan vršni element. Primjerice, nije moguće napisati „<h2>HelloWorld</h2> Welcome to our page ". Potrebno je omotati elemente u jedan vršni element. Umjesto <h2></h2> , što je legitimno, elemente možemo omotati u prazan element, u Reactu zvan „React Fragment", <> <h2></h2> </>. Omatanje u React Fragment se koristi kako bi se u višestrukom gnježđenju komponenti izbjegli nanizani elementi koji bi zahtjevali prikaz u HTML-strukturi.

5.1.2 Komponente i svojstva props

Komponente su JavaScriptove funckije koje primaju svojstva, „props“ kao argument. Ideja komponenti je razdvojiti UI u manje komponente koje imaju svoj smisao prikaza i funkcionalnosti. Nužno je da komponenta počinje velikim početnim slovom. Dio UI-ja što komponenta prikazuje, definirano je u „return statementu“. Slika 5.3 prikazuje primjer komponente.

```
const FruitList = (props) => {
  let list = (
    <ul>
      <li>Apple</li>
      <li>Banana</li>
      <li>Kiwi</li>
    </ul>
  );
  return <>{list}</>;
};
```

Slika 5.3 Primjer komponente

Temeljni princip kreiranja UI-ja je razdvojiti što je više moguće komponente u podkomponente kako akcija na pojedinoj unutarnjoj komponenti ne bi utjecala na cijelu vršnu komponentu, tj. kako akcija na jednoj komponenti ne bi povlačila ponovni prikaz na cijelom dijelu DOM-a. Primjer je dan na slici 5.4.

```
const FruitList = (props) => {
  let list = (
    <ul>
      <li>Apple</li>
      <li>Banana</li>
      <li>Kiwi</li>
    </ul>
  );
  return <>{list}</>;
};

const VegetableList = (props) => {
  return (
    <ul>
      <li>Potato</li>
    </ul>
  );
};

const UII = (props) => {
  <>
    <FruitList />
    <VegetableList />
  </>;
};
```

Slika 5.4 Dekompozicija komponenti

Svaka kompomenta prima argument „props“. Podaci iz propsa su „read-only“ i mogu se koristiti unutar funkcije (komponente) kojoj je predano. Primjer prikazuje slika 5.5.

```
const UI = (props) => {
  let carList = ["Volvo", "Audi", "Kia"];
  return <CarList list={carList} />;
};

const CarList = (props) => {
  return (
    <>
      <ul>
        {props.list.map((car) => {
          <p>Car {car}</p>;
        })}
      </ul>
    </>
  );
};
```

Slika 5.5 Argumenti props

5.1.3 Komponentne State i Lifecycle

Stanje u komponentama baziranima funkcijama se koristi pomoću kopče (engl. *hook*) useState() (objašnjeno u poglavlju 5.1.4 Kopče). Od prije je podržano u komponentama baziranima na razredima. “State” predstavlja stanje komponente koje se prati. Promjenom stanja dogodi se svaki put ponovni prikaz (engl. *re-render*) komponente. Usko uz stanje, vezan je životni ciklus komponente, čije je ponašanje bilo opisano pomoću metoda: componentDidMount(), koja se poziva nakon što je komponenta prikazana, componentDidUpdate(), koja se poziva na promjenu stanja ili propsa usporedbom s prethodnim i componentWillUnmount(), koja se poziva prilikom brisanja komponente iz DOM-a. Funkcionalnosti navedenih metoda moguće je implementirati pomoću kopče useEffect() koje je pojašnjeno u poglavlju (5.1.4 Kopče).

5.1.4 Kopče

U verziji Reacta prije v.16.8, za promjenu stanja komponente i implementaciju metoda životnog ciklusa, kao i niz drugih Reactovih mogućnosti (npr. kontekst, optimizacija komponenti, „routing“...), React je koristio „class based“ komponente. Ukoliko su komponente bile pisane kao funkcije koje koriste npr. stanje, metode životnog tijeka itd., potrebno ih je pretvoriti u „class based“ komponente. Nakon verzije Reacta 16.8 komponente kao funkcije imaju mogućnost upravljati stanjem, metodama životnog ciklusa itd. pomoću funkcija zvanih kopče (engl. *hooks*) Moguće je pisati i vlastitu funkciju kopče.

Postoje dva pravila za funkcije kopče:

1. Kopče mogu biti korišteni samo u komponentama kao funkcijama.
2. Kopče mogu biti pozvani jedino na „top level“ komponente (ne u ugnježdenim funkcijama, petljama i uvjetima)

Moguće je koristiti istu kopču više puta u funkciji, a osnovna namjena kopči je ponovo korištenje logike sa stanjem (engl. *reuse statefull logic*).

U nastavku ovog podpoglavlja navedene su i ukratko opisane ugrađene metode kopči u Reactu i React Router DOM-u koje se koriste u ovom završnom radu.

- useState() je kopča koja omogućuje postavljanje varijable stanja u komponenti. Metoda vraća polje s dva elementa: stanje i setter metodu za ažuriranje stanja. Metoda može primiti prvotnu vrijednost stanja koje može biti bilo što (broj, polje, objekt...). Primjer korištenja je:
  const [name, setName] = useState(„Ana“);
- useHistory() je kopča iz react-router-doma koja vraća instancu povijesti (engl. *history*) koja se koristi pri navigiranju u aplikaciji. Primjer poziva je: let history = useHistory(); history.push(„/home“);
- useRef() kopča se koristi onda kada ne želimo ponovni prikaz komponente na svaku promjenu nekog stanja, tj. ref nije stanje nego „value container“. Primjerice, kada se koristi u komponenti forma npr. pisanje maila, nepotrebno je vrijednost tog ulaza postavljati kao „state“, jer će na svaku promjenu tj. upisivanjem svakog znaka ići re-ponovni prikaz komponente. Iz tog razloga povezuje se ulazni element s ref kontejnerom pomoću atributa „ref“, kojem pristupamo pozivom nameRef.current.value. Primjer prikazuje slika 5.6.

```
const NewMail = (props) => {
  const mailRef = useRef();

  const handleMailSubmit = (event) => {
    event.preventDefault();
    MailService.sendMail(mailRef.current.value);
  };

  return (
    <>
      <form onSubmit={handleMailSubmit}>
        <div>
          <input type="text" placeholder="Write your mail" ref={mailRef} />
          <button type="submit">Send</button>
        </div>
      </form>
    </>
  );
};
```

Slika 5.6 Primjer korištenja useRef()

- useEffect() je kopča koja se koristi za upravljanje „side-effectsima“ unutar komponente. „Side-effects“ su radnje koje se događaju nevezano za samu Reactovu biblioteku, npr. dohvat podataka ili ručno mjenjanje DOM-a. Kopča useEffect() prima dva argumenta – funkciju koju izvršava i opcionalno polje zavisnosti prema kojima se

ponovno izvršava radnja koju predstavlja prva funkcija. Ukoliko se ne preda polje zavisnosti useEffect() pokreće metodu na svaki ponovni prikaz komponente. Ukoliko želimo da se useEffect pokreće na promjenu nekog stanja, možemo ga dodati u polje zavisnosti pa će React usporedbom trenutnog stanja i prethodnog odrediti hoće li se ili ne izvršiti ponovno efekt (funkcija predana kao prvi argument). Ukoliko se kao drugi argument preda prazno polje, efekt će se primjeniti samo na prvi render. Ukoliko je potrebno funkcija (prvi argument) može vratiti metodu koja se primjenjuje na brisanje komponente iz DOM-a. Primjer korištenja prikazan je ispod u slici 5.7 u kojem pozivamo sve pozivnice koje korisnik ima i spremamo ih u stanje „invitations“ koje se u nastavku programskog koda koristi za prikaz.

```
const [invitations, setInvitations] = useState([]);

useEffect(() => {
  let username = AuthenticationService.getLoggedInUsername();
  InvitationService.getAllUserInvitations(username)
    .then((response) => {
      setInvitations(response.data);
    })
    .catch(() => {
      console.log("Unsuccessfull getting all user invitations");
    });
}, []);
```

Slika 5.7 Primjer korištenja kopče useEffect()

5.2 Generiranje Reactove aplikacije i dodatnih biblioteka

Generiranje Reactove aplikacije moguće je ostvariti naredbom u naredbenom retku „npx create-react-app training-app“. Osim Reacta instaliraju se i react-dom i react-scripts... Nakon generiranja projekta u datoteku „package.json“ upisuju se sve ovisnosti. Kako bi aplikacija zadovoljila potrebe zahtjeva potrebno je dodati još ovisnosti koje su navedene u tablici 5.1. Navedena tablica opisuje naziv ovisnosti, kratki opis i naredbu za instalaciju.

Naziv zavisnosti	Opis zavisnosti	Naredba u command line
Bootstrap	CSS radni okvir za izgradnju responzivnih aplikacija, sadrži niz unaprijed stiliziranih komponenti	npm install react-bootstrap bootstrap
React Router DOM	Biblioteka koja omogćuje korisniku navigiranje kroz komponente	npm install react-router-dom@5
Axios	HTTP biblioteka koja omogućuje slanje zahtjeva prema resursima	npm install axios
Moment	Biblioteka za formatiranje datuma	npm install moment

Tablica 5.1 Dodatne potrebne ovisnosti

U strukturu projekta, u mapi „src“, potrebno je definirati sljedeće mape kako bi imali dobar pregled komponenti i datoteka po ulogama. To su mape:

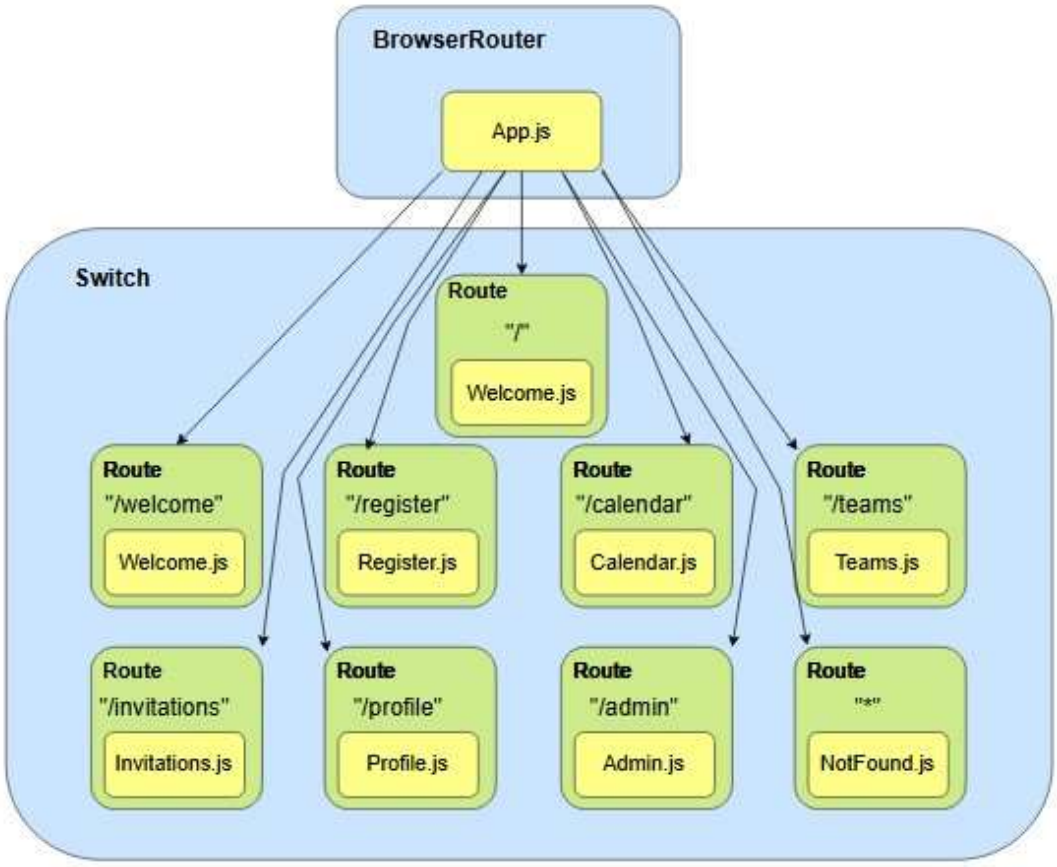
- „components“ - mapa za komponente
- „pages“ - mapa za komponente koje predstavljaju stranice u aplikaciji
- „styles“ - mapa za css datoteke
- „service“ - mapa koja sadrži razrede s definiranim metodama za dohvat i slanje podataka pomoću axios biblioteke

5.3 Izgradnja klijentskog dijela aplikacije

U ovom potpoglavlju opisan je tijek izrade klijentskog dijela uz pomoć prethodno navedenih biblioteka. U primjeru je opisano: definiranje stranica, sigurnost i slanje HTTP-zahtjeva prema serveru.

5.3.1 Komponente App.js i Routing

App.js je vršna komponenta u aplikaciji. S obzirom na potrebe navigiranja kroz stranice, u App.js formiramo usmjeravanje (engl. *routing*). Usmjeravanje je pojam vezan za navigiranje komponenti. Komponentu „App.js“ potrebno je omotati s komponentom <BrowserRouter> nakon čega je moguće definiranje ruta. Slika 5.8 opisuje koje rute želimo imati. Svaka ruta je omotana komponentom „<Route>“ kojoj se predaju atributi exact i path te pod sadržaj komponentu koju želimo prikazati na tu putanju. Atribut „exact“ govori da se komponenta prikazuje za točno određenu putanju, a „path“ opisuje URL na koji se postavlja pripadajuća komponenta.



Slika 5.8 Usmjeravanje

Slika 5.9 i 5.10 prikazuju programsko ostvarenje navigacije.

```
ReactDOM.render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>,
  document.getElementById("root")
);
```

Slika 5.9 Komponenta index.js

```
function App() {
  return (
    <div className="App">
      <Header />
      <main>
        <Switch>
          <Route exact path="/">
            <Redirect to="/welcome" />
          </Route>
          <Route exact path="/welcome">
            <Welcome />
          </Route>
          <Route exact path="/register">
            <Register />
          </Route>
          <Route exact path="/calendar">
            <Calendar />
          </Route>
          <Route exact path="/teams">
            <Teams />
          </Route>
          <Route exact path="/invitations">
            <Invitations />
          </Route>
          <Route exact path="/profile">
            <Profile />
          </Route>
          <Route exact path="/admin">
            <Admin />
          </Route>
          <Route path="*">
            <NotFound />
          </Route>
        </Switch>
      </main>
    </div>
  );
}
```

Slika 5.10 Programsko ostvarenje navigacije komponenti

5.3.2 Servisi i biblioteka Axios

Servisi na klijentskoj strani definiraju funkcije prilikom čijih poziva se šalju HTTP-zahtjevi prema poslužitelju. Axios je biblioteka koja omogućuje slanje HTTP-zahtjeva kao i presretanje HTTP-zahtjeva i odgovora. Slika 5.11 prikazuje servis za pozivnice.

```
import axios from "axios";

class InvitationService {
  //GET ALL USER INVITATIONS
  getAllUserInvitations(username) {
    return axios.get(
      `http://localhost:8080/api/users/getAllUserInvitations/${username}`
    );
  }

  //ACCEPT INVITATION
  acceptInvitation(object) {
    return axios.put("http://localhost:8080/api/teams/addUser", object);
  }

  //DECLINE = DELETE Invitation
  declineInvitation(id) {
    return axios.delete(`http://localhost:8080/api/invitations/${id}`);
  }
}

export default new InvitationService();
```

Slika 5.11 Prikaz servisa Invitation.js

5.3.3 Sigurnost, login, logout i presretanje HTTP zahtjeva

Sve metode vezane uz sigurnost, prijavu i odjavu korisnika nalaze se u datoteci AuthenticationService.js. Slika 5.12 prikazuje programski isječak koji opisuje poziv metode na prijavu (login) korisnika. Na klik gumba za prijavu u aplikaciju pokreće se metoda „executeBasicAuthenticationService“. Ukoliko dobijemo pozitivan odgovor, spremamo korisničko ime korisnika u sessionStorage i postavljamo „axios interceptor“ koji na svaki sljedeći HTTP-zahtjev s korisničke strane postavlja podatke korisnika u „HTTP header“.

```
class AuthenticatinService {
  //metoda za slanje na server je li postoji user -> login -> Login.jsx/Welcome.js
  executeBasicAuthenticationService(username, password) {
    //saljemo auth header i ako prođe s tim headerom tj. dobijemo odgovor znaci da postoji user
    let basicAuthHeader = "Basic " + window.btoa(username + ":" + password);
    return axios.get("http://localhost:8080/basicAuth", {
      headers: { authorization: basicAuthHeader },
    });
  }

  //metoda za spremanje logiranog usera u sessionStorage
  registerSuccessfullLogin(username, password) {
    sessionStorage.setItem("authenticatedUser", username);
    //kad usera loginamo postavljamo axios interceptor i dajemo username i password koji ce ici u header
    let basicAuthHeader = "Basic " + window.btoa(username + ":" + password);
    this.setupAxiosInterceptor(basicAuthHeader);
  }
```

Slika 5.12 Prijava korisnika

Slika 5.13 prikazuje postavljanje interceptora Axios koji presreće svaki zahtjev i šalje podatke o korisniku prema poslužitelju unutar HTTP-zaglavlja.

```
//definiranje interceptora
setupAxiosInterceptor(basicAuthHeader) {
  axios.interceptors.request.use((config) => {
    if (this.isUserLoggedIn()) {
      config.headers.authorization = basicAuthHeader;
    }
    return config;
  });
}
```

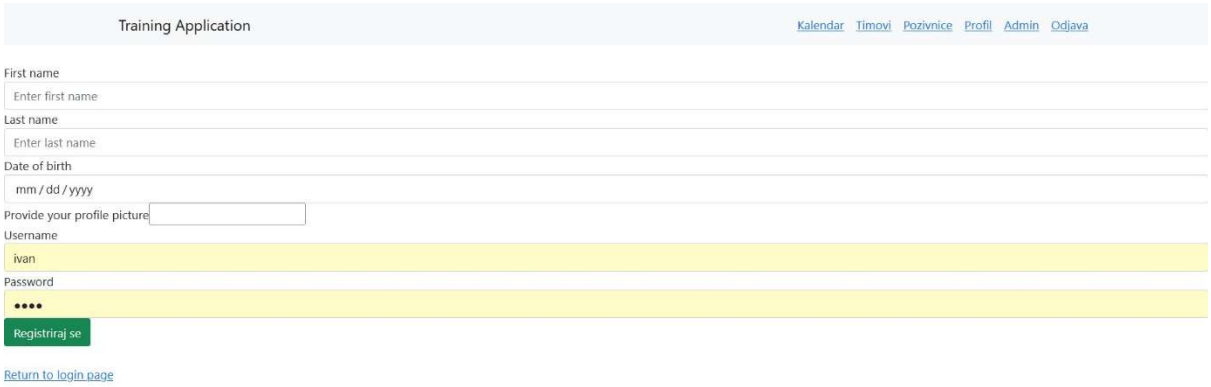
Slika 5.13 Definiranje interceptora biblioteke Axios

Odjava korisnika odvija se putem jednostavnog poziva metode pozivom „sessionStorage.removeItem(„authenticatedUser“)“.

5.4 Izgled i navigacija korisnika kroz aplikaciju

U ovom potpoglavlju opisani su izgled i navigacija korisnika kroz aplikaciju. Početno korisnik je usmjeren na prijavu u aplikaciju (slika 5.14). Ukoliko korisnik nema račun, može ga klikom na gumb „Registracija“ napraviti. Izgled forme za registraciju korisnika prikazan je na slici 5.15. Prijavom u aplikaciju korisnik ima linkove u zaglavlju kako bi se navigirao kroz istu.

Slika 5.14 Prijava korisnika u aplikaciju

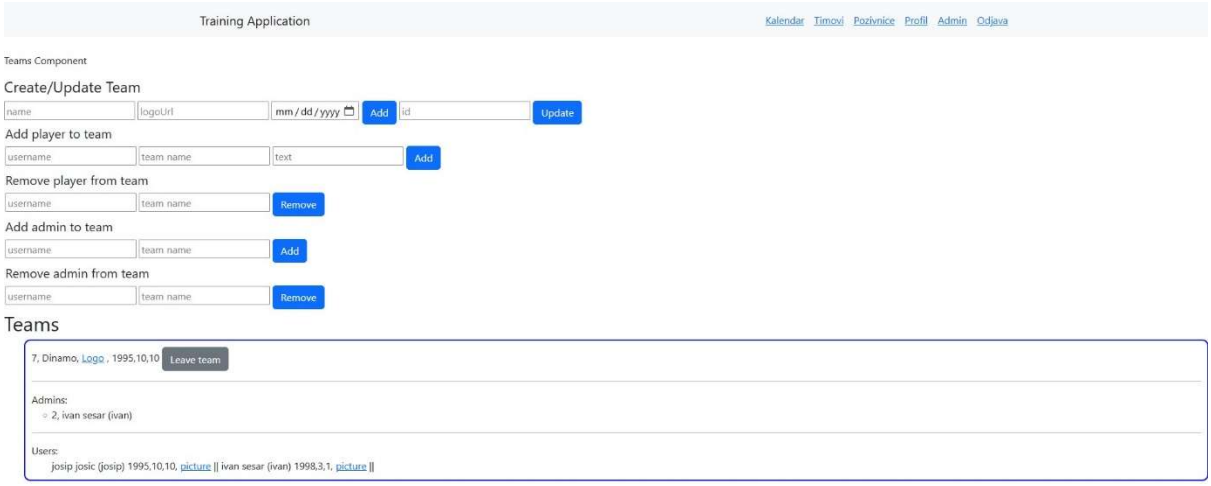


Slika 5.15 Prikaz obrasca za registraciju korisnika

Kada se korisnik prijavi u aplikaciju, preusmjerava ga se na kalendar treninga. Bilo individualni ili timski, treninzi su prikazani u obliku tablice sortirani prema datumu i vremenu održavanja od najranijeg do najkasnijeg. Korisnik na stranici kalendara klikom na određene gumbove ima mogućnost postavljanja novog timskog ili individualnog treninga (preusmjerava ga se na obrazac za stvaranja treninga), brisanja individualnog treninga ili otkazivanje dolaska na timski trening. Izgled prikazuje slika 5.16.

Slika 5.16 Kalendar individualnih i timskih treninga

Pod linkom „Timovi“ korisnika se preusmjerava na komponentu u kojoj može stvoriti novi tim, dodavati/brisati korisnike/administratore tima ovisno o ovlasti koje ima. I ima pregled svih timova u kojima se nalazi. Klikom na gumb „Leave Team“ korisnik može napustiti tim. Izgled prikazuje slika 5.17.



Slika 5.17 Team komponenta

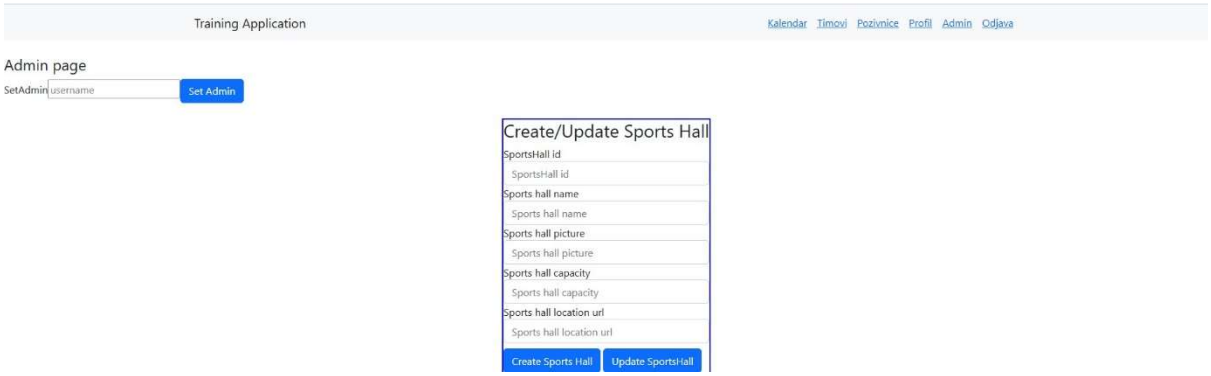
Pod linkom “Pozivnice” korisnik ima pregled svih pozivnica u timove gdje klikom na gumb “Accept” može prihvatiti pozivnicu odnosno klikom na gumb “Decline” odbiti pozivnicu od tima. Izgled prikazuje slika 5.18.

Slika 5.18 Izgled stranice s pozivnicama

Pod linkom “Profil” korisnika se preusmjerava na obrazac s osobnim podacima u kojemu je moguće ažurirati podatke o korisniku ili deaktivirati profil. Izgled prikazuje slika 5.19.

Slika 5.19 Izgled komponente “Profil”

Pod linkom “Admin” korisnik ima mogućnost dodavanja novog administratora aplikacije, stvaranje i ažuriranje sportske dvorane i pregled svih sportskih dvorana, korisnika i timova u aplikaciji. Ukoliko je korisnik administrator aplikacije ima mogućnost brisanja navedenih. Izgled prikazuju slika 5.20 i 5.21.



Slika 5.20 Izgled komponente “Admin” - prvi dio

Slika 5.21 Izgled komponente “Admin” - drugi dio

Klikom na link “Odjava” korisnik izlazi iz aplikacije.

Zaključak

Nakon izrade ovog završnog rada stekao sam nova iskustva i znanja vezano uz "full-stack" razvoj web aplikacija. Prilikom izrade rada istaknuo bih važnost snalaženja u potrazi za informacijama o pojedinim temama te njihovu primjenu u samom radu. Ponajviše sam se koristio tečajevima na web stranici Udemy [1-4], koja nudi niz tečajeva vezanih uz temu - programiranja. Niti jedna web stranica kao ni Udemyjev tečaj ne obuhvaća sve što je potrebno za izradu rada, već je bilo nužno potražiti informacije iz više izvora i primijeniti ih. Osim tečajeva koristio sam i se web stranicama kao što su npr. "www.baeldung.com" [5], www.bezkoder.com [6]. Posebno bih istaknuo kako je važno prije početka programiranja imati jasan uvid u zadatak. Pod jasnim uvidom smatram kako je potrebno iz teksta zadatka dobro analizirati i izvući svaku funkcionalnost koja se traži da bude implementirana. S dobrom analizom teksta zadatka i sistematičnosti u programiranju može se postići jasan i pregledan rad (gledano kao programski kod). Osobno sam zadovoljniji s napretkom pri izradi poslužitelja u odnosu na klijentsku stranu. Implementacija ovog rada nije kraj učenja nego tek početak za daljni razvoj npr. vezano za sigurnost rada otvoren je prostor za korištenje JWT tokena ili OAuth2 za ovjeru. Na klijenstkoj strani pak, otvoren je prostor za učenje o podršci kao što je npr. Redux, kvalitetnije stiliziranje aplikacije pomoću radnog okvira Bootstrap, ručno stiliziranje (pisanje .css datoteka), upoznavanje s radnim okvirom Next JS itd [7].

Literatura

[1] Darby C., Spring Boot 3: Learn Spring 6, Spring REST API, Spring MVC, Spring Security, Thymeleaf, JPA & Hibernate, (2023, ožujak). Poveznica: https://www.udemy.com/course/spring-hibernate-tutorial/; pristupljeno 10. rujna 2022.

[2] Bytes E., Spring Security Zero to Master along with JWT, OAUTH2, (2023, ožujak). Poveznica: https://www.udemy.com/course/spring-security-zero-to-master/; pristupljeno 10. siječnja 2023.

[3] Schwarzmuller M., React – The Complete Guide(incl Hooks, ReactRouter, Redux), (2023, veljača). Poveznica: https://www.udemy.com/course/react-the-complete-guide-incl-redux/; pristupljeno 10. prosinca 2022.

[4] Traversy B., Bootstrap 4 from Scratch With 5 Projects, (2018, lipanj). Poveznica: https://www.udemy.com/course/bootstrap-4-from-scratch-with-5-projects/; pristupljeno 10. listopada 2022.

[5] Baeldung, Poveznica: https://www.baeldung.com/; pristupljeno 10. rujna 2022.

[6] Bezkoder, (2022 listopad). Poveznica: https://www.bezkoder.com/; pristupljeno 20. listopada 2022.

[7] W3schools. Poveznica: https://www.w3schools.com/; pristupljeno 10. rujna 2022.

Popis slika

Popis tablica

,

Sažetak

Naslov: “Web aplikacija za individualno i timsko dogovaranje sportskih treninga u dvoranama”.

Sažetak: Na temelju zadatka završnog rada – izrade web aplikacije za individualno i timsko dogovaranje sportskih treninga u dvoranama, stvoren je prošireni tekst zadatka kako bi bili jasniji detalji. Iz proširenog teksta zadatka, definirani su entiteti, relacije između entiteta i atributi entiteta iz kojeg je napravljen ER model baze podataka. Iz ER modela baze podataka kreiran je relacijski model baze podataka. Sukladno tomu definirani su entiteti na poslužiteljskoj strani. Nakon povezivanja entiteta stvoreni su repozitoriji, nakon njih servisi koji sadrže metode za poslovnu logiku unutar aplikaciije, nakon njih nadzornici. Potom je kreirana podrška za hvatanje iznimki, zatim modeli za zahtjev i modeli za odgovor te konverteri koji sadrže statičke metode za pretvorbu entiteta u neki od modela i obrnuto. Po završetku na poslužiteljskoj strani ostvarena je podrška za sigurnost uz pomoć radnog okvira Spring Security. Na klijentskoj strani definirane su stranice aplikacije i u njima dohvat željenih podataka uz pomoć biblioteke Axios gdje su sve metode prema pozivu na poslužitelj smještene pod mapu “service“. Zadnje je stvorena podrška sigurnosti uz pomoć “sessionStoragea“ i metoda Axios interceptora.

Ključne riječi: razvoj web aplikacija, Spring Boot, Spring Security, REST API, JPA, React, Axios, usmjeravanje.

Summary

Title: “Web application for individual and team scheduling of sports trainings in gyms”

Summary: Based on the final thesis assignment – construction of a web application for individual and team scheduling of sports trainings in gyms, an expanded text of the assignment was made to make details clearer. From the expanded text entities, relations between entities and attributes of entities were defined, from which an ER database model was created. From the ER database model, a relation database model was made. Accordingly, application entities on the server side were defined. After connecting entities with each other, the repositories were created, and after them, the services which contain methods for business logic inside of an application were defined. Thereafter, controllers were created to handle HTTP requests to resources. Afterwards, support for exception handling was implemented, and models for request and response, as well as converters which contains static methods for converting entities to models and vice versa were made. By the end of implementation of the server side, security support by Spring Security framework was realized. On the client side, all application pages were defined, and, inside of them, data fetching was enabled with a help of the Axios library. All methods which communicate with the server were defined in the folder “service”. Lastly, security support with browser session storage and Axios interceptor methods was enabled.

Key words: web development, SpringBoot, Spring Security, REST API, JPA, React, Axios, routing.