Izrada full stack aplikacije Studomat za prijavu ispita

Maruna, Marko

Undergraduate thesis / Završni rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka / Sveučilište u Rijeci**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:195:942230

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-08-26**

Sveučilište u Rijeci
Fakultet informatike
i digitalnih tehnologija

Repository / Repozitorij:

Repository of the University of Rijeka, Faculty of Informatics and Digital Technologies - INFORI Repository

zir.nsk.hr

DIGITALNA KNJIŽNICA

DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

Sveučilište u Rijeci, Fakultet informatike i digitalnih tehnologija

Sveučilišni prijediplomski studij Informatika

Marko Maruna

Izrada full stack aplikacije Studomat za prijavu ispita

Završni rad

Mentor: izv. prof. dr. sc. Marija Brkić Bakarić

Rijeka, 13.09.2023

FIDIT
Sveučilište u Rijeci
Fakultet informatike
i digitalnih tehnologija
www.inf.uniri.hr

Rijeka, 26.5.2023.

Zadatak za završni rad

Pristupnik: Marko Maruna

Naziv završnog rada: Izrada full stack aplikacije Studomat za prijavu ispita

Naziv završnog rada na engleskom jeziku: Development of the full-stack application Studomat for exam registration

Sadržaj zadatka: Cilj ovog rada je istražiti razvoj full stack aplikacija korištenjem Spring boot razvojnog okruženja za backend, te React.js razvojnog okruženja za frontend. U uvodnom dijelu dan je prikazan popularnih tehnologija za izradu višeplatformskih aplikacija, a potom je detaljnije prikazan razvojni okvir Spring boot, te React.js. U sklopu rada je osmišljena, razvijena i testirana aplikacija Studomat za prijavljivanje ispita te je prikazan proces izrade aplikacije. Na kraju rada izneseni su zaključci i preporuke za usvajanje i korištenje razvojnog okvira Spring boot, te razvojnog okvira React.js.

Mentor

Izv. prof. dr. sc. Marija Brkić Bakarić

Voditelj za završne radove

Doc. dr. sc. Miran Pobar

Zadatak preuzet: 26.5.2023.

(potpis pristupnika)

Adresa: Radmile Matejčić 2
51000 Rijeka, Hrvatska
Tel: +385(0)51 584 700
E-mail: ured@inf.uniri.hr
OIB: 64218323816
IBAN: HR1524020061400006966
UNIRI

Sažetak

Cilj ovog završnog rada je prikaz kako rade tehnologije React.js i Spring boot na temelju izrade full stack aplikacije za online prijavu ispita. Aplikacija je namijenjena za učitavanje i spremanje podataka o kolegijima i korisnicima te omogućuje pregled, uređivanje i brisanje korisnika, kolegija i testova. Unutar aplikacije implementirane su razne funkcionalnosti koje su prikazane i objašnjene u nastavku.

Ključne riječi: React.js, Spring Boot, MySql, Liquidbase, full stack, router, autentifikacija

Sadržaj

1. Uvod

U digitalno doba obrazovanja, razvoj inovativnih rješenja koje olakšavaju procese u obrazovnim institucijama postaje ključno. U srcu ove potrebe leži izazov kako omogućiti studentima da se jednostavno prijavljuju na ispite, profesorima da učinkovito upravljaju ispitima i ocjenjivanjem, te administratorima da učinkovito upravljaju korisničkim računima. Upravo tu ulogu preuzima ovaj projekt – aplikacija za online prijavu ispita koja koristi tehnologije React.js i Spring Boot.

Uz gore navedene tehnologije također za izradu projekta će se koristiti i MySql baza podataka kao i Liquidbase za verzioniranje iste, te razne React.js biblioteke.

Korišteni jezici i tehnologije:

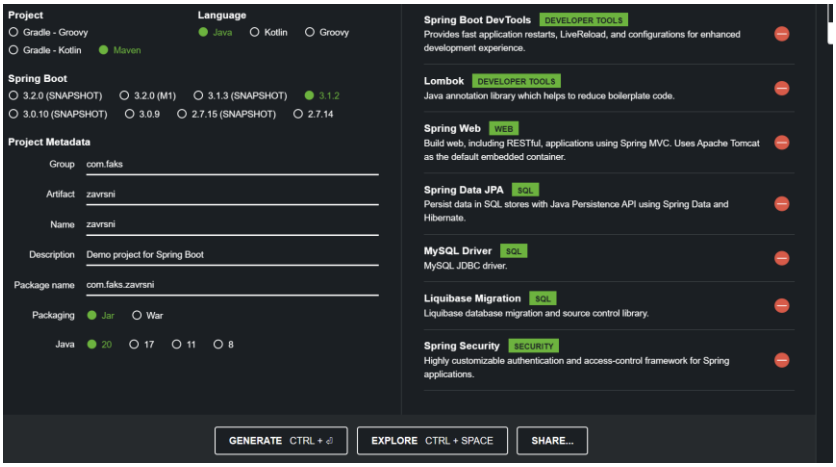
- React.js je vodeća open source javascript biblioteka izrađena od strane Facebook-a za izradu dinamičkih korisničkih sučelja. Njegova komponentna struktura čini ga savršenim izborom za izradu responzivnih i interaktivnih web aplikacija
- Spring Boot je open source, enterprise level Java okvir (*eng. Framewor*k) za izradu stabilnih i sigurnih Java aplikacija. Uzima osnovne koncepte Spring Frameworka i transformira ih u zrele konfiguracije, omogućavajući programerima da se više usmjere na izgradnju poslovne logike, a manje na poslovnu infrastrukturu.
- MySql je moćan open source relacijski sustav za upravljanje bazama podataka, često se koristi u izradi web aplikacija. Njegova sposobnost skladištenja i upravljanja strukturama podataka čini ga ključnim alatom za pohranu informacija u organizirane tablice i retke.
- Liquidbase je snažan open source alat za verzioniranje i upravljanje strukturom baze podataka. Jedna od glavnih prednosti Liquidbase-a je mogućnost definiranja promjena u bazi podataka kao „changeset“. Svaka promjena je dokumentirana i osigurava precizno praćenje evolucije baze podataka tijekom vremena.
- HTML (*eng. HyperText Markup Language*) je osnovni jezik za izradu struktura web stranica. On omogućava programerima da definiraju strukturu i sadržaj web stranice kroz korisnički čitljiv kod.

Koristio sam tehnologije React i Spring kako bih dublje razumio svijet modernog web razvoja. React i Spring činili su se kao odličan izbor za stvaranje responzivnih i skalabilnih web aplikacija.

2. Backend

2.1 Postavke projekta

Za početak rada u Spring Boot okruženju potrebno je inicijalizirati projekt pomoću Spring initializr web stranice te na samoj stranici odabrati jezik u kojem će se pisati projekt, *build* tool koji će se koristiti za izradu i upravljanje projektom, osnovne podatke o projektu kao što su grupa, artefakt i ime, te zavisnosti. Sve dosad spomenute opcije su prikazane na slici 1.



Slika 1. Inicijalizacija Spring Boot Projekta

2.1.1 Zavisnosti

Pri kreiranju Spring projekta odabrali smo nekoliko zavisnosti (engl. *dependency*) koji će nam biti potrebni za izradu backend-a ove aplikacije. Sve zavisnosti možete vidjeti na slici 1.

Spring Boot Dev Tools će nam služiti za automatski reload projekta tokom faze izrade radi lakšeg testiranja de će biti ugašen za finalnu verziju aplikacije.

Lombok nam služi za jednostavnije postavljanje entiteta u projektu, to jest da ne trebamo raditi gettere, settere i konstruktore, te nam još pruža razne mogućnosti koje ćemo dalje istražiti u nastavku projekta.

Spring Web služi za omogućavanje web funkcionalnosti projektu, to jest omogućava nam izradu endpoint-a na koje će frontend moći komunicirati sa backendom.

Spring Data JPA služi za omogućavanje funkcionalnosti za komunikaciju između backend-a i baze podataka, te ima mogućnost za samostalnu izradu upita (engl. *query*).

MySQL Driver daje aplikaciji osnovne mogućnosti za komunikaciju sa MySql bazom podataka.

Liquidbase Migration omogućava korištenje Liquidbase migracija u Spring projektu.

Spring Security nam daje mogućnost autentifikacije i autorizacije korisnika za korištenje određenih dijelova aplikacije.

2.1.2 Spring konfiguracija

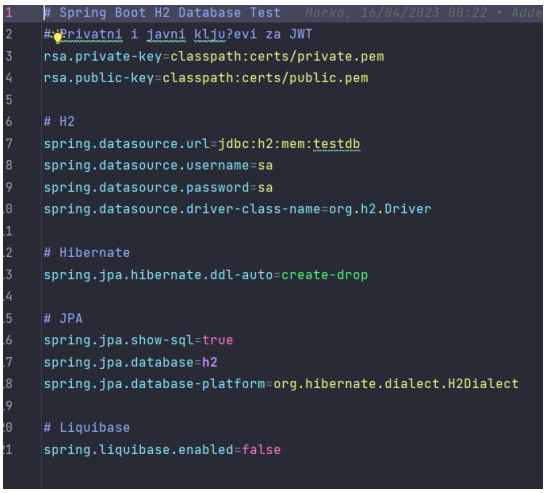
Na slici 2 možemo vidjeti konfiguraciju za naš Spring projekt. U konfiguraciji određujemo sve potrebne postavke za projekt, a u nastavku će biti objašnjene najvažnije:

- rsa.private-key – označava lokaciju privatnog ključa,
- rsa.public-key – označava lokaciju javnog ključa,
- spring.datasource.url – označava lokaciju baze podataka,
- spring.datasource.driver – označava na koju bazu podataka će se Spring spojiti,
- spring.jpa.hibernate.ddl-auto – označava kako će Spring postupati sa bazom podataka, mi koristimo *none* jer ne želimo da Spring sam stvara i izmjenjuje tablice,
- spring.jpa.database – označava na kojoj bazi podataka će JPA raditi,
- spring.jpa.database-platform – specificira dijalekt koji će Hibernate koristiti,
- spring.liquidbase.enabled – omogućuje rad Liquidbase-a u projektu,
- spring.devtools.restart.enabled – označava da li želimo automatski ponovno pokrenuti aplikaciju ako dođe do promjene.

```
# Privatni i javni ključevi za JWT   You, Yesterday • Uncommitted change
rsa.private-key=classpath:certs/private.pem
rsa.public-key=classpath:certs/public.pem

# MySQL
spring.datasource.url=jdbc:mysql://localhost:6612/zavrsni
spring.datasource.username=root
spring.datasource.password=password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Hibernate
spring.jpa.hibernate.ddl-auto=none

# JPA
spring.jpa.show-sql=true
spring.jpa.database=mysql
spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect

# Liquibase
spring.liquibase.enabled=true
spring.liquibase.change-log=classpath:db/changelog/changelog-master.xml
spring.liquibase.contexts=development, production

#Disable restarting
spring.devtools.restart.enabled = false
```

Slika 2. Spring standardna konfiguracija

Na slici 3 možemo vidjeti Spring konfiguraciju za testiranje aplikacije. Jedine promjene su te da je Liquidbase onemogućen, Hibernate je postavljen na *create-drop* što znači da će pri svakom paljenju sam stvoriti sve tablice u bazi podataka, te ih izbrisati tokom gašenja. Također za testiranje smo se prebacili sa MySql baze podataka na H2 bazu podataka jer je ona u memoriji i briše sve podatke nakon svakog restart-a što nam odgovara za testiranje.



Slika 3. Spring test konfiguracija

2.2 Izrada Entiteta

Na slici 4 možemo vidjeti jedan od entiteta u ovome projektu. Da bi klasa postala entitet iznad klase trebamo staviti oznaku @Entity, te u samoj klasi označiti koje polje iz baze podataka je primarni ključ sa oznakom @Id. Na polje id također smo dodali i oznaku @GeneratedValue koja govori Spring aplikaciji da će tokom kreiranja novog entiteta morati sam generirati novi id. Iznad same klase smo dodali oznake @AllArgsConstructor, @NoArgsConstructor, @Getter, @Setter i @Builder. Te oznake nam pomažu tokom kreiranja same klase na način da one umjesto nas kreiraju sve potrebne konstruktore, gettere i settere. Iznad svakog polja koristili smo oznaku @Column koja nam omogućuje da postavimo ime polja i njegova ograničenja iz baze podataka.

Slika 4 Izgled Entiteta

2.2.1 Povezivanje Entiteta

Da bi definirali relacije između entiteta moramo koristiti anotacije @OneToOne, @OneToMany, @ManyToOne i @ManyToMany. Ove anotacije definiraju kako su određeni entiteti povezani u bazi podataka.

Na slici 5 možemo vidjeti da je entitet User povezan sa entitetom UserProfile pomoću relacije @OneToOne što bi značilo da svaki User će biti povezan sa samo jednim UserProfilom. Unutar anotacije @OneToOne koristimo polje mappedBy kako bi Spring znao pomoću kojeg polja u bazi podataka će povezati ova dva entiteta, te koristimo polje *cascade* da označimo na koji način se brišu entiteti. *Cascade* ALL nam govori da će se u slučaju brisanja entiteta User automatski obrisati i UserProfile entitet koji je povezan s tim entitetom. Entitet User također je povezan sa entitetom Subject pomoću relacije @ManyToMany što bi značilo da više entiteta User može biti povezano na više entiteta Subject.

```
@OneToOne(
        mappedBy = "user",
        cascade = CascadeType.ALL
)
private UserProfile userProfile;
@ManyToMany(
        mappedBy = "students",
        cascade = CascadeType.DETACH
)
private List<Subject> subjects;
```

Slika 5 Veze između entiteta

2.2.2 Izrada tablice u bazi podataka

Za izradu tablica u bazi podataka koristimo Liquidbase. Na slici 6 možemo vidjeti primjer izrade tablice za entitet User. Za svaku izmjenu u bazi podataka moramo kreirati novi changeSet te unutar changeSet-a unijeti svoje promjene. Za izradu nove tablice koristimo oznaku createTable te u polje tableName unosimo ime tablice. Svaki stupac u toj tablici označavamo sa oznakom *Column* te unutar te oznake dajemo ime tog stupca te vrstu podatka za taj stupac. Ako za primjer uzmemo stupac *id* za vrstu smo stavili BIGINT te smo stavili oznaku autoIncrement koja omogućava bazi podataka da tokom izrade svakog novog retka stupac *id* podigne za jedan broj. Također za svaki stupac možemo postaviti neka ograničenja na način da unutar oznake column stavimo oznaku constraints te unutar nje stavimo sva ograničenja za taj stupac. Za stupac id postavili smo ograničenja nullable što označava da taj stupac ne smije biti null te smo postavili ograničenja koja govore da će taj stupac biti primarni ključ te tablice, te smo dali ime tom primarnom ključu u našoj tablici.

Jednom kada pokrenemo Liquidbase skriptu ona će provjeriti dali postoji novi changeSet te ga izvršiti, nakon izvršavanja changeSet-a Liquidbase zaključava taj changeSet te ne dopušta nekakve promjene na njemu.

```xml
<changeSet id="user" author="Marko Maruna">
    <createTable tableName="users">
        <column name="id" type="BIGINT" autoIncrement="true">
            <constraints primaryKey="true" nullable="false" primaryKeyName="pk_user_id"/>
        </column>
        <column name="username" type="VARCHAR(255)">
            <constraints nullable="false" unique="true"/>
        </column>
        <column name="password" type="VARCHAR(255)">
            <constraints nullable="false"/>
        </column>
        <column name="role" type="VARCHAR(255)">
            <constraints nullable="false"/>
        </column>
        <column name="enabled" type="BOOLEAN">
            <constraints nullable="false"/>
        </column>
    </createTable>
```

Slika 6 Izrada tablice u pomoću Liquidbase-a

2.2.3 Povezivanje entiteta sa bazom podataka

Kako bi naša aplikacija mogla vršiti promjene u bazi podataka za neki entitet potrebno je izraditi repositorij za taj entitet. Na slici 7 možemo vidjeti primjer izrade repozitorija. Kako bi kreirali repozitorij za entitet User, prvo kreiramo novo sučelje koji će nasljeđivati JpaRepository. Unutar JpaRepository zagrada ćemo označiti vrstu entitet i vrstu ključa za taj entitet. U našem slučaju to će biti entitet User, a ključ će biti tipa Long. Kako bi Spring znao da se ovo sučelje koristi kao repozitorij moramo iznad samog sučelja staviti anotaciju @Repository. JpaRepository sam kreira osnovne CRUD funkcije za manipulaciju baze podataka. Te funkcije su *create*, *read*, *update* i *delete*. U slučaju da nam trebaju neke složenije funkcije možemo ih dodati u ovo sučelje. Za neke od tih funkcija JpaRepository će automatski kreirati sql upit na način da će iz imena funkcije zaključiti što mi želimo, dok za funkcije koje neće biti automatski generirane sami moramo napisati sql upit na način da iznad funkcije stavimo anotaciju @Query te u polje value napišemo naš upit.

```java
@Repository
public interface UserRepository extends JpaRepository<User, Long> {   Marko,
    7 usages  Marko
    Optional<User> findByUsername(String username);

    2 usages  mmaruna
    List<User> findAllByUsernameContaining(String username);

    2 usages  mmaruna
    List<User> findAllByRole(Role role);

    1 usage  mmaruna
    @Modifying
    @Transactional
    @Query(value = "DELETE FROM user_subject where user_subject.user_id=?1",
            nativeQuery = true)
    void deleteConnections(Long id);
}
```

Slika 7 User repository

2.3 Implementacija JWT sigurnosti

U ovoj aplikaciji implementirali smo jwt (JSON web token) sigurnost kako bismo osigurali autentifikaciju i autorizaciju korisnika prilikom pristupa različitim dijelovima aplikacije.

2.3.1 Postavljanje sigurnosne konfiguracije

Na slici 8 možemo vidjeti osnovne dijelove sigurnosne konfiguracije u našoj aplikaciji. Iznad same konfiguracije postavljamo anotacije @Configuration i @EnableWebSecurity. Te anotacije govore Spring-u da je ova klasa konfiguracija te da pokrene sigurnosne dijelove naše aplikacije. U ovoj klasi također označavamo da će se za enkripciju lozinki koristiti BCryptPasswordEncoder te način na koji ćemo kodirati i dekodirati naše jwt tokene. U ovome slučaju za to smo koristili set rsa ključeva.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    4 usages
    private final RsaKeys rsaKeys;

    Marko
    public SecurityConfig(RsaKeys rsaKeys) { this.rsaKeys = rsaKeys; }

    Marko
    @Bean
    public BCryptPasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); }
    Marko, 09/04/2023 19:02 • Added jwt token and basic token services
    Marko +1
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {...}

    Marko
    @Bean
    JwtDecoder jwtDecoder() { return NimbusJwtDecoder.withPublicKey(rsaKeys.publicKey()).build(); }

    Marko
    @Bean
    JwtEncoder jwtEncoder() {
        JWK jwk = new RSAKey.Builder(rsaKeys.publicKey()).privateKey(rsaKeys.privateKey()).build();
        JWKSource<SecurityContext> jwks = new ImmutableJWKSet<>(new JWKSet(jwk));
        return new NimbusJwtEncoder(jwks);
    }
}
```

Slika 8 SecurityConfig

Na slici 9 je prikazan glavni dio sigurnosne konfiguracije. SecurityFilterChain predstavlja lanac sigurnosnih filtera koji se primijenjuju na dolazne zahtjeve kako bi se primijenila pravila sigurnosti. Svaki filter u lancu odgovoran je za određeni aspekt sigurnosti, kao što su autentifikacija, autorizacija, zaštita od CSRF-a, itd.

U ovome filteru označavamo da omogućujemo cross-origins resorce sharing, onemogućili smo zaštitu za CSRF, u oznaci authorizeHttpRequests označavamo koji korisnik može pristupiti kojem dijelu aplikacije, sessionManagement smo postavili na STETELESS što znači da aplikacija neće imati sesije već će se svi potrebni podatci dobivati iz jwt tokena, za autentifikaciju na stranici smo omogućili jwt te http basic pomoću oznaka oatuh2ResourceServer te httpBasic.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    return http
            .cors() CorsConfigurer<HttpSecurity>
            .and() HttpSecurity
            .csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests(auth -> auth
                    .requestMatchers("/api/auth/**") AuthorizedUrl
                        .permitAll() AuthorizationManagerRequestMat...
                    .requestMatchers("/api/file/download/**") AuthorizedUrl
                        .permitAll() AuthorizationManagerRequestMat...
                    .requestMatchers("/api/subject/teacher/**") AuthorizedUrl
                        .hasAnyAuthority( ...authorities: "SCOPE_ADMIN", "SCOPE_TEACHER") AuthorizationMa
                    .requestMatchers("/api/test/teacher/**") AuthorizedUrl
                        .hasAnyAuthority( ...authorities: "SCOPE_ADMIN", "SCOPE_TEACHER") AuthorizationMa
                    .requestMatchers("/api/admin/**") AuthorizedUrl
                        .hasAuthority("SCOPE_ADMIN") AuthorizationManagerRequestMat...      You, 10/08/2
                    .anyRequest().authenticated()
            )
            .sessionManagement(session -> session
                    .sessionCreationPolicy(SessionCreationPolicy.STATELESS) // Disable session
            )
            .oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt) // Enable JWT
            .httpBasic(Customizer.withDefaults()) // Enable basic authentication
            .build();
}
```

Slika 9 securityFilterChain

2.3.2 JWT token servis

Kako bi smo mogli generirati te dohvaćati podatke iz jwt tokena moramo napraviti servis koji će sadržavati navedene funkcionalnosti. Na slici 10 prikazane su funkcije unutar Token servisa.

Funkcija generateToken će nam služiti za izradu tokena kada se korisnik uspješno prijavi. Ova funkcija će uzeti trenutno vrijeme te korisničko ime i ulogu korisnika, te će potom sa tim podatcima napraviti novi jwt token koji će vrijediti 12 sati.

Funkcija getUsernameFromToken će nam služiti za dohvaćanje korisničkog imena iz tokena. Ova funkcija će se koristiti u većini drugih servisa.

```
2 usages  Marko *
public String generateToken(Authentication authentication) {
    Instant now = Instant.now();

    String scope = authentication.getAuthorities().stream() Stream<capture of extends G
            .map(GrantedAuthority::getAuthority) Stream<String>
            .collect(Collectors.joining());

    JwtClaimsSet claims = JwtClaimsSet.builder()
            .issuer("self")
            .issuedAt(now)
            .expiresAt(now.plus( amountToAdd: 12, ChronoUnit.HOURS))
            .subject(authentication.getName())
            .claim( name: "scope", scope)
            .build();

    return this.encoder.encode(JwtEncoderParameters.from(claims)).getTokenValue();
}

3 usages  Marko *
public String getUsernameFromToken(String bearer) {
    String token = bearer.substring( beginIndex: 7);
    return String.valueOf(this.decoder.decode(token).getSubject());
}
```

Slika 10 Token servis

2.3.3 Http basic autentikacija

Prije nego li korisniku dodijelimo jwt token moramo potvrditi da taj korisnik ima račun i to radimo na način da ga autenticiramo pomoću http basic autentifikacije. Nakon što je korisnik uspješno autentificiran, dodjeljujemo mu jwt token.

Da bi smo mogli koristiti httpBasic način autentifikacije, Spring treba znati iz kojeg će entiteta uzimati korisničko ime i šifru za autentifikaciju. Na slici 11 je prikazano da entitet User implementira sučelje UserDetails kako bi Spring mogao upravljati autentifikacijom i autorizacijom korisnika. Sučelje UserDetails definira potrebne informacije o korisniku koje Spring koristi kako bi provodio sigurnosne funkcije.

```
public class User implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;
    @Column(
            name = "username",
            unique = true,
            nullable = false
    )
    private String username;
    @Column(
            name = "password",
            nullable = false
    )
```

Slika 11 Http basic entitet

Također servis od entiteta treba implementirati UserDetailsService te unutar toga servisa trebamo Springu dati funkciju za pronalazak korisnika pomoću korisničkog imena. Izgled te funkcije možete vidjeti na slici 12.

```
5 usages  Marko *
@Override    Marko, 09/04/2023 19:02 • Added jwt token and basic token services
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    return userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException("User not found"));
}
```

Slika 12 funkcija loadUserByUsername

2.4 Spring kontroleri

Spring kontroleri su ključni dijelovi svake Spring aplikacije, posebno web aplikacija. Oni služe kao posrednici između korisničkog zahtjeva i poslovne logike. Glavna svrha kontrolera je obrađivati dolazne HTTP zahtjeve, izvoditi potrebne operacije te generirati i vraćati odgovarajuće HTTP odgovore. Kontroleri omogućuju da aplikacija reagira na korisničke zahtjeve, pružajući željene informacije ili akcije.

Kako bi smo izradili vlastiti Spring kontroler prvo kreiramo klasu iznad koje stavljamo @RestController, @CrossOrigin te @RequestMapping anotacije. Te anotacije govore Springu da će sljedeća klasa biti rest kontroler, te da koristi cross-origin header tokom slanja odgovora. RequestMapping nam omogućuje da postavimo url na kojem će biti dostupan kontroler, u ovome primjeru to bi bio url /api/auth.

Unutar klase kontrolera kreiramo funkcije koje će izvršavati određene radnje. Na slici 13 možemo vidjeti funkciju *signup*, ta funkcija iznad sebe sadrži anotaciju @PostMapping što govori Springu da će se na url /api/auth/signup moći slati POST zahtjev. Također na slici možemo vidjeti da ova funkcija prima SignupDTO vrstu podataka, te da će se ti podatci nalaziti u tijelu zahtjeva što se može vidjeti iz anotacije @RequestBody. Nakon što Spring potvrdi da je dobio ispravne podatke on će te podatke poslati na servisni dio aplikacije te korisniku vratiti odgovor u obliku ResponseEntity-a.

```
@RestController
@CrossOrigin
@RequestMapping("/api/auth")
public class Aut   Fetching Documentation...
    Marko, 11/04/...  ...dpoints
    3 usages
    private final UserService userService;

    Marko
    public AuthController(UserService userService) { this.userService = userService; }

    Marko +1
    @GetMapping("/login")
    public ResponseEntity<String> login(Authentication authentication) {
        return ResponseEntity.ok(userService.login(authentication));
    }

    mmaruna
    @PostMapping("/signup")
    public ResponseEntity<String> signup(@Validated @RequestBody SignupDTO data) throws CostumeErrorException {
        return ResponseEntity.ok(userService.signup(data));
    }

}
```

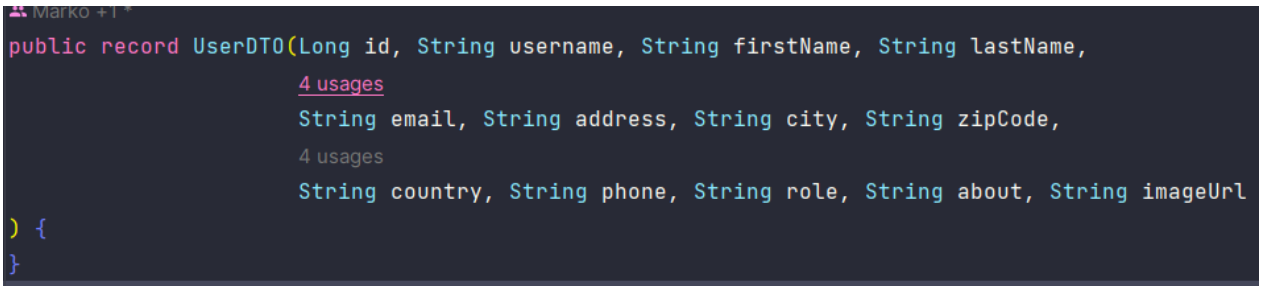
Slika 13 Spring kontroler

2.5 Upotreba DTO-ova za efikasni prijenosa podataka

Tokom razvoja aplikacije imali smo potrebu za preciznim upravljanjem prijenosom podataka između različitih slojeva aplikacije. Kako bi se postigla bolja organizacija i kontrola komunikacije između slojeva, odlučili smo primijeniti koncept objekta za prijenos podataka (engl. DTO).

Koncept DTO-ova smo koristili kako bi mogli ograničavati količinu podataka između slojeva backend-a i frontend-a. U nekim slučajevima bili su nam potrebni samo neki atributi entiteta da bi generirali odgovor za frontend. Na primjer ako neki korisnik zatraži podatke o drugom korisniku mi mu ne želimo poslati entitet traženog korisnika jer bi taj entitet sadržavao osjetljive informacije kao npr. šifru, već bismo kreirali novi entitet sa manjom količinom atributa koji ne sadrži osjetljive informacije, te taj isti poslali kao odgovor. Taj način rada pridonosi efikasnosti prijenosa podataka preko mreže te ubrzava odaziv aplikacije.

Na slici 14 možemo vidjeti DTO objekt koji aplikacija šalje kada frontend zatraži podatke o nekom korisniku. Ovaj DTO sadrži podatke iz entiteta User ali i entiteta UserProfile te nam pomaže u smanjenju korištenja mreže na način da umjesto da backend vraća 2 odgovora na poziv, jedan za User entitet i drugi za UserProfile, vraća samo jedan koji sadrži sve potrebne podatke.



Slika 14 UserDTO

2.5.1 Mapiranje DTO objekata

U razvoju naše Spring aplikacije jedna od ključnih komponenti koja je definirala efikasnost i preciznost prijenosa podataka između različitih slojeva jest upotreba „mappera“. Mapper je posebna komponenta koja služi za mapiranje podataka iz jednog objekta u drugi. Mi ćemo ih koristiti za popunjavanje DTO objekata iz entiteta.

Na slici 15 je prikazan *mapper* koji nam služi za mapiranje UserDTO objekta iz objekta User. Unutar metode *apply()* koristili smo podatke iz entiteta User kako biste stvorili objekt UserDTO. Svaki atribut iz entiteta koristili smo kako bismo postavili odgovarajući atribut u DTO objektu. Na ovaj način se preslikavaju podaci iz entiteta u DTO, s tim da možemo birati koje atribute želimo prenijeti.

```java
@Service
public class UserDtoMapper implements Function<User, UserDTO> {
    Marko +1
    @Override
    public UserDTO apply(User user) {
        return new UserDTO(
                user.getId(),
                user.getUsername(),
                user.getUserProfile().getFirstName(),
                user.getUserProfile().getLastName(),
                user.getUserProfile().getEmail(),
                user.getUserProfile().getAddress(),
                user.getUserProfile().getCity(),
                user.getUserProfile().getZipCode(),
                user.getUserProfile().getCountry(),
                user.getUserProfile().getPhoneNumber(),
                user.getRole().name(),
                user.getUserProfile().getAboutMe(),
                user.getUserProfile().getImageUrl()
        );
    }
```

Slika 15 UserDTO mapper

2.6 Servisni sloj aplikacije

U arhitekturi aplikacije, servisni sloj predstavlja ključnu komponentu koja se koristi za organizaciju servisne logike i koordinaciju između različitih dijelova aplikacije. Servisni sloj služi kao posrednik između sloja repositorijay i kontroler sloja aplikacije, osiguravajući da se servisna logika izvršava na pravilan i strukturiran način.

Zašto koristimo servisni sloj?

1. **Razdvajanje odgovornosti**: Servisni sloj omogućava jasno odvajanje servisne logike od ostatka aplikacije. Ovo čini našu aplikaciju modularnom i olakšava promjene u servisnoj logici bez utjecaja na druge dijelove sustava.
2. **Čitljivost i održavanje**: Servisna logika aplikacije može biti kompleksna, i stvaranje svega u jednom sloju može dovesti do teško održivog i loše čitljivog koda. Servisni sloj omogućava organiziranje servisne logike u manje, specifične usluge koje su lakše čitljive, testirane i održive.

3. **Višekratna uporaba**: Servisni sloj omogućava da se ista poslovna logika koristi na različitim mjestima unutar aplikacije, bez potrebe za dupliciranjem koda. Ovo povećava ponovno korištenje i smanjuje mogućnost pogrešaka.

U nastavku na slici 16 je prikazana *signup* funkcija iz UserService klase. Kao što možemo vidjeti u servisnom sloju se odrađuje sva logika aplikacije. Ova funkcija sa kontroler sloja dobiva SignupDTO koji sadrži *username*, *password* i *passwordConfirmation* atribute, te nakon primanja tih podataka naš servisni sloj provjerava dali se *password* atributi podudaraju, te da li *username* već postoji. Ako je sve u redu onda naš servisni sloj kreira novi entitet User na način da pozove signupDtoMapper koji će mapirati podatke iz SignupDTO na entitet User, nakon što se to obavi naš servisni sloj poziva sloj repositorija kako bi novi entitet spremio u bazu podataka, te ga poveže sa entitetom UserProfile. U slučaju da se *password* ne podudara ili je *username* već iskorišten naš servisni sloj vraća Response sa kodom 404 te odgovarajućom porukom.

```
@Override
public String signup(SignupDTO data) throws CostumeErrorException {
    if (data.password().equals(data.passwordConfirmation())) {
        if (userRepository.findByUsername(data.username()).isEmpty()) {
            User user = signupDtoMapper.apply(data);
            UserProfile userProfile = new UserProfile();

            userRepository.save(user);

            userProfile.setUser(user);

            userProfileService.saveProfile(userProfile);

            return "User created";
        }
        throw new CostumeErrorException("Username already exists", HttpStatus.BAD_REQUEST);
    }

    throw new CostumeErrorException("Password's don't match", HttpStatus.BAD_REQUEST);
}
```

Slika 16 Signup funkcija

2.7 Testiranje – Osiguravanje kvalitete

U razvoju aplikacije, implementacija testiranja postala je ključna komponenta za osiguranje kvalitete, pouzdanosti i stabilnosti softvera. Testiranje se koristi kako bi se provjerilo da li aplikacija ispravno funkcionira, da li zadovoljava zadane funkcionalnosti te da li se ponaša dosljedno u različitim situacijama.

Postoji nekoliko ključnih razloga zašto koristiti testiranje u izradi aplikacije:

1. **Identificiranje i ispravljanje pogrešaka**: Testiranje omogućava otkrivanje potencijalnih grešaka ili nesavršenosti u kodu prije nego što aplikacija dođe u ruke korisnika. To pomaže u sprječavanju neželjenih situacija poput rušenja aplikacije ili neispravnog funkcioniranja.
2. **Održavanje kvalitete koda**: Redovito testiranje osigurava da kvaliteta koda ostane visoka tijekom vremena. Nova ažuriranja i promjene u kodu mogu utjecati na postojeći funkcionalni dio aplikacije. Testiranje omogućava provjeru da li su svi dijelovi aplikacije i dalje funkcionalni nakon promjena.
3. **Sigurnost i stabilnost**: Testiranje doprinosi sigurnosti i stabilnosti aplikacije. Aplikacija koja je dobro testirana ima manju vjerojatnost da će imati ozbiljne sigurnosne propuste ili neočekivane padove.

2.7.1 Kreiranje testova

Za testiranje aplikacije koristili smo jedinične (engl. *unit*) testove, oni ne testiraju cijelu aplikaciju od jednom već testiraju dio po dio na način da ako želimo testirati da li signup funkcija unutar servisnog sloja radi prvo ćemo sve ostale dijelove aplikacije „imitirati“ (engl. *mock*), to jest dijelovi koje ne testiramo se neće pozivati već ćemo mi postaviti odgovarajuće odgovore za te dijelove aplikacije. Na slici 17 možemo vidjeti da ćemo testirati UserService, a sve ostale dijelove imitiramo. To prepoznajemo po anotaciji iznad servisa.

```
@SpringBootTest
class UserServiceImplTest {

    @Autowired
    private UserService userService;
    @MockBean
    private UserGetService userGetService;
    @MockBean
    private UserRepository userRepository;
    @MockBean
    private TokenService tokenService;    mmaruna, 10
    @MockBean
    private SignupDtoMapper signupDtoMapper;
    @MockBean
    private UserProfileService userProfileService;
```

Slika 17 UserService test

Na slici 18 su prikazana 2 testa u UserService sloju. Prvi test će testirati da li signup funkcija radi očekivano kada pošaljemo sve valjane podatke. Sa anotacijom when().thenReturn() određujemo što će vraćati imitirane funkcije kada ih funkcija *signup* pozove, to moramo napraviti za svaki poziv unutar funkcije *signup*. Nakon što odredimo sve odgovore imitiranih funkcija, možemo pozvati funkciju koju testiramo. U dolje navedenom primjeru smo pozvali funkciju *assertDoesNotThrow()* koja provjerava da li funkcija unutar nje vraća neku iznimku, ako funkcija ništa ne vraća onda test vraća da je sve u redu.

U testu *signupFail()* provjeravamo da li će naša funkcija baciti iznimku kada su *password* i *passwordConfirmation* polja unutar SignupDTO-a različiti. Pošto naša *signup* funkcija prvo to provjerava ne trebamo ništa imitirati već samo pozvati funkciju *assertThrows()* koja će provjeriti da li naša funkcija baca određenu iznimku.

```
mmaruna
@Test
@DisplayName("Test signup - success")
void signup() {
    when(userRepository.findByUsername("username")).thenReturn( t: Optional.empty());
    when(signupDtoMapper.apply(SignupDtoUtil.SignupDTO())).thenReturn(UserUtil.generate());
    when(userRepository.save(any())).thenReturn(UserUtil.generate());
    when(userProfileService.saveProfile(any())).thenReturn(UserUtil.generate().getUserProfile());
    assertDoesNotThrow(() -> userService.signup(SignupDtoUtil.SignupDTO()));
}

mmaruna
@Test
@DisplayName("Test signup - fail password mismatch")
void signupFail() {
    assertThrows(CostumeErrorException.class, () -> userService.signup(SignupDtoUtil.SignupDTObad()));
}   mmaruna, 10/08/2023 20:53 • Added userService/userGetServece tests
```

Slika 18 Testiranje funkcija

U gore navedenim testovima često pozivamo Util funkcije. One nam služe kako bi smanjili količinu koda u svakom testu, umjesto da uvijek ručno pišemo entitete koje šaljemo, Util klase u sebi imaju već predefinirane entitete koje one generiraju. Na slici 19 možemo vidjeti SignupDtoUtil klasu koja za nas može generirati dvije vrste SignupDTO-a. Jedna je ispravna, a druga ima različite passworde.

```
public final class SignupDtoUtil {
    3 usages  mmaruna *
    public static SignupDTO SignupDTO() {
        return new SignupDTO(
                username: "username",
                password: "password",
                passwordConfirmation: "password");
    }

    1 usage  mmaruna *
    public static SignupDTO SignupDTObad() {
        return new SignupDTO(
                username: "username",
                password: "password",
                passwordConfirmation: "password1");
    }
}
```

Slika 19 Testing utils

2.7.2 Konfiguracija za testove

Kada testiramo aplikaciju ne želimo da se događanja u testovima pojave u glavnoj bazi podataka. Na primjer ako testiramo spremišni sloj ne želimo da se ti testovi rade na produkcijskoj bazi podataka, zato kada testiramo aplikaciju Springu dodjeljujemo novi konfiguracijski podatak. Na slici 20 možemo vidjeti novi konfiguracijski podatak koji mijenja bazu podataka sa MySql-a na H2 bazu podataka koja će biti u memoriji, te gasimo Liquidbase i dopuštamo Springu da testnu bazu podataka sam ispuni tablicama. To radimo na način da opciju spring.jpa.hibernate.ddl-auto postavimo na create-drop što znači da će prilikom svakog pokretanja Spring napraviti tablice te ih tokom gašenja testova izbrisati.

```
# Spring Boot H2 Database Test   Marko, 16/04/2023 00:22 • Add
rsa.private-key=classpath:certs/private.pem
rsa.public-key=classpath:certs/public.pem

# H2
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.username=sa
spring.datasource.password=sa
spring.datasource.driver-class-name=org.h2.Driver

# Hibernate
spring.jpa.hibernate.ddl-auto=create-drop

# JPA
spring.jpa.show-sql=true
spring.jpa.database=h2
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

# Liquibase
spring.liquibase.enabled=false
```

Slika 20 Spring konfiguracija za testove

2.8 Izrada MySql Baze podataka

Za izradu baze podataka u našoj aplikaciji koristili smo Docker. Docker je alat koji omogućava izoliranje aplikacija i njihovih komponenti u "kontejnere", što je pristup koji donosi brojne prednosti za razvoj, testiranje i distribuciju aplikacije.

2.8.1 Docker compose file

Na slici 21 možemo vidjeti primjer izrade MySql baze podataka pomoću docker-compose podatka, u nastavku će biti objašnjenje svih konfiguracija:

- Version: verzija docker-compose filea
- Services: svi servisi koji će biti pokrenuti s ovim file-om
- Db: ime usluge za MySql bazu podataka
- Container_name: naziv kontejnera koji će biti stvoren za MySql bazu podataka
- Restart: postavlja način ponovnog pokretanja kontejnera nakon što se zaustavi, osim ako ga korisnik namjerno zaustavi.
- Image: slika koju ćemo koristiti kako temelj ovog kontejnera
- Command: komande koje će se izvršiti tokom pokretanja kontejnera
- Environment: postavlja okruženje za MySQL kontejner, uključujući administratorsku lozinku i ime baze podataka.
- Ports: Definira preslikavanje porta između host stroja i kontejnera (6612:3306). Ovo omogućava pristup MySQL bazi putem porta 6612 na hostu
- Volumes: Povezuje volumen na hostu s volumenom unutar kontejnera, omogućavajući trajno pohranjivanje podataka iz MySQL baze

```yaml
version: '2'
services:
  db:
    container_name: zavrsni-db
    restart: unless-stopped
    image: mysql:5.7
    command: --innodb_use_native_aio=0
    environment:
      MYSQL_ROOT_PASSWORD: 'password'
      MYSQL_DATABASE: 'zavrsni'
    ports:
      - 6612:3306
    volumes:
      - ./db:/var/lib/mysql
```

Slika 21 MySql docker compose

3. Frontend

U ovom dijelu rada, usmjereni smo na detaljnije razmatranje ključnih komponenti unutar naše React aplikacije, bez dubinskog ulaska u kod. Fokus će biti na razumijevanju funkcionalnosti i svrhe svake komponente, kao i načina na koji te komponente doprinose cjelokupnom korisničkom iskustvu. Razmotrit ćemo što svaka komponenta omogućava korisnicima, kako se koristi, i kako se uklapa u šire okruženje aplikacije.

3.1 Postavljanje okruženja

Za postavljenje projekta koristili smo „create-react-app“. Na slici 22 možemo vidjeti korištenu komandu, ona će nam kreirati osnovnu react aplikaciju po imenu zavrsni-client u trenutnom folder-u.

```
PS C:\Users\mmaruna\WebstormProjects\zavrsni-client> npx create-react-app zavrsni-client .
```

Slika 22 kreiranje react aplikacije

Nakon kreiranja aplikacije trebamo dodati sve potrebne zavisnosti kao što su react-router, react-cookie, axios te material-ui. To vršimo pomoću naredbe na slici 23.

```
npm install @mui/material @emotion/react @emotion/styled
react-router-dom localforage match-sorter sort-by axios react-cookie
```

Slika 23 Instaliranje dependancy-a

3.2 Razdvajanje aplikacije na prijavljeni i neprijavljeni dio

Da bismo osigurali da samo prijavljeni korisnici mogu pristupiti određenim stranicama naše aplikacije, koristili smo koncept zaštićenih ruta. Ovaj pristup osigurava da samo korisnici koji su autentificirani (prijavljeni) imaju pristup osjetljivim informacijama i funkcijama.

3.2.1 Provjera autentifikacije kroz kolačiće

Za implementaciju ovog mehanizma autentifikacije, koristili smo HTTP kolačiće (cookies). Kada korisnik uspješno izvrši prijavu, server šalje kolačić koji sadrži informacije o korisniku. Naša zaštićena ruta zatim provjerava postojanje ovog kolačića kako bi odredila je li korisnik prijavljen ili ne.

Na slici 24 možemo vidjeti komponentu koja provjerava dali je korisnik prijavljen u našu aplikaciju na način da provjeri da li korisnik ima postavljen kolačić. Ako korisnik ne posjeduje taj kolačić aplikacija će ga preusmjeriti na portal za prijavu, a u suprotnom će mu prikazati stranicu kojoj želi pristupiti.

```
2 usages   mmaruna
export const ProtectedRoute = () => {
    const [cookie : {access_token?: any} ] = useCookies( dependencies: ['access_token']);

    return cookie.access_token ? <Outlet /> : <Navigate to={"/auth/login/"} replace={true}/>;
}
```

Slika 24 Funkcija za zaštićene rute

3.2.2 Implementacija zaštićene rute

Za implementaciju zaštićenih ruta koristili smo react-router na način da postavimo ProtectedRoute kao element aplikacije koji se poziva prije svakog zaštićenog elementa, a rute koje služe za autentifikaciju korisnika smo odvojili u poseban dio koji nije u skupu zaštićenih ruta.

```
const router : Router = createBrowserRouter( routes: [
    {
        path: "/",
        element: <ProtectedRoute/>,
        errorElement: <Error/>,
        children: [
            {path: "/"...}
        ]
    },
    {
        path: "/auth",
        element: <Auth/>,
        errorElement: <Error/>,

        children: [...]
    }
]);

const root : Root = ReactDOM.createRoot(document.getElementById( elementId: 'root'));
root.render(
    <React.StrictMode>
        <CookiesProvider>
                <RouterProvider router={router}/>
        </CookiesProvider>
    </React.StrictMode>
);
```

Slika 25 Rute u aplikaciji

3.3 Komunikacija sa backendom kroz axios

Kako bismo omogućili našoj React frontend aplikaciji interakciju s backendom i dinamičko dohvaćanje te slanje podataka, koristili smo Axios, popularnu JavaScript biblioteku za upravljanje HTTP zahtjevima. Axios nam omogućava jednostavno izvođenje asinkronih HTTP zahtjeva prema backend serveru.

3.3.1 Inicijalizacija axios instanci

Na slici 26 možemo vidjeti inicijalizaciju axios instance. Ovako postavljamo globalnu konfiguraciju za sve zahtjeve, osnovni dio URL-a te povratne kodove koje smatramo valjanima. U ovome primjeru bazu URL-a uzimamo iz env varijable kako bi nam poslije bilo jednostavnije izgraditi finalnu verziju aplikacije te je pokrenuti u docker okruženju.

```
import axios from "axios";

5+ usages  mmaruna
export default axios.create({
    baseURL: `http://${process.env.REACT_APP_API_BASE_URL}:8080/api`,
    validateStatus: (status : number ) => {
        return status >= 200 && status < 500
    }
});
```

Slika 26 Inicijalizacija axios instance

3.3.2 Primjer axios GET/POST zahtjeva

Na slici 27 možemo vidjeti primjer GET zahtjeva. Zahtjev radimo na način da pozovemo Api.get te unutar funkcije unesemo url endpoint-a i potrebne headere za taj endpoint. Na ovoj slici smo pozivali *endpoint* te za header smo poslali „Authorization“ unutar kojega smo stavili *basic* autentifikaciju koja radi na principu da *username* i *password* upišemo u obliku base64, a za ostale get funkcije ćemo koristiti *bearer* token umjesto *basic* autentifikacije.

Slika 27 GET zahtjev

Na slici 28 prikazan je POST zahtjev. Jedina razlika između GET i POST zahtjeva je ta što u POST zahtjevu šaljemo podatke, te unutar headera moramo dodati i vrstu podataka koje šaljemo.

Slika 28 POST zahtjev

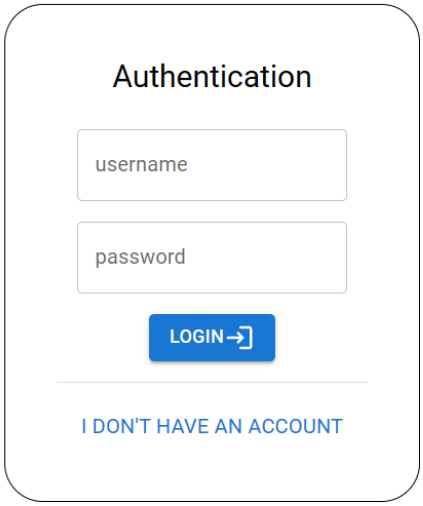
U ostatku projekta koristimo još neke vrste zahtjeva kao što su PUT i DELETE. Ti zahtjevi rade na istom principu kao POST i GET. PUT zahtjev izgleda identično kao POST, a DELETE kao GET.

3.4 Funkcionalnosti za studente

Ovaj odjeljak posvećen je komponentama i funkcionalnostima unutar naše aplikacije koje su posebno prilagođene studentima, ali važno je napomenuti da su dostupne za sve korisnike. Razmotrit ćemo kako ove komponente omogućavaju svim korisnicima bolje upravljanje svojim akademskim putem, pregled predmeta, sudjelovanje u testiranju i praćenje svog napretka.

3.4.1 Autentifikacija korisnika

Komponenta za prijavu (login) igra ključnu ulogu u autentifikaciji korisnika u našoj aplikaciji. Ova komponenta omogućava korisnicima da unesu svoje korisničko ime i lozinku kako bi pristupili zaštićenim dijelovima aplikacije. Izgled komponente možete vidjeti na slici 29.



Slika 29 Login komponenta

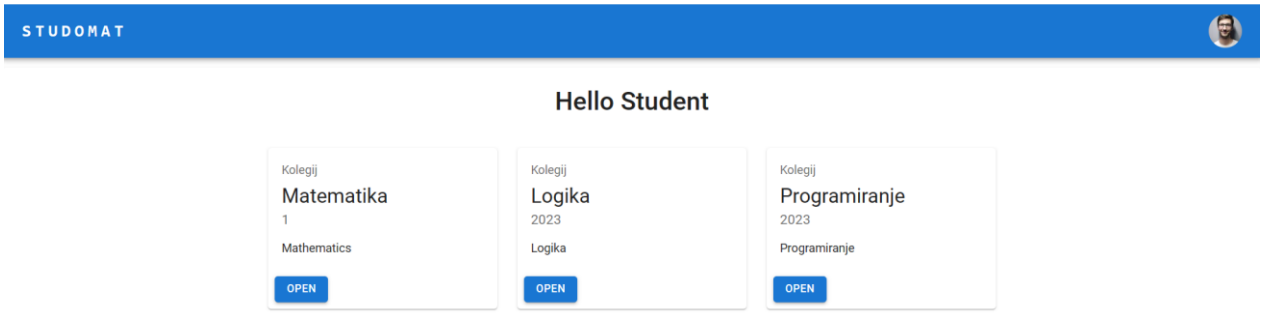
Na slici 27 možemo vidjeti najbitnije dijelove komponente *login*. Kada korisnik unosi svoje podatke oni se spremaju u varijable *username* i *password* te se pritiskom na gumb *Login* poziva funkcija *callLogin* koja te podatke pomoću axios-a šalje na backend. Ako je odgovor od backend-a pozitivan (kod 200), funkcija će odgovor spremiti u kolačić te korisnika preusmjeriti na početnu stranicu nakon čega će svi podatci o korisniku biti spremljeni u globalnom kontekstu, a u suprotnom korisniku će se ispisati poruka „Login failed“. Komponenta za registraciju ima istu funkcionalnost samo ima jedno polje više koje služi za potvrdu lozinke.

```
1 usage   mmaruna
const callLogin = async () : Promise<void>  => {
    const response : AxiosResponse<any>  = await login(username, password);

    if(response.status === 200) {
        setMessage( value: "");
        setCookies(
             name: "access_token",
            response.data,
             options: []);

        navigate("/")
    } else {
        setMessage( value: "Login failed.");
    }
}
```

Slika 30 Login funkcija

3.4.2 Glavna stranica aplikacije

Glavna stranica ("Home" – slika 31) predstavlja središnji prikaz za korisnike nakon prijave. Ova komponenta omogućava prikaz informacija o korisniku i predmetima koje je korisnik pohađao. Ako je korisnik administrator ili profesor na stranici će se prikazati dodatne mogućnosti za ulazak u Admin ili Teacher panel.



Slika 31 Home komponenta

Ova komponenta prikazuje dobrodošlicu korisniku s njegovim imenom na temelju podataka dostupnih u globalnom kontekstu. Također, komponenta dohvaća predmete koje korisnik pohađa putem funkcije getUserSubjects (slika 32), te rezultate sprema u varijablu subjects. Nakon što funkcija dohvati sve podatke varijablu loading postavlja na *false*.

```
const { user, } = useContext(UserContext);
const [ cookie : {access_token?: any} , , ] = useCookies( dependencies: ['access_token']);
const [ subjects : [] , setSubjects ] = useState( initialState: []);
const [ loading : boolean , setLoading ] = useState( initialState: true);

useEffect( effect: () : void => {
    getUserSubjects( cookie.access_token )
        .then((r : any | [] ) : void => {
            setSubjects(r);
            setLoading( value: false);
        }).catch((e) : void => {
            console.log(e);
        });
}, deps: [])
```

Slika 32 Home komponenta

Na slici 33 možemo vidjeti html glavne stranice i način na koji se ta stranica izrađuje. Pošto naša stanica ne može znati koliko predmeta korisnik polaže morali smo implementirati logiku za prikaz predmeta. Dok naša stranica ne učita korisnikove predmete prikazuje animirani krug koji predstavlja da je stranica u

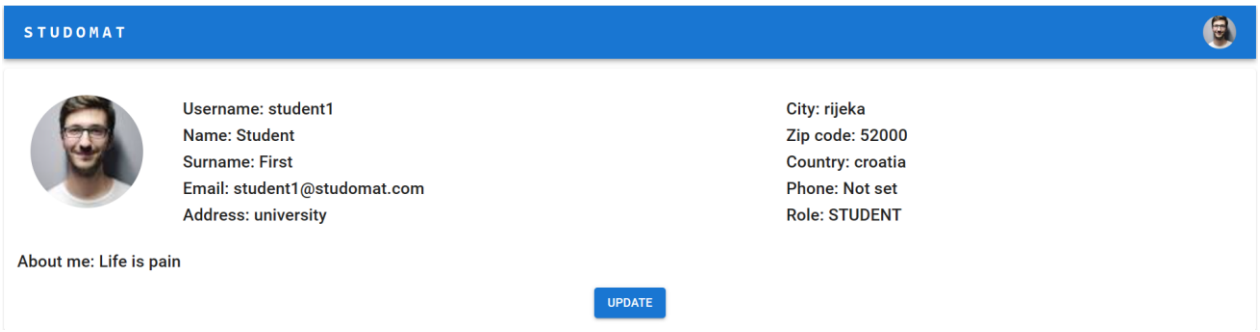
stanju pripreme. Nakon što stranica dobije odgovor sa backend servera, te se predmeti postave u varijablu subject tada stranica svaki predmet iz varijable subject mapira na komponentu SubjectCard. Na taj način smo uspjeli izraditi stranicu koja nema predefiniranu količinu komponenata na sebi, već se sama prilagođava količini predmeta svakog korisnika.

```
<Card sx={{my: 1.5}}>
    <Box sx={{display: {md: 'flex'}, flexDirection: {md: "column"}}}>
        <Box sx={{p: 2, mx: "auto"}}>
            <Typography variant="h6" sx={{fontSize: 30}}>
                Hello {user.firstName}
            </Typography>
        </Box>
        { !loading ?
            <>
                <Grid container spacing={3} columnSpacing={{ xs: 1, md: 3}} rows={{ xs: 1, sm: 3, md: 4 }}
                    alignItems="center" justifyContent="center" sx={{p: 2}}>
                    {subjects.map((subject) => (
                        <SubjectCard key={subject.id} subjects={subject} home={true}/>
                    ))}
                </Grid>
            </>
            :
            <Box justifyContent="center" justifyItems="center" sx={{p: 2}}>
                <CircularProgress />
            </Box>
        }
    </Box>
</Card>
```

Slika 33 html Home komponente

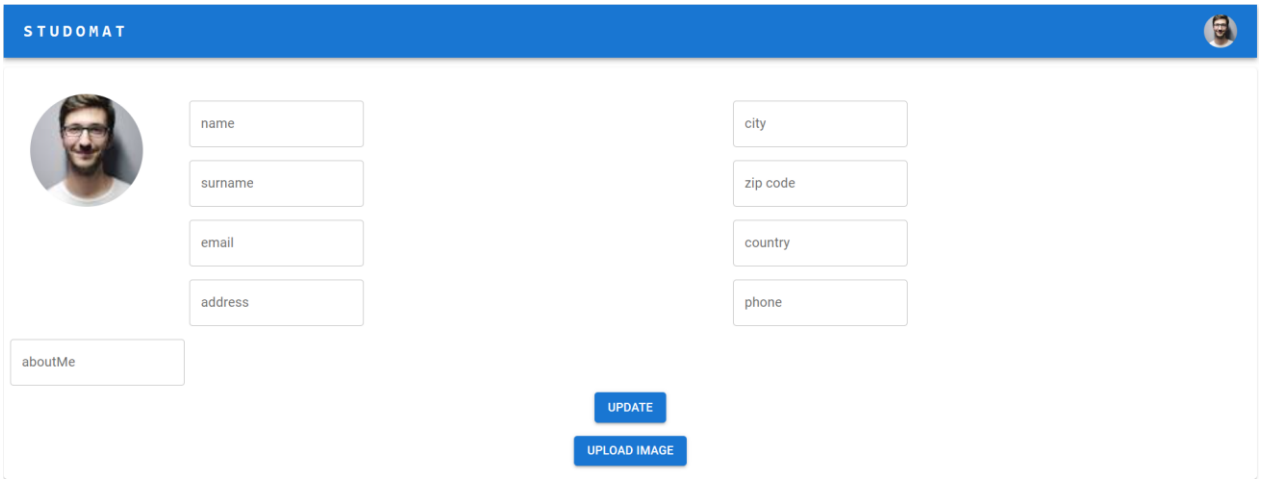
3.4.3 Korisnički profil

Komponenta "Profil" (slika 34) omogućava korisnicima da pregledaju svoje osobne podatke i informacije pohranjene u njihovom korisničkom profilu. Ova komponenta omogućava korisnicima da vide svoju sliku, korisničko ime, ime, prezime, e-mail, adresu i ostale detalje koji su uneseni prilikom registracije. Također, omogućava im da vide svoju ulogu unutar aplikacije.



Slika 34 Profil komponenta

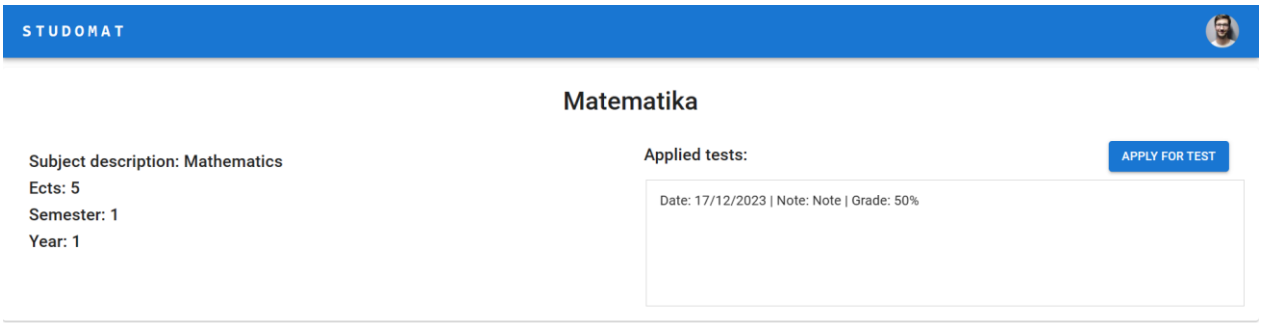
Korisnicima je omogućeno da ažuriraju svoje podatke klikom na gumb "Ažuriraj". Ovaj gumb pokreće navigaciju prema komponenti za ažuriranje profila (slika 35) gdje mogu unijeti nove ili izmijeniti postojeće informacije.



Slika 35 ProfileUpdate komponenta

3.4.4 Komponenta za predmet

Komponenta "MySubject" (slika 36) igra ključnu ulogu u našoj aplikaciji omogućavajući korisnicima pregled i upravljanje predmetima koje su upisali. Osim prikaza informacija o predmetu, ova komponenta omogućava korisnicima prijavu za ispitne rokove i pregled rezultata testiranja.



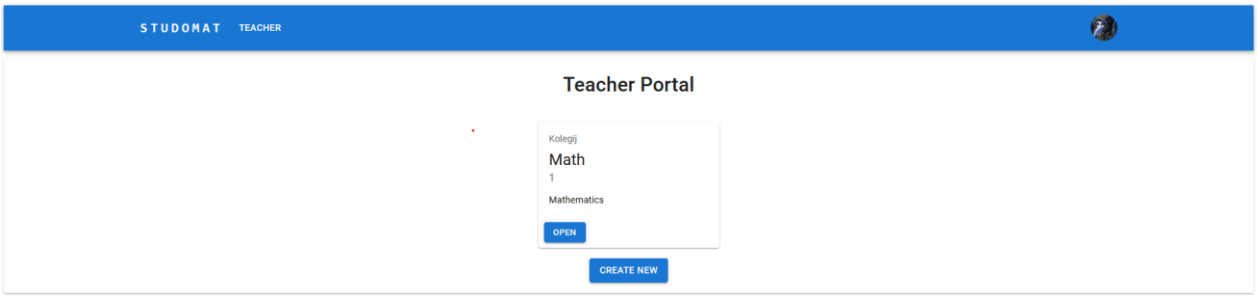
Slika 36 mySubject komponenta

3.5 Funkcionalnosti za Profesore

Ovaj odjeljak istražuje komponente i funkcionalnosti unutar naše aplikacije koje su posebno namijenjene profesorima, no važno je napomenuti da su također dostupne za korisnike s administratorskim ovlastima. Proučit ćemo kako ove komponente omogućuju profesorima bolje upravljanje i nadzor nad svojim akademskim resursima, predmetima i testovima.

3.5.1 Portal za profesore

Komponenta "TeacherPortal" (slika 37) predstavlja ključni dio naše aplikacije namijenjen profesorima. Ova komponenta omogućava profesorima pregled i upravljanje svim predmetima koje predaju. Kroz ovaj portal, profesori mogu brzo pristupiti informacijama o svojim predmetima, a također imaju mogućnost stvaranja novih predmeta. Ovaj dio aplikacije olakšava profesorima organizaciju i vođenje njihovih predavanja te pruža bolji uvid u njihov rad unutar aplikacije.

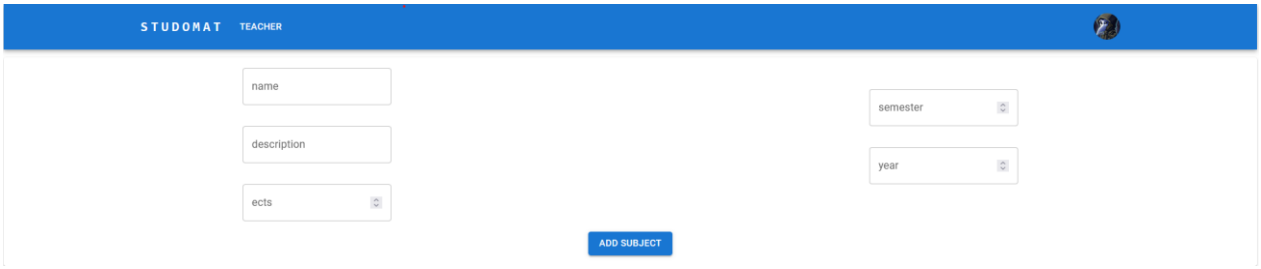


Slika 37 Teacher portal

Kao što možemo vidijeti iz priložene slike kada se prijavimo sa korisnikom koji ima prava profesora u zaglavlju se pojavi opcija teacher koja nam pruža pristup portalu za profesore.

3.5.2 Dodavanje predmeta

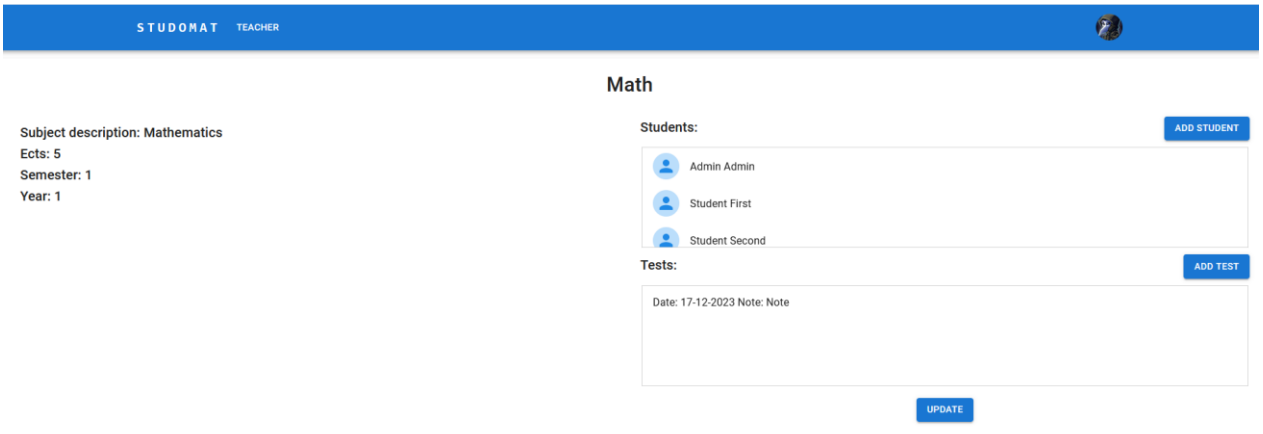
Komponenta "AddSubject" (slika 38) omogućava profesorima dodavanje novih predmeta u sustav. Ova komponenta pruža jednostavan način za unos svih relevantnih informacija o predmetu, uključujući naziv, opis, broj ECTS bodova, semestar i godinu studija na kojoj se predmet izvodi. Nakon unosa svih potrebnih podataka, korisnik može kliknuti na gumb "Dodaj predmet" kako bi spremljeni podaci postali dio aplikacije.



Slika 38 Dodavanje predmeta

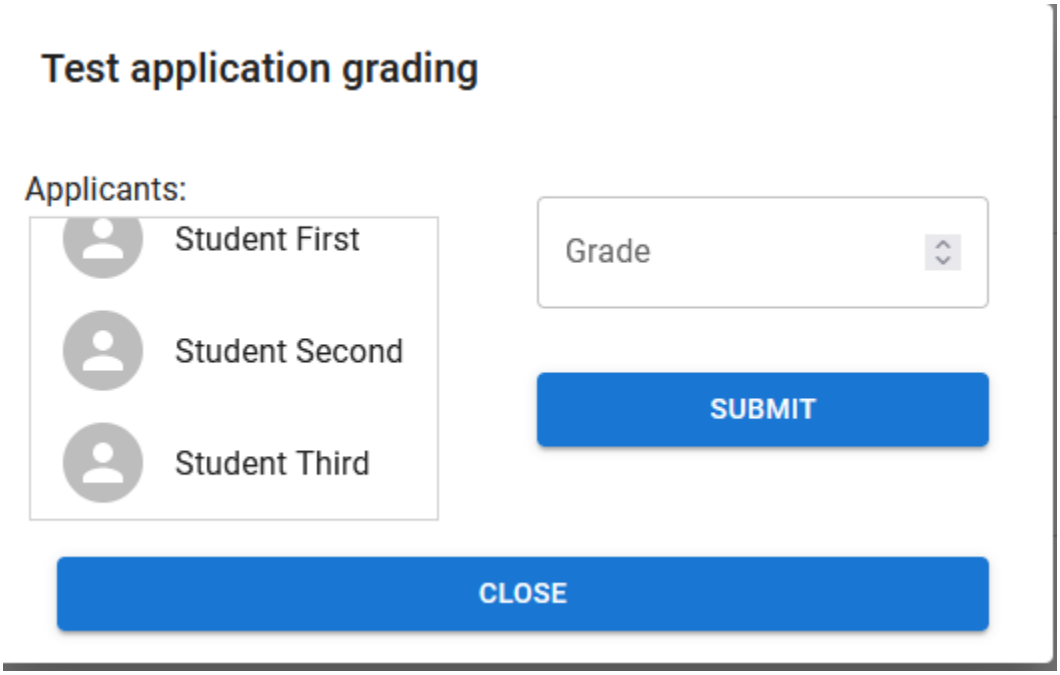
3.5.3 Pregled predmeta

Komponenta "Subject" (slika 39) je ključan alat za profesore. Omogućava im pregled svih važnih informacija o odabranom predmetu. Sadrži detalje kao što su naziv predmeta, opis, broj ECTS bodova, semestar i godina studija na kojoj se predmet izvodi.



Slika 39 Subject komponenta

Osim toga, profesorima se pruža mogućnost upravljanja studentima na predmetu, gdje mogu dodavati nove studente. Također, komponenta pruža uvid u sve testove koji su dodijeljeni ovom predmetu i omogućava profesorima dodavanje novih testova te ocjenjivanje postojećih (slika 40).

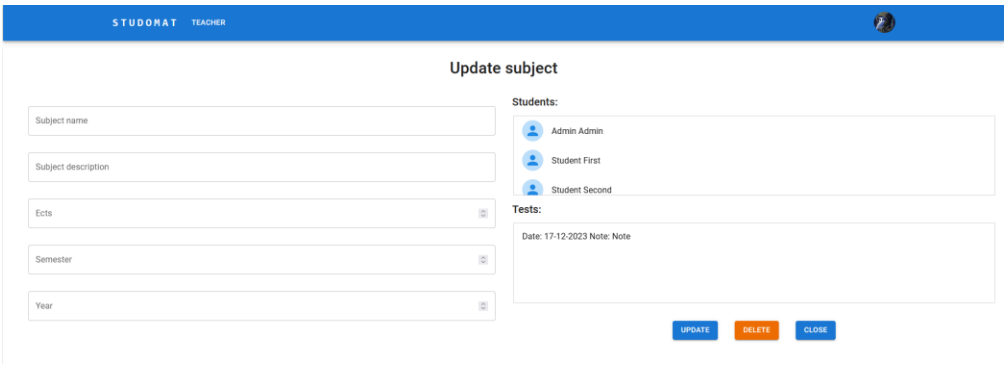


Slika 40 Ocjenjivanje testova

Ova komponenta je ključna za organizaciju i upravljanje predmetima iz perspektive profesora, pružajući im sve alate i informacije potrebne za učinkovito vođenje svojih predavanja i ocjenjivanje studenata.

3.5.4 Ažuriranje predmeta

Komponenta "UpdateSubject" (slika 41) je ključna za profesore jer im omogućava ažuriranje i upravljanje informacijama o odabranom predmetu. Profesori koriste ovu komponentu kako bi mijenjali naziv predmeta, opis, broj ECTS bodova, semestar i godinu studija na kojoj se predmet izvodi.



Slika 41 Ažuriranje predmeta

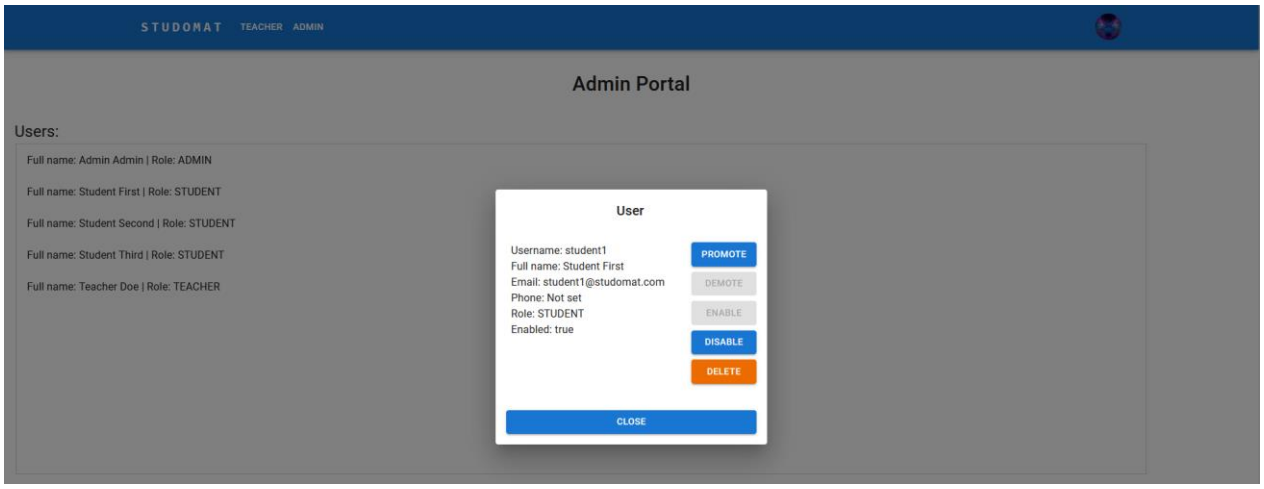
Osim toga, profesori mogu pregledavati i upravljati studentima na predmetu, dodavati ili uklanjati studente te pregledavati informacije o testovima i izmjenjivati ih.

3.6 Funkcionalnosti za administratore

U administrativnom dijelu sustava, imate širok spektar mogućnosti za upravljanje korisnicima. To uključuje promociju i demociju korisnika na različite razine pristupa, brisanje korisničkih računa koji više nisu potrebni, te aktiviranje ili deaktiviranje korisničkih računa prema potrebama.

3.6.1 Administratorski portal

Administratorski panel (slika 42) je ključni alat za upravljanje korisnicima i održavanje reda u sustavu. Ova komponenta omogućuje administratorima da pregledavaju, uređuju i upravljaju korisnicima sustava s lakoćom.



Slika 42 Administratorski panel

Evo kako je ovaj panel važan za administratore:

- **Pregled korisnika**: Panel omogućuje administratorima brz i jednostavan pregled svih korisnika u sustavu. To je korisno za praćenje broja korisnika i njihovih uloga.
- **Upravljanje ulogama**: Administratori mogu promovirati ili degradirati korisnike, omogućavajući im prilagodbu uloga i odgovornosti u sustavu
- **Brisanje korisnika**: Ako je potrebno, administratori mogu brisati korisničke račune koji više nisu potrebni ili su prestali biti aktivni.
- **Omogući/Onemogući korisnika**: Administratorski panel omogućuje administratorima da brzo omoguće ili onemoguće korisničke račune prema potrebi.

Ovaj administratorski panel je neophodan za održavanje sigurnosti i funkcionalnosti sustava, pružajući administratorima sredstva za pravilno upravljanje korisničkim računima i pristupom resursima.

4. Dokerizacija aplikacije

U ovoj sekciji detaljno ćemo razmotriti postupak Dockerizacije. Dockerizacija je moćna tehnika koja omogućuje izolaciju aplikacijskih komponenata u kontejnere, što pojednostavljuje njihovu distribuciju i izvođenje na različitim okolinama. Kroz ovu sekciju, prikazat ćemo korake za stvaranje Dockerfile-ova za Backend i Frontend, prijenos Docker slika na Docker Hub i korištenje Docker Compose-a za bezbolnu integraciju ovih komponenata.

4.1 Kreiranje Dockerfile-ova

U ovoj sekciji opisat ćemo korake za kreiranje Dockerfile-ova za Backend i Frontend komponente naše aplikacije. Dockerfile je konfiguracijska datoteka koja definira kako će se izgraditi Docker slika i koje će se komponente unutar slike koristiti.

4.1.2 Dockerfile za frontend

U ovom Dockerfile-u (slika 43), prvo se izrađuje privremena slika za izgradnju Frontend-a, zatim se kopiraju potrebni resursi i izvršavaju se komande za izgradnju. Nakon toga, kreira se finalna slika koja sadrži izgrađene resurse i Nginx poslužitelj za posluživanje Frontend-a.

```
FROM node as build
WORKDIR ./app
COPY package.json package-lock.json ./
RUN npm install
COPY . ./

ENV REACT_APP_API_BASE_URL="127.0.0.1"

RUN npm run build

FROM nginx
COPY --from=build /app/build /var/www
COPY nginx.conf /etc/nginx/nginx.conf
EXPOSE 80
ENTRYPOINT ["nginx","-g","daemon off;"]
```

Slika 43 Frontend Dockerfile

Za pokretanje Dockerfile-a koristimo komandu „*docker build -t frontend .*“.

4.1.3 Dockerfile za backend

Prvi korak bio je izgraditi izvršnu JAR datoteku iz izvornog koda Backend komponente pomoću Maven alata. Ova JAR datoteka sadrži cijelu Backend aplikaciju, uključujući sve njene ovisnosti. Nakon toga kreiramo Dockerfile koji će koristiti sliku OpenJDK za izvršavanje naše backend aplikacije.

```
FROM openjdk
WORKDIR /app
COPY ./ZavrsniRad-1.6.jar /app
CMD ["java", "-jar", "ZavrsniRad-1.6.jar"]
```

Slika 44 Backend Dockerfile

4.2 Objavljivanje slika na Docker Hub-u

Za objavljivanje slika na Docker Hub-u prvo moramo povezati računalo sa našim korisničkim računom pomoću komande *„docker login“*. Nakon toka kreiramo naše slike pomoću komande *„docker build“ (slika 45)*.

```
PS C:\Users\mmaru\Documents\IdeaProjects\Zavrsni_Server\support\docker> docker build -t backend .
[+] Building 1.6s (8/8) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 31B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/openjdk:latest
 => [internal] load build context
 => => transferring context: 42B
 => [1/3] FROM docker.io/library/openjdk@sha256:9b448de897d211c9e0ec635a485650aed6e28d4eca1efbc34940560a480b3f1f
 => CACHED [2/3] WORKDIR /app
 => CACHED [3/3] COPY ./ZavrsniRad-1.6.jar /app
 => exporting to image
 => => exporting layers
 => => writing image sha256:8d111b1898489771fd126f3ccef69c76f60f90ce0f71b9085ca9da6fe6d73428
 => => naming to docker.io/library/backend
```

Slika 45 docker build

Prije nego što objavite sliku, morate je tagirati s Docker Hub korisničkim imenom i verzijom. Koristite naredbu „*docker tag*“ (slika 46) kako biste to učinili.

```
docker tag backend marki2121/zavrsni-backend:0.0
```

Slika 46 docker tag

Nakon što ste tagirali svoje slike, možete ih objaviti na Docker Hub pomoću naredbe „*docker push*“ (slika 47).

```
PS C:\Users\mmaru\Documents\IdeaProjects\Zavrsni_Server\support\docker> docker push marki2121/zavrsni-backend:0.0
The push refers to repository [docker.io/marki2121/zavrsni-backend]
348c5bf64aa7: Pushed
3b8d75ec1774: Pushed
56285d9a7760: Layer already exists
077bff59ce57: Layer already exists
9cd9df9ffc97: Layer already exists
0.0: digest: sha256:f2bf138b13b82b6cdf7ffc0b5850f77a5b9b133b0c3e6ee5303e8aad646a1bec size: 1373
```

Slika 47 docker push

4.3 Konfiguracija Docker Compose-a

U ovoj sekciji, opisat ćemo kako konfigurirati Docker Compose za našu aplikaciju. Docker Compose će nam omogućiti da pokrenemo sve kontejnere (frontend, backend, i bazu podataka) kao dio jednog korištenja YAML konfiguracije. To izvršavamo pomoću naredbe „*docker compose up*“.

4.3.1 Backend dio konfiguracije

Na slici 48 prikazan je dio Docker Compose konfigracije koja pokreće backend dio aplikacije. U nastavku ćemo detaljno opisati ovu konfiguraciju:

- **container_name:** postavlja ime kontenjera
- **restart:** postavke za restart kontenjera
- **image:** slika korištena za pokretanje kontenjera
- **enviroment:** postavke okolišne varijable za konfiguraciju Spring Boot aplikacije. Unutar ove konfiguracije postavljamo URL za bazu podataka koju će backend koristiti
- **ports:** Povezuje port 8080 na lokalnom računalu s portom 8080 u kontejneru**.**
- **depends_on:** označava da je kontenjer ovisan o drugom kontenjeru, to jest backend dio se neće pokrenuti prije nego se pokrene baza podataka
- **networks:** govori na koje mreže povezujemo ovaj servis

```
backend:
  container_name: zavrsni-backend
  restart: on-failure
  image: marki2121/zavrsni-backend:1.7
  environment:
    SPRING_APPLICATION_JSON: '{
      "spring.datasource.url": "jdbc:mysql://db:3306/zavrsni?useSSL=false"
    }'
  ports:
    - "8080:8080"
  depends_on:
    - db
  networks:
    - backend
    - frontend
```

Slika 48 docker compose backend

4.3.2 Frontend dio konfguracije

Na slici 49 prikazan je dio Docker Compose konfigracije koja pokreće frontend dio aplikacije.

```
frontend:
  container_name: zavrsni-frontend
  restart: on-failure
  image: marki2121/zavrsni-frontend:1.7
  ports:
    - "9000:80"
  depends_on:
    - backend
    - db
  networks:
    - frontend
```

Slika 49 docker compose frontend

5. Zaključak

Predmet ovog završnog rada bio je prikaz razvoja full stack aplikacije pomoću *Spring Boot* i *React* razvojnih okruženja. Opisao se postupak razvoja web aplikacije te su navedeni i opisani svi koraci potrebni za izradu aplikacije. To uključuje izradu modela, njihovo povezivanje, izradu baze podataka te sve potrebne CRUD funkcionalnosti, autentifikaciju i validaciju korisnika, te sve potrebne funkcionalnosti za izradu, te kontrolu nad kolegijima.

Ovaj projek sam odabrao iz razloga što me zanima izrada backend dijela. Odabrani je Spring jer se još nisam susreo s tim razvojnim okvirom, a htio sam proširiti svoje znanje i razvojna okruženja za izradu backend dijela aplikacije.

Unutar svijeta razvoja Java aplikacija, Spring Boot nije uvijek prvi izbor, budući da Java ekosustav nudi širok spektar različitih okvira i tehnologija. No, Spring Boot se ističe kao iznimno popularan okvir zbog svoje sposobnosti da pojednostavi razvoj aplikacija, posebno web aplikacija i mikroservisa.

Također, zainteresirao sam se za React zbog njegove široke upotrebe i utjecaja u svijetu razvoja modernih web aplikacija. Odlučio sam dublje istražiti ovaj popularni JavaScript okvir kako bih bolje razumio njegove temeljne koncepte i mogućnosti, posebno u pogledu izgradnje frontend dijela aplikacija.

Vjerujem da ću u budućnosti koristiti kombinaciju Spring Boot i React zbog njihove izvanredne sinergije i sposobnosti za razvoj kompleksnih web aplikacija.

Popis slika

Literatura

1. „*Intruduction*“. Spring boot, https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started.html#getting-started.introducing-spring-boot (pristupljeno 15. svibnja 2023.).

2. „Spring boot“. Wikipedia, https://en.wikipedia.org/wiki/Spring_Boot (pristupljeno 15. svibnja 2023.).

3. „*Core properties*“. Spring boot, https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html (pristupljeno 16. svibnja 2023.).

4. „*JWT*“. Spring boot, https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/jwt.html (pristupljeno 23. svibnja 2023.).

5. „*Accessing Data with JPA*“. Spring boot, https://spring.io/guides/gs/accessing-data-jpa/ (pristupljeno 5. lipnja 2023.).

6. „*Liquidbase*“. Wikipedia, https://en.wikipedia.org/wiki/Liquibase (pristupljeno 9. kolovoza 2023.).

7. „*Docker“.* Wikipedia, https://en.wikipedia.org/wiki/Docker (pristupljeno 9. kolovoza 2023.).

8. „*React“.* Wikipedia, https://en.wikipedia.org/wiki/React_(software) (pristupljeno 9. kolovoza 2023.).

9. Material UI, *https://mui.com/material-ui/* (pristupljeno 22. kolovoza 2023.).

10. React router, https://reactrouter.com/en/main (pristupljeno 24. kolovoza 2023.).

11. React, https://react.dev/ (pristupljeno 26. kolovoza 2023.).

12. „*Try Docker Compose“.* Docker, *https://docs.docker.com/compose/gettingstarted/* (pristupljeno 5. rujna 2023.).