UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Milica Kostadinović

RAZVOJ VEB APLIKACIJE ZA UPRAVLjANjE DOGAĐAJIMA I PRIJAVAMA UČESNIKA

master rad

Beograd, 2026.

Mentor:

dr Mirjana Maljković Ružičić, docent
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

prof. dr Vesna Marinković, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Staša Vujičić Stanković, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Naslov master rada: Razvoj veb aplikacije za upravljanje događajima i prijavama učesnika

Rezime: U ovom radu predstavljeno je projektovanje i implementacija veb aplikacije za upravljanje događajima sa podrškom za komunikaciju u realnom vremenu i obradu konkurentnih prijava na događaje ograničenog kapaciteta. Cilj rada bio je razvoj jedinstvenog informacionog sistema koji objedinjuje upravljanje korisnicima, organizaciju događaja, prijavu i odjavu učesnika, kontrolu kapaciteta, upravljanje listom čekanja i razmenu poruka između učesnika i organizatora.

Serverski deo sistema realizovan je primenom okvira Spring Boot i REST arhitekture, dok je korisnički interfejs razvijen korišćenjem biblioteke React. Za skladištenje podataka korišćen je sistem za upravljanje relacionim bazama podataka PostgreSQL, autentifikacija i autorizacija implementirane su korišćenjem okvira Spring Security i JWT tokena, dok je komunikacija u realnom vremenu ostvarena primenom WebSocket protokola.

Poseban doprinos rada predstavlja implementacija benchmark modula namenjenog analizi performansi optimističkog i pesimističkog zaključavanja pri velikom broju istovremenih prijava korisnika na događaje ograničenog kapaciteta. Modul omogućava izvođenje kontrolisanih eksperimenata i poređenje vremena izvršavanja, broja uspešnih prijava, broja konflikata pri optimističkom zaključavanju i konzistentnosti podataka za oba pristupa zaključavanju.

Rezultati sprovedenih eksperimenata omogućavaju analizu prednosti i nedostataka optimističkog i pesimističkog zaključavanja u različitim uslovima opterećenja sistema, čime se pruža osnova za izbor odgovarajućeg mehanizma u informacionim sistemima sa velikim brojem konkurentnih zahteva.

Ključne reči: upravljanje događajima, veb aplikacija, Spring Boot, React, WebSocket, PostgreSQL, JWT, optimističko zaključavanje, pesimističko zaključavanje, konkurentno programiranje.

Sadržaj

Glava 1

Uvod

Razvoj informaciono-komunikacionih tehnologija i intenzivna digitalizacija poslovnih procesa doveli su do značajnih promena u načinu organizacije i realizacije različitih događaja. Konferencije, seminari, radionice, sportske manifestacije, kulturni događaji i drugi oblici okupljanja danas se u velikoj meri oslanjaju na informacione sisteme koji omogućavaju efikasno upravljanje učesnicima, organizatorima i svim pratećim aktivnostima. Pored potrebe za jednostavnijom administracijom događaja, savremeni korisnici očekuju brz pristup informacijama, mogućnost prijavljivanja putem interneta, trenutno obaveštavanje o promenama i komunikaciju u realnom vremenu.

U praksi se organizacija događaja veoma često zasniva na korišćenju više međusobno nepovezanih alata, kao što su elektronska pošta, Google Forms obrasci, Excel tabele ili različite aplikacije za razmenu poruka. Iako ovakav pristup može biti dovoljan za organizaciju manjih događaja, on postaje neefikasan kada broj učesnika raste ili kada postoji potreba za složenijim upravljanjem registracijama i komunikacijom. Ručno evidentiranje prijava povećava mogućnost nastanka grešaka, otežava praćenje raspoloživih mesta, usporava obradu zahteva i komplikuje razmenu informacija između organizatora i učesnika. Zbog toga se javlja potreba za jedinstvenim informacionim sistemom koji objedinjuje sve ključne funkcionalnosti neophodne za uspešnu organizaciju događaja.

Savremene veb aplikacije omogućavaju centralizaciju svih aktivnosti u okviru jedinstvenog sistema kojem korisnici pristupaju korišćenjem internet pregledača, bez potrebe za instalacijom dodatnog softvera. Na taj način omogućeno je jednostavno pravljenje događaja, upravljanje korisničkim nalozima, elektronsko prijavljivanje učesnika, automatska kontrola kapaciteta, vođenje liste čekanja, kao i komunikacija između korisnika u realnom vremenu. Integracijom navedenih funkcionalnosti unapređuje se efikasnost organizacije događaja, smanjuje mogućnost ljudskih grešaka i poboljšava korisničko isku-

stvo.

Jedan od najvećih izazova u razvoju informacionih sistema za upravljanje događajima predstavlja obrada velikog broja istovremenih prijava korisnika na događaje ograničenog kapaciteta. U situacijama kada više korisnika u istom trenutku pokušava da rezerviše poslednja raspoloživa mesta, neophodno je obezbediti konzistentnost podataka i sprečiti pojavu nekonzistentnih stanja u bazi podataka. Ovakvi problemi predstavljaju tipičan primer konkurentnog pristupa podacima i zahtevaju primenu odgovarajućih mehanizama upravljanja transakcijama. U savremenim informacionim sistemima najčešće se koriste optimističko i pesimističko zaključavanje, pri čemu izbor odgovarajuće strategije direktno utiče na performanse, propusnost i pouzdanost sistema.

Pored pouzdane obrade transakcija, savremene veb aplikacije sve češće zahtevaju razmenu podataka u realnom vremenu. Tradicionalni model komunikacije zasnovan na HTTP protokolu funkcioniše po principu zahtev–odgovor i nije pogodan za scenarije u kojima korisnici očekuju trenutno obaveštavanje o promenama u sistemu. Iz tog razloga u modernim aplikacijama široku primenu pronalazi WebSocket protokol, koji omogućava dvosmernu komunikaciju između klijenta i servera bez potrebe za periodičnim osvežavanjem stranice. Ovakav način komunikacije posebno je značajan u sistemima koji podržavaju razmenu poruka, praćenje statusa događaja i ažuriranje podataka u realnom vremenu.

Predmet ovog master rada jeste projektovanje, implementacija i analiza veb aplikacije namenjene upravljanju događajima, registracijama učesnika i komunikaciji u realnom vremenu. Razvijeni sistem objedinjuje funkcionalnosti upravljanja korisnicima, pravljenja i administracije događaja, prijavljivanja i odjavljivanja učesnika, automatskog upravljanja listom čekanja, kao i razmene javnih i privatnih poruka između učesnika i organizatora.

Serverski deo aplikacije razvijen je primenom okvira Spring Boot i REST arhitekture, dok je korisnički interfejs implementiran korišćenjem biblioteke React. Za skladištenje i upravljanje podacima korišćen je sistem za upravljanje bazama podataka (SUBP) PostgreSQL, koji obezbeđuje pouzdanu obradu transakcija i očuvanje integriteta podataka. Funkcionalnosti koje zahtevaju komunikaciju u realnom vremenu implementirane su korišćenjem WebSocket protokola.

Poseban doprinos ovog rada predstavlja razvoj benchmark modula namenjenog eksperimentalnom poređenju performansi optimističkog i pesimističkog zaključavanja prilikom obrade konkurentnih prijava na događaje ograničenog kapaciteta. Modul omogućava izvođenje kontrolisanih eksperimenata simulacijom velikog broja istovremenih korisnika, pri čemu se prikupljaju podaci o ukupnom vremenu izvršavanja, broju uspešnih prijava, broju konfli-

kata pri optimističkom zaključavanju i konzistentnosti podataka. Na osnovu dobijenih rezultata moguće je objektivno analizirati prednosti i ograničenja oba pristupa i proceniti njihovu pogodnost za primenu u informacionim sistemima sa velikim brojem konkurentnih zahteva.

Rad je organizovan u osam poglavlja. Nakon uvoda, u drugom poglavlju predstavljene su teorijske osnove tehnologija i koncepata na kojima se zasniva razvijeni sistem. Treće poglavlje posvećeno je projektovanju sistema, uključujući arhitekturu aplikacije, model baze podataka, projektovanje REST API-ja, bezbednosne arhitekture, komunikacije u realnom vremenu i benchmark modula. U četvrtom poglavlju opisana je implementacija serverskog i klijentskog dela sistema, kao i baze podataka. Peto poglavlje prikazuje izgled i najvažnije funkcionalnosti razvijene aplikacije iz perspektive korisnika. U šestom poglavlju predstavljena je evaluacija performansi optimističkog i pesimističkog zaključavanja kroz sprovedene benchmark eksperimente. U sedmom poglavlju opisana je upotreba alata zasnovanih na veštačkoj inteligenciji tokom izrade rada i razvoja aplikacije. U osmom poglavlju dati su zaključci, sumirani ostvareni rezultati rada i predstavljeni mogući pravci budućeg razvoja sistema.

Izvorni kod razvijene aplikacije dostupan je u javnom repozitorijumu: `https://github.com/milicak33/event_management`.

Glava 2

Teorijske osnove

2.1 Veb aplikacije

Veb aplikacije predstavljaju jednu od najznačajnijih kategorija savremenih informacionih sistema i danas se primenjuju u gotovo svim oblastima poslovanja, obrazovanja, državne uprave i elektronske trgovine, kao i u drugim oblastima. Njihova široka primena rezultat je razvoja internet tehnologija, povećanja dostupnosti računarskih mreža i potrebe za centralizovanim upravljanjem podacima i poslovnim procesima. Za razliku od tradicionalnih desktop aplikacija, kojima je potrebno pristupiti sa konkretnog računara i koje zahtevaju instalaciju na korisničkom uređaju, veb aplikacije omogućavaju pristup putem internet pregledača, nezavisno od operativnog sistema ili vrste uređaja koji korisnik koristi [1, 2].

Osnovna karakteristika veb aplikacija jeste podela sistema na klijentski i serverski deo. Korisnik ostvaruje interakciju sa aplikacijom putem internet pregledača, dok se obrada zahteva, izvršavanje poslovne logike i upravljanje podacima realizuju na serverskoj strani. Ovakav pristup omogućava jednostavnije održavanje sistema, centralizovano upravljanje podacima i istovremenu dostupnost aplikacije velikom broju korisnika [1].

Savremene veb aplikacije uglavnom su zasnovane na višeslojnoj arhitekturi, pri čemu svaki sloj ima jasno definisanu odgovornost. Najčešće se razlikuju sloj korisničkog interfejsa (frontend), aplikacioni sloj (backend) i sloj za skladištenje podataka. Razdvajanje ovih slojeva omogućava nezavisan razvoj pojedinačnih komponenti sistema, jednostavnije održavanje aplikacije i lakše proširivanje funkcionalnosti u skladu sa novim zahtevima korisnika [2].

Komunikacija između klijentskog i serverskog dela aplikacije najčešće se ostvaruje korišćenjem HTTP protokola i REST servisa, pri čemu se podaci razmenjuju u JSON (JavaScript Object Notation) formatu. U poslednjih

nekoliko godina sve veću primenu imaju i tehnologije koje omogućavaju komunikaciju u realnom vremenu, poput WebSocket protokola, čime se korisnicima omogućava trenutno dobijanje informacija bez potrebe za periodičnim osvežavanjem stranice [3, 4].

Pored funkcionalnosti, jedan od najvažnijih zahteva savremenih veb aplikacija jeste bezbednost. Budući da ovakvi sistemi često obrađuju osetljive podatke korisnika, neophodno je obezbediti pouzdanu autentifikaciju, autorizaciju i zaštitu komunikacije između klijenta i servera. Iz tog razloga u razvoju modernih aplikacija široku primenu imaju mehanizmi kao što su Spring Security i JSON Web Token (JWT), koji omogućavaju bezbednu kontrolu pristupa resursima sistema [5, 6].

2.2 Arhitektura klijent–server

Arhitektura klijent–server predstavlja jedan od najrasprostranjenijih modela za razvoj distribuiranih informacionih sistema i osnovu funkcionisanja savremenih veb aplikacija. U ovom modelu odgovornosti sistema podeljene su između klijentskog i serverskog dela, pri čemu klijent inicira zahteve, dok server obrađuje pristigle zahteve, izvršava poslovnu logiku i vraća odgovarajući odgovor korisniku [7, 2].

Klijentski deo sistema zadužen je za interakciju sa korisnikom i prikaz podataka. U slučaju veb aplikacija, ulogu klijenta najčešće ima internet pregledač u kojem se izvršava korisnički interfejs razvijen korišćenjem savremenih tehnologija kao što su HTML, CSS i JavaScript, odnosno odgovarajućih biblioteka i okvira poput biblioteke React. Klijent prikuplja korisničke zahteve, šalje ih serveru putem mreže i prikazuje rezultate obrade u odgovarajućem obliku.

Serverski deo sistema odgovoran je za obradu zahteva pristiglih od klijenata. Njegove osnovne funkcije obuhvataju izvršavanje poslovne logike, autentifikaciju i autorizaciju korisnika, komunikaciju sa sistemom za upravljanje bazama podataka, validaciju ulaznih podataka i formiranje odgovora koji se vraćaju klijentu. Budući da se poslovna logika nalazi na jednom mestu, održavanje sistema je jednostavnije, a izmene funkcionalnosti mogu se izvršiti bez potrebe za promenama na korisničkim uređajima [1].

Komunikacija između klijenta i servera ostvaruje se putem računarske mreže korišćenjem standardnih mrežnih protokola. U najvećem broju savremenih aplikacija razmena podataka odvija se korišćenjem HTTP protokola, pri čemu klijent šalje zahtev odgovarajućem resursu na serveru, a server vraća odgovor koji najčešće sadrži podatke u JSON formatu. Ovakav pristup omogućava jednostavnu integraciju različitih klijentskih aplikacija sa istim

serverskim sistemom.

Jedna od najvažnijih prednosti arhitekture klijent–server jeste jasno razdvajanje odgovornosti između pojedinih delova sistema. Korisnički interfejs može se razvijati i unapređivati nezavisno od serverskog dela aplikacije, dok izmene poslovne logike ne zahtevaju ponovno instaliranje aplikacije na korisničkim uređajima. Pored toga, ovakav pristup omogućava bolju skalabilnost sistema, jednostavnije održavanje i efikasnije upravljanje bezbednosnim mehanizmima [2].

2.3 REST arhitektura

REST (Representational State Transfer) predstavlja arhitektonski stil za razvoj distribuiranih informacionih sistema koji je definisan 2000. godine u doktorskoj disertaciji [3]. Danas REST predstavlja najrasprostranjeniji pristup za implementaciju veb servisa i komunikaciju između klijentskih i serverskih aplikacija. Njegova popularnost proizlazi iz jednostavnosti implementacije, nezavisnosti od programske platforme i dobre skalabilnosti, zbog čega ga koristi veliki broj savremenih informacionih sistema.

REST nije protokol niti programski okvir, već skup arhitektonskih principa koji definišu način organizacije komunikacije između klijenta i servera. Osnovna ideja jeste da server svoje funkcionalnosti izlaže kroz resurse kojima se pristupa korišćenjem jedinstvenih URI adresa. Klijent nad tim resursima izvršava različite operacije koristeći standardne HTTP metode, pri čemu server vraća odgovarajući odgovor zajedno sa odgovarajućim HTTP statusnim kodom [3].

Jedno od osnovnih svojstava REST arhitekture jeste komunikacija bez čuvanja stanja (engl. *stateless*). To znači da server ne čuva informacije o prethodnim zahtevima klijenta, već svaki zahtev mora sadržati sve podatke neophodne za njegovu obradu. Na ovaj način omogućena je bolja skalabilnost sistema, jednostavnije balansiranje opterećenja između više serverskih instanci i lakše održavanje aplikacije [8].

REST servisi najčešće koriste HTTP protokol kao komunikacioni mehanizam. HTTP definiše skup metoda kojima se određuje operacija koja će biti izvršena nad određenim resursom. U tabeli 2.1 prikazane su metode koje se najčešće koriste u REST servisima.

Tabela 2.1: Najčešće HTTP metode u REST arhitekturi

Metoda	**Namena**
GET	Preuzimanje jednog ili više resursa.
POST	Pravljenje novog resursa na serveru.
PUT	Potpuna izmena postojećeg resursa.
PATCH	Delimična izmena postojećeg resursa.
DELETE	Brisanje resursa.

Pored metoda, HTTP definiše i statusne kodove koji predstavljaju rezultat obrade zahteva. Kodovi iz opsega 2xx označavaju uspešno izvršavanje zahteva, 4xx ukazuju na greške nastale usled nepravilnog zahteva klijenta, dok kodovi iz opsega 5xx označavaju greške na serverskoj strani. Korišćenje standardizovanih statusnih kodova omogućava jednostavnije razvijanje i integraciju klijentskih aplikacija sa serverskim sistemom.

Podaci se u REST servisima najčešće razmenjuju korišćenjem JSON formata. JSON predstavlja lagan tekstualni format za razmenu podataka koji je jednostavan za čitanje i obradu kako ljudima, tako i računarima. Zahvaljujući svojoj jednostavnosti i širokoj podršci u različitim programskim jezicima, JSON je postao de facto standard za razmenu podataka između veb aplikacija [9].

2.4 Okvir Spring Boot

Spring Boot predstavlja okvir otvorenog koda namenjen razvoju aplikacija u programskom jeziku Java, zasnovan na okviru Spring Framework. Razvijen je sa ciljem pojednostavljivanja razvoja poslovnih aplikacija uklanjanjem potrebe za opsežnom konfiguracijom i omogućavanjem brže implementacije sistema spremnih za produkciju [5, 10].

Za razliku od tradicionalnog okvira Spring Framework, koji zahteva značajnu količinu XML ili Java konfiguracije, Spring Boot uvodi koncept automatske konfiguracije (engl. *Auto Configuration*). Na osnovu biblioteka koje se nalaze u projektu i definisanih podešavanja, okvir automatski konfiguriše veliki broj komponenti sistema, čime se značajno smanjuje količina koda koju programer mora ručno da napiše [11].

Jedna od najvažnijih karakteristika okvira Spring Boot jeste mogućnost razvoja samostalnih aplikacija koje u sebi sadrže ugrađeni aplikacioni server, kao što su Apache Tomcat, Jetty ili Undertow. Zahvaljujući tome, nije potrebno posebno instalirati niti konfigurisati aplikacioni server, već se aplikacija može pokrenuti direktno izvršavanjem generisanog JAR fajla. Ovakav

pristup pojednostavljuje razvoj, testiranje i distribuciju aplikacije [10].

Spring Boot se zasniva na principu inverzije kontrole (engl. *Inversion of Control*, IoC) i mehanizmu ubrizgavanja zavisnosti (engl. *Dependency Injection*, DI). Inverzija kontrole podrazumeva da upravljanje pravljenjem i životnim ciklusom objekata nije odgovornost same aplikacije, već se prepušta Spring okruženju, odnosno IoC kontejneru. Objekti kojima upravlja Spring kontejner u Spring terminologiji nazivaju se Spring bean objekti (engl. *Spring beans*). Kontejner je zadužen za njihovo pravljenje, konfigurisanje, povezivanje i upravljanje njihovim životnim ciklusom.

U okviru Spring okruženja, klase označene odgovarajućim anotacijama, kao što su `@Component`, `@Service`, `@Repository` i `@Controller`, mogu biti automatski prepoznate, a njihove instance registrovane kao objekti kojima upravlja IoC kontejner. Pored automatskog otkrivanja komponenti, ovi objekti se mogu eksplicitno definisati i u konfiguracionim klasama. Na taj način IoC kontejner raspolaže objektima koji čine aplikaciju i može da upravlja njihovim međusobnim zavisnostima.

Ubrizgavanje zavisnosti predstavlja jedan od osnovnih načina realizacije principa IoC. Umesto da objekat sam kreira druge objekte od kojih zavisi, potrebne zavisnosti mu obezbeđuje Spring kontejner. Na primer, ukoliko servisna klasa zavisi od repozitorijuma, Spring kontejner kreira potrebne objekte i objekat repozitorijuma prosleđuje servisnoj klasi. Ubrizgavanje zavisnosti može se realizovati preko konstruktora, metoda ili polja, pri čemu se u praksi često preporučuje ubrizgavanje preko konstruktora.

Na ovaj način IoC, DI i objekti kojima upravlja Spring kontejner predstavljaju međusobno povezane koncepte: IoC predstavlja princip prema kojem se upravljanje objektima prepušta Spring kontejneru, termin *beans* odnosi se na objekte kojima taj kontejner upravlja, dok DI predstavlja mehanizam kojim kontejner povezuje ove objekte i obezbeđuje njihove međusobne zavisnosti. Time se smanjuje direktna povezanost između komponenti i olakšavaju održavanje, proširivanje i testiranje aplikacije.

Razvoj REST servisa u okviru Spring Boot zasniva se na korišćenju anotacija koje pojednostavljuju definisanje kontrolera i mapiranje HTTP zahteva. Anotacije poput `@RestController`, `@RequestMapping`, `@GetMapping`, `@PostMapping`, `@PutMapping` i `@DeleteMapping` omogućavaju jednostavnu implementaciju REST API-ja, pri čemu okvir automatski vrši konverziju Java objekata u JSON format i obrnuto [11].

2.5 Biblioteka React

Biblioteka React predstavlja otvorenu biblioteku namenjenu razvoju modernih korisničkih interfejsa korišćenjem programskog jezika JavaScript. Razvila ju je kompanija Meta (nekada Facebook) sa ciljem pojednostavljivanja razvoja složenih jednostraničnih veb aplikacija (engl. *Single Page Applications, SPA*) [12, 13]. Zahvaljujući komponentnom pristupu i efikasnom mehanizmu ažuriranja korisničkog interfejsa, React je danas jedna od najčešće korišćenih biblioteka za razvoj frontend aplikacija.

Osnovu biblioteke React čine komponente, koje predstavljaju nezavisne i ponovo iskoristive celine korisničkog interfejsa. Svaka komponenta može sadržati sopstvenu logiku, stanje i prikaz podataka, dok se složeniji interfejsi formiraju njihovim međusobnim povezivanjem. Ovakav pristup omogućava jednostavnije održavanje aplikacije, veću preglednost izvornog koda i ponovnu upotrebu funkcionalnosti u različitim delovima sistema [12].

Jedna od najvažnijih karakteristika biblioteke React jeste Virtual DOM (engl. *Virtual Document Object Model*). Umesto direktnog ažuriranja stvarnog DOM stabla nakon svake izmene podataka, React prvo ažurira njegovu virtuelnu reprezentaciju, zatim određuje minimalan skup promena i primenjuje ih na stvarni DOM. Time se značajno smanjuje broj operacija nad korisničkim interfejsom i poboljšavaju performanse aplikacije, naročito kod sistema sa velikim brojem interaktivnih elemenata [13].

Savremene React aplikacije zasnivaju se na funkcionalnim komponentama i React Hooks mehanizmu, koji omogućava upravljanje stanjem aplikacije i životnim ciklusom komponenti bez potrebe za korišćenjem klasnih komponenti. Najčešće korišćeni Hook-ovi su `useState` za upravljanje stanjem i `useEffect` za izvršavanje operacija kao što su slanje HTTP zahteva ili obrada podataka nakon prikazivanja komponente [12].

Komunikacija React aplikacije sa serverskim delom sistema najčešće se ostvaruje korišćenjem HTTP zahteva prema REST servisima. U praksi se za ovu namenu često koriste biblioteke kao što su Axios [14] ili ugrađena Fetch API funkcionalnost JavaScript jezika. Dobijeni podaci se obrađuju i prikazuju korisniku kroz odgovarajuće React komponente.

2.6 PostgreSQL

PostgreSQL predstavlja relacioni sistem za upravljanje bazama podataka otvorenog koda, razvijen sa ciljem obezbeđivanja visokih performansi, pouzdanosti i usklađenosti sa SQL standardom [15]. Zahvaljujući bogatom skupu funkcionalnosti, podršci za transakcije i velikoj skalabilnosti, PostgreSQL se

danas koristi u velikom broju poslovnih informacionih sistema.

Relacione baze podataka organizuju informacije u obliku tabela koje se sastoje od redova i kolona. Povezanost između tabela ostvaruje se korišćenjem primarnih i stranih ključeva, čime se omogućava očuvanje referencijalnog integriteta podataka i efikasno izvršavanje složenih upita [16].

Jedna od najvažnijih karakteristika PostgreSQL sistema jeste podrška za ACID transakcije (Atomicity, Consistency, Isolation, Durability), koje obezbeđuju pouzdanu obradu podataka čak i u uslovima velikog broja istovremenih korisnika. Transakcioni mehanizam garantuje da će sve operacije biti izvršene u celini ili neće biti izvršene uopšte, čime se sprečava nastanak nekonzistentnih stanja baze podataka [15].

PostgreSQL implementira MVCC (engl. *Multi-Version Concurrency Control*), mehanizam koji omogućava istovremeno izvršavanje velikog broja operacija čitanja i pisanja uz minimalan broj konflikata između transakcija. Ovakav pristup doprinosi većoj propusnosti sistema i predstavlja osnovu za implementaciju različitih strategija upravljanja konkurentnim pristupom podacima [15].

2.7 Spring Security i JWT

Bezbednost predstavlja jedan od najvažnijih aspekata razvoja savremenih veb aplikacija, naročito kada sistem obrađuje podatke korisnika i omogućava pristup funkcionalnostima koje nisu namenjene svim korisnicima. Mehanizmi autentifikacije i autorizacije imaju zadatak da obezbede da sistemu mogu pristupiti isključivo legitimni korisnici, kao i da svaki korisnik može koristiti samo one funkcionalnosti za koje poseduje odgovarajuća ovlašćenja [17, 18].

Autentifikacija predstavlja proces provere identiteta korisnika, odnosno potvrdu da je korisnik zaista ono za šta se predstavlja. Najčešće se realizuje proverom korisničkog imena ili elektronske adrese i lozinke. Nakon uspešne autentifikacije, sistem utvrđuje prava pristupa korisnika kroz proces autorizacije. Tokom autorizacije određuje se kojim resursima i funkcionalnostima korisnik može pristupiti u skladu sa svojom ulogom u sistemu [17].

Spring Security predstavlja zvanični bezbednosni okvir Spring ekosistema namenjen implementaciji autentifikacije, autorizacije i zaštite aplikacija od različitih bezbednosnih pretnji. Okvir omogućava zaštitu REST servisa, upravljanje korisničkim ulogama, filtriranje HTTP zahteva i jednostavnu integraciju sa različitim mehanizmima autentifikacije [19].

U savremenim REST aplikacijama sve češće se koristi autentifikacija zasnovana na JSON Web Token (JWT) tehnologiji. JWT predstavlja kompaktan i digitalno potpisan token koji server generiše nakon uspešne prijave

korisnika. Token sadrži skup informacija (engl. *claims*) koje omogućavaju identifikaciju korisnika i proveru njegovih ovlašćenja bez potrebe za čuvanjem sesije na serverskoj strani [6].

JWT se sastoji od tri logičke celine razdvojene tačkama:

- Header – sadrži informacije o tipu tokena i algoritmu korišćenom za digitalni potpis;
- Payload – sadrži podatke o korisniku, poput identifikatora, elektronske adrese, korisničke uloge i vremena isteka tokena;
- Signature – predstavlja digitalni potpis kojim se garantuje integritet tokena i sprečava njegova neovlašćena izmena.

Kod autentifikacije zasnovane na JWT-u, nakon uspešne prijave korisnika server generiše token i vraća ga klijentu. Klijent zatim token prosleđuje u zaglavlju narednih HTTP zahteva korišćenjem *Authorization* zaglavlja u obliku:

```
Authorization: Bearer <JWT>
```

Po prijemu zahteva, Spring Security filter proverava validnost tokena, njegov digitalni potpis i vreme isteka. Ukoliko je token ispravan, korisnik se smatra autentifikovanim i dozvoljava mu se pristup odgovarajućim REST servisima. U suprotnom, zahtev se odbacuje i server vraća odgovarajući HTTP statusni kod (najčešće 401 Unauthorized ili 403 Forbidden) [19, 6].

Prednost JWT autentifikacije u odnosu na tradicionalne sesije ogleda se u činjenici da server ne čuva informacije o korisničkim sesijama. Svaki zahtev sadrži sve neophodne informacije za proveru identiteta korisnika, zbog čega je ovakav pristup posebno pogodan za distribuirane i mikroservisne aplikacije, kao i sisteme sa velikim brojem istovremenih korisnika [6].

2.8 Protokol WebSocket

Klasična komunikacija između klijentskog i serverskog dela veb aplikacije najčešće se zasniva na HTTP protokolu, koji funkcioniše po principu zahtev–odgovor. U ovom modelu klijent inicira svaki zahtev, dok server može poslati odgovor isključivo kao reakciju na pristigli zahtev. Iako je ovakav način komunikacije dovoljan za većinu funkcionalnosti savremenih veb aplikacija, nije pogodan za scenarije u kojima je potrebno trenutno obaveštavanje korisnika o promenama u sistemu [4].

WebSocket predstavlja mrežni protokol koji omogućava uspostavljanje trajne dvosmerne komunikacije između klijenta i servera preko jedne TCP konekcije. Nakon inicijalnog HTTP zahteva za uspostavljanje veze, komunikacija se nastavlja korišćenjem WebSocket protokola, pri čemu obe strane mogu slati poruke u bilo kom trenutku bez potrebe za ponovnim uspostavljanjem konekcije [4].

Podrška za WebSocket u okviru Spring Boot okvira implementirana je korišćenjem modula Spring WebSocket. Za organizaciju komunikacije često se koristi STOMP (Simple Text Oriented Messaging Protocol) [4, 20, 21], koji omogućava slanje poruka na unapred definisane destinacije i jednostavno upravljanje pretplatama korisnika [20].

2.9 Optimističko zaključavanje

U informacionim sistemima koji omogućavaju istovremeni pristup velikog broja korisnika istim podacima može doći do konflikata prilikom ažuriranja zapisa u bazi podataka. Kako bi se očuvala konzistentnost podataka, primenjuju se različite strategije upravljanja konkurentnim pristupom. Jedna od najčešće korišćenih strategija jeste optimističko zaključavanje [22, 23].

Optimističko zaključavanje polazi od pretpostavke da su konflikti između korisnika relativno retki i da nije potrebno zaključavati podatke tokom njihovog čitanja. Umesto toga, sistem dozvoljava istovremeni pristup istim podacima, dok se prilikom upisa proverava da li je druga transakcija u međuvremenu izmenila zapis.

Najčešći način implementacije optimističkog zaključavanja zasniva se na korišćenju verzionog broja zapisa. Svaki entitet poseduje atribut koji predstavlja njegovu trenutnu verziju. Prilikom uspešne izmene podataka verzija se automatski uvećava. Ukoliko druga transakcija pokuša da sačuva zastarelu verziju istog zapisa, sistem detektuje konflikt i odbacuje izvršavanje transakcije [23].

Prednosti optimističkog zaključavanja ogledaju se u velikoj propusnosti sistema, odsustvu dugotrajnih zaključavanja baze podataka i dobroj skalabilnosti pri velikom broju operacija čitanja. Sa druge strane, u sistemima sa velikim brojem istovremenih izmena može doći do povećanog broja konflikata i potrebe za ponovnim izvršavanjem transakcija.

2.10 Pesimističko zaključavanje

Za razliku od optimističkog pristupa, pesimističko zaključavanje polazi od pretpostavke da su konflikti između korisnika česti i da ih je potrebno sprečiti pre nego što do njih dođe. Zbog toga sistem zaključava podatke odmah nakon njihovog preuzimanja, čime se drugim transakcijama onemogućava njihova izmena sve do završetka tekuće transakcije [22].

Hibernate predstavlja ORM (*Object-Relational Mapping*) okvir za programski jezik Java koji omogućava mapiranje objekata aplikacije na tabele relacione baze podataka i upravljanje njihovim čuvanjem i izmenama [23].

U relacionim bazama podataka pesimističko zaključavanje najčešće se realizuje korišćenjem SQL mehanizama za zaključavanje redova, dok se u okviru Hibernate okvira može primeniti korišćenjem režima `PESSIMISTIC_WRITE`. Na taj način sistem za upravljanje bazama podataka garantuje da samo jedna transakcija može menjati određeni zapis u datom trenutku.

Osnovna prednost pesimističkog zaključavanja jeste gotovo potpuno eliminisanje konflikata između konkurentnih transakcija. Međutim, zaključavanje zapisa može dovesti do čekanja drugih korisnika, povećanja vremena odziva sistema i pojave uzajamnog blokiranja (engl. *deadlock*), pri kojem transakcije međusobno čekaju oslobađanje resursa [22].

2.11 Benchmark analiza performansi

Benchmark analiza predstavlja postupak sistematskog merenja performansi softverskog sistema u unapred definisanim uslovima rada. Cilj benchmark testiranja jeste objektivna procena efikasnosti različitih implementacionih rešenja, identifikacija potencijalnih uskih grla i analiza ponašanja sistema pri različitim nivoima opterećenja [24].

Performanse informacionog sistema mogu se analizirati korišćenjem različitih metrika. Najčešće korišćene mere uključuju ukupno vreme izvršavanja, medijanu vremena odziva, prosečno vreme odziva, 95. percentil vremena odziva (P95), broj uspešno izvršenih operacija i propusnost sistema izraženu brojem obrađenih zahteva u jedinici vremena.

U sistemima koji omogućavaju konkurentan pristup podacima benchmark analiza ima poseban značaj, jer omogućava poređenje različitih strategija upravljanja konkurentnim transakcijama. Rezultati benchmark testiranja mogu pokazati kako se performanse sistema menjaju sa povećanjem broja istovremenih korisnika i pomoći pri izboru odgovarajuće strategije zaključavanja.

Glava 3

Projektovanje sistema

Projektovanje informacionog sistema predstavlja jednu od najvažnijih faza razvoja softvera, tokom koje se definiše arhitektura sistema, organizacija podataka, način komunikacije između pojedinačnih komponenti i mehanizmi za realizaciju funkcionalnih i nefunkcionalnih zahteva. Dobro projektovana arhitektura omogućava jednostavnije održavanje sistema, veću modularnost, lakše proširivanje funkcionalnosti i bolju skalabilnost.

U okviru ovog rada razvijena je veb aplikacija za upravljanje događajima i komunikaciju u realnom vremenu zasnovana na arhitekturi klijent–server. Klijentski deo sistema implementiran je korišćenjem biblioteke React, dok je serverski deo razvijen primenom okvira Spring Boot. Za skladištenje podataka korišćen je sistem za upravljanje bazama podataka PostgreSQL, dok je komunikacija u realnom vremenu realizovana primenom WebSocket protokola. Autentifikacija i autorizacija korisnika implementirane su korišćenjem okvira Spring Security i JWT mehanizma.

U ovom poglavlju prikazano je projektovanje razvijenog sistema. Najpre je opisana arhitektura aplikacije, zatim projektovanje baze podataka, organizacija REST API-ja, bezbednosna arhitektura, komunikacija u realnom vremenu, kao i projektovanje benchmark modula za analizu performansi različitih strategija zaključavanja.

3.1 Arhitektura sistema

Razvijeni informacioni sistem zasnovan je na višeslojnoj arhitekturi klijent–server, koja omogućava jasno razdvajanje korisničkog interfejsa, poslovne logike i sloja za skladištenje podataka. Ovakva organizacija sistema doprinosi većoj modularnosti, lakšem održavanju i mogućnosti nezavisnog razvoja pojedinačnih komponenti aplikacije.

Arhitektura sistema sastoji se od četiri osnovne celine:

- klijentske aplikacije razvijene korišćenjem biblioteke React;
- serverske aplikacije razvijene korišćenjem okvira Spring Boot;
- baze podataka PostgreSQL;
- komunikacionog sloja zasnovanog na protokolu WebSocket.

Korisnik ostvaruje interakciju sa sistemom putem klijentske aplikacije razvijene korišćenjem biblioteke React, koja se izvršava u internet pregledaču. Korisnički interfejs omogućava prijavu korisnika, pregled događaja, registraciju na događaje, administraciju sistema i razmenu poruka sa drugim korisnicima.

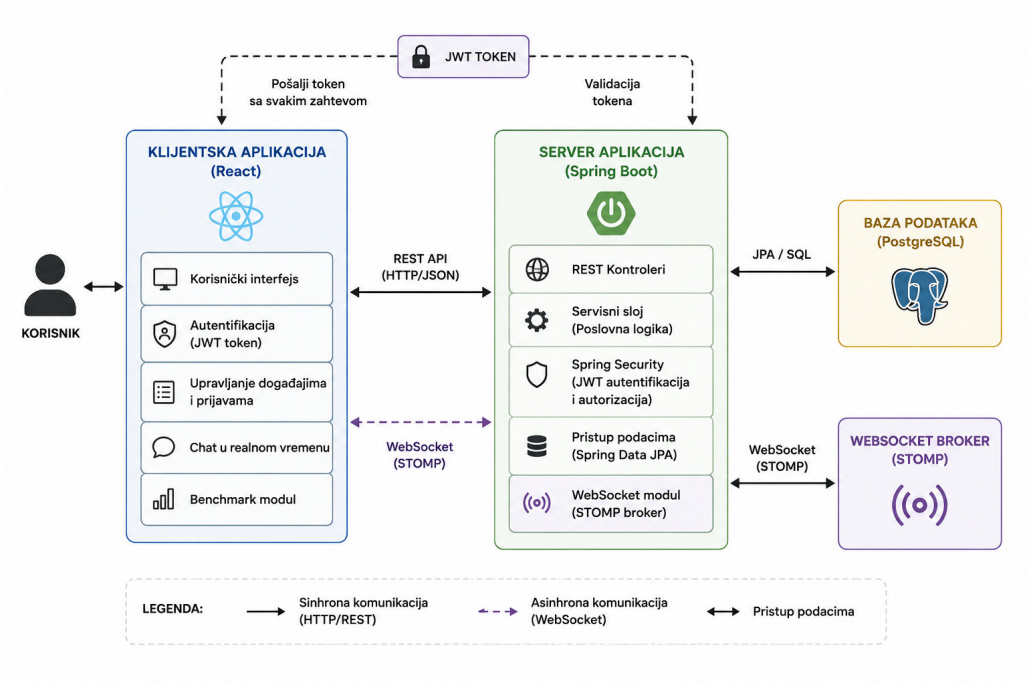
Klijentska aplikacija ostvaruje komunikaciju sa serverskim delom sistema korišćenjem REST servisa [3]. HTTP zahtevi koriste JSON format za razmenu podataka, dok se za funkcionalnosti koje zahtevaju trenutno obaveštavanje korisnika koristi WebSocket komunikacija [20].

Serverski deo sistema razvijen je primenom okvira Spring Boot i organizovan je u više logičkih slojeva. Kontrolerski sloj prima zahteve od klijenta, servisni sloj implementira poslovnu logiku, dok repozitorijumi ostvaruju komunikaciju sa sistemom za upravljanje bazom podataka PostgreSQL korišćenjem biblioteke Spring Data JPA. Autentifikacija i autorizacija korisnika realizovane su korišćenjem okvira Spring Security i JWT mehanizma.

Podaci o korisnicima, događajima, registracijama i porukama trajno se čuvaju u bazi podataka. Za obezbeđivanje konzistentnosti podataka tokom konkurentnih prijava na događaje korišćeni su mehanizmi optimističkog i pesimističkog zaključavanja.

Komunikacija u realnom vremenu realizovana je pomoću WebSocket protokola i STOMP komunikacionog modela. Na ovaj način omogućena je trenutna razmena poruka između organizatora i učesnika bez potrebe za periodičnim slanjem HTTP zahteva.

Na Slici 3.1 prikazana je višeslojna arhitektura razvijenog sistema i komunikacija između njegovih osnovnih komponenti. Klijentska aplikacija komunicira sa serverskim delom putem REST servisa i WebSocket konekcije, dok serverska aplikacija pristupa PostgreSQL bazi podataka radi trajnog čuvanja podataka.



Slika 3.1: Višeslojna arhitektura razvijenog informacionog sistema

3.2 Projektovanje baze podataka

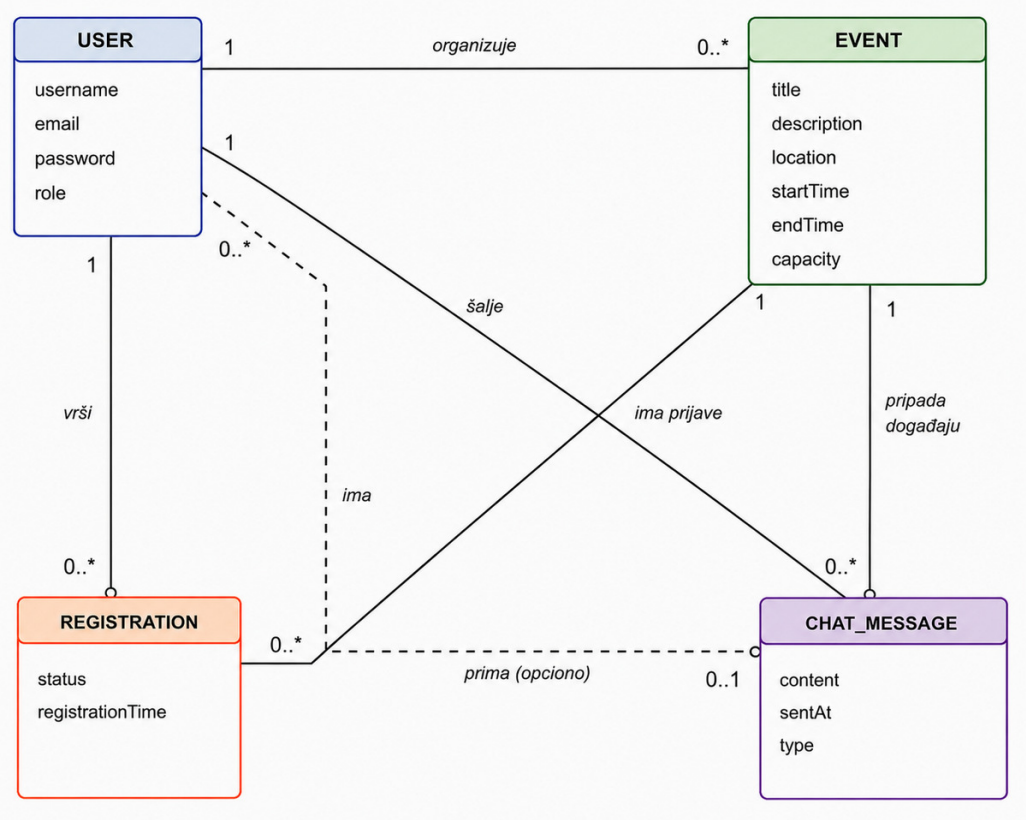
Projektovanje baze podataka predstavlja važan deo razvoja informacionog sistema, jer definiše način organizacije i čuvanja podataka koji podržavaju poslovne procese aplikacije. Model baze podataka mora da obezbedi konzistentnost podataka, očuvanje referencijalnog integriteta i efikasno izvršavanje operacija nad podacima.

U okviru razvijene aplikacije korišćen je sistem za upravljanje relacionim bazama podataka PostgreSQL, dok je mapiranje Java objekata na relacione tabele realizovano primenom okvira Hibernate ORM i biblioteke Spring Data JPA [15, 23, 25]. Entitetske klase mapirane su na odgovarajuće tabele u bazi podataka, dok su veze između tabela implementirane korišćenjem odgovarajućih JPA anotacija.

Model baze podataka projektovan je tako da podrži upravljanje korisnicima, organizaciju događaja, prijave učesnika i komunikaciju u realnom vremenu. Sastoji se od četiri osnovna entiteta: *User*, *Event*, *Registration* i *ChatMessage*. Njihove međusobne veze omogućavaju efikasno evidentiranje svih aktivnosti korisnika u okviru sistema.

Konceptualni model baze podataka prikazan je dijagramom na Slici 3.2.

Dijagram predstavlja konceptualni prikaz osnovnih entiteta sistema i njihovih međusobnih veza, nezavisno od načina implementacije u relacionoj bazi podataka.

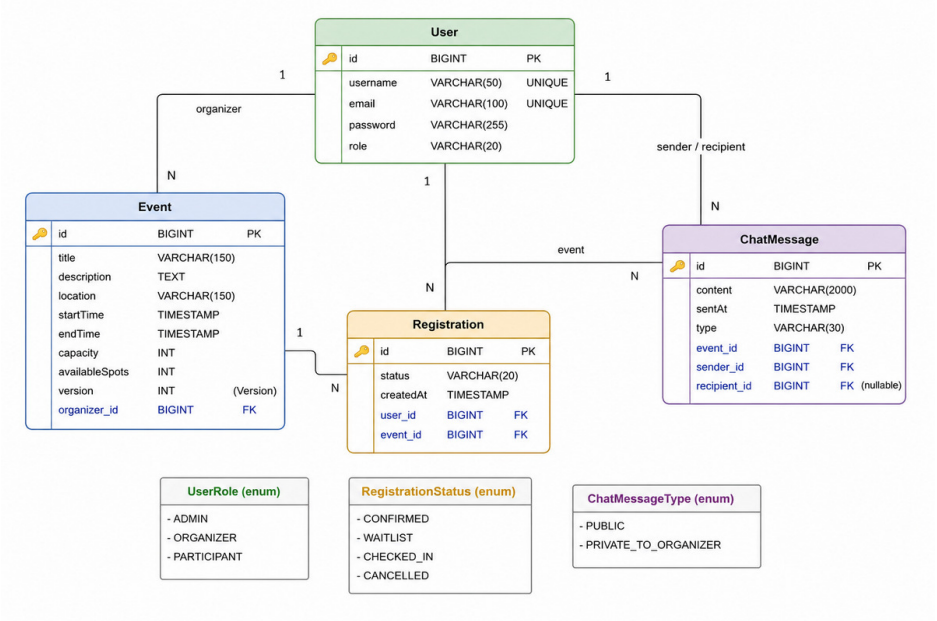


Slika 3.2: Konceptualni model baze podataka

3.2.1 Logički model baze podataka

Logički model baze podataka prikazuje tabele, njihove atribute i veze koje su implementirane u relacionoj bazi podataka PostgreSQL.

Na Slici 3.3 prikazan je, pomoću dijagrama tabela, logički model baze podataka razvijene aplikacije.



Slika 3.3: Logički model baze podataka

3.2.2 Entitet User

Entitet *User* predstavlja registrovanog korisnika sistema i sadrži podatke neophodne za autentifikaciju, autorizaciju i identifikaciju korisnika. Svakom korisniku dodeljen je jedinstveni identifikator, korisničko ime, adresa elektronske pošte, lozinka i korisnička uloga.

Korisničke uloge definisane su enumeracijom *UserRole*, koja razlikuje administratore, organizatore i učesnike sistema. Na osnovu dodeljene uloge određuju se prava pristupa pojedinim funkcionalnostima aplikacije.

Entitet *User* povezan je sa entitetima *Event*, *Registration* i *ChatMessage*, čime je omogućeno evidentiranje organizovanih događaja, prijava korisnika i razmenjenih poruka.

Pregled atributa entiteta *User* prikazan je u Tabeli 3.1.

Tabela 3.1: Atributi entiteta User

Atribut	**Tip**	**Opis**
id	Long	Jedinstveni identifikator korisnika.
username	String	Korisničko ime.
email	String	Adresa elektronske pošte korisnika.
password	String	Lozinka korisnika (heširana).
role	UserRole	Korisnička uloga u sistemu.

3.2.3 Entitet Event

Entitet *Event* predstavlja događaj koji organizator pravi u sistemu. Sadrži sve informacije potrebne za prikaz događaja korisnicima i upravljanje prijavama.

Pored osnovnih podataka, entitet sadrži atribut *availableSpots*, koji predstavlja broj slobodnih mesta na događaju, kao i atribut *version*. Ovaj atribut koristi se za implementaciju optimističkog zaključavanja i omogućava bezbedno izvršavanje konkurentnih operacija nad podacima bez narušavanja njihovog integriteta.

Pregled atributa entiteta *Event* prikazan je u Tabeli 3.2.

Tabela 3.2: Atributi entiteta Event

Atribut	**Tip**	**Opis**
id	Long	Jedinstveni identifikator događaja.
title	String	Naziv događaja.
description	String	Opis događaja.
location	String	Lokacija održavanja događaja.
startTime	LocalDateTime	Vreme početka događaja.
endTime	LocalDateTime	Vreme završetka događaja.
capacity	Integer	Maksimalan broj učesnika.
availableSpots	Integer	Broj slobodnih mesta.
version	Long	Verzija zapisa za optimističko zaključavanje.
organizer	User	Organizator događaja.

3.2.4 Entitet Registration

Entitet *Registration* predstavlja prijavu korisnika na određeni događaj. Njegova osnovna uloga jeste realizacija veze više-prema-više između entiteta User i Event.

Pored reference na korisnika i događaj, čuvaju se status prijave i vreme njenog pravljenja. Na ovaj način omogućeno je praćenje svih prijava i njihovog trenutnog stanja.

Pregled atributa entiteta *Registration* prikazan je u Tabeli 3.3.

Tabela 3.3: Atributi entiteta Registration

Atribut	**Tip**	**Opis**
id	Long	Jedinstveni identifikator prijave.
user	User	Korisnik koji se prijavio.
event	Event	Događaj na koji je izvršena prijava.
status	RegistrationStatus	Trenutni status prijave.
createdAt	LocalDateTime	Datum i vreme prijave.

3.2.5 Entitet ChatMessage

Entitet *ChatMessage* koristi se za čuvanje poruka koje korisnici razmenjuju u okviru konkretnog događaja. Sistem podržava javne poruke, koje su vidljive svim učesnicima događaja, kao i privatne poruke namenjene isključivo organizatoru događaja.

Pored sadržaja poruke čuvaju se podaci o pošiljaocu, primaocu (kod privatnih poruka), događaju kojem poruka pripada, vremenu slanja i tipu poruke.

Pregled atributa entiteta *ChatMessage* prikazan je u Tabeli 3.4.

Tabela 3.4: Atributi entiteta ChatMessage

Atribut	**Tip**	**Opis**
id	Long	Jedinstveni identifikator poruke.
content	String	Tekst poruke.
sentAt	LocalDateTime	Datum i vreme slanja.
type	ChatMessageType	Tip poruke (javna ili privatna).
sender	User	Pošiljalac poruke.
recipient	User	Primalac privatne poruke.
event	Event	Događaj kojem poruka pripada.

3.2.6 Veze između entiteta

Veze između entiteta projektovane su u skladu sa poslovnim zahtevima sistema i pravilima relacionog modela baze podataka. Korišćenjem odgovarajućih JPA anotacija omogućeno je očuvanje referencijalnog integriteta podataka i jednostavno upravljanje međusobno povezanim objektima.

Između entiteta *User* i *Event* uspostavljena je veza jedan-prema-više. Jedan korisnik sa ulogom organizatora može praviti više događaja, dok svaki događaj ima tačno jednog organizatora. Ova veza realizovana je korišćenjem atributa *organizer* u entitetu *Event*.

Entitet *Registration* predstavlja vezu između korisnika i događaja. Jedan korisnik može biti prijavljen na više događaja, dok jedan događaj može imati više prijavljenih korisnika. Zbog toga je veza više-prema-više između entiteta *User* i *Event* realizovana posredstvom entiteta *Registration*, koji pored referenci na korisnika i događaj sadrži i dodatne informacije o statusu prijave i vremenu njenog pravljenja.

Entitet *ChatMessage* povezan je sa entitetom *Event*, čime se obezbeđuje da svaka poruka pripada tačno jednom događaju. Istovremeno, svaka poruka povezana je sa korisnikom koji ju je poslao, dok se kod privatnih poruka evidentira i korisnik kojem je poruka namenjena. Na ovaj način omogućena je realizacija javne komunikacije između svih učesnika događaja, kao i privatne komunikacije između učesnika i organizatora.

Projektovani model baze podataka obezbeđuje efikasno čuvanje podataka, podržava sve funkcionalnosti razvijene aplikacije i predstavlja osnovu za implementaciju poslovne logike na serverskoj strani sistema.

3.3 Projektovanje REST API-ja

Komunikacija između klijentske i serverske strane aplikacije realizovana je primenom REST arhitekture [3]. Serverski deo sistema implementira skup REST servisa koji omogućavaju izvršavanje svih funkcionalnosti aplikacije, uključujući autentifikaciju korisnika, upravljanje događajima, prijave na događaje, upravljanje korisnicima i razmenu poruka.

REST servisi implementirani su korišćenjem okvira Spring Boot, pri čemu je svaki skup funkcionalnosti organizovan u zaseban kontroler. Na ovaj način ostvarena je jasna podela odgovornosti između pojedinih delova sistema, što doprinosi većoj modularnosti i jednostavnijem održavanju aplikacije.

Klijentska aplikacija komunicira sa REST servisima serverske aplikacije slanjem HTTP zahteva, pri čemu se podaci razmenjuju u JSON formatu. Za slanje zahteva i obradu odgovora koristi se biblioteka Axios.

3.3.1 Organizacija kontrolera

Najvažniji kontroleri serverske aplikacije su:

- *AuthController* – zadužen je za registraciju i prijavu korisnika u sistem, kao i za izdavanje JWT tokena koji se koristi za autentifikaciju narednih zahteva;
- *EventController* – omogućava pravljenje, izmenu, brisanje i prikazivanje događaja, kao i pretragu dostupnih događaja;

- *RegistrationController* – upravlja prijavama korisnika na događaje, otkazivanjem prijava i pregledom prijavljenih učesnika;
- *UserController* – namenjen je administraciji korisnika sistema i omogućava pregled i upravljanje registrovanim korisnicima;
- *ChatController* – realizuje komunikaciju između učesnika događaja i organizatora i podržava razmenu javnih i privatnih poruka.

3.3.2 Najvažnije REST krajnje tačke

REST API razvijene aplikacije organizovan je prema resursima nad kojima se izvršavaju operacije. Za svaki resurs definisane su odgovarajuće HTTP metode i URI putanje. Na taj način klijentska aplikacija može da izvršava autentifikaciju korisnika, upravlja događajima i prijavama, razmenjuje poruke i pokreće benchmark testove.

Pregled najvažnijih REST krajnjih tačaka prikazan je u Tabelama 3.5–3.10.

Tabela 3.5: REST krajnje tačke za autentifikaciju korisnika

Metoda	**Putanja**	**Opis**	**Pristup**
POST	`/api/auth/register`	Registracija novog korisnika u sistemu.	Javno
POST	`/api/auth/login`	Provera kredencijala korisnika i izdavanje JWT tokena.	Javno

Tabela 3.6: REST krajnje tačke za upravljanje korisnicima

Metoda	**Putanja**	**Opis**	**Pristup**
POST	`/api/users`	Administratorsko kreiranje novog korisnika.	ADMIN
GET	`/api/users`	Preuzimanje liste svih registrovanih korisnika.	ADMIN

Tabela 3.7: REST krajnje tačke za upravljanje događajima

Metoda	**Putanja**	**Opis**	**Pristup**
POST	`/api/events`	Pravljenje novog događaja.	Autentifikovan
GET	`/api/events`	Preuzimanje liste svih događaja.	Autentifikovan
GET	`/api/events/{id}`	Preuzimanje podataka o izabranom događaju.	Autentifikovan
PUT	`/api/events/{id}`	Izmena podataka postojećeg događaja.	Autentifikovan
DELETE	`/api/events/{id}`	Brisanje izabranog događaja.	Autentifikovan

Tabela 3.8: REST krajnje tačke za upravljanje prijavama

Metoda	**Putanja**	**Opis**	**Pristup**
POST	`/api/events/{eventId}/registrations/optimistic`	Prijava korisnika uz primenu optimističkog zaključavanja.	Autentifikovan
POST	`/api/events/{eventId}/registrations/pessimistic`	Prijava korisnika uz primenu pesimističkog zaključavanja.	Autentifikovan
GET	`/api/events/{eventId}/registrations`	Preuzimanje prijava za određeni događaj.	Autentifikovan
DELETE	`/api/events/{eventId}/registrations`	Otkazivanje prijave korisnika na događaj.	Autentifikovan

Tabela 3.9: REST krajnje tačke za razmenu poruka

Metoda	**Putanja**	**Opis**	**Pristup**
GET	`/api/events/{eventId}/chat`	Preuzimanje poruka koje su dostupne prijavljenom korisniku u okviru događaja.	Autentifikovan
POST	`/api/events/{eventId}/chat`	Slanje javne poruke ili privatne poruke organizatoru događaja.	Autentifikovan

Tabela 3.10: REST krajnje tačke benchmark modula

Metoda	Putanja	Opis	Pristup
POST	`/api/benchmark/run`	Pokretanje benchmark testa za izabrani režim zaključavanja.	ADMIN
POST	`/api/benchmark/compare`	Pokretanje uporednog testa optimističkog i pesimističkog zaključavanja.	ADMIN

Javno dostupne krajnje tačke ograničene su na registraciju i prijavu korisnika. Nakon uspešne prijave serverska aplikacija izdaje JWT token, koji klijentska aplikacija prosleđuje u zaglavlju svakog narednog zaštićenog zahteva.

Krajnje tačke za upravljanje korisnicima i pokretanje benchmark testova dostupne su isključivo administratorima. Ostale krajnje tačke zahtevaju autentifikovanog korisnika.

3.4 Projektovanje bezbednosne arhitekture

Bezbednost predstavlja jedan od najvažnijih aspekata razvijene aplikacije, budući da sistem upravlja korisničkim nalozima, prijavama na događaje i komunikacijom između učesnika. Iz tog razloga implementiran je mehanizam za autentifikaciju i autorizaciju zasnovan na okviru Spring Security i JWT (JSON Web Token) tehnologiji.

3.4.1 Autentifikacija korisnika

Proces autentifikacije započinje prijavom korisnika na sistem unosom adrese elektronske pošte i lozinke. Nakon prijema zahteva, serverska aplikacija proverava ispravnost prosleđenih kredencijala korišćenjem okvira Spring Security [17].

U slučaju uspešne autentifikacije generiše se JWT token koji se koristi za autentifikaciju narednih zahteva. Način generisanja, prosleđivanja i validacije tokena detaljnije je opisan u nastavku.

3.4.2 Autorizacija i korisničke uloge

Nakon uspešne autentifikacije, sistem vrši autorizaciju korisnika kako bi odredio kojim funkcionalnostima može pristupiti. Autorizacija je implementirana korišćenjem korisničkih uloga definisanih u okviru Spring Security me-

hanizma. Na osnovu uloge svakom korisniku dodeljuju se odgovarajuća prava pristupa resursima sistema.

Razvijena aplikacija podržava tri osnovne korisničke uloge: administrator, organizator i učesnik. Svaka od ovih uloga ima jasno definisana ovlašćenja, čime se obezbeđuje bezbedan pristup funkcionalnostima sistema i sprečava neovlašćeno izvršavanje operacija.

Pregled korisničkih uloga i njihovih osnovnih ovlašćenja prikazan je u Tabeli 3.11.

Tabela 3.11: Korisničke uloge i ovlašćenja

Uloga	**Ovlašćenja**
ADMIN	Upravljanje korisnicima sistema, pregled svih događaja, pristup benchmark modulu i administrativnim funkcionalnostima aplikacije.
ORGANIZER	Pravljenje, izmena i brisanje sopstvenih događaja, pregled prijava učesnika, upravljanje događajima, razmena javnih poruka i prijem privatnih poruka od učesnika.
PARTICIPANT	Pregled dostupnih događaja, prijava i odjava sa događaja, razmena javnih poruka u okviru događaja i slanje privatnih poruka organizatoru.

Autorizacija se izvršava na osnovu korisničke uloge sadržane u JWT tokenu. Prilikom obrade svakog zahteva Spring Security proverava identitet korisnika i njegovu ulogu, nakon čega dozvoljava ili odbija pristup traženom resursu. Na ovaj način obezbeđuje se da svaka funkcionalnost sistema bude dostupna isključivo korisnicima koji imaju odgovarajuća ovlašćenja [17].

3.4.3 JWT mehanizam

Za autentifikaciju korisnika u razvijenoj aplikaciji primenjen je JWT (JSON Web Token) mehanizam. Nakon uspešne prijave korisnika, serverska aplikacija generiše token koji predstavlja dokaz uspešne autentifikacije i koristi se prilikom svih narednih zahteva upućenih REST servisima.

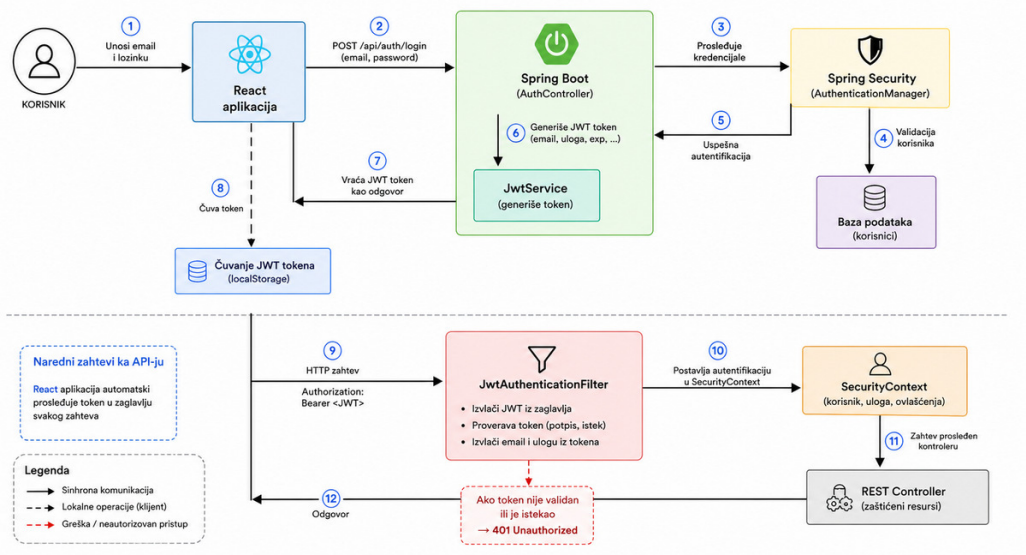
Prilikom generisanja tokena u njega se smeštaju podaci o identitetu korisnika, pre svega adresa elektronske pošte, kao i njegova korisnička uloga. Token je digitalno potpisan tajnim ključem, čime se obezbeđuje njegova autentičnost i sprečava neovlašćena izmena sadržaja [6].

Klijentska aplikacija čuva dobijeni JWT token i prosleđuje ga u HTTP zaglavlju svakog narednog zahteva korišćenjem Authorization zaglavlja i Bearer

šeme. Na ovaj način nije potrebno da korisnik ponovo unosi svoje kredencijale prilikom svake komunikacije sa serverskom aplikacijom.

Na ovaj način ostvarena je bezbedna autentifikacija bez čuvanja sesije na serverskoj strani, što predstavlja jednu od osnovnih karakteristika REST arhitekture [3, 6]. Primena JWT mehanizma omogućava jednostavno skaliranje sistema, smanjuje opterećenje servera i obezbeđuje efikasnu zaštitu REST servisa.

Tok autentifikacije, od slanja kredencijala do pristupa zaštićenim resursima korišćenjem JWT tokena, prikazan je na Slici 3.4.



Slika 3.4: Proces autentifikacije korišćenjem JWT mehanizma

3.4.4 Zaštita REST servisa

Zaštita REST servisa realizovana je korišćenjem okvira Spring Security. Bezbednosna pravila centralizovano su definisana u konfiguracionoj klasi *SecurityConfig*, koja određuje koje putanje su javno dostupne, koje zahtevaju autentifikaciju i koje su ograničene na korisnike sa određenom ulogom [17].

Krajnje tačke namenjene registraciji i prijavi korisnika, koje pripadaju putanji `/api/auth/**`, dostupne su bez prethodne autentifikacije. Ovakav pristup je neophodan kako bi novi korisnici mogli da prave nalog, a postojeći korisnici da se prijave u sistem i dobiju JWT token. Ostale REST krajnje tačke zahtevaju da korisnik bude uspešno autentifikovan.

Prilikom svakog zahteva ka zaštićenom resursu klijentska aplikacija prosleđuje JWT token u HTTP zaglavlju *Authorization*, korišćenjem Bearer šeme.

Zahtev zatim prolazi kroz komponentu *JwtAuthenticationFilter*, koja izdvaja token, proverava njegovu ispravnost i utvrđuje identitet korisnika. Ukoliko je token validan, podaci o autentifikovanom korisniku smeštaju se u objekat *SecurityContext*, nakon čega se zahtev prosleđuje odgovarajućem kontroleru.

U slučaju da token nije prosleđen, nije ispravan ili mu je istekao period važenja, korisniku se onemogućava pristup zaštićenom resursu. Serverska aplikacija tada vraća odgovarajući HTTP status, najčešće `401 Unauthorized`. Kada je korisnik uspešno autentifikovan, ali nema odgovarajuća ovlašćenja za izvršavanje određene operacije, vraća se status `403 Forbidden`.

Pored provere autentifikacije, određene krajnje tačke zaštićene su i na osnovu korisničkih uloga. Krajnje tačke za upravljanje korisnicima dostupne su administratorima, kao i funkcionalnosti benchmark modula. Ostale funkcionalnosti dostupne su autentifikovanim korisnicima, dok se dodatna poslovna pravila, poput provere vlasništva nad događajem i vidljivosti privatnih poruka, izvršavaju u servisnom sloju aplikacije.

Primena ovakve bezbednosne arhitekture omogućava centralizovanu kontrolu pristupa, razdvajanje autentifikacije od poslovne logike i zaštitu podataka od neovlašćenog korišćenja. Istovremeno, korišćenjem JWT tokena očuvana je priroda REST komunikacije bez čuvanja stanja [17].

3.5 Projektovanje komunikacije u realnom vremenu

Jedna od ključnih funkcionalnosti razvijene aplikacije jeste razmena poruka između učesnika i organizatora događaja u realnom vremenu. Za realizaciju ove funkcionalnosti korišćen je WebSocket protokol u kombinaciji sa STOMP (Simple Text Oriented Messaging Protocol) komunikacionim modelom [20].

Primena WebSocket komunikacije omogućava trenutno prikazivanje novih poruka učesnicima događaja bez potrebe za periodičnim slanjem HTTP zahteva.

3.5.1 Arhitektura komunikacije

Komunikacija u realnom vremenu zasniva se na modelu klijent–server. Nakon prijavljivanja na sistem, klijentska aplikacija razvijena korišćenjem biblioteke React uspostavlja WebSocket konekciju sa serverskom aplikacijom. Uspostavljena konekcija ostaje aktivna tokom rada aplikacije, čime se omogućava trenutno slanje i prijem poruka.

Serverski deo aplikacije koristi Spring WebSocket modul za obradu dolaznih poruka i njihovo prosleđivanje odgovarajućim korisnicima. Svaka poruka pripada konkretnom događaju, dok se distribucija poruka u realnom vremenu vrši korišćenjem posebnih STOMP tema (engl. *topics*) povezanih sa pojedinačnim događajima. Klijentska aplikacija se pretplaćuje na temu odgovarajućeg događaja, čime prima poruke i obaveštenja koja se odnose upravo na taj događaj.

Na ovaj način svaki događaj poseduje sopstveni kanal za komunikaciju, čime je omogućeno da korisnici primaju isključivo poruke koje se odnose na događaj u kojem učestvuju [20].

3.5.2 WebSocket konfiguracija

WebSocket komunikacija konfigurisana je korišćenjem klase *WebSocketConfig*. U okviru konfiguracije definisana je krajnja tačka za uspostavljanje WebSocket veze, kao i prefiksi koji određuju način razmene poruka između klijenta i servera.

Klijentska aplikacija uspostavlja vezu sa serverom preko definisane WebSocket krajnje tačke, nakon čega se pretplaćuje na odgovarajući komunikacioni kanal. Kada server primi novu poruku, ona se prosleđuje svim korisnicima koji su pretplaćeni na kanal odgovarajućeg događaja.

Ovako organizovana komunikacija omogućava jednostavno proširenje sistema i efikasno rukovanje većim brojem istovremenih korisnika [20].

3.5.3 Tok razmene poruka

Tok razmene poruka započinje kada korisnik unese tekst poruke u korisničkom interfejsu i odabere opciju za njeno slanje. Klijentska aplikacija razvijena korišćenjem biblioteke React formira zahtev koji sadrži identifikator događaja, sadržaj poruke i informacije o njenom tipu, nakon čega ga prosleđuje serverskoj aplikaciji.

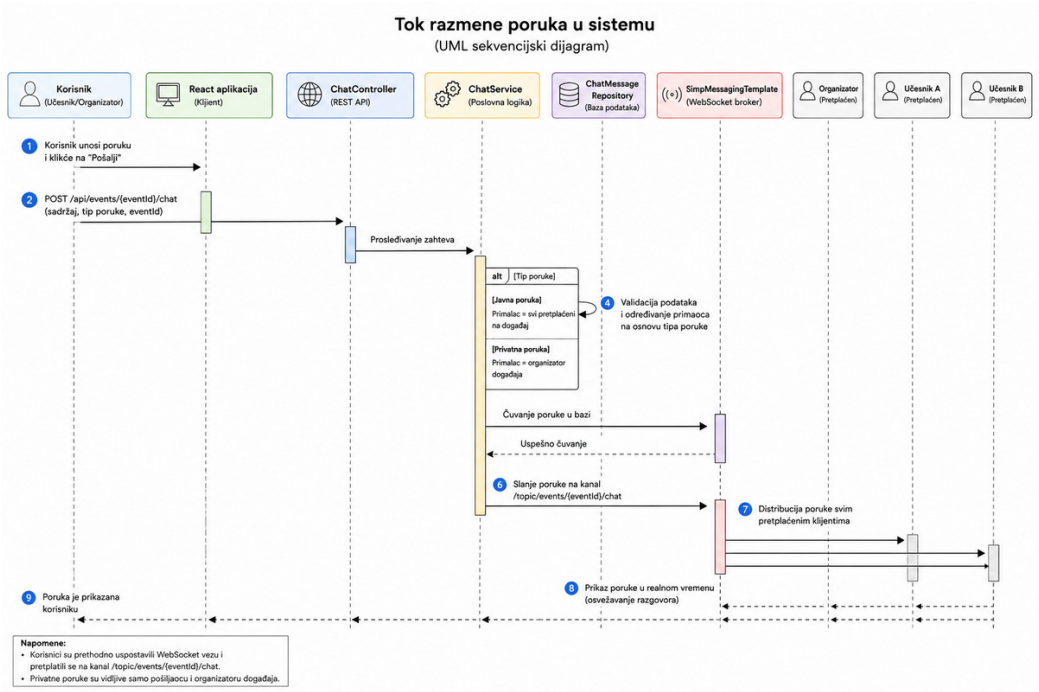
Po prijemu zahteva, serverska aplikacija proverava identitet korisnika na osnovu JWT tokena i određuje pošiljaoca poruke. Zatim se vrši validacija podataka i određuje da li je u pitanju javna ili privatna poruka. Kod privatnih poruka kao primalac se automatski postavlja organizator događaja.

Nakon uspešne obrade poruka se čuva u bazi podataka, čime se obezbeđuje njena trajna dostupnost i mogućnost kasnijeg pregleda istorije komunikacije. Nakon toga serverska aplikacija korišćenjem komponente *SimpMessagingTemplate* prosleđuje obaveštenje svim klijentima koji su pretplaćeni na komunikacioni kanal odgovarajućeg događaja.

Klijentske aplikacije koje su pretplaćene na odgovarajući WebSocket kanal trenutno primaju novu poruku ili obaveštenje o njenom slanju, nakon čega osvežavaju prikaz razgovora bez potrebe za ponovnim učitavanjem stranice ili slanjem dodatnih HTTP zahteva. Na ovaj način ostvaruje se komunikacija u realnom vremenu između svih učesnika događaja.

Kod privatnih poruka dodatno se primenjuju pravila vidljivosti, tako da sadržaj poruke mogu da vide isključivo njen pošiljalac i organizator događaja. Time se obezbeđuje poverljivost komunikacije uz zadržavanje jedinstvenog sistema za razmenu poruka.

Opisani tok razmene poruka između klijentske aplikacije, serverskog dela sistema i SUBP prikazan je na Slici 3.5.



Slika 3.5: Tok razmene poruka u sistemu

3.6 Projektovanje benchmark modula

Pored osnovnih funkcionalnosti za upravljanje događajima i komunikaciju u realnom vremenu, razvijena aplikacija poseduje i poseban benchmark modul namenjen ispitivanju ponašanja sistema pri velikom broju istovremenih prijava na događaj. Cilj ovog modula jeste analiza performansi i pouzdanosti dva pristupa konkurentnom pristupu podacima: optimističkog i pesimističkog

zaključavanja.

Benchmark modul omogućava automatsko pokretanje većeg broja paralelnih korisničkih zahteva koji simuliraju prijavljivanje učesnika na isti događaj. Tokom izvršavanja prate se ukupno vreme izvršavanja, broj uspešno izvršenih i odbijenih prijava, kao i broj konflikata nastalih pri primeni optimističkog zaključavanja. Na osnovu prikupljenih rezultata moguće je uporediti efikasnost oba mehanizma zaključavanja u različitim uslovima opterećenja [23, 22].

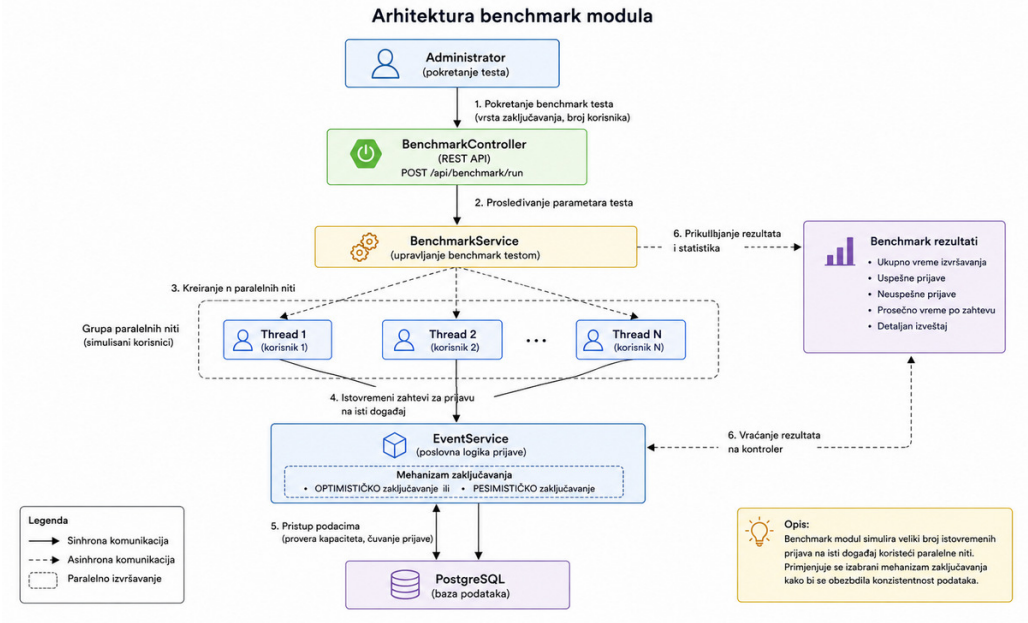
Modul je implementiran kao zaseban deo serverske aplikacije i dostupan je preko posebnih REST krajnjih tačaka namenjenih administratorima sistema. Na ovaj način omogućeno je jednostavno pokretanje eksperimenata bez uticaja na svakodnevni rad korisnika aplikacije.

3.6.1 Arhitektura benchmark modula

Benchmark modul sastoji se od kontrolera, servisnog sloja i pomoćnih komponenti koje generišu veliki broj paralelnih zahteva prema aplikaciji. Administrator pokreće benchmark test korišćenjem odgovarajuće REST krajnje tačke, nakon čega serverska aplikacija pravi definisani broj konkurentnih niti koje istovremeno pokušavaju da izvrše prijavu korisnika na isti događaj.

Svaka nit predstavlja jednog simuliranog korisnika. Tokom izvršavanja testa servisni sloj primenjuje odabrani mehanizam zaključavanja, odnosno optimističko ili pesimističko zaključavanje, kako bi obezbedio konzistentnost podataka prilikom istovremenog pristupa istom događaju [23].

Osnovne komponente benchmark modula i njihov međusobni odnos prikazani su na Slici 3.6.



Slika 3.6: Arhitektura benchmark modula

3.6.2 Scenario izvršavanja benchmark testa

Izvršavanje benchmark testa započinje izborom vrste zaključavanja i definisanjem broja paralelnih korisnika koji će učestvovati u testu. Nakon pokretanja testa serverska aplikacija pravi odgovarajući broj niti koje gotovo istovremeno pokušavaju da izvrše prijavu na isti događaj.

Tokom izvršavanja svaka nit prolazi kroz identičan proces prijavljivanja. U zavisnosti od odabranog mehanizma zaključavanja deo zahteva može biti privremeno blokiran ili ponovo izvršen zbog konflikta prilikom ažuriranja podataka.

Po završetku svih niti benchmark modul izračunava ukupno vreme izvršavanja, broj uspešnih i neuspešnih prijava, kao i broj konflikata nastalih pri optimističkom zaključavanju. Kod ponovljenog benchmark testiranja rezultati pojedinačnih izvršavanja agregiraju se izračunavanjem medijane ukupnog vremena izvršavanja, interkvartilnog raspona (IQR) ukupnog vremena i medijane broja optimističkih konflikata. Dobijeni rezultati predstavljaju osnovu za analizu performansi i poređenje optimističkog i pesimističkog zaključavanja, koje je detaljno predstavljeno u Poglavlju 6.

Glava 4

Implementacija sistema

U ovom poglavlju prikazana je implementacija razvijene veb aplikacije za upravljanje događajima i komunikaciju u realnom vremenu. Na osnovu arhitekture i modela sistema predstavljenih u prethodnom poglavlju realizovane su funkcionalnosti serverskog i klijentskog dela aplikacije, kao i mehanizmi za bezbednu autentifikaciju korisnika, komunikaciju u realnom vremenu i upravljanje konkurentnim pristupom podacima.

Serverska aplikacija implementirana je korišćenjem okvira Spring Boot i programskog jezika Java. Organizovana je u više logičkih slojeva koji obuhvataju kontrolere, servisni sloj, repozitorijume i modele podataka. Za trajno čuvanje podataka korišćena je baza podataka PostgreSQL uz primenu biblioteke Spring Data JPA i okvira Hibernate ORM [25, 23].

Klijentski deo sistema razvijen je korišćenjem biblioteke React i omogućava korisnicima jednostavno korišćenje svih funkcionalnosti aplikacije putem savremenog veb interfejsa. Komunikacija između klijenta i servera ostvarena je REST servisima, dok je razmena poruka u realnom vremenu realizovana pomoću WebSocket tehnologije i protokola STOMP.

U nastavku poglavlja opisani su najvažniji aspekti implementacije pojedinačnih komponenti sistema, uz prikaz karakterističnih delova izvornog koda i objašnjenje njihove uloge u radu aplikacije.

4.1 Implementacija serverske aplikacije

Serverski deo razvijene aplikacije implementiran je korišćenjem okvira Spring Boot. Korišćenjem okvira Spring Boot omogućena je jednostavna konfiguracija projekta, integracija različitih Spring modula i razvoj REST servisa uz minimalnu količinu konfiguracionog koda.

Aplikacija je organizovana prema višeslojnoj arhitekturi, pri čemu je sva-

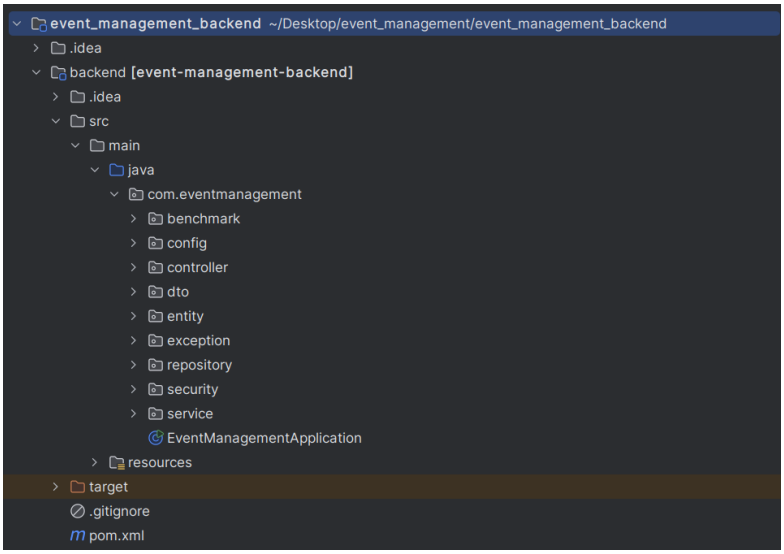
ki sloj odgovoran za jasno definisan skup funkcionalnosti. Takva organizacija doprinosi modularnosti sistema, jednostavnijem održavanju i lakšem testiranju pojedinačnih komponenti [1].

Serverski deo sistema sastoji se od modela podataka, repozitorijuma, servisnog sloja, REST kontrolera, bezbednosnih komponenti i konfiguracionih klasa. Komunikacija između ovih slojeva ostvarena je primenom mehanizma ubrizgavanja zavisnosti.

4.1.1 Organizacija projekta

Izvorni kod serverske aplikacije organizovan je u više paketa, pri čemu svaki paket predstavlja jednu funkcionalnu celinu sistema. Ovakva organizacija omogućava jasno razdvajanje odgovornosti između pojedinih delova aplikacije i olakšava njeno dalje proširivanje i održavanje.

Najvažniji paketi razvijene aplikacije prikazani su na Slici 4.1. Paket *controller* sadrži REST kontrolere koji obrađuju HTTP zahteve korisnika. Paket *service* implementira poslovnu logiku sistema, dok paket *repository* omogućava komunikaciju sa sistemom za upravljanje bazom podataka korišćenjem biblioteke Spring Data JPA [25].



Slika 4.1: Organizacija paketa serverske aplikacije

Modeli podataka nalaze se u paketu *entity*, dok paket *dto* sadrži objekte za prenos podataka (engl. Data Transfer Object, DTO) namenjene razmeni podataka između klijentske i serverske aplikacije. Bezbednosne komponente, uključujući JWT autentifikaciju i konfiguraciju okvira Spring Security, nalaze

se u paketu *security*. Konfiguracija WebSocket komunikacije izdvojena je u poseban paket *websocket*, dok se konfiguracione klase sistema nalaze u paketu *config*.

4.1.2 Model podataka

Model podataka implementiran je korišćenjem Java klasa označenih anotacijom `@Entity`. Svaka klasa predstavlja jednu tabelu u PostgreSQL bazi podataka, dok su međusobne veze između entiteta realizovane korišćenjem odgovarajućih JPA anotacija, kao što su `@OneToMany`, `@ManyToOne` i `@JoinColumn`. Na ovaj način ostvareno je objektno-relaciono mapiranje između Java objekata i relacionih tabela baze podataka [23, 25].

U razvijenoj aplikaciji implementirani su entiteti *User*, *Event*, *Registration* i *ChatMessage*. Pored osnovnih atributa, entitet *Event* koristi anotaciju `@Version`, koja omogućava realizaciju optimističkog zaključavanja prilikom konkurentnih prijava na događaje. Ovaj mehanizam automatski vodi računa o verzijama zapisa i sprečava nekonzistentno ažuriranje podataka [23].

Primer dela implementacije entiteta `Event` prikazan je u Listingu 4.1. Poseban značaj za realizaciju optimističkog zaključavanja ima atribut `version`, označen anotacijom `@Version`.

```
1  public class Event {
2
3      @Id
4      @GeneratedValue(strategy = GenerationType.IDENTITY)
5      private Long id;
6
7      @Column(nullable = false)
8      private String title;
9
10     @Column(nullable = false)
11     private Integer capacity;
12
13     @Column(nullable = false)
14     private Integer availableSpots;
15
16     @ManyToOne(fetch = FetchType.LAZY)
17     @JoinColumn(name = "organizer_id")
18     private User organizer;
19
20     @Version
21     private Long version;
22 }
```

Listing 4.1: Deo implementacije entiteta Event

4.1.3 REST servisi

Komunikacija između klijentske i serverske aplikacije implementirana je korišćenjem REST arhitekture. Implementacija REST servisa realizovana je korišćenjem anotacija Spring Web modula, kao što su `@RestController`, `@RequestMapping`, `@GetMapping`, `@PostMapping`, `@PutMapping`, `@PatchMapping` i `@DeleteMapping`.

Svaki kontroler odgovoran je za određenu funkcionalnu celinu sistema. Zahtevi koje klijentska aplikacija šalje prosleđuju se odgovarajućem servisnom sloju, u kojem se izvršava poslovna logika aplikacije. Nakon obrade zahteva rezultat se vraća klijentskoj aplikaciji u JSON formatu.

Organizacija REST kontrolera i njihove odgovornosti detaljnije su opisane u Odeljku 3.3.1.

4.1.4 Implementacija JWT autentifikacije i autorizacije

Bezbednost aplikacije implementirana je korišćenjem Spring Security okruženja i JSON Web Token (JWT) tehnologije [17, 6]. Nakon uspešne prijave korisnika generiše se JWT token koji sadrži identitet korisnika i njegovu ulogu u sistemu.

Prilikom svakog narednog zahteva klijentska aplikacija prosleđuje token u HTTP zaglavlju *Authorization*. JWT filter proverava validnost tokena i iz njega izdvaja identitet korisnika, nakon čega se u okviru Spring Security konteksta uspostavlja autentifikacija i proveravaju odgovarajuća ovlašćenja za pristup traženom resursu.

U okviru razvijene aplikacije implementirane su tri korisničke uloge:

- administrator,
- organizator,
- učesnik.

Za generisanje i validaciju JWT tokena implementirana je posebna servisna klasa *JwtService*. Pored toga, razvijen je JWT filter koji se izvršava pre obrade svakog HTTP zahteva i automatski autentifikuje korisnika ukoliko je prosleđeni token validan.

Generisanje JWT tokena prikazano je u Listingu 4.2, dok je postupak validacije tokena prikazan u Listingu 4.3.

```
1 public String generateToken(String email, String role) {
2     return Jwts.builder()
3             .subject(email)
```

```
4               .claim("role", role)
5               .issuedAt(new Date())
6               .expiration(new Date(
7                       System.currentTimeMillis() + expiration))
8               .signWith(getSigningKey())
9               .compact();
10  }
```

Listing 4.2: Generisanje JWT tokena

```
1   public boolean isTokenValid(String token) {
2       try {
3           return extractClaims(token)
4                   .getExpiration()
5                   .after(new Date());
6       } catch (Exception e) {
7           return false;
8       }
9   }
```

Listing 4.3: Validacija JWT tokena

4.1.5 Implementacija komunikacije u realnom vremenu

Komunikacija u realnom vremenu implementirana je korišćenjem WebSocket protokola i STOMP komunikacionog modela.

Konfiguracija WebSocket komunikacije realizovana je posebnom konfiguracionom klasom u kojoj su definisane krajnje tačke za uspostavljanje veze i destinacije preko kojih se razmenjuju poruke. Nakon uspostavljanja konekcije klijentska aplikacija se pretplaćuje na odgovarajući kanal koji pripada izabranom događaju.

Kada korisnik pošalje novu poruku, ona se najpre čuva u bazi podataka, nakon čega se pomoću klase *SimpMessagingTemplate* prosleđuje svim trenutno povezanim korisnicima. Na ovaj način svi učesnici događaja dobijaju novu poruku bez potrebe za osvežavanjem stranice [20].

Pored javnih poruka implementirana je i mogućnost slanja privatnih poruka organizatoru događaja. Serverska aplikacija proverava prava pristupa svakoj poruci i obezbeđuje da privatne poruke budu vidljive isključivo njihovom pošiljaocu i organizatoru događaja.

4.1.6 Implementacija benchmark modula

Jedna od specifičnih funkcionalnosti razvijene aplikacije predstavlja benchmark modul, čija je svrha poređenje performansi optimističkog i pesimističkog

zaključavanja prilikom istovremenih prijava korisnika na isti događaj. Modul je implementiran kao poseban REST servis kojem mogu pristupiti isključivo korisnici sa administratorskom ulogom.

Implementacija benchmark modula zasniva se na simulaciji većeg broja istovremenih korisničkih zahteva. Prilikom pokretanja testa aplikacija kreira definisani broj niti, pri čemu svaka nit pokušava da izvrši prijavu na isti događaj. Na ovaj način simuliraju se uslovi koji mogu nastati kada veliki broj korisnika istovremeno pokušava da rezerviše ograničen broj mesta.

Za realizaciju optimističkog zaključavanja entitet *Event* sadrži atribut označen anotacijom `@Version`. Prilikom konkurentne izmene Hibernate proverava verziju entiteta i detektuje situaciju u kojoj je druga transakcija u međuvremenu izmenila isti podatak. U benchmark modulu takvi konflikti rešavaju se mehanizmom ponovnog pokušaja.

```
1  private void registerOptimisticWithRetry(
2          Long eventId,
3          Long userId,
4          AtomicInteger optimisticConflicts) {
5
6      for (int attempt = 1;
7           attempt <= MAX_OPTIMISTIC_ATTEMPTS;
8           attempt++) {
9
10         try {
11             registrationService.registerOptimistic(
12                     eventId, userId);
13             return;
14
15         } catch (RuntimeException exception) {
16
17             if (!isOptimisticConflict(exception)) {
18                 throw exception;
19             }
20
21             optimisticConflicts.incrementAndGet();
22
23             if (attempt == MAX_OPTIMISTIC_ATTEMPTS) {
24                 throw exception;
25             }
26
27             long delayMs =
28                     ThreadLocalRandom.current().nextLong(
29                             MIN_RETRY_DELAY_MS,
30                             MAX_RETRY_DELAY_MS + 1);
31
32             LockSupport.parkNanos(
33                     TimeUnit.MILLISECONDS.toNanos(delayMs));
```

```
34
35              if (Thread.currentThread().isInterrupted()) {
36                  throw new IllegalStateException(
37                          "Optimistic registration retry was interrupted",
38                          exception);
39              }
40          }
41      }
42
43      throw new IllegalStateException(
44              "Optimistic registration failed unexpectedly");
45  }
```

Listing 4.4: Mehanizam ponovnog pokušaja kod optimističkog zaključavanja

Kao što je prikazano u Listingu 4.4, ponovno izvršavanje primenjuje se samo kada je detektovan konflikt optimističkog zaključavanja. Broj pokušaja ograničen je na 30, dok se između uzastopnih pokušaja primenjuje kratko randomizovano čekanje u intervalu od 0 do 2 ms. Implementacija ne koristi eksponencijalni *backoff*, odnosno vreme čekanja se ne povećava sa brojem neuspešnih pokušaja.

Pesimističko zaključavanje realizovano je na nivou repozitorijuma korišćenjem režima `PESSIMISTIC_WRITE`. Metoda za učitavanje događaja uz pesimističko zaključavanje prikazana je u Listingu 4.5.

```
1  @Lock(LockModeType.PESSIMISTIC_WRITE)
2  @Query("select e from Event e where e.id = :id")
3  Optional<Event> findByIdForUpdate(@Param("id") Long id);
```

Listing 4.5: Upit sa pesimističkim zaključavanjem

Prilikom poziva metode `findByIdForUpdate` odgovarajući zapis događaja učitava se uz pesimističko zaključavanje za upis. Time se sprečava da konkurentne transakcije istovremeno menjaju isti zapis, dok ostali zahtevi moraju da sačekaju oslobađanje zaključavanja.

Upotreba ove metode prilikom registracije korisnika prikazana je u Listingu 4.6.

```
1  @Transactional
2  public RegistrationResponse registerPessimistic(
3          Long eventId,
4          Long userId) {
5
6      Event event = eventRepository.findByIdForUpdate(eventId)
7              .orElseThrow(
8                      () -> new RuntimeException("Event not found"));
9
10     return registerInternal(event, userId);
```

```
11 }
```

Listing 4.6: Registracija korisnika uz pesimističko zaključavanje

Na ovaj način dva implementirana pristupa različito rešavaju konkurentni pristup istom događaju. Kod optimističkog pristupa konflikt se detektuje nakon konkurentne izmene i operacija se po potrebi ponavlja, dok se kod pesimističkog pristupa konkurentna izmena sprečava zaključavanjem odgovarajućeg zapisa u bazi podataka.

Za svaki izvršeni benchmark test beleže se ukupno vreme izvršavanja, broj uspešno registrovanih korisnika, broj korisnika na listi čekanja, broj konflikata nastalih pri optimističkom zaključavanju i konzistentnost konačnog stanja sistema. Benchmark modul omogućava i ponovljeno izvršavanje istog scenarija radi dobijanja reprezentativnijih rezultata. Kod ponovljenog izvršavanja izračunavaju se medijana ukupnog vremena izvršavanja, interkvartilni raspon (IQR) ukupnog vremena i medijana broja optimističkih konflikata. Detaljna metodologija testiranja, broj ponavljanja i analiza dobijenih rezultata predstavljeni su u Poglavlju 6.

Rezultati benchmark testova prikazuju se administratoru putem odgovarajućeg korisničkog interfejsa, gde je moguće izvršiti testove sa različitim brojem korisnika i kapacitetima događaja i uporediti ponašanje implementiranih mehanizama zaključavanja.

4.2 Implementacija klijentske aplikacije

Klijentski deo aplikacije razvijen je korišćenjem JavaScript biblioteke React, koja omogućava razvoj interaktivnih korisničkih interfejsa zasnovanih na komponentama. Korisnički interfejs sastoji se od većeg broja nezavisnih komponenti koje međusobno razmenjuju podatke korišćenjem React mehanizama za upravljanje stanjem i rutiranjem.

Za komunikaciju sa serverskom aplikacijom korišćena je biblioteka Axios, koja omogućava slanje HTTP zahteva REST servisima i obradu odgovora servera. Nakon uspešne autentifikacije JWT token se čuva u lokalnoj memoriji pregledača i automatski prosleđuje uz zahteve ka zaštićenim resursima. Ovakav pristup izabran je zbog jednostavnosti implementacije u okviru razvijenog prototipa. Međutim, čuvanje tokena u lokalnoj memoriji pregledača predstavlja bezbednosno ograničenje, jer token može biti dostupan zlonamernom JavaScript kodu u slučaju XSS napada. U produkcionom okruženju prikladnije rešenje bilo bi čuvanje autentifikacionih podataka u odgovarajuće konfigurisanom `HttpOnly` kolačiću, kojem JavaScript kod nema direktan pristup.

Korisnički interfejs prilagođen je različitim tipovima korisnika, tako da administrator, organizator i učesnik imaju pristup isključivo funkcionalnostima koje odgovaraju njihovoj ulozi u sistemu.

4.2.1 Organizacija React projekta

Klijentska aplikacija organizovana je prema komponentnoj arhitekturi, pri čemu svaka funkcionalna celina predstavlja zasebnu React komponentu. Na ovaj način omogućena je ponovna upotreba komponenti, jednostavnije održavanje izvornog koda i lakše proširivanje aplikacije novim funkcionalnostima.

Glavne komponente aplikacije predstavljaju stranice za prijavu i registraciju korisnika, pregled događaja, upravljanje događajima, pregled prijava učesnika, benchmark modul i sistem za razmenu poruka u realnom vremenu.

Navigacija između stranica implementirana je korišćenjem biblioteke React Router, koja omogućava definisanje javnih i zaštićenih ruta. Nakon uspešne prijave korisnik se automatski preusmerava na odgovarajuću početnu stranicu u skladu sa svojom ulogom u sistemu [26].

4.2.2 Implementacija autentifikacije korisnika

Autentifikacija korisnika realizovana je putem forme za prijavu koja od korisnika zahteva unos adrese elektronske pošte i lozinke. Nakon unosa podataka klijentska aplikacija šalje POST zahtev odgovarajućem REST servisu koristeći Axios biblioteku.

U slučaju uspešne autentifikacije serverska aplikacija vraća JWT token.

Na osnovu uloge korisnika određuje se koje stranice i funkcionalnosti će biti dostupne nakon prijave. Ovakav pristup omogućava jednostavnu implementaciju kontrole pristupa i sprečava neovlašćeno korišćenje administrativnih funkcionalnosti.

4.2.3 Implementacija upravljanja događajima

Upravljanje događajima realizovano je kroz skup React komponenti koje omogućavaju pregled, pravljenje, izmenu i brisanje događaja. Organizator može unositi osnovne informacije o događaju, uključujući naziv, opis, vreme održavanja, lokaciju i maksimalan broj učesnika.

Pregled svih događaja implementiran je kao dinamička tabela koja se automatski osvežava nakon svake izmene podataka. Za komunikaciju sa serverskom aplikacijom koriste se REST servisi koji vraćaju podatke u JSON formatu, dok se prikaz podataka ažurira promenom stanja React komponenti.

Učesnicima sistema omogućen je pregled dostupnih događaja, prijava na događaj, otkazivanje prijave i pregled sopstvenih prijava. Organizator može pregledati listu prijavljenih učesnika za događaje koje organizuje, dok administrator ima pristup svim događajima u sistemu.

4.2.4 Implementacija korisničkog interfejsa za komunikaciju

Sistem za razmenu poruka implementiran je kao posebna React komponenta koja omogućava komunikaciju između učesnika događaja i organizatora u realnom vremenu. Nakon otvaranja stranice događaja uspostavlja se WebSocket konekcija sa serverskom aplikacijom korišćenjem STOMP klijenta.

Sve pristigle poruke automatski se prikazuju korisnicima bez potrebe za osvežavanjem stranice. Pored javnih poruka implementirana je mogućnost slanja privatnih poruka organizatoru događaja, pri čemu se odgovarajuća ograničenja pristupa proveravaju na serverskoj strani.

Korisnički interfejs omogućava pregled istorije razgovora, unos novih poruka i trenutno prikazivanje svih promena koje nastaju tokom komunikacije između učesnika događaja.

4.3 Implementacija baze podataka

Komunikacija između aplikacije i baze podataka realizovana je korišćenjem biblioteke Spring Data JPA i okvira Hibernate ORM. Hibernate omogućava objektno-relaciono mapiranje (ORM), pri čemu se Java klase automatski mapiraju na odgovarajuće tabele baze podataka, dok se objekti aplikacije prevode u SQL upite bez potrebe za njihovim ručnim pisanjem [23, 25].

Pristup podacima realizovan je preko repozitorijuma koji nasleđuju interfejs *JpaRepository*. Na ovaj način omogućeno je automatsko izvršavanje osnovnih CRUD operacija, dok su složeniji upiti implementirani korišćenjem metoda izvedenih iz naziva ili prilagođenih JPQL upita [25].

Posebna pažnja posvećena je obezbeđivanju konzistentnosti podataka prilikom istovremenih prijava korisnika na isti događaj. U tu svrhu korišćeni su mehanizmi optimističkog i pesimističkog zaključavanja koje podržava Hibernate, čime je sprečeno prekoračenje maksimalnog broja učesnika i obezbeđena ispravnost podataka čak i u uslovima velikog broja konkurentnih zahteva [23, 22].

Implementacija baze podataka direktno prati logički model opisan u Poglavlju 3.2.1. Mapiranje tabela i njihovih međusobnih veza definisano je kroz JPA entitete i odgovarajuće anotacije, dok Hibernate ORM obezbeđuje

objektno-relaciono mapiranje i komunikaciju između aplikacionog modela i relacione baze podataka.

Glava 5

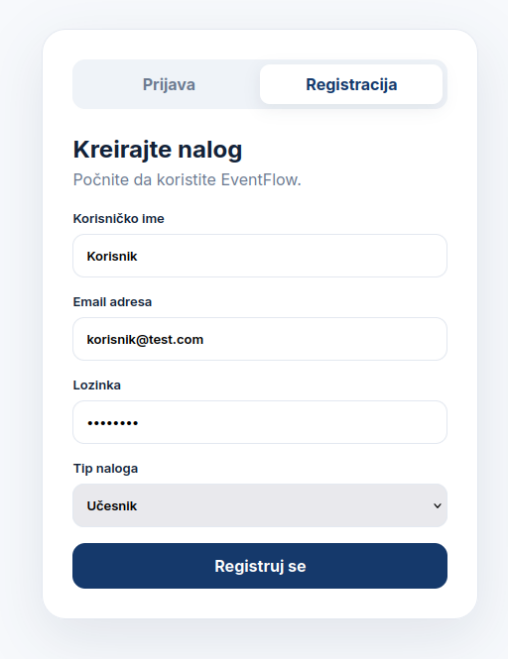
Izgled i funkcionalnosti aplikacije

U ovom poglavlju prikazane su najvažnije funkcionalnosti razvijene veb aplikacije iz perspektive korisnika sistema. Funkcionalnosti koje su korisniku dostupne zavise od njegove uloge, pri čemu sistem razlikuje učesnika, organizatora i administratora. U nastavku su predstavljene registracija i prijava korisnika, pregled i upravljanje događajima, prijava na događaje, pregled sopstvenih prijava, komunikacija u realnom vremenu i administrativne funkcionalnosti.

5.1 Registracija korisnika

Novom korisniku omogućeno je pravljenje korisničkog naloga putem forme za registraciju. Prilikom registracije korisnik unosi potrebne podatke, na osnovu kojih se kreira njegov nalog u sistemu. Uspešnom registracijom korisniku se dodeljuje uloga učesnika. Korisnik se zatim može prijaviti u aplikaciju i koristiti funkcionalnosti dostupne toj korisničkoj ulozi.

Forma za registraciju novog korisnika prikazana je na Slici 5.1.

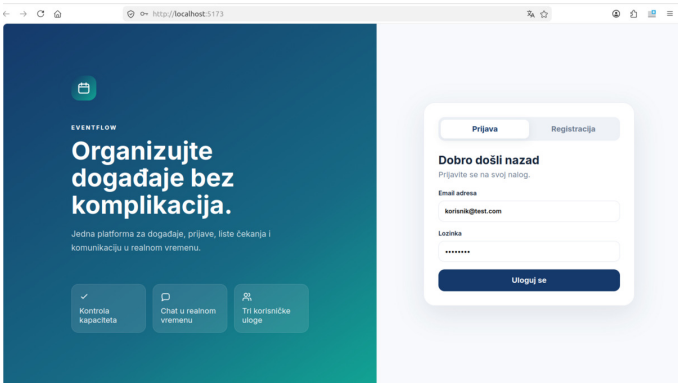


Slika 5.1: Forma za registraciju novog korisnika

5.2 Prijava korisnika

Pristup funkcionalnostima aplikacije omogućen je nakon uspešne prijave korisnika. Korisnik unosi adresu elektronske pošte i lozinku, nakon čega sistem proverava unete podatke i, u slučaju uspešne autentifikacije, omogućava pristup funkcionalnostima u skladu sa njegovom korisničkom ulogom.

Forma za prijavu korisnika prikazana je na Slici 5.2.



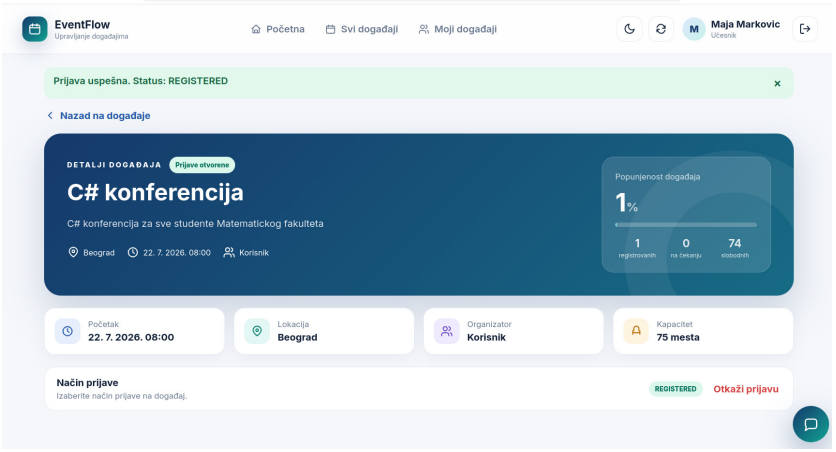
Slika 5.2: Forma za prijavu korisnika

5.3 Prijava na događaj

Učesnik se može prijaviti na događaj ukoliko postoje raspoloživa mesta. Nakon uspešne prijave broj slobodnih mesta automatski se umanjuje. Korisniku je omogućeno i otkazivanje prijave, pri čemu se oslobođeno mesto dodeljuje prvom korisniku sa liste čekanja, ukoliko takav korisnik postoji.

Kada je maksimalan broj učesnika dostignut, novi korisnici se smeštaju na listu čekanja. Na taj način sistem omogućava upravljanje prijavama uz poštovanje definisanog kapaciteta događaja.

Primer prijave korisnika na događaj prikazan je na Slici 5.3.

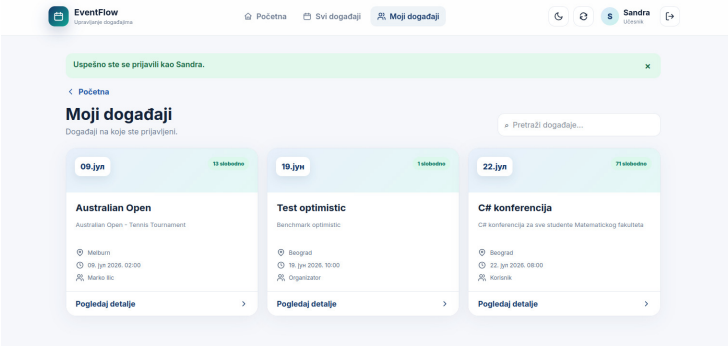


Slika 5.3: Prijava korisnika na događaj

5.4 Pregled prijavljenih događaja

Korisniku je omogućen pregled događaja na koje je prijavljen, čime na jednom mestu može pratiti svoje prijave i pristupiti detaljima odabranog događaja. Ovaj pregled olakšava korisniku upravljanje sopstvenim prijavama na događaje.

Pregled događaja na koje je korisnik prijavljen prikazan je na Slici 5.4.

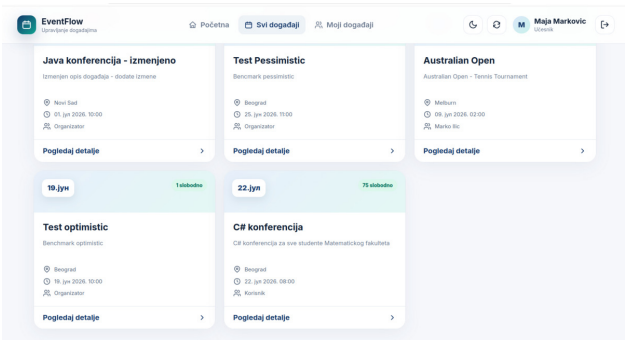


Slika 5.4: Pregled događaja na koje je korisnik prijavljen

5.5 Pregled događaja

Učesnicima je omogućen pregled objavljenih događaja, dok administrator ima pristup kompletnom pregledu događaja u sistemu. Pregled omogućava korisnicima da pristupe osnovnim informacijama o dostupnim događajima i odaberu događaj za koji žele da prikažu detaljnije informacije.

Prikaz liste događaja dat je na Slici 5.5.

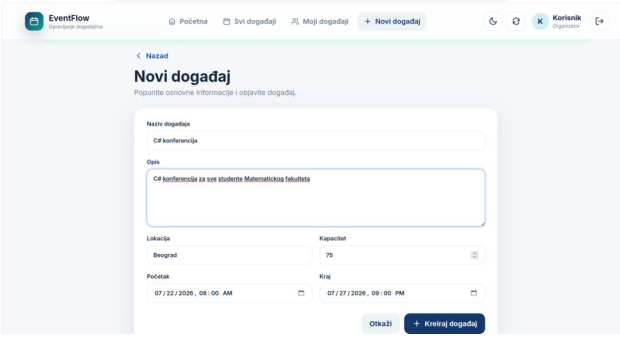


Slika 5.5: Lista događaja

5.6 Upravljanje događajima

Organizatoru su dostupne funkcionalnosti za pravljenje, izmenu i brisanje događaja. Prilikom kreiranja događaja organizator unosi osnovne podatke, kao što su naziv, opis, mesto održavanja, vreme početka i završetka, kao i maksimalan broj učesnika. Nakon kreiranja događaj postaje dostupan u pregledu događaja.

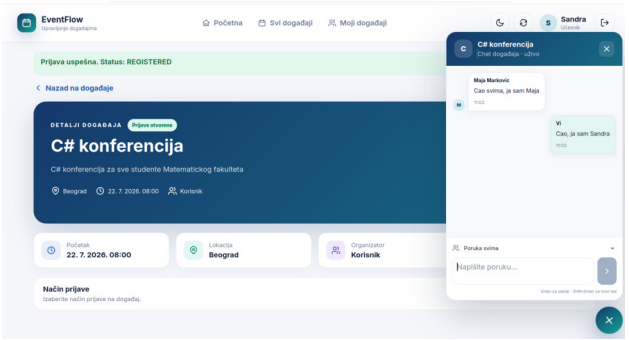
Forma za pravljenje novog događaja prikazana je na Slici 5.6.



Slika 5.6: Forma za pravljenje događaja

5.7 Komunikacija u realnom vremenu

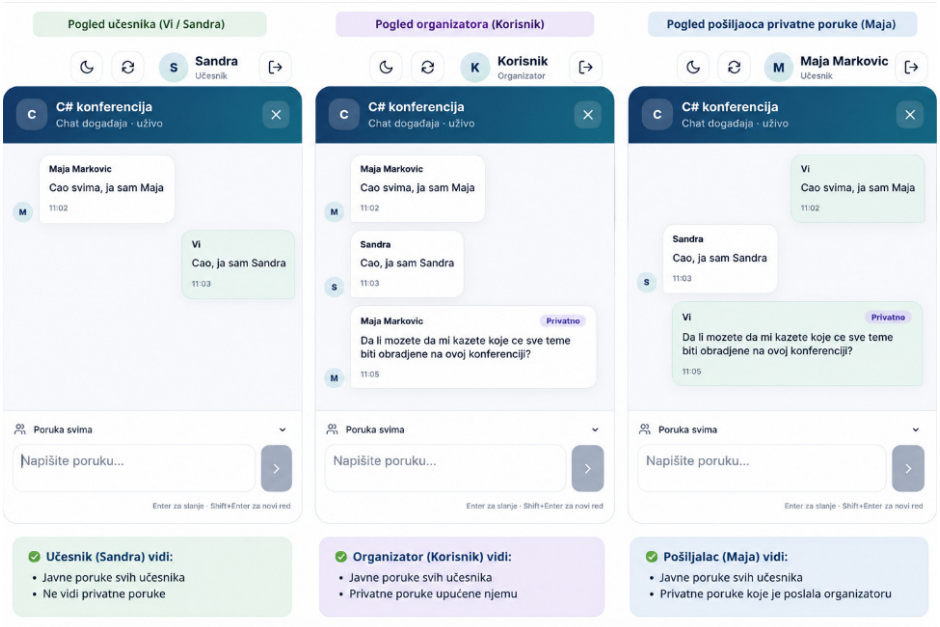
U okviru svakog događaja korisnicima je omogućena komunikacija u realnom vremenu. Učesnici mogu slati javne poruke koje su dostupne svim korisnicima povezanima sa događajem. Primer javne komunikacije prikazan je na Slici 5.7.



Slika 5.7: Razmena javnih poruka u okviru događaja

Pored javnih poruka, učesniku je omogućeno slanje privatne poruke organizatoru događaja. Privatna poruka dostupna je samo njenom pošiljaocu i organizatoru, dok ostali učesnici nemaju pristup njenom sadržaju.

Na Slici 5.8 prikazana je vidljivost poruka iz perspektive različitih korisnika. Levo je prikazan pogled učesnice Sandre, koja vidi javne poruke, ali nema pristup privatnoj poruci drugog učesnika. U sredini je prikazan pogled organizatora, kome su dostupne javne poruke i privatne poruke koje su mu upućene. Desno je prikazan pogled učesnice Maje, koja vidi javne poruke i privatnu poruku koju je uputila organizatoru.

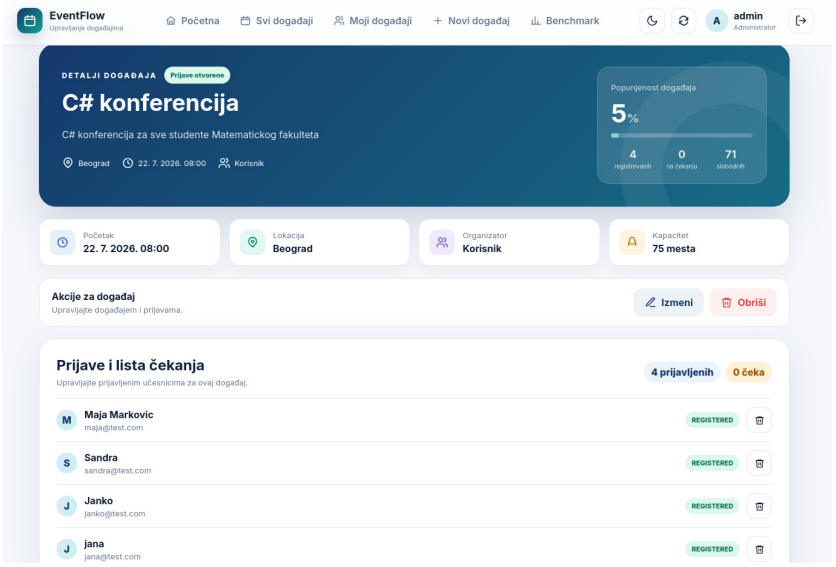


Slika 5.8: Vidljivost javnih i privatnih poruka u zavisnosti od korisnika

5.8 Administrativne funkcionalnosti

Administrator ima pristup detaljnom pregledu događaja i podacima o prijavljenim učesnicima. U okviru pregleda događaja prikazane su informacije o kapacitetu i popunjenosti događaja, broju prijavljenih učesnika i korisnika na listi čekanja. Administratoru su dostupne i funkcionalnosti za izmenu i brisanje događaja, kao i upravljanje prijavama učesnika.

Administrativni pregled detalja događaja i prijavljenih učesnika prikazan je na Slici 5.9.



Slika 5.9: Administrativni pregled događaja i prijavljenih učesnika

Prikazane funkcionalnosti obuhvataju osnovne načine interakcije korisnika sa razvijenom aplikacijom. Razdvajanjem funkcionalnosti prema korisničkim ulogama omogućeno je da učesnici, organizatori i administratori imaju pristup operacijama u skladu sa svojim ovlašćenjima. Pored upravljanja događajima i prijavama, aplikacija omogućava i komunikaciju u realnom vremenu, čime je podržana neposredna razmena informacija između učesnika i organizatora događaja.

Glava 6

Evaluacija performansi

U ovom poglavlju prikazana je evaluacija performansi razvijene aplikacije sa posebnim osvrtom na konkurentnu prijavu korisnika na događaje. Cilj evaluacije je poređenje optimističkog i pesimističkog mehanizma zaključavanja pri različitim nivoima konkurentnog opterećenja. U nastavku su opisani testno okruženje i metodologija benchmark testiranja, nakon čega su predstavljeni i analizirani dobijeni rezultati.

6.1 Testno okruženje

Benchmark testiranje sprovedeno je u lokalnom razvojnom okruženju. Serverska i klijentska aplikacija izvršavane su na istom računaru, dok je baza podataka PostgreSQL bila instalirana lokalno.

Serverski deo aplikacije razvijen je korišćenjem programskog jezika Java u verziji 21 i okvira Spring Boot u verziji 3.5.0. Klijentski deo implementiran je korišćenjem biblioteke React u verziji 19.2.7, dok je za skladištenje podataka korišćen sistem za upravljanje bazom podataka PostgreSQL u verziji 14.23. Sve komponente sistema izvršavane su na operativnom sistemu Ubuntu 22.04 LTS.

Osnovne karakteristike hardverskog i softverskog okruženja korišćenog tokom benchmark testiranja prikazane su u Tabeli 6.1.

Tabela 6.1: Konfiguracija testnog okruženja

Komponenta	**Vrednost**
Operativni sistem	Ubuntu 22.04 LTS
Java	OpenJDK 21.0.11
Spring Boot	3.5.0
React	19.2.7
PostgreSQL	14.23
Skup konekcija (*connection pool*)	HikariCP, maksimalno 10 konekcija
Razvojno okruženje	IntelliJ IDEA, Visual Studio Code
Veb pregledač	Google Chrome
Procesor	Intel Core i5-12500H, 12 jezgara i 16 logičkih niti
RAM memorija	16 GB

6.2 Metodologija benchmark testiranja

Benchmark testiranje sprovedeno je sa ciljem poređenja dva mehanizma konkurentnog pristupa podacima: optimističkog i pesimističkog zaključavanja. Testiranje je izvršeno korišćenjem benchmark modula implementiranog u okviru aplikacije, koji omogućava simulaciju većeg broja istovremenih korisničkih zahteva usmerenih ka istom događaju.

Za svaki test unapred je definisan broj korisničkih niti koje istovremeno pokušavaju da izvrše prijavu na isti događaj. Na ovaj način simulirani su uslovi u kojima veći broj korisnika konkurentno pristupa istom resursu.

Za paralelno izvršavanje benchmark zahteva korišćen je `ExecutorService` kreiran metodom `Executors.newVirtualThreadPerTaskExecutor()`, pri čemu se svaki simulirani korisnički zahtev izvršava u zasebnoj virtuelnoj niti.

Konkurentni zahtevi generisani u okviru benchmark modula nisu upućivani preko HTTP sloja aplikacije, već su direktno pozivane odgovarajuće metode servisnog sloja zadužene za registraciju korisnika. Na taj način benchmark je usmeren na poređenje ponašanja optimističkog i pesimističkog zaključavanja u aplikacionom i perzistentnom sloju, bez uticaja dodatnog vremena potrebnog za obradu HTTP zahteva i mrežnu komunikaciju.

Za oba mehanizma zaključavanja izvršeni su testovi sa različitim brojem zahteva i različitim kapacitetima događaja. Tokom pojedinačnih izvršavanja praćeni su ukupno vreme izvršavanja, broj uspešno registrovanih korisnika, broj korisnika na listi čekanja, broj konflikata nastalih pri optimističkom zaključavanju i konzistentnost konačnog stanja sistema. Za poređenje ponovljenih izvršavanja korišćena je medijana ukupnog vremena izvršavanja, dok je

raspršenje rezultata prikazano interkvartilnim rasponom (IQR). Pored toga, izračunata je medijana broja konflikata nastalih pri optimističkom zaključavanju, kako bi se sagledao uticaj ponovnih pokušaja na performanse ovog mehanizma.

Pre početka merenja svaki benchmark scenario izvršen je pet puta radi zagrevanja JVM-a. Rezultati ovih izvršavanja nisu uključeni u analizu. Nakon faze zagrevanja, svaki scenario je izvršen 30 puta za svaki posmatrani mehanizam zaključavanja. Korišćenjem većeg broja ponavljanja smanjen je uticaj pojedinačnih odstupajućih merenja na konačne rezultate.

Pre svakog pojedinačnog izvršavanja formiran je novi testni događaj sa zadatom vrednošću kapaciteta i nov skup testnih korisnika, čime je obezbeđeno da svako od 30 merenih ponavljanja započne iz istog početnog stanja. Nakon završetka svakog izvršavanja testni podaci su uklonjeni.

Kod optimističkog zaključavanja konflikti koji nastaju prilikom konkurentne izmene istog podatka rešavaju se mehanizmom ponovnog pokušaja (engl. *retry*). Ponovno izvršavanje primenjuje se samo kada je detektovan konflikt optimističkog zaključavanja, dok se ostale greške ne ponavljaju. Maksimalan broj pokušaja po zahtevu ograničen je na 30.

Između dva uzastopna pokušaja uvedeno je kratko randomizovano čekanje u intervalu od 0 do 2 ms, čime se smanjuje verovatnoća da niti koje su istovremeno doživele konflikt odmah ponovo pokušaju pristup istom podatku. Implementacija ne koristi eksponencijalni *backoff*, odnosno vreme čekanja se ne povećava sa brojem neuspešnih pokušaja.

Vrednost od 30 pokušaja predstavlja konfiguraciono ograničenje eksperimentalne postavke i izabrana je kako bi se broj ponovnih izvršavanja ograničio, a istovremeno omogućilo da zahtev u uslovima povećane konkurentnosti ima dovoljno prilika da se uspešno završi.

Svi testovi sprovedeni su u istim hardverskim i softverskim uslovima, kako bi dobijeni rezultati bili međusobno uporedivi.

6.3 Rezultati benchmark testiranja

Benchmark testiranje sprovedeno je kroz sedam scenarija sa različitim brojem istovremenih zahteva i različitim kapacitetima događaja. Na ovaj način analizirano je ponašanje optimističkog i pesimističkog zaključavanja u uslovima različitog nivoa konkurencije.

Testirani su sledeći scenariji:

- 10 zahteva i kapacitet događaja 10;
- 50 zahteva i kapacitet događaja 50;

- 50 zahteva i kapacitet događaja 20;
- 100 zahteva i kapacitet događaja 50;
- 200 zahteva i kapacitet događaja 50;
- 200 zahteva i kapacitet događaja 100;
- 500 zahteva i kapacitet događaja 100.

Za svaki scenario izvršeno je po 30 merenih pokretanja sa optimističkim i pesimističkim zaključavanjem, nakon prethodno sprovedene faze zagrevanja JVM-a. Rezultati su predstavljeni medijanom ukupnog vremena izvršavanja i interkvartilnim rasponom (IQR), a za optimističko zaključavanje prikazana je i medijana broja konflikata. Pored toga, provereno je da li je stanje sistema nakon svih izvršavanja ostalo konzistentno.

Rezultati benchmark testiranja prikazani su u Tabeli 6.2.

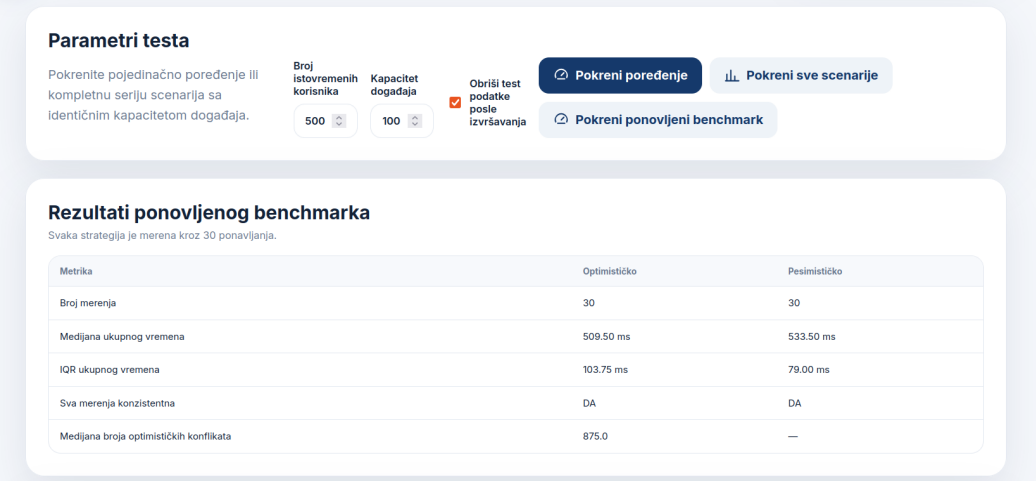
Tabela 6.2: Rezultati ponovljenog benchmark testiranja za različite scenarije

Scenario	**Zaključavanje**	**Med. (ms)**	**IQR (ms)**	**Opt. konflikti**	**Konzist.**
10 / 10	Optimističko	15,00	3,75	27,0	DA
	Pesimističko	13,00	3,00	–	DA
50 / 50	Optimističko	187,50	33,75	363,0	DA
	Pesimističko	51,00	4,75	–	DA
50 / 20	Optimističko	38,50	15,75	178,0	DA
	Pesimističko	48,00	9,75	–	DA
100 / 50	Optimističko	187,00	20,50	438,0	DA
	Pesimističko	117,50	31,25	–	DA
200 / 50	Optimističko	243,50	88,00	439,5	DA
	Pesimističko	239,50	48,25	–	DA
200 / 100	Optimističko	424,50	95,50	875,5	DA
	Pesimističko	241,50	45,25	–	DA
500 / 100	Optimističko	509,50	103,75	875,0	DA
	Pesimističko	533,50	79,00	–	DA

Kolona „Opt. konflikti” odnosi se na medijanu broja konflikata detektovanih pri optimističkom zaključavanju. Ova metrika nije primenljiva na pesimističko zaključavanje, jer se kod njega konkurentni pristup kontroliše zaključavanjem odgovarajućeg zapisa, pri čemu konkurentni zahtevi čekaju na oslobađanje zaključavanja. Zbog toga je za pesimističko zaključavanje u ovoj koloni prikazana oznaka „–”.

Za svako pojedinačno merenje benchmark modul dodatno proverava konzistentnost stanja događaja nakon završetka konkurentnih zahteva. Provera obuhvata uslov da broj uspešno registrovanih korisnika ne prekorači kapacitet događaja, da broj preostalih slobodnih mesta nije negativan i da zbir broja registrovanih korisnika i preostalih slobodnih mesta odgovara početnom kapacitetu događaja.

U svih sedam testiranih scenarija, kod oba mehanizma zaključavanja, svih 30 merenih izvršavanja zadovoljilo je definisane uslove konzistentnosti. Primer prikaza rezultata ponovljenog benchmark testa za scenario sa 500 istovremenih zahteva i kapacitetom događaja 100 dat je na Slici 6.1.



Slika 6.1: Rezultati ponovljenog benchmark testa za 500 zahteva i kapacitet 100

6.4 Analiza rezultata

Rezultati ponovljenog benchmark testiranja pokazuju da performanse optimističkog i pesimističkog zaključavanja zavise od nivoa konkurentnosti i odnosa između broja zahteva i kapaciteta događaja. Korišćenje medijane kao reprezentativne vrednosti omogućava poređenje tipičnih vremena izvršavanja uz manji uticaj pojedinačnih odstupajućih merenja, dok interkvartilni raspon (IQR) ukazuje na raspršenje dobijenih rezultata. Medijana broja konflikata kod optimističkog zaključavanja dodatno omogućava sagledavanje učestalosti ponovnih pokušaja pri konkurentnom ažuriranju istog događaja.

Kod najmanjeg scenarija, sa 10 zahteva i kapacitetom 10, pesimističko zaključavanje ostvarilo je nešto nižu medijanu ukupnog vremena izvršavanja. Medijana je iznosila 13,00 ms, u poređenju sa 15,00 ms kod optimističkog

pristupa. IQR je iznosio 3,00 ms kod pesimističkog i 3,75 ms kod optimističkog zaključavanja. Kod optimističkog pristupa medijana broja konflikata iznosila je 27,0.

Izraženija razlika uočena je u scenariju sa 50 zahteva i kapacitetom 50. Medijana vremena izvršavanja kod pesimističkog pristupa iznosila je 51,00 ms, dok je kod optimističkog iznosila 187,50 ms. Istovremeno, medijana broja optimističkih konflikata iznosila je 363,0. Ovaj rezultat ukazuje na dodatni trošak optimističkog pristupa usled konflikata i ponovnog izvršavanja operacija u uslovima konkurentnog pristupa istom događaju. IQR optimističkog pristupa iznosio je 33,75 ms, dok je kod pesimističkog iznosio 4,75 ms.

U scenariju sa 50 zahteva i kapacitetom 20 optimističko zaključavanje ostvarilo je nižu medijanu ukupnog vremena izvršavanja, 38,50 ms, u odnosu na 48,00 ms kod pesimističkog zaključavanja. Medijana broja optimističkih konflikata iznosila je 178,0. IQR je, međutim, bio veći kod optimističkog pristupa i iznosio je 15,75 ms, dok je kod pesimističkog iznosio 9,75 ms. Ovaj scenario pokazuje da niža medijana vremena izvršavanja ne mora istovremeno da znači i manje raspršenje rezultata.

Kod 100 zahteva i kapaciteta 50 pesimističko zaključavanje ostvarilo je nižu medijanu ukupnog vremena izvršavanja, 117,50 ms, u poređenju sa 187,00 ms kod optimističkog zaključavanja. Medijana broja optimističkih konflikata iznosila je 438,0. IQR optimističkog pristupa iznosio je 20,50 ms, dok je kod pesimističkog iznosio 31,25 ms.

U scenariju sa 200 zahteva i kapacitetom 50 razlika između medijana bila je mala. Optimističko zaključavanje ostvarilo je medijanu od 243,50 ms, dok je kod pesimističkog ona iznosila 239,50 ms. Medijana broja optimističkih konflikata iznosila je 439,5. IQR optimističkog pristupa iznosio je 88,00 ms, a pesimističkog 48,25 ms, što ukazuje na veće raspršenje vremena izvršavanja kod optimističkog pristupa u ovom scenariju.

Kod 200 zahteva i kapaciteta 100 pesimističko zaključavanje ostvarilo je medijanu od 241,50 ms, dok je kod optimističkog pristupa ona iznosila 424,50 ms. U ovom scenariju zabeležena je najveća medijana broja optimističkih konflikata, koja je iznosila 875,5. IQR je takođe bio veći kod optimističkog pristupa i iznosio je 95,50 ms, u poređenju sa 45,25 ms kod pesimističkog zaključavanja.

Pri najvećem testiranom opterećenju, sa 500 zahteva i kapacitetom događaja 100, optimističko zaključavanje ponovo je ostvarilo nižu medijanu ukupnog vremena izvršavanja. Ona je iznosila 509,50 ms, dok je kod pesimističkog pristupa iznosila 533,50 ms. Medijana broja optimističkih konflikata bila je 875,0. IQR optimističkog pristupa iznosio je 103,75 ms, a pesimističkog 79,00 ms.

Posmatranjem broja konflikata može se uočiti da njihova veća učesta-

lost može predstavljati značajan dodatni trošak optimističkog zaključavanja, jer konflikt zahteva ponovno izvršavanje operacije. To je naročito uočljivo u scenariju 50/50, u kojem je medijana broja optimističkih konflikata iznosila 363,0, a pesimističko zaključavanje ostvarilo je znatno nižu medijanu ukupnog vremena izvršavanja. Ipak, broj konflikata nije jedini faktor koji određuje ukupne performanse. U scenariju 500/100 medijana broja optimističkih konflikata iznosila je 875,0, ali je optimistički pristup uprkos tome ostvario nižu medijanu ukupnog vremena od pesimističkog. Kod pesimističkog zaključavanja konkurentni zahtevi čekaju oslobađanje zaključavanja, pa na ukupno vreme izvršavanja može uticati i serijalizacija pristupa zaključanom zapisu.

Važan rezultat sprovedenog testiranja odnosi se i na konzistentnost podataka. U svih sedam scenarija, za oba mehanizma zaključavanja, svih 30 merenih izvršavanja zadovoljilo je definisane uslove konzistentnosti. Time je potvrđeno da u okviru sprovedenih eksperimenata nijedan od posmatranih pristupa nije doveo do prekoračenja kapaciteta događaja, negativnog broja slobodnih mesta ili narušavanja odnosa između broja registracija i raspoloživog kapaciteta.

Dobijeni rezultati ne ukazuju na to da je jedan mehanizam zaključavanja bezuslovno bolji u svim posmatranim uslovima. Pesimističko zaključavanje ostvarilo je niže medijane ukupnog vremena izvršavanja u pet od sedam testiranih scenarija: 10/10, 50/50, 100/50, 200/50 i 200/100. Optimističko zaključavanje ostvarilo je niže medijane u scenarijima 50/20 i 500/100. Rezultati takođe pokazuju da broj optimističkih konflikata predstavlja važan faktor pri tumačenju performansi, ali da sam po sebi nije dovoljan za određivanje efikasnijeg mehanizma zaključavanja u svim uslovima.

6.5 Diskusija

Rezultati sprovedenog benchmark testiranja pokazuju da izbor između optimističkog i pesimističkog zaključavanja zavisi od karakteristika konkretnog opterećenja sistema. Nijedan od posmatranih mehanizama nije ostvario bolje rezultate u svim testiranim scenarijima, što ukazuje na to da broj konkurentnih zahteva sam po sebi nije dovoljan za izbor odgovarajućeg pristupa.

Kod optimističkog zaključavanja više transakcija može istovremeno da pristupa istom podatku, dok se konflikt otkriva prilikom pokušaja izmene. U implementiranom rešenju konflikti se rešavaju mehanizmom ponovnog pokušaja (engl. *retry*), kojim se neuspešna operacija ponovo izvršava. Između ponovnih pokušaja primenjuje se kratko randomizovano čekanje. Ovakav pristup omogućava izbegavanje dugotrajnog držanja zaključavanja nad podacima, ali ponovljena izvršavanja predstavljaju dodatni trošak kada dođe do

konflikata.

Sa druge strane, kod pesimističkog zaključavanja pristup podacima se kontroliše zaključavanjem na nivou baze podataka. Na taj način se izbegavaju konflikti karakteristični za optimistički pristup, ali konkurentni zahtevi mogu čekati na oslobađanje zaključanog resursa. Takvo čekanje može postati značajan faktor pri većem broju istovremenih zahteva, što se može odraziti na ukupno vreme izvršavanja sistema.

Rezultati ponovljenih merenja pokazuju različito ponašanje dva pristupa u zavisnosti od scenarija. Pesimističko zaključavanje ostvarilo je nižu medijanu ukupnog vremena izvršavanja u scenarijima 10/10, 50/50, 100/50, 200/50 i 200/100. Optimističko zaključavanje ostvarilo je nižu medijanu u scenarijima 50/20 i 500/100.

Posebno izražena razlika u korist pesimističkog zaključavanja uočena je u scenariju sa 50 zahteva i kapacitetom 50. Medijana ukupnog vremena izvršavanja pesimističkog pristupa iznosila je 51,00 ms, dok je kod optimističkog pristupa iznosila 187,50 ms. Istovremeno, medijana broja optimističkih konflikata iznosila je 363,0. Ovakav rezultat ukazuje na to da ponovna izvršavanja izazvana konfliktima mogu predstavljati značajan dodatni trošak optimističkog pristupa.

Pri najvećem testiranom opterećenju, sa 500 zahteva i kapacitetom 100, optimističko zaključavanje ostvarilo je nižu medijanu ukupnog vremena izvršavanja, koja je iznosila 509,50 ms, u poređenju sa 533,50 ms kod pesimističkog zaključavanja. Medijana broja optimističkih konflikata u ovom scenariju iznosila je 875,0. Ovaj rezultat pokazuje da veliki broj konflikata ne mora nužno da znači i veće ukupno vreme izvršavanja u odnosu na pesimistički pristup, jer kod pesimističkog zaključavanja konkurentni zahtevi mogu čekati na oslobađanje zaključanog zapisa.

Pored medijane, interkvartilni raspon pokazuje da pristup sa nižom medijanom ne mora u svakom slučaju imati i manje raspršenje rezultata. Na primer, u scenariju 50/20 optimističko zaključavanje ostvarilo je nižu medijanu ukupnog vremena izvršavanja, 38,50 ms u odnosu na 48,00 ms kod pesimističkog pristupa, ali je njegov IQR iznosio 15,75 ms, u poređenju sa 9,75 ms kod pesimističkog pristupa. Zbog toga je prilikom poređenja mehanizama korisno posmatrati i reprezentativno vreme izvršavanja i raspršenje dobijenih rezultata.

Rezultati koji uključuju broj optimističkih konflikata dodatno pokazuju da ova metrika predstavlja važan pokazatelj ponašanja optimističkog mehanizma, ali se ne može posmatrati izolovano. Na ukupne performanse utiču i drugi faktori, uključujući broj konkurentnih zahteva, kapacitet događaja, trošak ponovnog izvršavanja operacija i čekanje na oslobađanje zaključavanja kod pesimističkog pristupa.

Sa stanovišta razvijene aplikacije, oba pristupa obezbeđuju kontrolu konkurentnog pristupa prilikom prijave korisnika na događaje, ali na različite načine. U svim sprovedenim merenjima oba mehanizma zadovoljila su definisane uslove konzistentnosti podataka. Optimističko zaključavanje predstavlja pogodan pristup kada je prihvatljivo rešavanje konflikata ponovnim pokušajima, dok pesimističko zaključavanje predstavlja pristup kojim se konflikti sprečavaju zaključavanjem resursa, uz cenu mogućeg čekanja konkurentnih zahteva na oslobađanje zaključavanja.

Dobijene rezultate treba posmatrati u okviru uslova u kojima je benchmark sproveden. Testiranje je izvršeno u lokalnom razvojnom okruženju, pri čemu su komponente sistema i baza podataka izvršavane na istom računaru. Benchmark zahtevi nisu prolazili kroz HTTP sloj aplikacije, već su direktno pozivane metode servisnog sloja, pa izmerena vremena ne obuhvataju trošak mrežne komunikacije i obrade HTTP zahteva. Rezultati zato prvenstveno omogućavaju međusobno poređenje dva implementirana mehanizma u kontrolisanim uslovima, dok bi za donošenje opštijih zaključaka o ponašanju sistema u produkcionom okruženju bilo potrebno sprovesti dodatna testiranja na distribuiranoj infrastrukturi i sa različitim profilima opterećenja.

Na osnovu sprovedene evaluacije može se zaključiti da ne postoji univerzalno optimalan mehanizam zaključavanja za sve scenarije. Rezultati potvrđuju značaj prilagođavanja strategije konkurentnog pristupa očekivanom opterećenju sistema, kao i praćenja više pokazatelja performansi prilikom donošenja odluke o izboru odgovarajućeg mehanizma.

Glava 7

Upotreba alata zasnovanih na veštačkoj inteligenciji

Tokom izrade master rada korišćen je *ChatGPT* kao dodatna podrška u pojedinim fazama razvoja aplikacije i pisanja rada. Alat je korišćen za razmatranje mogućih načina implementacije određenih funkcionalnosti, analizu problema koji su se pojavljivali tokom razvoja, kao i za predloge za poboljšanje i organizaciju programskog koda. Takođe je korišćen kao pomoć pri jasnijem formulisanju i jezičkom oblikovanju pojedinih delova teksta.

Predlozi dobijeni na ovaj način služili su kao polazna osnova i nisu automatski uključivani u konačnu verziju rada. Predložena programska rešenja prilagođavana su zahtevima razvijenog sistema, a njihovo funkcionisanje proveravano je kroz implementaciju i testiranje aplikacije. Prilikom izrade teorijskog dela rada korišćena je literatura navedena u bibliografiji, dok je konačan izbor, provera i prilagođavanje sadržaja izvršeno u skladu sa ciljevima i predmetom ovog master rada.

Glava 8

Zaključak

Savremeni sistemi za upravljanje događajima zahtevaju pouzdano upravljanje velikim brojem korisnika, jednostavnu organizaciju događaja i mogućnost komunikacije u realnom vremenu između učesnika i organizatora. Istovremeno, jedan od značajnih izazova predstavlja očuvanje konzistentnosti podataka u situacijama kada veliki broj korisnika istovremeno pristupa istim resursima, kao što su prijave na događaje ograničenog kapaciteta.

U okviru ovog rada razvijena je veb aplikacija za upravljanje događajima i komunikaciju u realnom vremenu koja omogućava registraciju i autentifikaciju korisnika, upravljanje događajima, prijavu učesnika, administraciju sistema i razmenu javnih i privatnih poruka korišćenjem WebSocket tehnologije. Serverski deo sistema implementiran je korišćenjem okvira Spring Boot, dok je korisnički interfejs razvijen primenom biblioteke React. Za skladištenje i upravljanje podacima korišćen je sistem za upravljanje bazama podataka PostgreSQL, a zaštita pristupa realizovana je primenom JWT autentifikacije i sistema korisničkih uloga.

Posebna pažnja posvećena je implementaciji konkurentnog pristupa podacima prilikom prijavljivanja korisnika na događaje ograničenog kapaciteta. U okviru sistema realizovana su dva različita mehanizma zaključavanja – optimističko i pesimističko zaključavanje – čime je omogućeno njihovo direktno poređenje u identičnim uslovima izvršavanja. Pored same implementacije razvijen je i benchmark modul koji automatski izvršava testove sa različitim brojem istovremenih korisnika, prikuplja relevantne metrike performansi i proverava konzistentnost sistema nakon završetka svakog scenarija.

U radu su predstavljene ključne funkcionalnosti sistema iz perspektive različitih korisničkih uloga. Prikazani su proces registracije i prijave korisnika, pregled i upravljanje događajima, prijavljivanje i odjavljivanje učesnika, komunikacija u realnom vremenu, razmena privatnih poruka i administrativne funkcionalnosti. Na ovaj način prikazan je način interakcije učesnika,

organizatora i administratora sa razvijenom aplikacijom.

Jedan od značajnih doprinosa ovog rada predstavlja razvoj benchmark modula koji omogućava objektivno poređenje različitih strategija zaključavanja u okviru iste aplikacije. Modul automatski formira testne scenarije, izvršava veliki broj paralelnih korisničkih zahteva, prikuplja statističke pokazatelje performansi i proverava konzistentnost sistema, čime omogućava detaljnu analizu ponašanja aplikacije pod različitim nivoima opterećenja.

Benchmark testiranje pokazalo je da oba implementirana mehanizma zaključavanja uspešno obezbeđuju konzistentnost podataka i sprečavaju prekoračenje kapaciteta događaja čak i pri velikom broju istovremenih korisničkih zahteva. Dobijeni rezultati ukazuju da između optimističkog i pesimističkog zaključavanja ne postoji univerzalno najbolje rešenje, već da njihove performanse zavise od karakteristika konkretnog opterećenja. Analiza broja konflikata pokazala je da konflikti i ponovna izvršavanja mogu predstavljati značajan dodatni trošak optimističkog pristupa, ali da njihov broj sam po sebi nije dovoljan za određivanje efikasnijeg mehanizma. Kod pesimističkog zaključavanja konkurentni zahtevi mogu čekati na oslobađanje zaključanog resursa, što takođe može uticati na ukupno vreme izvršavanja.

Dalji razvoj sistema mogao bi obuhvatiti implementaciju distribuiranog izvršavanja benchmark testova na više serverskih instanci, integraciju sistema za keširanje, primenu mehanizama za obradu događaja zasnovanih na redovima poruka, kao i implementaciju dodatnih strategija konkurentnog pristupa podacima. Takođe, interesantan pravac daljeg razvoja predstavlja evaluacija performansi sistema u cloud okruženju i analiza njegovog ponašanja pri znatno većem broju istovremenih korisnika.

Na osnovu sprovedene implementacije i izvršenih testiranja može se zaključiti da razvijeni sistem uspešno ispunjava postavljene funkcionalne zahteve i predstavlja pouzdano rešenje za upravljanje događajima i komunikaciju u realnom vremenu. Istovremeno, razvijeni benchmark modul omogućava detaljnu analizu performansi različitih mehanizama zaključavanja i može poslužiti kao osnova za dalja istraživanja u oblasti konkurentnog pristupa podacima u savremenim veb aplikacijama.

Bibliografija

[1] R. S. Pressman and B. R. Maxim, *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 9 ed., 2020.

[2] I. Sommerville, *Software Engineering*. Pearson, 10 ed., 2016.

[3] R. T. Fielding, *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[4] Internet Engineering Task Force, "The WebSocket Protocol (RFC 6455)." `https://www.rfc-editor.org/rfc/rfc6455`, 2011.

[5] VMware, "Spring Framework reference documentation." `https://docs.spring.io/spring-framework/reference/`, 2025. Pristupljeno: jul 2026.

[6] Internet Engineering Task Force, "JSON Web Token (JWT) (RFC 7519)." `https://www.rfc-editor.org/rfc/rfc7519`, 2015.

[7] A. S. Tanenbaum and M. V. Steen, *Distributed Systems*. CreateSpace, 3 ed., 2017.

[8] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs*. O'Reilly Media, 2013.

[9] Internet Engineering Task Force, "The JavaScript Object Notation (JSON) data interchange format (RFC 8259)." `https://www.rfc-editor.org/rfc/rfc8259`, 2017.

[10] VMware, "Spring Boot reference documentation." `https://docs.spring.io/spring-boot/documentation.html`, 2025. Pristupljeno: jul 2026.

[11] C. Walls, *Spring in Action*. Manning, 6 ed., 2022.

[12] Meta Platforms, Inc., "React documentation." https://react.dev/, 2025. Pristupljeno: jul 2026.

[13] D. Flanagan, *JavaScript: The Definitive Guide*. O'Reilly Media, 7 ed., 2020.

[14] Axios Contributors, "Axios documentation." https://axios-http.com/, 2025. Pristupljeno: jul 2026.

[15] PostgreSQL Global Development Group, "PostgreSQL documentation." https://www.postgresql.org/docs/, 2025. Pristupljeno: jul 2026.

[16] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*. Pearson, 7 ed., 2017.

[17] L. Spilca, *Spring Security in Action*. Manning, 2020.

[18] OWASP Foundation, "OWASP Authentication Cheat Sheet." https://cheatsheetseries.owasp.org, 2024. Pristupljeno: jul 2026.

[19] VMware, "Spring Security reference documentation." https://docs.spring.io/spring-security/reference/, 2025. Pristupljeno: jul 2026.

[20] VMware, "Spring Framework WebSocket documentation." https://docs.spring.io/spring-framework/reference/web/websocket.html, 2025. Pristupljeno: jul 2026.

[21] STOMP Protocol, "STOMP Protocol Specification Version 1.2." https://stomp.github.io/stomp-specification-1.2.html, 2012. Pristupljeno: jul 2026.

[22] M. Kleppmann, *Designing Data-Intensive Applications*. O'Reilly Media, 2017.

[23] Hibernate Team, "Hibernate ORM user guide." https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html, 2025. Pristupljeno: jul 2026.

[24] Oracle, "Java Microbenchmark Harness (JMH)." https://openjdk.org/projects/code-tools/jmh/, 2025. Pristupljeno: jul 2026.

[25] VMware, "Spring Data JPA reference documentation." https://docs.spring.io/spring-data/jpa/reference/, 2025. Pristupljeno: jul 2026.

[26] React Router Team, “React Router documentation.” https://reactrouter.com/, 2025. Pristupljeno: jul 2026.

Biografija

Milica Kostadinović rođena je 3. jula 1999. godine. Osnovne akademske studije završila je na Matematičkom fakultetu Univerziteta u Beogradu, na studijskom programu Informatika, nakon čega je na istom fakultetu upisala master akademske studije.

Trenutno je zaposlena u kompaniji NCR Atleos, gde stiče profesionalno iskustvo u oblasti razvoja softvera. Njena profesionalna interesovanja obuhvataju razvoj softvera, veb tehnologije i razvoj informacionih sistema.