UNIVERZITET U BANJOJ LUCI

ELEKTROTEHNIČKI FAKULTET

GENERISANJE VEB APLIKACIJA NA OSNOVU ŠEME BAZE PODATAKA

Master rad

Mentor:
Doc. dr Mihajlo Savić

Kandidat:
Danijela Vukosav

Banja Luka, septembar 2025.

UNIVERSITY OF BANJA LUKA

FACULTY OF ELECTRICAL ENGINEERING

GENERATION OF WEB APPLICATIONS BASED ON A DATABASE SCHEME

Master thesis

Mentor:
Asst. Prof. Mihajlo Savić, PhD

Candidate:
Danijela Vukosav

Banja Luka, September 2025.

Tema: **GENERISANJE VEB APLIKACIJA NA OSNOVU ŠEME BAZE PODATAKA**

Mentor: **Dr Mihajlo Savić, docent,**
Elektrotehnički fakultet,
Univerzitet u Banjoj Luci

Naučna oblast: **Inženjerstvo i tehnologija**

Naučno polje: **Elektrotehnika, elektronika i informaciono inženjerstvo**

Ključne riječi: *generisanje koda, jednostranične veb aplikacije, REST API, relaciona baza podataka*

Klasifikaciona oznaka (CERIF): **T120**

Sažetak: Veb aplikacije i aplikacije zasnovane na REST API-ju mogu se smatrati najčešćim tipom aplikacija danas i njihov broj stalno raste. Zbog pritiska da se proizvede više koda i da se to brže uradi, programeri koriste različite programske jezike, biblioteke i razvojene okvire pogodne za takve aplikacije. Dvije često korištene tehnologije su Spring na serverskoj strani i React na klijentskoj strani. Iako svaka tehnologija pruža zadovoljavajuću funkcionalnost, količina koda koju je potrebno proizvesti nije trivijalna, posebno u slučajevima kada aplikacija mora da obezbijedi čak i osnovne bezbjednosne i revizorske funkcionalnosti. Jedno rješenje za ovaj problem je upotreba alata za generisanje koda koji proizvode i serverski i klijentski kod iz modela. Iako postoji širok spektar generatora, nijedan od analiziranih ne ispunjava sve zahtjeve moderne veb aplikacije. U ovom radu je opisana veb aplikacija za generisanje koda koja generiše Spring i React kod zasnovan na relacionom modelu podataka u DDL ili JSON formatu, te pruža programerima jednostavan i efikasan alat za generisanje potpuno funkcionalnog osnovnog koda.

Topic: **GENERATION OF WEB APPLICATIONS BASED ON A DATABASE SCHEME**

Mentor: **Dr Mihajlo Savić, Assistant Professor,**
Faculty of Electrical Engineering,
University of Banja Luka

Scientific area: **Engineering and technology**

Scientific field: **Electrical engineering, electronics and information engineering**

Keywords: *code generation, single-page web applications, REST API, relational database*

Classification label (CERIF): **T120**

Abstract: Web applications and applications based on REST APIs can be considered the most common types of applications today, and their number is constantly growing. Due to the pressure to produce more code and to do so faster, developers rely on various programming languages, libraries, and development frameworks suitable for such applications. Two frequently used technologies are Spring on the server side and React on the client side. Although each technology provides satisfactory functionality, the amount of code that needs to be produced is not trivial, especially in cases where the application must provide even basic security and auditing features. One solution to this problem is the use of code generation tools that produce both server-side and client-side code from a model. Although a wide range of generators exists, none of those analyzed meet all the requirements of a modern web application. This paper presents a web application for code generation that produces Spring and React code based on a relational data model in DDL or JSON format, providing developers with a simple and efficient tool for generating fully functional boilerplate code.

Mojoj porodici

LISTA KORIŠTENIH SKRAĆENICA

1NF - First Normal Form
2NF - Second Normal Form
3NF - Third Normal Form
4NF - Fourth Normal Form
5NF - Fifth Normal Form
ABAC - Attribute-Based Access Control
AOP - Aspect-Oriented Programming
API - Application Programming Interface
AJAX - Asynchronous JavaScript and XML
B2B - Business-to-Business
B2C - Business-to-Consumer
BCNF - Boyce-Codd Normal Form
BLOB - Binary Large Object
CI/CD - Continuous Integration/Continuous Deployment
CRUD - Create, Retrieve, Update, Delete
CTE - Common Table Expressions
CSRF - Cross-Site Request Forgery
CLI - Command-Line Interface
DAC - Discretionary Access Control
DBMS - Database Management System
DDL - Data Definition Language
DI - Dependency Injection
ECDSA - Elliptic Curve Digital Signature Algorithm
GIS - Geographic Information System
GUI - Graphical User Interface
HMAC - Hash-based Message Authentication Code
HTML - HyperText Markup Language
HOC - Higher-Order Components
IAM - Identity and access management
ID - Identifier/Identification
IDE - Integrated Development Environment
IoC - Inversion of Control
JDBC - Java Database Connectivity
JDL - JHipster Domain Language
JMS - Java Message Service
JNDI - Java Naming and Directory Interface
JPA - Java Persistence API
JPQL - Jakarta Persistence Query Language
JSON - JavaScript Object Notation
JTA - Java Transaction API
JSX - JavaScript XML
JWT - JSON Web Token
LAMP - Linux, Apache, MySQL, PHP/Python/Perl
LCDP - Low-Code Development Platforms
LDAP - Lightweight Directory Access Protocol
LLM - Large Language Model
MAC - Mandatory Access Control

MDD - Model-Driven Development
MFA - Multi-factor Authentication
MEAN - MongoDB, Express.js, Angular, Node.js
MVC - Model-View-Controller
OAuth - Open Authorization
ORM - Object-Relational Mapping
OTP - One-Time Password
OLTP - Online Transaction Processing
OLAP - Online Analytical Processing
POJO - Plain Old Java Object
RAG - Retrieval-Augmented Generation
RBAC - Role-Based Access Control
RDBMS - Relational Database Management System
REST - Representational State Transfer
RSA - Rivest, Shamir, Adleman
SPA - Single Page Application
SpEL - Spring Expression Language
SRP - Single Responsibility Principle
SoC - Separation of Concerns
SQL - Structured Query Language
T4 - Text Template Transformation Toolkit
T-SQL - Transact Structured Query Language
URL - Uniform Resource Locator
VS Code - Visual Studio Code
XML - Extensible Markup Language
XSS - Cross-Site Scripting
YAML - YAML Ain't Markup Language

SADRŽAJ

Uz rad je priložena elektronska verzija rada.

1. UVOD

1.1. Predmet istraživanja

Savremeno inženjerstvo softvera nalaže sve viši stepen automatizacije rutinskih i repetitivnih procesa u razvoju složenih aplikacionih sistema. U kontekstu razvoja veb aplikacija, poseban značaj dobijaju segmenti koji zahtijevaju čestu i standardizovanu implementaciju, poput definisanja strukture i šeme baze podataka, organizovanja arhitekturnih slojeva te realizovanja osnovnih mehanizama za manipulaciju podacima (operacije poput kreiranja, čitanja, ažuriranja i brisanja). Ove aktivnosti su podložne automatizaciji jer se njihovi obrasci u velikoj mjeri ponavljaju i rijetko se suštinski razlikuju između različitih aplikacija i domena. Značajan dio procesa implementacije veb aplikacija podrazumijeva primjenu unaprijed poznatih principa i šablona koji obuhvataju standardizovanu organizaciju direktorijuma, inicijalnu implementaciju korisničke autentifikacije i autorizacije, kao i podršku za manipulaciju podacima. Ipak, bez obzira na postojanje ovih šablona, većina razvojnih timova i dalje troši značajan dio vremena na manuelnu implementaciju ovih komponenti, što ne samo da usporava razvojni proces, već i povećava mogućnost unošenja grešaka i narušavanja konzistentnosti koda.

Softverska industrija već neko vrijeme teži uvođenju rješenja koja bi omogućila brzu i pouzdanu generičku implementaciju osnovnih aplikativnih struktura, ali i koda uopšte. Analiza trenutno dostupnih generatora koda osnovnih CRUD funkcionalnosti ukazuje na postojanje brojnih alata koji automatizuju taj proces. Sa druge strane, istovremeno se pokazuju izražena ograničenja njihove primjene. Najčešće se ovi generatori opredjeljuju isključivo za jednu stranu arhitekture, ili serversku ili klijentsku, pri čemu integrisana rješenja koja objedinjuju obje komponente ostaju nedovoljno razvijena i dostupna. Pored ovih tehničkih aspekata, većina aktuelnih generatora ne obezbjeđuje mehanizme za prilagođavanje generisanih aplikacija samom korisniku. Metode modelovanja podataka koje podržavaju aktuelni generatori često su ograničene na sopstvene, interno definisane modele, što značajno otežava njihovu primjenu u praksi. Široko prihvaćeni formati poput SQL DDL skripte ili strukturisanih JSON šema rijetko su podržani.

Upravo u ovom kontekstu, predmet istraživanja je sistematska analiza, dizajn i implementacija alata za automatsko generisanje kompletnih veb aplikacija. Istraživanje se detaljno bavi identifikacijom i prevazilaženjem uočenih ograničenja. Jedna od njih je podrška različitim standardizovanim tipovima ulaznih podatka u sistem (DDL, JSON). Dok je druga bitna stavka generisanje gotovih, funkcionalnih rješenja koja integrišu i serverski i klijentski dio aplikacione arhitekture uz implementirane mehanizme pristupa, sigurnosti, nadzora i kontrole nad podacima. Na taj način, istraživanje doprinosi proširenju teorijskih osnova i metodoloških pristupa u oblasti automatizovanog razvoja, te nudi praktično rješenje za realne izazove sa kojima se suočavaju razvojni timovi. Optimalan ishod sistema treba da obezbijedi maksimalnu produktivnost, smanji vjerovatnoću unosa grešaka i omogući fleksibilnost u budućem razvoju. Istovremeno se smanjuju ukupni troškovi i vrijeme razvoja.

1.2. Cilj istraživanja

Cilj istraživanja je razvoj, validacija i teorijska evaluacija sveobuhvatnog sistema za automatsko generisanje veb aplikacija na osnovu šema baze podataka. Istraživanje je usmjereno na prevazilaženje ograničenja poznatih generatora koda i adresiranje ključnih

izazova u vezi sa brzinom, efikasnošću, fleksibilnošću i kvalitetom generisanog softverskog rješenja. U fokusu je dizajn i implementacija alata koji omogućava razvoj potpuno funkcionalnih aplikacija integrisanih sa savremenim principima višeplatformske arhitekture, visokim stepenom sigurnosti i kontrolisanim pristupom podacima. Poseban akcenat stavljen je na bezbjednost i privatnost korisničkih podataka. Generisani kod se isporučuje isključivo korisniku, bez zadržavanja ili dijeljenja šeme baze podataka, poslovne logike ili ideja sa bilo kojim trećim licem ili sistemom. Na taj način, korisnik ostvaruje potpuno vlasništvo nad dobijenim kodom i potpunu kontrolu nad sopstvenom aplikacijom, čime se eliminiše rizik od neovlaštenog pristupa, zloupotrebe ili komercijalnog iskorištavanja njegovih rješenja od strane drugih korisnika ili eksternih servisa.

Centralni cilj rada ogleda se u implementaciji generatora koda čija je osnovna svrha da automatizuje inicijalne faze razvoja kroz sljedeće definisane zadatke:

- Omogućavanje generisanja aplikacija polazeći od univerzalno prihvaćenih ulaznih formata, prvenstveno DDL skripti i strogo definisanih JSON šema, čime se proširuje dostupnost i pristupačnost alata za širok spektar korisnika.

- Generisanje serverskih i klijentskih aplikacija, kao nezavisnih ili integrisanih komponenti, prilagodljivih zahtjevima projekta.

- Uvođenje i automatizacija naprednih funkcionalnosti, kao što su autentifikacija i granularna autorizacija korisnika, praćenje promjena podataka, kao i mogućnost kontrole vidljivosti, filtriranja i sortiranja podataka po entitetima i kolonama.

- Validacija kvaliteta generisanog koda i izlaznih sistema upotrebom alata za statičku analizu, kao i praktičnom provjerom kroz različite šeme i nivoe kompleksnosti podataka.

Cilj ovog istraživanja je da ubrza proces izgradnje šablonskih veb aplikacija, poboljša trenutne metode automatizacije razvoja i omogući fleksibilnu osnovu za buduće širenje i inovacije u softverskom inženjeringu. Očekivanja su da sistem, koji je razvijen kroz ovaj rad, pruži jednostavan i pouzdan alat, pogodan za upotrebu kako u poslovnom, tako i u obrazovnom okruženju.

1.3. Metodologija

Metodološki okvir istraživanja zasnovan je na kombinaciji teorijske analize, sistematskog pregleda postojećih rješenja i praktične implementacije inovativnog sistema za generisanje veb aplikacija. Pristup istraživanju je organizovan tako da omogući temeljnu evaluaciju relevantnih naučnih, tehničkih i praktičnih aspekata izabrane teme, uz proceduru za potvrdu efikasnosti i kvaliteta razvijenog rješenja.

Proces istraživanja je obuhvatao sljedeće faze:

1. Teorijska analiza i sistematizacija postojećih rješenja - U prvoj fazi sprovedena je detaljna analiza savremenih tehnologija i alata iz domena automatizovanog generisanja koda. Posebna pažnja posvećena je aktuelnim rješenjima (kao što su JHipster, Spring Roo, OpenAPI Generator, Yeoman i CodeSmith), pri čemu su identifikovane njihove prednosti, ograničenja i stepen podrške za funkcionalnosti

poput autentifikacije, autorizacije, nadzora i rukovanja kompleksnim modelima podataka.

2. Definisanje zahtjeva i projektovanje rješenja - Na osnovu rezultata analize, formulisani su funkcionalni i nefunkcionalni zahtjevi za sistem koji će biti razvijen. U ovoj fazi izvršeno je definisanje arhitekture i modularnih komponenti generatora koda. Poseban akcenat stavljen je na parametarsku fleksibilnost i mogućnost interaktivne konfiguracije proizvoda sistema.

3. Praktična implementacija generatora koda - Treća faza uključuje dizajn i razvoj samog generatora koda. Implementacija je realizovana upotrebom Spring Boot-a i React-a. Poseban akcenat stavljen je na izradu parserske i transformacione logike šema baza podataka. Razvijen je grafički korisnički interfejs koji omogućava unos i validaciju ulaznih šema, konfiguraciju funkcionalnosti i jednostavno preuzimanje generisanog izvornog koda.

4. Testiranje, validacija i evaluacija sistema - Nakon implementacije, sprovedena su testiranja funkcionalnosti na raznovrsnim primjerima ulaznih šema različite kompleksnosti. Potom je izvršena i statička analiza kvaliteta generisanog koda upotrebom specijalizovanih alata. Kvantitativna mjerenja uključivala su procjenu vremena generisanja, potrošnje memorije i skalabilnosti rješenja, dok su kvalitativni aspekti vrednovani upoređivanjem sa postojećim alatima i analizom dometa podržanih funkcionalnosti.

5. Komparativna analiza sa postojećim rješenjima - Na osnovu dobijenih rezultata testiranja i evaluacije, urađena je uporedna analiza implementiranog rješenja i drugih često korištenih sistema za generisanje aplikacija, sa akcentom na novine, ograničenja, prednosti i preporuke za buduća unapređenja.

Ovakav višefazni metodološki pristup omogućava sveobuhvatno ispitivanje i verifikaciju postavljenih istraživačkih ciljeva. Kombinovanjem teorijske analize, evaluacije postojećih rješenja i praktične implementacije, postignut je balans između naučne utemeljenosti i realnih inženjerskih zahtjeva. Važno je naglasiti da su procesi testiranja i validacije sprovedeni na reprezentativnim scenarijima, koji su simulirali korisničke potrebe u različitim kontekstima upotrebe i sa različitim nivoima kompleksnosti podataka. Uz komparativnu analizu sa postojećim tehnologijama, ovakav metod istraživanja omogućava identifikaciju ne samo inovativnih aspekata razvijenog sistema, već i potencijalnih prostora za unapređenje. Dobijeni rezultati potvrđuju efikasnost i funkcionalnu cjelovitost rješenja, što je od velikog značaja za praktičnu primjenu i za dalje unapređenje sistema u skladu sa potrebama savremenog softverskog inženjeringa i tržišta.

1.4. Struktura rada

Rad je organizovan tako da obuhvati ključne teorijske, tehničke i praktične aspekte automatizovanog generisanja veb aplikacija. Strukturu rada čine:

1. Baze podataka – Poglavlje koje opisuje teorijsku osnovu relacionih baza podataka, sa objašnjenjem principa modelovanja i upravljanja podacima, što je važno za razumijevanje načina na koji se podaci organizuju i koriste u informacionim sistemima.

2. Spring Boot - U ovom dijelu rada detaljno se analiziraju karakteristike i prednosti Spring Boot razvojnog okvira. Posebna pažnja je posvećena njegovoj ulozi u pojednostavljenju razvoja, obezbjeđivanju sigurnosti i efikasnoj komunikaciji sa bazom podataka.

3. React - Fokus je na mogućnostima i osobinama React biblioteke za razvoj korisničkog interfejsa na strani klijenta, uz objašnjenje kako to React doprinosi stvaranju savremenih, interaktivnih i responzivnih veb aplikacija.

4. Prijedlog i opis rješenja za generisanje aplikacija - Ovo poglavlje detaljno opisuje arhitekturu i način funkcionisanja razvijenog sistema za generisanje veb aplikacija na osnovu šeme baze podataka. Prikazane su glavne funkcionalnosti i data su jasna uputstva za korištenje alata.

5. Evaluacija implementiranog rješenja - U ovom dijelu rada predstavljeni su rezultati eksperimentalnog testiranja sistema, uključujući kvantitativne mjere performansi i kvaliteta generisanog koda, kao i ograničenja sistema.

6. Zaključak – Sistematski opisan rad uz praktična zapažanja, analizu koristi i potencijalna unapređenja rješenja, kao i širi značaj dobijenih rezultata za praksu i budući razvoj ove oblasti.

Na kraju rada data su zaključna razmatranja, kao i preporuke i smjernice za buduća istraživanja. Nakon zaključka, navedena je korištena literatura.

1.5. Objavljeni rezultati istraživanja

Objavljen je naučni rad u naučnom časopisu International Journal of Electrical Engineering and Computing (IJEEC):

- D. Vukosav, D. Banjac, M. Ljubojević, and M. Savić, “CodeCrafter – Efficient Code Generator for Modern Single-page Web Applications,” International Journal of Electrical Engineering and Computing, vol. 9, no. 1, pp. 1–9, Jun. 2025, doi: 10.7251/IJEEC2501001V.

2. BAZE PODATAKA

U savremenom digitalnom dobu, podaci predstavljaju jedan od najvrjednijih resursa svake organizacije. Efikasnost prikupljanja, čuvanja, organizovanja i pristupa podacima je ključna stavka za donošenje odluka vezanih za optimizaciju poslovnih procesa i razvoj informacionih sistema. Upravo iz tog razloga, baze podataka zauzimaju centralno mjesto u informacionim tehnologijama jer omogućavaju strukturisano skladištenje i upravljanje velikim količinama podataka.

Baza podataka (eng. *Database*) predstavlja organizovanu zbirku podataka, koja je sistematski sačuvana i koncipirana tako da omogućava efikasno unošenje, ažuriranje i pretraživanje podataka kroz različite načine manipulacije. Najčešće se baza podataka definiše kao „kolekcija međusobno povezanih podataka, organizovanih na način koji olakšava njihovo korištenje i održavanje“ [1].

U savremenom kontekstu, baza podataka nije samo skladište podataka, već i infrastruktura koja obezbjeđuje integritet, sigurnost, konzistentnost i pristupačnost podataka uz pomoć specijalizovanih softverskih sistema, a to su sistemi za upravljanje bazom podataka (eng. *Database Management System - DBMS*). DBMS je softverski alat koji omogućava definiciju, kreiranje, upit, ažuriranje i upravljanje bazom podataka, te pruža interfejs prema korisnicima i aplikacijama [2]. Ključne karakteristike baze podataka su:

- Konzistentnost i integritet podataka - Centralizovano upravljanje podacima minimizuje višestruko čuvanje. Pravila integriteta štite dosljednost podataka.
- Bezbjednost podataka - Pristup bazi je kontrolisan uz mogućnost definisanja različitih nivoa privilegija za korisnike.
- Dugotrajnost - Podaci su sačuvani čak i u slučaju pada sistema.
- Upotrebljivost i pristupačnost - Lako pretraživanje i izmjena podataka omogućena je uz pomoć standardizovanih upitnih jezika (npr. SQL za relacione baze).

2.1. Istorijat razvoja baza podataka

Koncept računarske baze podataka datira još iz pedesetih godina XX vijeka, sa pojavom prvih računara i potrebom za elektronskim skladištenjem podataka. Prva generacija baza su bile baze u obliku datoteka (eng. *File-based systems*), gdje su podaci skladišteni u jednostavnim sekvencijalnim datotekama. Ovakav pristup otežavao je pristup i ažuriranje podataka. Redundancija i nekonzistentnost su bile česte pojave [3].

U šezdesetim godinama su razvijeni prvi hijerarhijski i mrežni modeli baza podataka. IBM-ov IMS (eng. *Information Management System*), predstavljen 1966. godine, bio je prvi široko korišten hijerarhijski DBMS. Paralelno, mrežni model (npr. IDMS) dozvoljavao je uzajamne veze između zapisa, ali je manipulacija strukturom baze bila kompleksna [2].

Prava revolucija dolazi sedamdesetih godina sa objavljivanjem istraživanja Edgara F. Codda iz IBM-a, koji je postavio temelje relacionog modela podataka [4]. Relacioni model, koji podatke organizuje u tabele (relacije), omogućio je daleko veću fleksibilnost, konzistentnost i jednostavnost upravljanja podacima nego dotadašnji modeli. Prvi

komercijalni sistemi bazirani na ovom modelu pojavili su se krajem sedamdesetih i početkom osamdesetih godina (Oracle, Ingres, DB2).

Tokom posljednje dvije decenije, ekspanzijom interneta, pojavljuju se nove, tzv. NoSQL baze podataka koje nude veću skalabilnost i fleksibilnost u radu sa nestrukturisanim podacima. Paralele se povlače i prema razvoju baza koje podatke organizuju u grafove i tako omogućavaju efikasno skladištenje i analizu povezanih (grafovskih) informacija.

2.2. Najčešće vrste baza podataka

Razvoj informacionih tehnologija doveo je do pojave više vrsta baza podataka, prilagođenih potrebama različitih domena primjene. Najšire rasprostranjene vrste su:

1. **Relacione baze podataka** - Podaci su organizovani u tabele (relacije), gdje svaki red predstavlja jedan zapis, a svaka kolona pojedinačno svojstvo tog zapisa. Snažan formalni model i standardizovan jezik za manipulaciju podacima (SQL) čine relacione baze dominantnim rješenjem u poslovnim aplikacijama, veb servisima i transakcionim sistemima. Ključne osobine uključuju:

   - ACID pravila (atomičnost, konzistentnost, izolovanost, trajnost),
   - striktnu šemu podataka,
   - integritet podataka.

   Primjeri servera relacionih baza podatka su: Oracle DBMS, MySQL, MariaDB, Microsoft SQL Server, te PostgreSQL.

2. **NoSQL baze podataka -** NoSQL baze (eng. *Not Only SQL*) su nastale kao odgovor na potrebe za skalabilnošću, fleksibilnošću i efikasnim radom sa velikim količinama nestrukturisanih ili polustrukturisanih podataka. Izuzetno su popularne u domenima sa velikom količinom podataka (eng. *Big Data*), kao i u aplikacijama koje komuniciraju u realnom vremenu. Glavne klase NoSQL baza su:

   - *Document store* - Podaci se skladište u formi dokumenata (najčešće u JSON/BSON formatu). Najpoznatiji primjer je MongoDB.
   - *Key-Value store* - Koriste jednostavne asocijacije ključ-vrijednost. Najpoznatiji primjeri su Redis, Valkey i Memcached.
   - *Column-family store* - Podaci se fizički čuvaju u kolonama umjesto u redovima, što je često pogodno za analitičke aplikacije. Primjeri su MonetDB i Apache Cassandra.

   NoSQL baze često žrtvuju dio ACID svojstava za bolje performanse ili veću skalabilnost.

3. **Grafovske baze podataka** - Grafovske baze podataka specijalizovane su za rad sa velikom mrežom entiteta i njihovih veza. Jedinice informacija (čvorovi) su povezane preko grana (veza) koje mogu nositi dodatne atribute. Glavna prednost je jednostavnost izražavanja i efikasnost pretraživanja složenih veza i obrazaca.

Upotreba grafovskih baza je rasprostranjena u društvenim mrežama, analizi prevara, sistemima za preporuke, bioinformatici i drugim oblastima. Često korišteni primjeri su Neo4j i Amazon Neptune.

2.2.1. Poređenje osnovnih tipova baza podataka

Različiti tipovi baza podataka omogućavaju optimalno skladištenje i manipulaciju podacima u zavisnosti od potreba konkretnog domena primjene - od tradicionalnih transakcionih aplikacija do društvenih mreža i analize podataka u realnom vremenu. Razumijevanje ovih pristupa i njihove evolucije je neophodno za pravilan izbor tehnologija i efikasan dizajn budućih softverskih rješenja. Poređenje osnovnih tipova baza podataka, po određenim osobinama i mogućnostima, je prikazano u tabeli 1.

Tabela 1: Poređenje tipova baza podataka

Model	**Struktura**	**Šema**	**Skalabilnost**	**Integritet**	**Primjer domena**
Relacione baze	Tabele	Fiksna	Vertikalna	Visok	Transakcione aplikacije
NoSQL	Dokument/ kolona/ključ-vrijednost	Slaba	Horizontalna	Srednji	Big Data, IoT, veb
Graf	Čvorovi-veze	Dinamična	Horizontalna	Srednji	Društvene mreže

2.3. Relacione baze podataka

Relacioni model podataka, koji je prvi put formalno opisan od strane E. F. Codd-a 1970. godine u radu "A Relational Model of Data for Large Shared Data Banks", predstavlja i opisuje matematičko-logički pristup organizaciji podataka. Osnovna ideja ovog modela je da se svi podaci predstavljaju u formi relacija, odnosno matematičkih skupova koji su slični tabelama. Codd je u okviru svog rada postavio i dvanaest principa koje svaki RDBMS treba da zadovolji, od kojih su najvažniji teorijska nezavisnost podataka i programska nezavisnost [2].

Ovaj model je bio revolucija u svijetu informacionih sistema tako što je ubrzao razvoj industrije baza podataka i omogućio stvaranje strukturnog upitnog jezika (eng. *Structured query language - SQL*), koji danas predstavlja osnovu gotovo svih komercijalnih rješenja. Relacioni model je uspio da prevaziđe ograničenja hijerarhijskih i mrežnih modela tako što je uveo nivo apstrakcije koji omogućava logički opis podataka nezavisan od fizičkog načina skladištenja. Podaci se organizuju u strukture nazvane tabele (relacije), gdje je svaki red u tabeli logička instanca (zapis), a svaka kolona predstavlja atribut ili osobinu entiteta koji se modeluje. Ključna obilježja relacionog modela su jednostavnost, dosljednost, lakoća upotrebe i formalno definisana pravila integriteta. Sve to omogućava robusno i pouzdano skladištenje, kao i napredan mehanizam za upite nad podacima putem standardizovanih jezika poput SQL-a [5].

Relacioni model, sa jasno definisanim pojmovima, omogućio je sistemima za upravljanje bazama podataka da zadovolje stroge zahtjeve za konzistentnošću, fleksibilnošću

i efikasnošću pristupa podacima u različitim aplikacijama i domenima. Osnovni pojmovi koji prate relacione baze podataka su:

1. **Tabela** (eng. *Table*) - Osnovni element relacione baze podataka. Svaka tabela je konceptualno ekvivalent skupu, gdje je svaki element tog skupa jedan red (eng. *record ili tuple*). Tabela je definisana kao dvodimenzionalna struktura sa tačno određenim brojem kolona i proizvoljnim brojem redova. U terminologiji relacionog modela, tabela se naziva relacija (eng. *Relation*).

2. **Red** (eng. *Record, Tuple*) - Red predstavlja jedan zapis u tabeli, odnosno, instancu entiteta koji se modeluje. Svaki red sadrži vrijednosti za sva svojstva (kolone) koja su definisana strukturom tabele. Redovi nisu sortirani i ne mogu imati identične vrijednosti za primarni ključ, čime se obezbjeđuje jedinstvenost.

3. **Kolona** (eng. *Attribute*) - Kolona, odnosno atribut, predstavlja jedno svojstvo ili karakteristiku entiteta (npr. ime, prezime, broj telefona). Svaka kolona ima jedinstveno ime unutar svoje tabele i unaprijed definisani tip podatka (npr. integer, varchar, date), što omogućava dosljednost i validaciju podataka u svakom zapisu.

4. **Ključ** (eng. *Key*) - Ključevi su elementi koji služe za strukturisanje i međusobno povezivanje podataka. Postoje sljedeće vrste ključeva:

    - Primarni ključ (eng. *Primary Key*) je jedan ili skup atributa čija je vrijednost jedinstvena za svaki red (npr. JMBG, broj indeksa, ID). Omogućava identifikaciju svakog zapisa i ne dozvoljava nedefinisane vrijednosti.

    - Strani ključ (eng. *Foreign Key*) predstavlja jedan ili skup atributa u tabeli koji jednoznačno referencira primarni ključ iz druge tabele. Strani ključevi omogućavaju logičko povezivanje tabela.

    - Kandidatski ključ (eng. *Candidate Key*) je svaki atribut ili kombinacija atributa koji zadovoljavaju uslov jedinstvenosti. Kandidatski ključ predstavlja minimalni superključ, što znači da ne sadrži nijedan suvišan atribut. Od više kandidatskih ključeva, jedan se bira za primarni ključ.

    - Superključ (eng. *Super Key*) čini skup atributa koji jedinstveno identifikuje svaki red u tabeli, s tim da može sadržati i atribute koji nisu neophodni.

Prethodno opisani pojmovi omogućavaju formalan i konzistentan prikaz podataka, uz jasno definisane veze i ograničenja, što čini osnovu za integritet i efikasnost relacionih baza.

2.4. Normalizacija baze podataka

Normalizacija je formalni postupak organizacije podataka u relacionoj bazi podataka koji ima za cilj minimizovanje redundancije, poboljšanje integriteta podataka i smanjenje mogućnosti anomalija pri izmjeni, brisanju ili unosu podataka [3]. Primjenom različitih normalnih formi, proces normalizacije obuhvata dekompoziciju tabela na manje i logički povezane tabele. Glavni razlozi za sprovođenje normalizacije su:

- eliminacija ponovljenih (redundantnih) podataka;

- obezbjeđivanje integriteta podataka (npr. izbjegavanje anomalija prilikom ažuriranja);
- pravljenje jednostavnijeg ili efikasnijeg modela podataka.

Proces normalizacije zasnovan je na matematičkoj teoriji skupova i funkcionalnih zavisnosti između atributa.

2.4.1. Normalne forme

Normalne forme predstavljaju stepene normalizacije koje treba zadovoljiti prilikom dizajniranja relacione baze. Svaki sljedeći nivo (viša normalna forma) podrazumijeva da su svi zahtjevi prethodne forme ispunjeni, uz nova dodatna pravila. Normalne forme se definišu kroz sljedeće nivoe:

1. **Prva normalna forma (1NF)**

   Tabela je u prvoj normalnoj formi ako su svi njeni atributi elementarni (nedjeljivi), odnosno ne sadrže skupove ili liste vrijednosti. Svaka vrijednost atributa u tabeli mora sadržati samo jednu vrijednost atributa. Primjer narušavanja 1NF je prikazan u tabeli 2.

   Tabela 2: Primjer narušavanja prve normalne forme

student_id	**ime**	**telefoni**
1	Marko	066121734,065233563

   Ukoliko tabela sadrži listu vrijednosti (kao kolona *telefoni* u tabeli 2), ona nije u 1NF. Kroz normalizaciju, ovakva tabela se može dekomponovati u dvije tabele sa kolonama:

   1. *student_id* i *ime*
   2. *id, student_id* i *telefon*.

2. **Druga normalna forma (2NF)**

   Tabela je u drugoj normalnoj formi ako je u 1NF i ako su svi neprimarni atributi potpuno funkcionalno zavisni od primarnog ključa [6]. To znači da, ukoliko tabela ima složeni primarni ključ (koji se sastoji od više kolona), svaki drugi atribut mora zavisiti od cijelog ključa, a ne samo od njegovog dijela.

   Primjer narušavanja 2NF je tabela sa sa kolonama: *student_id, predmet_id, predmet_naziv*. U tom slučaju, *predmet_naziv* je zavisan samo od *predmet_id*, a ne od cjelokupnog primarnog ključa (*student_id, predmet_id*). To ukazuje na parcijalnu funkcionalnu zavisnost. Po drugoj normalnoj formi, primjer treba da obuhvata dvije tabele. Prva tabela treba da ima kolone *student_id* i *predmet_id*, dok su *predmet_id* i *predmet_naziv* kolone druge tabele.

3. **Treća normalna forma (3NF)**

Tabela je u trećoj normalnoj formi ako je u 2NF i nijedan neključni atribut nije tranzitivno funkcionalno zavisan od primarnog ključa. To znači da ne postoji neključni atribut koji zavisi od drugog neključnog atributa [2].

Primjer narušavanja 3NF može biti tabela dizajnirana tako da ima kolone: *student_id, broj_indeksa, naziv_fakulteta*. Ako fakultet zavisi od indeksa, koji je već zavisan od *student_id*, postoji tranzitivna zavisnost. Da bi se zadovoljila treća normalna forma prethodnog primjera, potrebno je izdvojiti kolone u zasebne tabele: *student_id* i *broj_indeksa* u prvu tabelu, *broj_indeksa* i *naziv_fakulteta* u drugu.

4. Boyce-Codd normalna forma (BCNF)

Boyce-Codd normalna forma (BCNF) je stroža verzija 3NF. Tabela je u BCNF ukoliko je svaki određujući skup (eng. *determinant* - skup atributa od kojeg neki drugi atribut funkcionalno zavisi) kandidat za ključ. Drugim riječima, za svaku netrivijalnu funkcionalnu zavisnost X → Y, X mora biti superključ [1]. BCNF rješava probleme koji mogu nastati u situacijama gdje postoje složeni odnosi između više mogućih ključeva. Često se narušava u slučajevima kada, pored primarnog ključa, postoji još neki kandidat za ključ i funkcionalne zavisnosti nisu potpuno pokrivene primarnim ključem.

Primjer narušavanja BCNF može biti tabela u kojoj važe funkcionalne zavisnosti gdje kombinacija *student_id* i *predmet_id* određuje *profesor_id*, a pritom *profesor_id* određuje *predmet_id*. Tada dolazimo do situacije u kojoj *profesor_id* nije superključ, ali ipak određuje drugu kolonu *predmet_id*. Upravo zbog toga tabela ne zadovoljava uslove BCNF forme, jer postoji funkcionalna zavisnost u kojoj determinanta (*profesor_id*) nije superključ. Da bi ovaj primjer zadovoljio uslove BCNF forme, potrebno je izdvojiti dvije zasebne tabele sa sljedećim kolonama: *profesor_id* i *predmet_id* u prvu tabelu, *student_id* i *predmet_id* u drugu tabelu. Na ovaj način se eliminišu anomalije i svaka zavisnost polazi od primarnog ključa.

5. Četvrta normalna forma (4NF)

Tabela je u četvrtoj normalnoj formi (4NF) ako je u BCNF i dodatno, za svaku višestruku zavisnost u tabeli, lijeva strana te zavisnosti je superključ. Višestruke zavisnosti se javljaju kada određeni atributi zavise nezavisno jedan od drugog od istog ključa (eng. *Multivalued dependency*). 4NF eliminiše situacije gdje u jednoj tabeli postoji više nezavisnih listi vrijednosti za jedan entitet. Time se sprečavaju anomalije umetanja, brisanja i ažuriranja koje proizilaze iz takvih višestrukih (nezavisnih) zavisnosti.

Tabela 3: Primjer narušavanja četvrte normalne forme

student_id	**jezik**	**sport**
1	srpski	tenis

U tabeli 3 je prikazan primjer narušavanja 4NF. Jezik i sport su nezavisne višestruke vrijednosti vezane za istog studenta. Zbog toga, potrebno je normalizovati tabelu i podijeliti je u dvije nezavisne tabele sa kolonama:

1. *student_id* i *jezik*

2. *student_id* i *sport*.

6. Peta normalna forma (5NF) / Normalna forma projekcije-spajanja (PJNF)

Tabela je u petoj normalnoj formi ako je u 4NF i ako je svaka zavisnost spajanja (eng. *join*) u tabeli posljedica ključa. Drugim riječima, ako je moguće tabelu rastaviti u više tabela, a prilikom ponovnog spajanja se dobijaju isključivo originalni podaci, tada je tabela u 5NF.

5NF rješava vrlo rijetke slučajeve kada rekonstrukcija originalnog zapisa iz nekoliko podtabela može proizvesti lažne kombinacije podataka koji u stvarnosti nisu postojali. Ovakve situacije se rijetko javljaju u praksi, ali su moguće kada su u pitanju vrlo složeni odnosi između entiteta (tipično kod složenih veza „više na više“, na više atributa istovremeno).

Tabela 4: Primjer narušavanja pete normalne forme

proizvod	**dobavljač**	**potrošač**
Stolica	Dobavljač1	PotrošačA
Stolica	Dobavljač2	PotrošačB

Primjer narušavanja 5NF je prikazan u tabeli 4, jer ako se tabela rastavi na više tabela sa po dvije kolone i kasnije spoji, može se proizvesti kombinacija (npr. Stolica, Dobavljač1, PotrošačB) koja u stvarnosti ne postoji u originalnim podacima. Da bi se riješio taj problem, podatke je moguće organizovati u tri tabele sa kolonama:

1. *proizvod* i *dobavljač*,

2. *proizvod* i *potrošač*,

3. *dobavljač* i *potrošač* (ako je potrebno).

2.4.2. Denormalizacija

Denormalizacija je proces djelimičnog ili potpunog narušavanja normalnih formi, kombinovanjem više tabela ili uvođenjem redundantnih podataka u cilju poboljšanja performansi baze podataka [3]. Denormalizacija se obično sprovodi u sljedećim slučajevima:

- Ukoliko postoji veliki broj tabela, sa velikim brojem *join* operacija, gdje se značajno usporava izvršavanje upita.

- Kada je čitanje podataka znatno češće i kritičnije od ažuriranja (npr. sistemi za izvještavanje ili poslovnu analitiku).

- Ukoliko postoji potreba za agregiranim podacima koji se potražuju za izvještaje i statistiku.

Jedan od primjera denormalizacije može biti u aplikacijama gdje se često prikazuju zbirni podaci (npr. ukupna vrijednost narudžbe). Umjesto računanja, može se čuvati dodatna

kolona „ukupna_vrijednost” direktno u tabeli sa narudžbama (iako se ona tehnički može izračunati iz detalja narudžbe).

Treba naglasiti da denormalizacija uvodi rizik od nekonzistentnosti podataka, te se primjenjuje isključivo na osnovu analize upita, kao i performansi baze.

2.5. SQL - Standardni jezik za rad sa bazama podataka

Strukturirani upitni jezik (eng. *Structured Query Language - SQL*) predstavlja temeljni alat za rad sa relacionim bazama podataka. SQL je jedinstveni standard u gotovo svim današnjim informacionim sistemima zasnovanim na RDBMS pristupu. SQL je deklarativan jezik gdje korisnici i programeri opisuju šta žele da postignu (npr. koje podatke žele da pročitaju ili izmijene), dok stvarna implementacija „kako“ će baza izvršiti operacije ostaje skrivena iza složenog mehanizma izvršavanja, što omogućava izuzetnu fleksibilnost i portabilnost sistema [1].

SQL je bogat i fleksibilan jezik koji omogućava:

- definisanje strukture baze i pripadajućih objekata (tabele, indeksi, pogledi);
- manipulaciju podacima;
- upravljanje korisničkim privilegijama i bezbjednošću;
- transakcionu kontrolu, integritet i sigurnost.

SQL se funkcionalno dijeli na:

- **DDL (Data Definition Language)** - Jezik koji omogućava definisanje i upravljanje strukturom baze, tj. objektima unutar baze podataka (tabele, pogledi, indeksi, šeme itd). Osnovne DDL komande su:
  - CREATE - Kreiranje novih entiteta u bazi.
  - ALTER - Izmjena strukture postojećih entiteta.
  - DROP - Brisanje entiteta iz baze, zajedno sa svim podacima.
- **DML (Data Manipulation Language)** - Jezik za manipulaciju podacima obuhvata komande za rad sa samim podacima: unos, ažuriranje i brisanje podataka. Glavne DML komande su:
  - SELECT - Dohvatanje i pretraga podataka.
  - INSERT - Unos novih podataka.
  - UPDATE - Izmjena postojećih podataka.
  - DELETE - Brisanje podataka.

Važno je napomenuti da postoji i DQL (Data Query Language) koji obuhvata SELECT, ali ako se koriste i FROM ili WHERE u istom iskazu, kao što je gotovo uvijek slučaj, iskaz više nije dio DQL-a, nego DML-a.

- **DCL (Data Control Language)** - Jezik za kontrolu prava pristupa, koji se koristi za dodjeljivanje i ukidanje privilegija korisnicima nad entitetima u bazi podataka i to pomoću komandi kao što su:
  - GRANT - Dodjeljivanje prava korisnicima.
  - REVOKE - Oduzimanje prava.

  DCL je ključan za implementaciju bezbjednosti i kontrole pristupa, posebno u višekorisničkim sredinama.

2.6. MySQL

MySQL je jedan od najpoznatijih i najkorištenijih sistema za upravljanje relacionim bazama podataka (eng. *Relational Database Management System - RDBMS*). Prvu verziju je izdala kompanija MySQL AB 1995. godine, a danas ga razvija i održava kompanija Oracle Corporation [7]. Neke od najvažnijih karakteristika su:

- Otvoren izvorni kod - MySQL je slobodno dostupan i pod GNU GPLv2 licencom, ali postoje i komercijalne varijante sa naprednom podrškom i funkcijama.
- Višeplatformska podrška - Dostupan je za Linux, Windows, macOS i druge operativne sisteme.
- Podrška za SQL standard - Implementira značajan dio standarda SQL92 i SQL99, uz sopstveno proširenje funkcionalnosti.
- Podrška za više skladišnih mehanizama - MySQL podržava različite mehanizme skladištenja podataka (npr. InnoDB, MyISAM, Memory), što omogućava korisniku da izabere odgovarajući mehanizam u skladu sa zahtjevima aplikacije.
- Transakcije i ACID principi - Upotrebom InnoDB skladišta, MySQL podržava transakcije, strane ključeve i garantuje ACID svojstva: atomičnost, konzistentnost, izolovanost i trajnost.
- Replikacija i visoka dostupnost - Omogućava više tipova replika (master-slave, master-master) i podržava klastere za visoku dostupnost (MySQL Cluster).
- Visoka skalabilnost i performanse - Prepoznatljiv je po brzini izvršavanja upita, čak i pri obradi velikih količina podataka, kao i po podršci za horizontalnu skalabilnost u određenim verzijama.

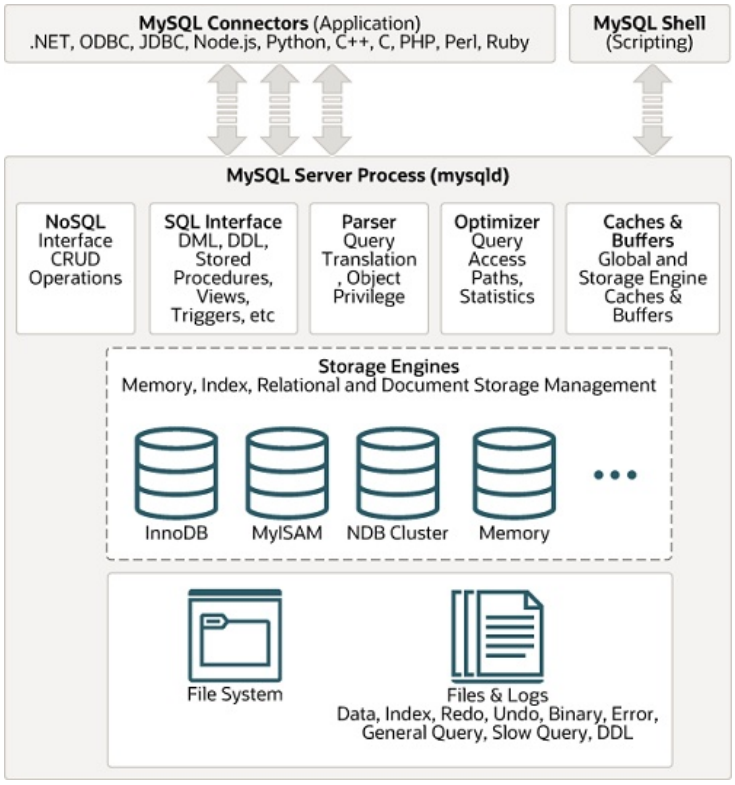
2.6.1. Arhitektura MySQL sistema

Arhitektura MySQL-a je pažljivo dizajnirana kako bi omogućila visoku modularnost, fleksibilnost i efikasnost u upravljanju podacima u različitim aplikacionim i produkcionim okruženjima. Sistem je organizovan u više slojeva koji međusobno sarađuju, a osnovnu

funkcionalnu podjelu čine konektorski sloj, SQL sloj i skladišni sloj, uz centralnu ulogu proširivog API-ja za skladišne mehanizme.

Konektorski sloj služi kao interfejs između korisničkih aplikacija i serverskog jezgra, omogućavajući podršku za širok spektar programskih jezika i tehnologija. Ovaj sloj upravlja autentifikacijom i autorizacijom korisnika, te obezbjeđuje siguran pristup bazi podataka.

Mjesto gdje se odvijaju ključni procesi logičkog upravljanja podacima u MySQL-u je tzv. SQL sloj. U ovom sloju dolazi do parsiranja i prevođenja SQL upita. Nakon toga se procjenjuje najbolji plan izvršavanja, te realizuju složene operacije. Posebno značajnu ulogu imaju algoritmi za keširanje upita i podataka, omogućavajući smanjenje kašnjenja i poboljšanje ukupnih performansi sistema. Upravo ovaj višeslojni pristup (prikazan na slici 1) omogućava prelazak sa relacionog na nerelacioni model, bez promjene osnovne serverske logike.



Slika 1: MySQL arhitektura [7]

Na najnižem nivou arhitekture smješten je skladišni sloj, odnosno mehanizam za fizičko upravljanje podacima i indeksima. Posebna karakteristika MySQL-a je tzv. Plugable Storage Engine API, koji omogućava serveru baze podataka da simultano podržava više različitih skladišnih mehanizama. Najznačajniji mehanizmi za skladištenje su:

- InnoDB - Podržava ACID, pogodan je za transakcione aplikacije.
- MyISAM - Optimizovan za brza čitanja, bez pune podrške za ACID.
- NDB Cluster - Koristi se za aplikacije koje zahtijevaju mala kašnjenja i visoku otpornost na greške.

- Memory storage engine - Često korišten za rad u distribuiranim okruženjima i brzu obradu podataka u radnoj memoriji računara.

Ovi mehanizmi komuniciraju sa fajl sistemom gdje se smještaju podaci, indeksi, log fajlovi, te sve ostale strukture bitne za dosljedno i bezbjedno upravljanje podacima i transakcijama.

Ovakav modularan dizajn omogućava jednostavno proširivanje i prilagođavanje različitim potrebama korisnika i tipovima opterećenja. Separacijom funkcionalnosti unutar svakog sloja sistema, postiže se visoka skalabilnost i lakše održavanje. Na slici 1 je prikazan dijagram arhitekture MySQL-a, gdje se jasno mogu uočiti svi ključni slojevi, interfejsi i tokovi podataka između pojedinih komponenti sistema.

2.6.2. Poređenje MySQL sistema sa drugim relacionim sistemima

Većina sistema za upravljanje relacionim bazama podataka (RDBMS) dijeli niz zajedničkih karakteristika koje čine osnovu njihove funkcionalnosti. Kao takvi, omogućavaju efikasno upravljanje podacima u različitim aplikacionim okruženjima. Prije svega, svi ovi sistemi zasnovani su na relacionom modelu podataka, što znači da podatke organizuju u vidu tabela koje mogu biti međusobno povezane.

Jedna od ključnih zajedničkih karakteristika je podrška za SQL, koji je standardni jezik za komunikaciju sa bazama podataka. Praktično svi poznati RDBMS sistemi implementiraju SQL standard, ili njegov podskup, te omogućavaju izvođenje osnovnih komandi za definisanje strukture baze, manipulaciju podacima, kao i upravljanje korisničkim pravima i kontrolom pristupa. Neki sistemi dodatno podržavaju i napredne SQL funkcionalnosti, uključujući rekurzivne upite, CTE (eng. *Common Table Expressions*), procedure i trigere. Pored toga, ovi sistemi u prvilu obezbjeđuju ACID svojstva, koja garantuju pouzdano izvršavanje transakcija. To znači da se svaka transakcija izvršava kao cjelina, a baza mora da se nalazi u konzistentnom stanju prije i nakon transakcije. Paralelne transakcije su izolovane jedna od druge. Trajni zapisi se čuvaju i nakon eventualnog kvara sistema. Navedena svojstva omogućavaju sigurno rukovanje podacima, uključujući i mogućnost vraćanja u prethodno stanje (eng. *Rollback*) u slučaju greške. Takođe, većina sistema podržava širok spektar programskih jezika i alata. Uz to, često su dostupni i ORM alati (eng. *Object-Relational Mapping*), kao i grafički korisnički interfejsi koji olakšavaju rad administratorima i programerima. Integracija sa raznovrsnim aplikacionim okruženjima čini ove sisteme pogodnim za razvoj kako jednostavnih aplikacija, tako i kompleksnih informacionih sistema.

Uprkos svim zajedničkim osobinama, među različitim RDBMS sistemima postoje i značajne razlike koje ih izdvajaju. Te razlike mogu se ogledati u načinu na koji implementiraju određene funkcionalnosti, stepenu podrške za standarde, performansama, mogućnostima skaliranja, bezbjednosnim mehanizmima, alatima za upravljanje, podrški za replikaciju i drugo. Neke od ovih karkteristika su prikazane u tabeli 5.

Tabela 5: Rezlike među RDBMS sistemima

Osobina	**Microsoft SQL Server**	**PostgreSQL**	**MySQL**
Licenca	Komercijalni softver	Otvoreni kod (PostgreSQL License)	Otvoreni kod (GNU GPL); vlasnička varijanta

Podrška za nadogradive mehanizme čuvanja	Jedan mehanizam	Jedan mehanizam, ali izuzetno proširiv	Više različitih skladišnih mehanizama (InnoDB, MyISAM, Memory, ...)
ACID / transakcije	Potpuno ACID	Potpuno ACID	Zajedno sa InnoDB, potpun ACID
Podrška za JSON	Podržano	Napredna obrada JSON objekata	Podržano
Podrška za napredne tipove	Podržano (XML, spatial, custom)	Podržano (arrays, hstore, custom)	Ograničeno (uglavnom standardni tipovi)
Replikacija/klaster	Napredan AlwaysOn/Clustering	Integrisana replikacija, log shipping	Više načina replikacije i organizacije klastera
Performanse	Izbalansiran OLTP i OLAP	Optimizovan za kompleksne upite i analitiku	Izuzetna brzina za upite, laka integracija sa veb aplikacijama
Podrška za proceduralni jezik	T-SQL (izuzetna proceduralna logika)	PL/pgSQL, PL/Python, PL/Perl	Ograničena
Popularnost za veb	Velika u enterprise i Windows okruženju	Popularan u data science, GIS, enterprise okruženjima	Visoka, posebno u LAMP/MEAN stekovima

3. SPRING BOOT

Početak XXI vijeka obilježen je velikim izazovima u razvoju Java aplikacija, prije svega kroz korištenje tada dominantnog Java EE (eng. *Enterprise Edition*) standarda. Java EE, iako robustan i skalabilan, patio je od suvišne kompleksnosti, zahtijevao je opsežnu konfiguraciju i korištenje kompleksnih komponenti poput EJB (eng. *Enterprise Java Beans*) koje su otežavale razvoj i održavanje aplikacija [8]. Ove okolnosti su dovele do potrebe za fleksibilnijim razvojnim okruženjem koje bi pojednostavilo razvoj poslovnih aplikacija u Javi.

Spring Framework je razvijen upravo kao odgovor na ove izazove. Osnivač projekta, Rod Johnson, prvi put je pisao o potrebi za novom paradigmom razvoja u svojoj knjizi "Expert One-on-One J2EE Design and Development" [9]. Iz ove inicijalne ideje, 2003. godine nastaje prva verzija Spring razvojnog okvira, sa ciljem da ponudi lakši, modularni okvir koji jednostavno integriše najbolje prakse, s tim da zadržava komponentne arhitekture i enterprise nivo robusnosti. Spring Framework je od samog početka bio softver otvorenog koda (eng. *Open-source*), čime je omogućio širokoj zajednici korisnika da doprinese njegovom razvoju [10]. Zahvaljujući aktivnoj zajednici i blagovremenom odgovoru na promjene u IT industriji, Spring je brzo evoluirao i tokom godina postao standard za razvoj modernih Java aplikacija [11].

3.1. Osnovne karakteristike Springa

Spring Framework je projektovan da pojednostavi razvoj enterprise aplikacija u Java programskom jeziku. Jedna od njegovih najvažnijih inovacija je inverzija kontrole (eng. *Inversion of Control - IoC*) i injektovanje zavisnosti (eng. *Dependency Injection - DI*). Ovaj pristup omogućava da objekti budu definisani putem konfiguracije, pri čemu se njihove zavisnosti automatski injektuju. Na taj način se postiže veća modularnost aplikacije, olakšava testiranje komponenti i značajno smanjuje snažna sprega između klasa. Pored toga, Spring pruža podršku za aspektno orijentisano programiranje (eng. *Aspect-Oriented Programming - AOP*), što omogućava elegantno razdvajanje poslovne logike od tehničkih aspekata sistema. Ova paradigma omogućava da se aspektno orijentisani zadaci implementiraju nezavisno od osnovne funkcionalnosti aplikacije, čime se povećava preglednost i održivost koda.

Jedna od istaknutijih prednosti Springa je i jednostavna integracija sa drugim Java tehnologijama, uključujući JDBC, JPA/Hibernate, JMS i druge. Korištenjem apstraktnih slojeva i šablona, Spring eliminiše potrebu za ponavljajućim kodom (eng. *Boilerplate code*), što ne samo da ubrzava razvoj, već olakšava i eventualnu migraciju ili zamjenu tehnologija u okviru projekta. Spring takođe nudi transakcionu podršku, omogućavajući deklarativno upravljanje transakcijama. Ovo je ključno za razvoj pouzdanih enterprise aplikacija, jer pojednostavljuje rukovanje kompleksnim transakcionim scenarijima bez potrebe za ručnim upravljanjem transakcijama u kodu.

Arhitektura Spring razvojnog okvira je modularna. Sastoji od više nezavisnih modula koji mogu biti korišteni samostalno ili u kombinaciji, u zavisnosti od potreba projekta. Neki od najčešće korištenih modula uključuju Spring Core, Spring MVC, Spring Data, Spring Security i druge.

Pored mehanizama koji olakšavaju razvoj, Spring pruža jednostavan način testiranja koda i logike. Zahvaljujući IoC mehanizmu, mogućnosti korištenja *mock* objekata i izolovanoj konfiguraciji, značajno je olakšano izvođenje jediničnih i integracionih testova. Time je u potpunosti omogućen stabilan i kvalitetan razvoj softverskih rješenja.

Spring se ne ističe samo mogućnostima i funkcionalnostima, već i filozofijom razvoja softvera koja podstiče dobre prakse, održivost i dugoročnu stabilnost softverskih rješenja. U nastavku su prikazani osnovni principi i vrijednosti koje oblikuju razvojni pristup unutar Spring ekosistema:

- „Don't Repeat Yourself" (DRY) i „Convention over Configuration" - Spring podstiče smanjenje šablonskog koda i naglašava dobre prakse konfiguracije kroz konvencije, automatsko otkrivanje komponenti (eng. *Component scanning*) i minimizaciju eksplicitne konfiguracije.

- Povećanje fleksibilnosti i prilagodljivosti - Spring podržava različite arhitekture (od monolitne do mikroservisne) i može da se koristiti samo u onim dijelovima koji su potrebni aplikaciji. Takođe, lako integriše postojeće i naslijeđene (eng. *legacy*) Java sisteme.

- Otvorenost i interoperabilnost - Kroz svoju otvorenost prema drugim razvojnim okvirima, standardima i raznim ekstenzijama, Spring je omogućio jednostavnu migraciju i integraciju sa raznovrsnim softverskim rješenjima.

- Fokus na deklarativnom programiranju - Umjesto imperativnog upravljanja nekim zadacima (poput transakcija ili bezbjednosti), Spring nudi deklarativni način podešavanja putem anotacija ili XML konfiguracije.

- Jednostavnost i produktivnost - Od samog nastanka, cilj Springa bio je da pojednostavi razvoj enterprise Java aplikacija bez kompromitovanja snage i fleksibilnosti [9].

Ove vrijednosti i principi učinili su da Spring postane izuzetno popularna platforma za razvoj Java aplikacija na globalnom nivou.

3.2. Spring moduli

Ekosistem Springa čini skup međusobno povezanih modula, koji omogućavaju razvoj skalabilnih, modularnih i lako testabilnih aplikacija. Prvobitno je bio zamišljen kao lagani kontejner za IoC i AOP, ali se vremenom razvio u bogat ekosistem koji pokriva skoro sve aspekte moderne veb aplikacije. Najvažniji moduli koji grade Spring ekosistem su:

- **Spring Core** - Predstavlja osnovu cjelokupnog razvojnog okvira, obezbjeđujući ključne mehanizme kao što su inverzija kontrole (IoC) i injektovanje zavisnosti (DI), kao i podršku kroz BeanFactory, ApplicationContext i različite pomoćne alate [12].

- **Spring Beans** - Ovaj modul realizuje definiciju bean-ova i manipulaciju njihovim životnim ciklusom. Omogućava detaljnu kontrolu konfiguracije objekata.

- **Spring Context** - Omogućava pristup kontekstu aplikacije i napredniju konfiguraciju (internacionalizacija, resursi, događaji u okviru aplikacije itd).

- **Spring Expression Language (SpEL)** - Služi za implementiranje dinamičkog izražavanja i evaluaciju izraza unutar konteksta Spring beanova.

- **Spring AOP** - Omogućava primjenu principa aspektno orijentisanog programiranja, kojim se izdvajaju funkcionalnosti koje nisu direktno vezane za osnovnu poslovnu logiku, poput evidentiranja događaja (logovanja), obezbjeđenja pristupa i kontrole transakcija, i drugo [13].

- **Spring Data Access/Integration** - Pruža podršku za pristup podacima i integraciju sa različitim spoljnim sistemima. Kroz ujednačen i pojednostavljen API je omogućen rad sa bazama podataka, ORM alatima, porukama i drugim resursima.

- **Spring JDBC** - Ovaj modul pojednostavljuje rad sa JDBC API-jem u Javi, omogućavajući lakšu interakciju sa relacionim bazama podataka. Obezbjeđuje šablone koji automatizuju rutinske zadatke poput otvaranja i zatvaranja konekcija, obrade izuzetaka i izvršavanja SQL upita, čime se značajno smanjuje količina šablonskog koda.

- **Spring ORM** - Modul omogućava integraciju sa popularnim objektno-relacionim mapiranjima (ORM) u Javi, kao što su Hibernate, Java Persistence API (JPA), JDO i iBATIS. Obezbjeđuje dosljedan način upravljanja perzistencijom podataka kroz jednostavne konfiguracije i automatizaciju transakcija, uz istovremeno iskorištavanje svih prednosti koje nude izabrani ORM alati.

- **Spring Transaction Management** - Pruža fleksibilan mehanizam za upravljanje transakcijama. Takođe, podržava i deklarativni i programski pristup, uz mogućnost integracije sa različitim tehnologijama za perzistenciju podataka.

- **Spring JMS** - Modul omogućava integraciju sa sistemima za razmjenu poruka, pružajući podršku za asinhronu komunikaciju između komponenti distribuiranih aplikacija. Kroz upotrebu šablona kao što je JmsTemplate, pojednostavljuje se slanje i primanje poruka uz automatizovano upravljanje konekcijama i resursima.

- **Spring Web** - Osnova za razvoj veb aplikacija. Omogućava razvoj veb aplikacija korištenjem Spring MVC arhitekture, sa podrškom za RESTful servise, kontrolere, obradu zahtjeva i prikaz podataka.

- **Spring Web MVC** - Omogućava razvoj veb aplikacija zasnovanih na Model-View-Controller (MVC) arhitekturi. Na taj način se razdvajaju prezentacioni, poslovni i upravljački sloj aplikacije, čime se postiže bolja organizacija koda i lakše testiranje. Spring Web MVC, takođe, pruža mehanizam za mapiranje HTTP zahtjeva, validaciju podataka, obradu formulara i generisanje odgovora kroz različite tehnologije (JSP, Thymeleaf itd).

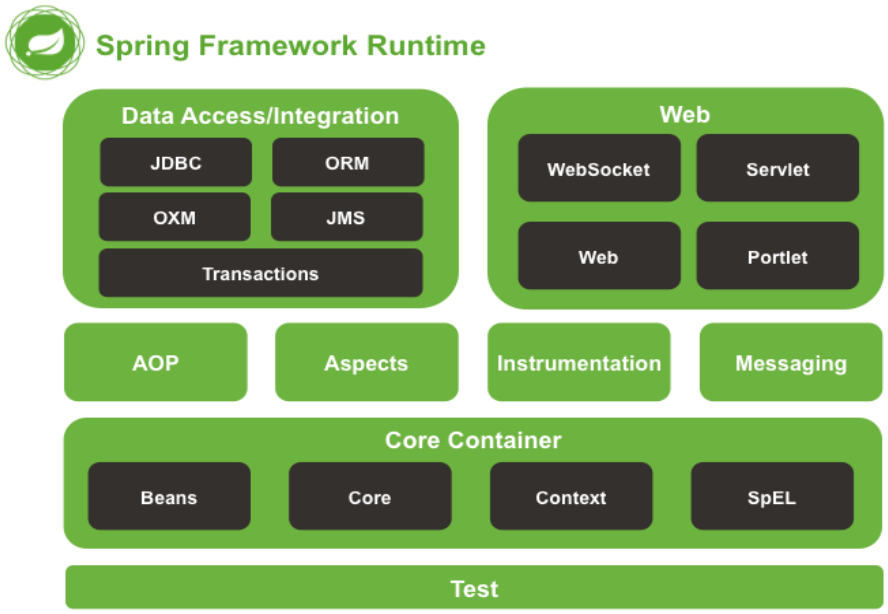
- **Spring Web Websocket** - Modul pruža punu podršku za dvosmjernu, asinhronu komunikaciju između klijenta i servera, putem WebSocket protokola. Ovaj modul je integrisan sa Spring Web i Spring Messaging komponentama, te tako omogućava jednostavnu implementaciju funkcionalnosti u realnom vremenu.

- **Spring Security** - Obezbjeđuje sveobuhvatne mehanizme za autentifikaciju i autorizaciju u Java aplikacijama. Modul omogućava zaštitu aplikacionih resursa kroz

fleksibilnu konfiguraciju pravila pristupa, integraciju sa različitim mehanizmima identiteta (poput LDAP, OAuth, JWT), kao i podršku za zaštitu od uobičajenih bezbjednosnih prijetnji (poput CSRF napada, session hijacking-a i drugih). Spring Security je dizajniran da bude lako proširiv i prilagodljiv različitim bezbjednosnim zahtjevima.

- **Spring Test** - Koristi se za testiranje komponenti razvijenih unutar Spring okvira. Obezbjeđuje jednostavno pisanje i izvođenje jediničnih i integracionih testova kroz integraciju sa popularnim testnim okvirima poput JUnit i TestNG. Modul olakšava simulaciju Spring konteksta, upravljanje transakcijama tokom testiranja i samo testiranje veb sloja.

Kontinualnim razvojem, ekosistem je proširen dodatnim projektima koji proširuju funkcionalnost za specifične potrebe i tehnologije [11]. Slika 2 prikazuje osnovni ekosistem Spring okvira.



Slika 2: Ekosistem Spring razvojnog okvira [14]

3.3. Spring Boot

Spring Boot predstavlja projekat unutar Spring platforme, osmišljen s ciljem da omogući brzu izradu produkciono spremnih aplikacija uz minimalnu konfiguraciju. Njegova filozofija počiva na principima koji omogućavaju jednostavniji i brži razvoj, što ga čini izuzetno pogodnim za savremene razvojne paradigme poput agilnog razvoja i mikroservisa.

Jedna od ključnih prednosti Spring Boot-a je brza izrada funkcionalnih aplikacija. Zahvaljujući auto-konfiguraciji i unaprijed pripremljenim starter modulima, moguće je pokrenuti potpuno funkcionalan prototip aplikacije u svega nekoliko minuta. Time se značajno skraćuje inicijalna faza razvoja i omogućavaju brze iteracije. Spring Boot takođe omogućava nezavisnu samostalnu arhitekturu, to jeste razvoj aplikacija koje ne zavise od eksterno instaliranih servera. Aplikacije se mogu jednostavno pokrenuti u bilo kom

okruženju, što pojednostavljuje serviranje (eng. *Deployment*) i testiranje. U kontekstu softverskih arhitektura, Spring Boot pruža podršku za mikroservisni pristup, omogućavajući laku integraciju sa komponentama Spring Cloud okruženja. Mehanizmi kao što su distribuirana konfiguracija, otkrivanje servisa (eng. *Service discovery*) i mehanizam osigurača (eng. *Circuit breaker*) predstavljaju osnovne komponente cloud-native aplikacija, a njihova implementacija pomoću Spring Boot-a znatno olakšava razvoj skalabilnih i otpornijih sistema. Neizostavna osobina Spring Boot-a jeste automatska konfiguracija aplikacije zasnovana na zavisnostima koje se nalaze u projektu. Minimalna potreba za ručnim podešavanjima olakšava razvoj i eliminiše veliki dio repetitivnog konfigurisanja koje je ranije bilo neophodno.

Zahvaljujući svojoj modularnosti, automatizaciji i mogućnosti za rad u cloud i mikroservisnim okruženjima, Spring Boot je postao široko prihvaćena tehnologija u razvoju modernih Java poslovnih rješenja.

3.4. Razlike između Spring Framework-a i Spring Boot-a

Spring Framework obuhvata bazni set biblioteka i alata za razvoj Java poslovnih aplikacija, ali zahtijeva opsežnu manuelnu konfiguraciju, postavljanje projekata i integraciju raznih modula. Sve to može biti jako zahtjevno. Pored XML konfiguracija, bilo je nužno postavljanje spoljašnjih zavisnosti (eng. *Dependencies*), definisanje bean-ova, transakcija i još mnogo manualnog rada. Spring Boot je nastao kao odgovor na izazove postavljanja tradicionalnog Spring okruženja. Njegova glavna svrha je pojednostavljeno kreiranje, konfiguracija i pokretanje novih Spring aplikacija [15]. Spring Boot to postiže pomoću sljedećih principa:

- Autokonfiguracija - Automatski detektuje koje su konfiguracije potrebne na osnovu dostupnih biblioteka u projektu, eliminišući potrebu ručne konfiguracije.
- Ugrađeni serveri - Spring Boot aplikacije startuju se kao samostalni Java programi, sa ugrađenim serverima (Tomcat, Jetty, Undertow). Nije potrebno koristiti eksterni aplikacioni server.
- Preporučene podrazumijevane vrijednosti (eng. *Opinionated defaults*) - Spring Boot donosi unaprijed postavljene optimalne vrijednosti za većinu konfiguracija, poput ugrađene baze i unaprijed definisanih profila za razvoj, testiranje i produkciju.
- Spring Boot Starter POM - Unaprijed definisane grupe zavisnosti (tzv. starteri) za najčešće korištene funkcionalnosti.
- Actuator - Omogućava nadzor, monitoring i upravljanje aplikacijom kroz REST pristupne tačke.

Spring Boot nije zamjena za Spring Framework, već je njegova nadgradnja koja pomaže razvoju brzih, modernih i testabilnih aplikacija [12]. U tabeli 6 prikazane su ključne razlike između Spring Framework-a i Spring Boot-a.

Tabela 6: Poređenje Spring Framework-a i Spring Boot-a

Karakteristika	**Spring Framework**	**Spring Boot**
Konfiguracija	Ručna, detaljna	Automatska
Pokretanje aplikacije	Eksterni server	Ugrađeni server
Upravljanje zavisnostima	Manuelno	Starter POMs
Ciljevi	Fleksibilnost	Produktivnost i brzina
Monitoring	Nije ugrađen	Actuator modul

3.5. Spring IoC i DI

Tekstualne CLI (eng. *Command-Line Interface*) aplikacije predstavljaju tipičan metod za implementaciju proceduralnog programiranja, gdje tok programa određuje programer i kod se izvršava sekvencijalno, dio po dio. Sa druge strane, aplikacije na operativnim sistemima sa grafičkim interfejsom omogućavaju da tok programa zavisi od korisničkih unosa i događaja, koji su dinamični. Dok su proceduralni načini programiranja bili dominantni, bilo je potrebno pronaći način da se kontrola toka programa premjesti sa tradicionalnog proceduralnog pristupa (gdje programer diktira tok) na eksterne izvore, kao što su razvojni okviri ili komponente, koji su određivali tok programa. Ovo pomjeranje kontrole je ono što se naziva inverzija kontrole (IoC). IoC predstavlja vrlo uopšten princip i dio je većine razvojnih okvira [16].

IoC princip podrazumijeva delegiranje procesa upravljanja objektima na specijalizovani upravljački kontejner, čime se smanjuje zavisnost između modula. Injektovanje zavisnosti, kao konkretan način implementacije IoC-a, omogućava eksplicitno definisanje zavisnosti između objekata, čime se olakšava konfiguracija i unaprjeđuje fleksibilnost aplikativnog koda. U okviru Spring okruženja, ovaj pristup omogućava definisanje klasa i njihovih međusobnih relacija na deklarativan način, a sam Spring kontejner preuzima odgovornost za instanciranje, konfigurisanje i upravljanje životnim ciklusom svih objekata u sistemu (tzv. bean-ova).

3.5.1. Bean

U okviru Spring ekosistema, pojam bean označava svaki objekat čijim životnim ciklusom (uključujući instanciranje, konfiguraciju i uništavanje) u potpunosti upravlja Spring kontejner. Svaki objekat koji je pod kontrolom Springa automatski postaje bean te se registruje u kontejneru, čime ulazi u kontrolisani tok upravljanja putem centralizovanih mehanizama aplikacije. Bean-ovi stoga čine temelj svake Spring aplikacije, predstavljajući ključne komponente kroz koje se ostvaruje aplikativna logika i međuzavisnosti složenih softverskih sistema [16]. Definisanje bean-a u Springu obuhvata nekoliko osnovnih koraka:

- određivanje klase koju bean predstavlja;
- popunjavanje svojstava (zavisnosti);
- određivanje opsega (eng. *Scope*) svakog bean-a;
- obrada različitih faza životnog ciklusa (poput inicijalizacije i uništavanja).

Spring omogućava deklaraciju bean-ova na više načina: tradicionalna XML konfiguracija, Java anotacije (*@Component*, *@Service*, *@Repository*, *@Controller*) i konfiguracione klase označene anotacijama *@Configuration* i *@Bean*. Ovakva fleksibilnost omogućava prilagođavanje i proširivost prema potrebama konkretnog softverskog problema te pogoduje primjeni principa čistog koda i održive arhitekture [14].

3.5.2. IoC kontejner

Inverzna kontrola predstavlja jedan od temeljnih principa na kojem počiva razvoj savremenih softverskih sistema u Springu, ali i u objektno-orijentisanom programiranju uopšte. Za razliku od tradicionalnog pristupa, kod kojeg objekti sami kontrolišu kreiranje i upravljanje svojim zavisnostima, IoC tu odgovornost prebacuje na specijalizovani kontejner. U kontekstu Springa, IoC kontejner je odgovoran za iniciranje i održavanje životnog ciklusa bean-ova, upravljanje njihovim zavisnostima i obezbjeđivanje infrastrukture za povezivanje različitih komponenti aplikacije.

Instanciranje objekata najčešće se ostvaruje refleksijom, dok se vezivanje zavisnosti sprovodi automatski, bez direktne intervencije korisnika u izvorni kod [17]. Zajedno s upravljanjem životnim ciklusima, IoC kontejner podržava i registraciju „post-processor" bean-ova, kao i osluškivače događaja (eng. *Event listeners*), čime dodatno proširuje funkcionalnosti. Implementacija Spring IoC kontejnera realizovana je putem ApplicationContext interfejsa i njegovih različitih realizacija, koje internim mehanizmima omogućavaju rukovanje objektima i njihovim zavisnostima. Ovaj princip kontrole se u praksi sinonimno koristi sa pojmom injektovanja zavisnosti, pošto Spring kontejner automatski injektuje, odnosno ubacuje, sve neophodne zavisnosti u bean tokom njegovog instanciranja. Spring omogućava realizaciju različitih tipova DI-a:

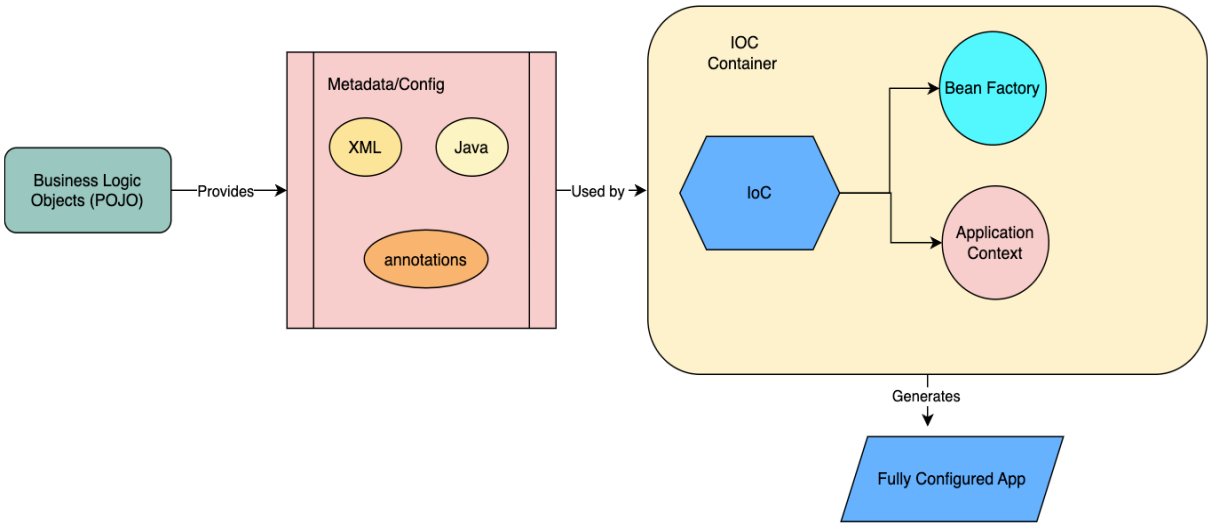
- *Constructor-based Dependency Injection* - Podrazumijeva prosljeđivanje svih zavisnosti objekta preko konstruktora, što obezbjeđuje njihov pravovremeni raspored prilikom kreiranja bean-a i pogoduje nepromjenljivosti komponenti.

- *Setter-based Dependency Injection* - Zasniva se na postavljanju zavisnosti kroz setter metode nakon instanciranja objekta, omogućavajući veću fleksibilnost, ali manje strogu kontrolu nad zavisnosti.

- *Field-based Dependency Injection* - Koristi direktno anotiranje polja (najčešće putem *@Autowired* anotacije), čime omogućava još jednostavnije deklarisanje zavisnosti. Korištenje ove metode olakšava automatsko prepoznavanje i popunjavanje referenci od strane kontejnera, ali može otežati testiranje i narušiti enkapsulaciju [18].

Anotacija *@Autowired* omogućava Springu da detektuje i ubaci odgovarajuće zavisnosti automatski, bez potrebe za eksplicitnom konfiguracijom bean-ova. S obzirom na to, dobra praksa nalaže da se *@Autowired* koristi prvenstveno na konstruktorima ili setter metodama radi boljeg nadzora i mogućnosti testiranja.

Slika 3 prikazuje arhitekturu Spring IoC kontejnera. Proces započinje kreiranjem poslovnih objekata (POJO) sa određenom konfiguracijom. Ova konfiguracija se zatim koristi od strane IoC kontejnera, koji kroz BeanFactory i ApplicationContext komponente instancira i upravlja bean-ovima. Na kraju, IoC kontejner generiše potpuno konfigurisanu aplikaciju spremnu za upotrebu.



Slika 3: Zavisnosti u IoC kontejneru [19]

3.5.3. ApplicationContext

ApplicationContext predstavlja centralni interfejs u Spring okruženju, koji omogućava kompletnu kontrolu nad IoC kontejnerom i njegovim funkcionalnostima. ApplicationContext je napredniji IoC kontejner koji je izgrađen na osnovu BeanFactory-ja i pruža dodatne mogućnosti kao što su objavljivanje događaja, razmjena poruka i internacionalizacija [20].

U okviru rada aplikacije, ApplicationContext je odgovoran za automatsko učitavanje definicija bean-ova iz različitih izvora, kao što su: XML datoteke, Java anotacije ili konfiguracione klase. Poslije toga, on preuzima zadatak instanciranja, konfiguracije i međusobnog povezivanja bean-ova, upravljajući pri tome i njihovim životnim ciklusom.

Postoji nekoliko različitih implementacija interfejsa ApplicationContext, koje su prilagođene specifičnim potrebama korisnika i različitim tipovima aplikacija. Klase kao što su ClassPathXmlApplicationContext i FileSystemXmlApplicationContext omogućavaju učitavanje konfiguracije iz XML fajlova, bilo sa classpath-a ili direktno iz fajl sistema. Sa druge strane, klasa AnnotationConfigApplicationContext je optimizovana za rad sa Java konfiguracionim klasama i anotacijama u samom kodu, čime se omogućava potpun prelazak sa XML konfiguracije na anotacioni pristup. Klasa GenericApplicationContext predstavlja opštu implementaciju interfejsa i omogućava manuelnu registraciju bean-ova. Za veb aplikacije, Spring nudi WebApplicationContext kao specijalizovanu implementaciju za integraciju sa Spring MVC okvirom i servlet kontejnerima.

Kao rezultat navedenih unapređenja, razvoj aplikacija postaje značajno jednostavniji, skalabilniji i brži, dok se istovremeno omogućava veći stepen fleksibilnosti u izboru konfiguracionog pristupa.

3.5.4. Životni ciklus bean-a

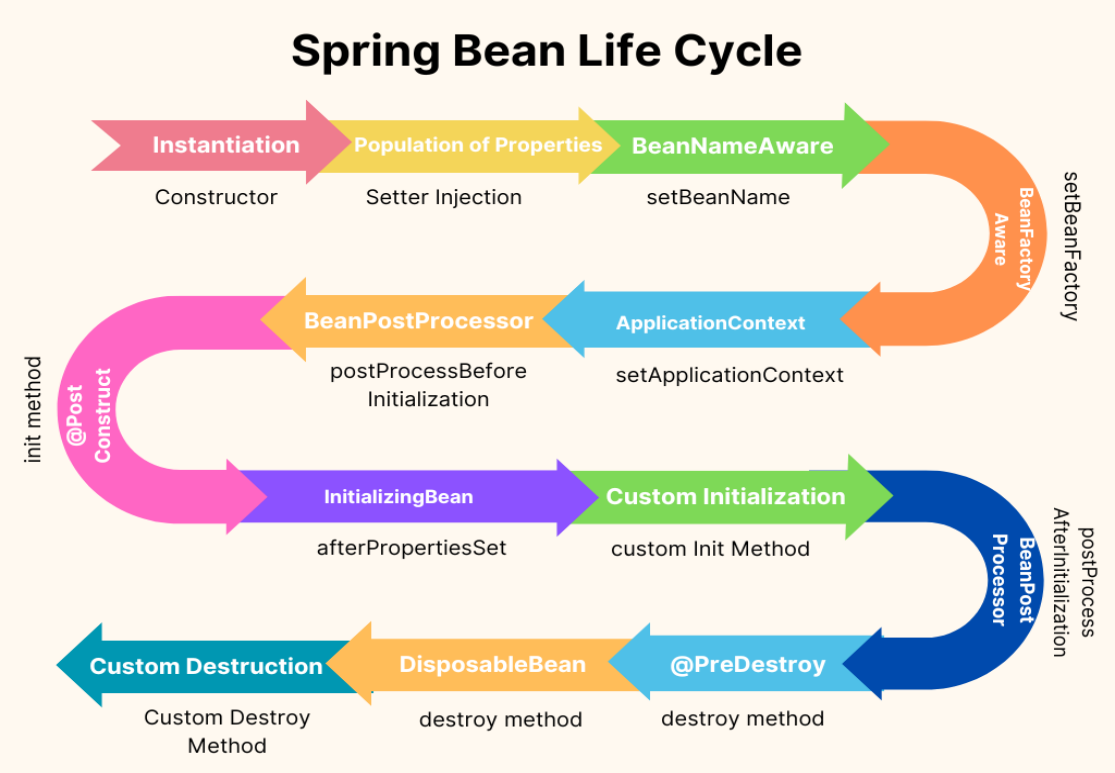
Životni ciklus bean-a u Spring okruženju obuhvata sve faze od njegovog definisanja do uništavanja, a upravljanje ovim procesom u potpunosti je povjereno IoC kontejneru. Prvi

korak, prikazan na slici 4 pod nazivom „Instantiation”, predstavlja učitavanje definicije bean-a pri pokretanju aplikacije, gdje Spring prepoznaje konfiguracije iz XML fajlova, Java konfiguracionih klasa ili anotacija. Na ovoj osnovi kontejner pristupa instanciranju samog objekta korištenjem Java refleksije (Constructor), čime omogućava visoku fleksibilnost pri kreiranju svih potrebnih instanci [21].

Sljedeća faza je popunjavanje svih zavisnosti, odnosno povezivanje bean-ova sa odgovarajućim resursima i drugim komponentama aplikacije. Ove zavisnosti mogu biti automatizovano injektovane kroz konstruktor, set metode ili direktno na polja, u skladu s izabranim načinom konfigurisanja.

Nakon što su svi podaci i reference obezbijeđeni, slijedi dio ciklusa u kojem kontejner obavlja posebne operacije. Ukoliko se implementiraju standardizovani interfejsi poput InitializingBean ili DisposableBean, ili su upotrebljene anotacije kao što su *@PostConstruct* i *@PreDestroy*, kontejner automatski izvršava predodređene metode za dodatno podešavanje ili oslobađanje resursa [14].

Slika 4 ilustruje kompletan životni ciklus Spring Boot bean-a, ilustrujući svaki korak kroz koji bean prolazi od instanciranja do uništavanja.



Slika 4: Životni ciklus Spring Boot Bean-a [22]

Bean-ovi mogu imati različit životni ciklus, koji se bira u skladu sa specifičnim zahtjevima aplikacije. Najčešće korišteni opsezi životnog ciklusa su:

- Singleton - Kreira se samo jedna instanca bean-a za čitavu aplikaciju.
- Prototype - Na svaki novi zahtjev se kreira nova instanca bean-a.

- Request - Bean se instancira za svaki HTTP zahtjev.
- Session - Bean postoji dok traje korisnička sesija.
- Application - Bean dijeljen na nivou čitavog veb konteksta.
- Websocket - Nadležan za upravljanje WebSocket konekcijama.

Izbor odgovarajućeg životnog ciklusa omogućava kontrolu nad resursima i ponašanjem bean-ova u različitim dijelovima aplikacije.

3.5.5. Prednosti Dependency Injection pristupa

Primjena DI principa u okviru IoC paradigme donosi brojne prednosti koje su doprinijele popularnosti Spring okvira. Najistaknutija prednost se ogleda u postizanju slabe sprege (eng. *Decoupling*) između komponenata. Klase više ne instanciraju niti kontrolišu svoje zavisnosti, već njihovo povezivanje i upravljanje povjeravaju IoC kontejneru. Ovakav pristup omogućava jasnije razgraničenje odgovornosti unutar aplikacionog koda, te u značajnoj mjeri pojednostavljuje refaktorisanje, ubrzava razvoj i unaprjeđuje održavanje sistema.

Još jedna ključna prednost DI pristupa je znatno olakšana testabilnost softverskih rješenja. S obzirom da su zavisnosti eksplicitno izložene, na primjer kroz konstruktore ili set metode, prilikom testiranja se sa lakoćom mogu koristiti zamjenske implementacije ili *mock* objekti. Spring dodatno olakšava ovu praksu integrisanim Spring Test modulom, čime se omogućava efektivna automatizovana provjera funkcionalnosti aplikacija na svim nivoima.

Fleksibilnost sistema se dodatno ogleda u mogućnosti jednostavne zamjene implementacija pojedinačnih bean-ova kroz promjenu konfiguracije, bez potrebe za izmjenama u izvornom kodu. Takva modularnost značajno doprinosi i ponovnoj upotrebljivosti koda. Samostalni bean-ovi se mogu lako koristiti u različitim dijelovima aplikacije ili integrisati u nove projekte.

Spring DI omogućava i jednostavno upravljanje konfiguracijama za različita okruženja putem sistema profila i eksternih *properties* fajlova. Time se olakšava postavljanje razvojnih, testnih i produkcijskih okruženja. Dodatno, korištenjem AOP programiranja u kombinaciji sa DI principima, moguće je implementirati funkcionalnosti koje se prepliću kroz više slojeva aplikacije a da time ne narušavaju poslovnu logiku.

3.6. Spring Data i rad sa bazama podataka

Efikasno i pouzdano upravljanje podacima predstavlja jedan od ključnih zahtjeva za izgradnju robusnih i skalabilnih aplikacija. Komunikacija sa bazama podataka je tradicionalno podrazumijevala pisanje opsežnog i često složenog koda za pristup, mapiranje i upravljanje podacima. Sve to je povećavalo složenost aplikacionog sloja i otežavalo održavanje. Razvojem Java ekosistema, pojavili su se standardizovani pristupi i alati koji značajno pojednostavljuju ove procese. Jedan od najvažnijih je JPA (eng. *Java Persistence API*), koji implementira univerzalni model za objektnu perzistenciju u relacionim bazama podataka. Napredne implementacije, poput Hibernate-a, su dodatno unaprijedile praktičnu primjenu ovih standarda, omogućujući specijalizovane ORM mehanizme [23]. Spring, u okviru Spring Data modula, integriše ove tehnologije i razvija napredne mehanizme za rad sa

perzistentnim slojem. Spring Data obezbjeđuje deklarativan i lako prilagodljiv način upravljanja podacima, čime se smanjuje količina ručno pisanog šablonskog koda, podržava efikasna izgradnja i održavanje složenih poslovnih rješenja. Ovaj pristup omogućava bržu i kvalitetniju izradu aplikacija, te doprinosi i većoj sigurnosti i konzistentnosti prilikom manipulacije podacima.

3.6.1. Java Persistence API

Java Persistence API (JPA) predstavlja standardizovanu specifikaciju unutar Java EE platforme, namijenjenu za objektnu perzistenciju i mapiranje objekata (eng. *Plain Old Java Objects - POJO*) na relacione baze podataka. JPA definiše skup anotacija i apstraktnih programskih interfejsa pomoću kojih se implementiraju mapiranje, manipulacija i upravljanje životnim ciklusom perzistentnih entiteta. Osnovna prednost ovog pristupa jeste što omogućava slabu spregu između aplikacione logike i konkretne tehnologije baze podataka, pružajući tako prenosivost i fleksibilnost koda [23].

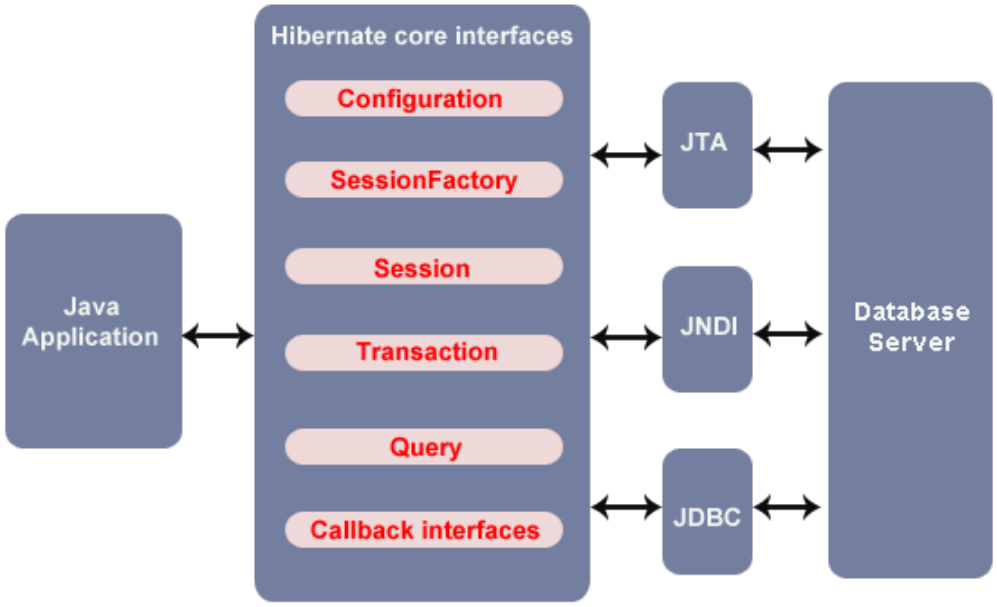
JPA specifikacija podržava različite napredne mogućnosti, uključujući složene upite putem JPQL jezika, upravljanje relacijama između entiteta, automatsko ažuriranje podataka i pozivanje metoda životnog ciklusa.

3.6.2. Hibernate

Hibernate je najrasprostranjenija i funkcionalno najbogatija implementacija JPA standarda. On omogućava napredno objektno-relaciono mapiranje (ORM), pri čemu se složena logika prevođenja između objekata i baza podataka automatski delegira Hibernate mehanizmu. Među ključnim karakteristikama Hibernate-a izdvajaju se: podrška za učitavanje komponenti sistema po potrebi (eng. *Lazy loading*), mehanizmi keširanja, kontrola transakcija i kompatibilnost sa različitim tipovima baza podataka [24]. Dodatno, Hibernate omogućava detaljnu konfiguraciju i optimizaciju upita, modelovanje relacija i efikasnog rukovanja velikim količinama podataka.

Jedan od osnovnih ciljeva Hibernate-a jeste da oslobodi programera manuelnog upravljanja transakcijama i sesijama, prebacujući ovu kompleksnost na Spring kontejner i prateće servise. Programer može da se koncentriše na opis poslovnih pravila i modela podataka, dok se tehnički dio sprovodi unutar aplikacije. Ovakav pristup značajno smanjuje šanse za pojavu grešaka, poboljšava održavanje i čini razvoj aplikacija efikasnijim.

Slika 5 prikazuje arhitekturu Hibernate-a. Java aplikacija komunicira sa Hibernate-om putem osnovnih interfejsa kao što su: Configuration, SessionFactory, Session, Transaction, Query i Callback interfejsi. Ovi interfejsi omogućavaju konfiguraciju, kreiranje i upravljanje sesijama i transakcijama, kao i izvršavanje upita. Hibernate se zatim povezuje sa bazom podataka uz pomoć nekog od servisa: JTA (Java Transaction API), JNDI (Java Naming and Directory Interface) i JDBC (Java Database Connectivity). Takva struktura omogućava transparentan i efikasan pristup podacima na serverskoj strani baze [25].



Slika 5: Arhitektura Hibernate-a

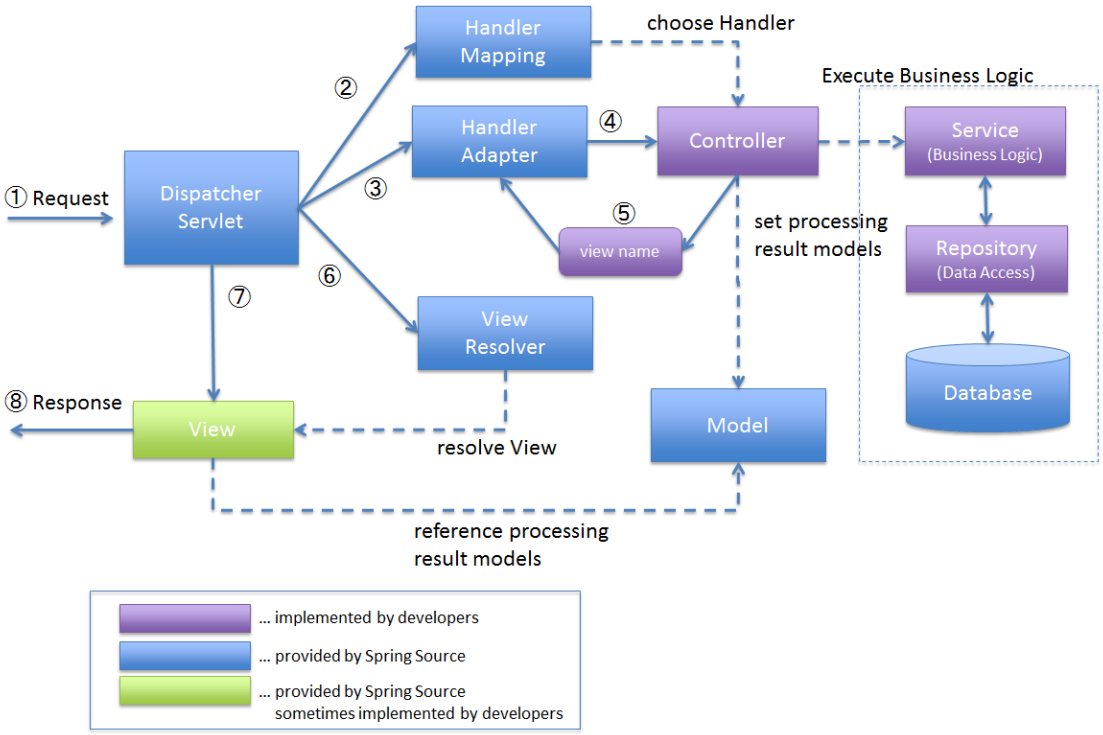
3.7. Spring MVC (Model-View-Controller)

Spring MVC predstavlja integralni dio Spring razvojnog okvira i omogućava izgradnju veb aplikacija primjenom koncepata Model-View-Controller (MVC) arhitekturalnog obrasca. MVC pristup doprinosi jasnom razdvajanju aplikacione logike na elemente:

- Model – Predstavlja podatke sa kojima aplikacija upravlja.
- View - Prezentacioni element koji je odgovoran za generisanje korisničkog interfejsa, bilo da je riječ o HTML stranicama, JSON podacima ili drugim formatima.
- Controller - Element koji prima korisničke zahtjeve, obrađuje ih i poziva potrebne servise. Takođe, ovaj sloj upravlja komunikacijom između modela i prikaza.

Ovakva segmentacija omogućava razvoj, testiranje i održavanje slojeva nezavisno, čime se postiže visoka modularnost veb aplikacija.

Standardna arhitektura Spring MVC aplikacije obuhvata četiri glavna elementa: kontroler (eng. *Controller*), servis (eng. *Service*), repozitorijum za pristup podacima (eng. *Repository*) i domenski model (eng. *Model/Entity*). Osnova svakog veb zahtjeva prolazi kroz DispatcherServlet koji je centralna komponenta Spring MVC-a.



Slika 6: Arhitektura i tok podataka u Spring MVC aplikaciji [26]

Na slici 6 je prikazan tok obrade HTTP zahtjeva u uobičajenoj Spring MVC aplikaciji:

1. Klijent šalje HTTP zahtjev, koji najprije prima DispatcherServlet.
2. DispatcherServlet upotrebljava HandlerMapping i HandlerAdapter za identifikaciju i pozivanje odgovarajućeg kontrolera.
3. Controller prepoznaje tip zahtjeva, te delegira zadatke servisnom sloju.
4. Sloj servisa sprovodi poslovnu logiku uz pomoć repozitorijuma i podatke iz baze reprezentuje kroz modele.
5. Nakon obrade, Controller prosljeđuje pripremljene podatke DispatcherServlet-u.
6. DispatcherServlet angažuje ViewResolver za lociranje odgovarajućeg prikaznog sloja.
7. Podaci iz modela se šalju View elementu, gdje se generiše konačni korisnički interfejs ili formatiran odgovor (npr. JSON objekat).
8. Na kraju, rezultat obrade se vraća klijentu kao HTTP odgovor.

Ovako granularno razdvajanje odgovornosti omogućava lako skaliranje aplikacije i nezavisno proširivanje svakog sloja odvojeno.

3.8. Spring Security

Povezanost javno dostupnih interfejsa i složenih softverskih arhitektura dovodi do velike izloženosti bezbjednosnim prijetnjama. Veb aplikacije su dominantna meta napada koji često rezultuju kompromitovanjem osjetljivih podataka, kao i zloupotrebom infrastrukture. Obezbjeđivanje veb aplikacija se može svesti na ostvarenje tri osnovna cilja:

- Autentifikacija korisnika - Aplikacija treba da pouzdano identifikuje korisnika ili softverski entitet koji pristupa sistemu.

- Autorizacija - Nakon što je identitet ustanovljen, aplikacija treba da omogući granularno upravljanje resursima i operacijama koje određeni korisnik ima pravo da izvrši.

- Obezbjeđivanje integriteta i povjerljivosti podataka - Čuvanje i razmjena podataka trebaju biti zaštićeni od neovlaštenog pristupa, modifikacije ili presretanja, koristeći savremene kriptografske i arhitekturalne mehanizme.

Bezbjednost softverskih aplikacija podrazumijeva zaštitu od različitih vrsta napada, među kojima se posebno izdvajaju:

- Krađa identiteta (eng. *phishing*) - Napadač se lažno predstavlja i tako pokušava da prevari korisnika kako bi došao do njegovih povjerljivih podataka (na primjer, do korisničkog imena i lozinke).

- XSS (*Cross-site scripting*) – Ostvaruje se tako što napadač pokušava da ubaci zlonamjerni kod (najčešće JavaScript) na stranicu koju će drugi korisnici posjetiti. Na taj način može doći do krađe podataka ili kompromitovanja naloga.

- *SQL injection* - Napad u kojem napadač ubacuje zlonamjerne SQL upite preko formi ili URL-ova aplikacije, čime može pristupiti ili izmijeniti podatke u bazi, ili u najgorem slučaju potpuno izbrisati podatke.

- Otimanje sesije (eng. *Session hijacking*) - Zloupotrebljava se aktivna sesija korisnika tako što napadač preuzima kontrolu nad njom, najčešće presretanjem kolačića ili drugih sesijskih tokena.

Zbog mogućih prijetnji, neophodno je da se prilikom razvoja posebna pažnja posveti implementaciji bezbjednosnih mjera i da se redovno održavanju postojeći zaštitni mehanizmi. U tom kontekstu, Spring Security je postao referentni modul za primjenu najboljih bezbjednosnih praksi u Java ekosistemu. Ovaj modul nudi integrisanu podršku za kontrolu autentifikacije, autorizacije i zaštite od najčešćih napada. Njegova filozofija se zasniva na fleksibilnoj i deklarativnoj zaštiti koja je primjenljiva i u tradicionalnim i u modernim mikroservisnim arhitekturama.

3.8.1. Autentifikacija

Autentifikacija (eng. *Authentication*) je proces kojim se potvrđuje identitet entiteta koji zahtijeva pristup sistemu. Pitanje na koje autentifikacija odgovara jeste: „Ko si ti?”. Pravilno implementirana autentifikacija je preduslov za bilo kakvu značajnu zaštitu resursa, jer se svi mehanizmi kontrole pristupa baziraju na tačnom poznavanju korisničkog identiteta. U

modernim sistemima postoji više metoda autentifikacije, koje se često klasifikuju prema sigurnosti i praktičnosti:

1. **Autentifikacija lozinkom (eng. *Password-based authentication*)** – Često korištena metoda, gdje korisnik dokazuje identitet unosom korisničkog imena i lozinke. Lozinke bi se trebale čuvati u heširanom obliku, nikad u izvornom tekstu. Iako jednostavna za implementaciju, ova metoda je podložna napadima poput phishing-a, brute-force pokušaja i curenja lozinki.

2. **Višefaktorska autentifikacija (eng. *Multi-factor Authentication - MFA*)** - Kombinuje više metoda i to najčešće:

   - nešto što korisnik zna (lozinka, PIN);
   - nešto što korisnik ima (token, smart kartica, telefon za SMS/OTP kod);
   - nešto što korisnik jeste (biometrijski podaci kao što su otisak prsta, prepoznavanje lica/dužice).

   MFA značajno povećava bezbjednost, jer smanjuje šanse kompromitovanja autentifikacije čak i u slučaju gubitka jednog faktora.

3. **Autentifikacija putem tokena (eng. *Token-based authentication*) -** Korisnik, nakon uspješne potvrde identiteta, dobija jedinstveni token (npr. JSON Web Token), koji se koristi za dalju autentifikaciju umjesto tradicionalne sesije ili kolačića. Ovakav način autentifikacije se široko primjenjuje i za SPA (eng. *Single Page Application*), a i za mikroservisne arhitekture.

4. **Delegirana autentifikacija** - Mehanizam u kojem aplikacija koristi neki drugi servis za provjeru identiteta korisnika, najčešće putem protokola OAuth2 ili OpenID Connect. U ovom modelu, korisnik se autentifikuje preko pouzdane treće strane (poput Googlea, Facebooka ili GitHuba), dok aplikacija preuzima pristupni token i koristi ga za upravljanje pravima identiteta. Ovakav metod omogućava upravljanje identitetom bez potrebe za lokalnim čuvanjem korisničkih kredencijala, te je posebno primjenjiv u B2B i B2C okruženjima.

5. **Klijentski sertifikati (eng. *Certificate-based authentication*) -** Korisnici ili aplikacije se identifikuju kriptografskim sertifikatima, koji se verifikuju na strani servera. Ovo je veoma siguran metod koji se koristi u sistemima visoke bezbjednosti (bankarstvo, infrastruktura, interni sistemi).

6. **Biometrija -** Temelji se na jedinstvenim fizičkim karakteristikama korisnika, kao što su otisak prsta, biometrija oka, prepoznavanje glasa i slično. Pogodna je za mobilne uređaje, fizičke pristupne kontrolere i ID kontrolne sisteme.

7. **Jednokratne lozinke (eng. *One-Time Passwords - OTP*) -** Pristup se omogućava kroz kod koji se generiše i važi ograničeno vrijeme (npr. Time-based OTP - TOTP ili HMAC-OTP). Često se implementira kao dodatni sloj u okviru MFA.

Kombinovanje više pomenutih metoda (npr. lozinka i OTP) značajno se povećava sigurnost sistema.

3.8.2. Autorizacija

Nadovezujući se na proces autentifikacije, autorizacija (eng. *Authorization*) zauzima važnu ulogu u očuvanju sigurnosti resursa i podataka. Dok proces autentifikacije odgovara na pitanje ko je korisnik, autorizacija određuje koje akcije definisanom korisniku ili entitetu su dozvoljene nad određenim resursima. Drugim riječima, u kojoj mjeri i pod kojim uslovima korisnik ili entitet može pristupiti funkcionalnostima i podacima sistema [27]. U suštini, autorizacija predstavlja skup mehanizama i pravila dizajniranih da upravljaju pravima pristupa i raspodjelom dozvola među korisnicima, procesima, ili drugim učesnicima u informacionom sistemu.

Modeli autorizacije se razvijaju u skladu sa potrebama organizacije, kao i stepenom povjerljivosti i osjetljivosti podataka kojima se upravlja. Među klasičnim pristupima ističu se:

- *Discretionary Access Control (DAC)* - Prava pristupa određuje vlasnik objekta (npr. korisnik može samostalno dodijeliti pristup fajlu).

- *Mandatory Access Control (MAC)* - Pravo pristupa određuje centralizovano pravilo, često implementirano u sigurnosno kritičnim okruženjima (vojni sistemi, državne institucije).

- *Role-Based Access Control (RBAC)* - Pristup se organizuje na osnovu pripadnosti ulozi. Dozvole se ne dodjeljuju direktno korisnicima, već apstraktnim ulogama koje predstavljaju funkcionalne ili organizacione pozicije. Korisnici nasljeđuju prava i dopuštene akcije onih uloga kojima su pridruženi, što značajno olakšava upravljanje i reviziju prava pristupa u velikim i kompleksnim informacionim sistemima. Ovo je često korišten model u modernim poslovnim aplikacijama.

Razvoj distribuiranih i dinamičkih arhitektura, naročito u kontekstu mikroservisnih modela, doveo je do upotrebe modela gdje su kontrole pristupa zasnovane na atributima (ABAC). Ovakav pristup omogućava donošenje odluka o autorizaciji na temelju skupa karakteristika subjekta, objekta i konteksta (npr. korisnička pozicija, vlasništvo, vrijeme pristupa, lokacija i slično), što omogućava izuzetno granularnu i dinamičnu kontrolu [28].

U mikroservisnim okruženjima značajne su delegirane autorizacije i token-bazirani pristupi, gdje se dio ili cijeli proces provjere prava pristupa povjerava eksternim autoritativnim servisima (tzv. IAM rješenjima) kroz standardne protokole kao što su OAuth 2.0 i OpenID Connect [29]. U ovom slučaju, dozvole i pravo na pristup resursima kodirani su unutar tokena tako da ih aplikacija validira pri svakom zahtjevu.

Mehanizmi autorizacije mogu se implementirati na različitim slojevima sistema: od prava pristupa na nivou pojedinačnih API ruta i funkcionalnosti, preko restrikcija nad specifičnim podacima, do definisanja detaljnih opsega resursa i operacija nad istim. Razvojni okviri, poput Spring Security modula, omogućavaju istovremeno centralizovano i deklarativno upravljanje politikama pristupa. Oslanjaju se na kombinaciju uloga, permisija, atributa i eksternih autoriteta, a sve to uz dosljednu primjenu principa najmanjih privilegija.

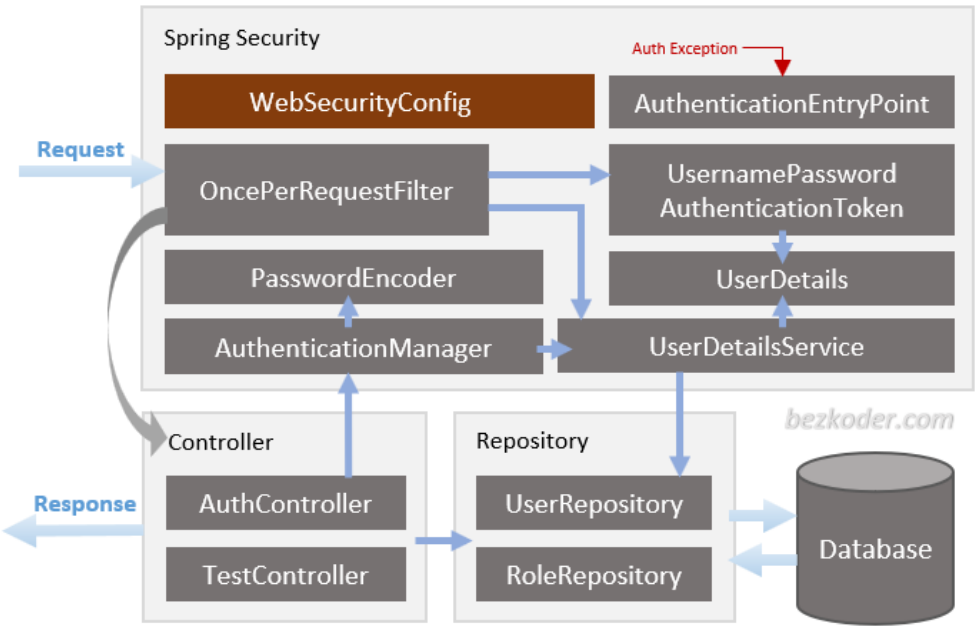
Efikasan sistem autorizacije nužno mora biti robustan, skalabilan i lako prilagodljiv promjenama organizacione i tehničke strukture. Implementacija ovakvog sistema direktno utiče na smanjenje rizika od neautorizovanog pristupa, minimizuje moguće štete

prouzrokovane kompromitovanim kredencijalima, te omogućava zadovoljavanje regulatornih i zakonskih zahtjeva u pogledu zaštite podataka. Autorizacija koja je pažljivo razvijena i sprovedena, čini nezaobilazan dio svakog naprednog informacionog sistema.

3.8.3. Konfiguracija Spring Security modula za REST API

Spring Security kao modul funkcioniše kroz lanac filtera (eng. *Security Filter Chain*), gdje se svi zahtjevi najprije prosljeđuju filterima koji provjeravaju validnost korisnika i prava pristupa. Osnovne komponente su:

- SecurityConfig - centralna klasa za konfiguraciju pravila bezbjednosti;
- UserDetailsService - interfejs za učitavanje korisničkih podataka iz baze;
- PasswordEncoder - sigurno čuvanje i verifikacija lozinki;
- AuthenticationEntryPoint – izuzetak koji se koristi kada autentifikacija ne uspije.



Slika 7: Lanac filtera Spring Security modula [30]

Slika 7 prikazuje arhitekturu Spring Security okvira u tipičnoj veb aplikaciji. Prikazan je tok autentifikacije i autorizacije, počevši od slanja korisničkog zahtjeva. Zahtjev najprije prolazi kroz bezbjednosni filter OncePerRequestFilter, čije ponašanje i pravila određuje konfiguracijska klasa WebSecurityConfig. Zatim filter prosljeđuje zahtjeve dalje prema komponentama zaduženim za autentifikaciju. AuthenticationManager provjerava korisničke podatke uz pomoć PasswordEncoder klase. Autentifikacioni token (UsernamePasswordAuthenticationToken) i korisnički detalji (UserDetails) su glavni elementi procesa prijave, pri čemu podatke o korisnicima i rolama preuzima UserDetailsService iz repozitorijuma (UserRepository, RoleRepository) koji komuniciraju sa bazom podataka. U slučaju greške pri autentifikaciji, proces se prebacuje na dio za obradu

izuzetaka. Njima upravlja AuthenticationEntryPoint. Nakon uspješne autentifikacije, kontroleri vraćaju odgovor korisniku.

```java
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .csrf( CsrfConfigurer<HttpSecurity> csrf -> csrf.disable())
                .authorizeHttpRequests( AuthorizationManagerRequestMat... auth -> auth
                        .requestMatchers("/api/auth/**").permitAll()
                        .requestMatchers("/api/admin/**").hasRole("ADMIN")
                        .anyRequest().authenticated()
                )
                .sessionManagement( SessionManagementConfigurer<HttpSecurity> sess -> sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
        return http.build();
    }

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

Slika 8: Primjer podešavanja Spring Security konfiguracije u kodu

U primjeru sa slike 8, sve rute osim „*/api/auth/***” zahtijevaju autentifikaciju, dok je ruta „*/api/admin/***” dostupna samo korisnicima sa ulogom *ADMIN*.

3.8.4. JSON Web Tokens (JWT)

Tradicionalni pristupi autentifikaciji u veb aplikacijama su podrazumijevali održavanje korisničke sesije na strani servera, gdje su podaci o identitetu i privilegijama korisnika čuvani u sesijskim skladištima (memorija, baza, distribuirani keš). Ovakav pristup otežava skaliranje aplikacije i povećava njenu složenost. U modernom razvoju, fokus je na servisima bez stanja (eng. *Stateless services*). To su aplikacije koje ne održavaju klijentsku sesiju i ne očekuju da svaki zahtjev sadrži dovoljno informacija za validaciju autentifikacije i autorizacije. Tu se pojavljuje JWT kao jednostavno, elegantno i standardizovano rješenje.

JWT predstavlja otvoreni industrijski standard (RFC 7519) za kompaktno i nezavisno prenošenje informacija između strana u vidu JSON objekta, koji može biti digitalno potpisan koristeći HMAC algoritme ili asimetrične ključeve (RSA/ECDSA) radi verifikacije integriteta i autentičnosti podataka [31].

Osnovna upotreba JWT-a je u implementaciji autentifikacije bez stanja, gdje server, nakon što uspješno verifikuje identitet korisnika, izdaje kriptografski potpisan JWT token i prosljeđuje ga klijentu. Klijent potom koristi ovaj token za pristup svim zaštićenim resursima, bez potrebe za ponovnim logovanjem sve dok je token validan.

Struktura JWT tokena

JWT se sastoji iz tri jasno odvojena dijela koja su kodovana Base64Url algoritmom i međusobno razdvojenih tačkom:

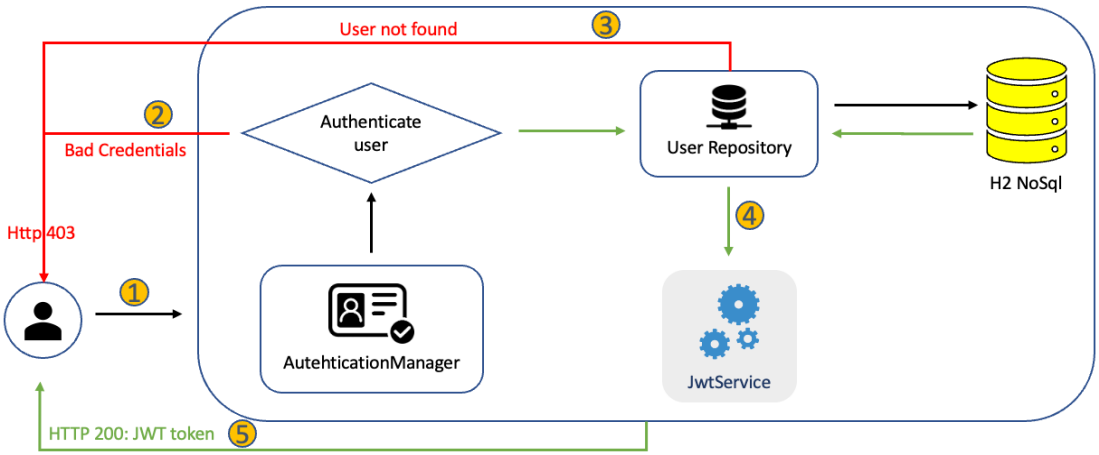
1. **Header** - Nosi informacije o tipu tokena (najčešće „JWT"”) i algoritmu koji se koristi za potpisivanje (npr. "alg": "HS256").

2. **Payload** - Sadrži skup tvrdnji (eng. *claim*) koji predstavljaju informacije o korisniku i tokenu. Standardne tvrdnje su:
   - sub (subject) - subjekat koga token identifikuje, tipično user ID,
   - exp (expiry) - vrijeme isteka tokena,
   - iat (issued at) - vrijeme izdavanja,
   - roles ili sličan korisnički claim za privilegije (npr. korisničke uloge, prava),
   - proizvoljni podaci, s tim da se ne preporučuje slanje povjerljivih informacija jer JWT može biti pročitan od strane svakoga ko ga posjeduje (npr. putem browser alatki).
3. **Signature** - Kreira se tako što se header i payload kodiraju i konkateniraju, a zatim se nad njima primijeni odgovarajući kriptografski algoritam uz korištenje privatnog ključa. Na ovaj način, svaki pokušaj izmjene sadržaja tokena može biti otkriven verifikacijom potpisa na serveru.

Tok autentifikacije i autorizacije sa JWT tokenima

U nastavku je prikazan uobičajeni proces autentifikacije i autorizacije koristeći JWT token. Prvi proces obuhvata inicijalnu autentifikaciju korisnika putem „*/login*" pristupne tačke i dobijanje JWT tokena, dok drugi proces opisuje način na koji se svaki naredni zahtjev prema zaštićenim resursima autorizuje.

1. Autentifikacija korisnika i izdavanje JWT tokena (slika 9)

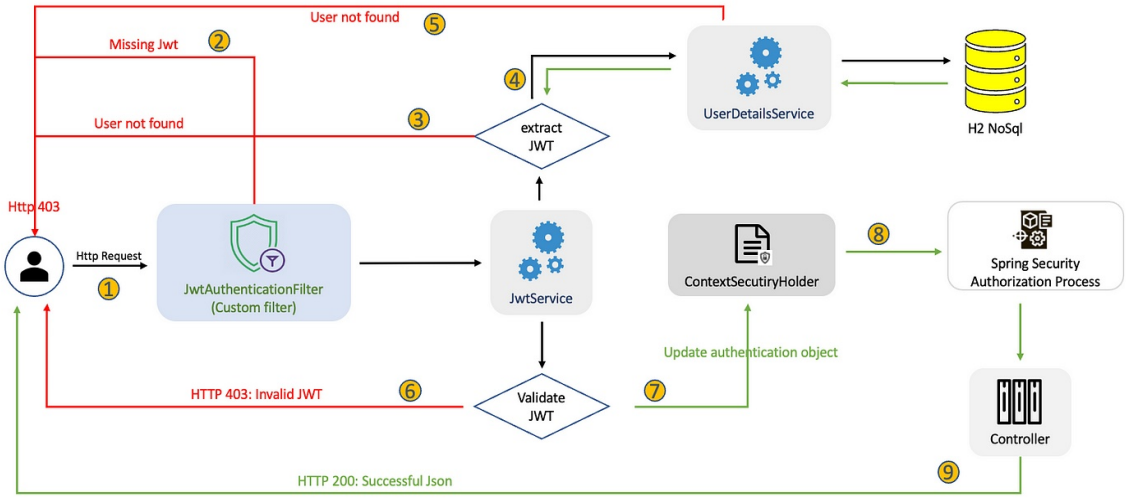


Slika 9: Tok zahtjeva za autentifikaciju korisnika [32]

Proces započinje slanjem korisničkih kredencijala na „*/login*" pristupnu tačku (korak 1 na dijagramu). U pozadini, odgovarajući menadžer autentifikacije preuzima zadatak provjere identiteta. Ako su kredencijali pogrešni, korisniku se vraća HTTP 403 status sa naznakom nepravilne autentifikacije (korak 2). U slučaju da korisnički nalog nije pronađen, sistem takođe vraća odgovarajuću grešku (korak 3).

Pri uspješnom prepoznavanju korisnika, podaci se povlače iz korisničkog repozitorijuma, te se zatim pristupa generisanju JWT tokena putem JWT servisa (korak 4). Token sadrži sve neophodne podatke: digitalni potpis, vremensko ograničenje važenja, podatke o identitetu korisnika, kao i skup tvrdnji, uključujući uloge i dozvole definisane prema sigurnosnoj politici sistema. JWT se potom dostavlja klijentu kao standardizovani odgovor (korak 5).

2. Autorizacija korisnika pri svakom zahtjevu (slika 10)



Slika 10: Tok validacije JWT tokena za HTTP zahtjev [32]

Za svaki naredni zahtjev prema zaštićenim resursima, klijent šalje dobijeni JWT kao dio HTTP zaglavlja. Na serverskoj strani, specijalizovani filter procesira dolazni zahtjev (korak 1). Ukoliko token nedostaje ili nije validan, korisnik odmah dobija HTTP odgovor sa 403 statusom (koraci 2 i 3).

Validni tokeni se predaju JWT servisu koji ih dekodira i provjerava (korak 4). U procesu validacije posebno se analizira integritet potpisa, validnost vremena važenja, identitet subjekta i prava pristupa (claim-ovi). Informacije iz tokena se potom upisuju u sigurnosni kontekst aplikacije (ContextSecurityHolder), ažurirajući autentifikacioni objekat (korak 7). Tek nakon uspješno sprovedene validacije i autorizacije, zahtjev se propušta do odgovarajućeg kontrolera i odgovor se vraća klijentu.

Ovakav pristup ima niz prednosti. Neke od njih su potpuna arhitektura bez stanja i sesija, odlična skalabilnost i interoperabilnost među mikroservisima. Sam JWT je dobro standardizovan i podržan u većini modernih razvojnih okruženja. S obzirom da nije potreban centralni server za održavanje stanja sesija, svi potrebni detalji o identitetu i pravima korisnika sadržani su u samom tokenu, što odgovara zahtjevima distribuiranih i cloud aplikacija. Sigurnost ovakvog sistema oslanja se na ispravno upravljanje privatnim ključevima, ograničavanje vremena važenja tokena, kao i zaštitu tokena od krađe i zloupotrebe.

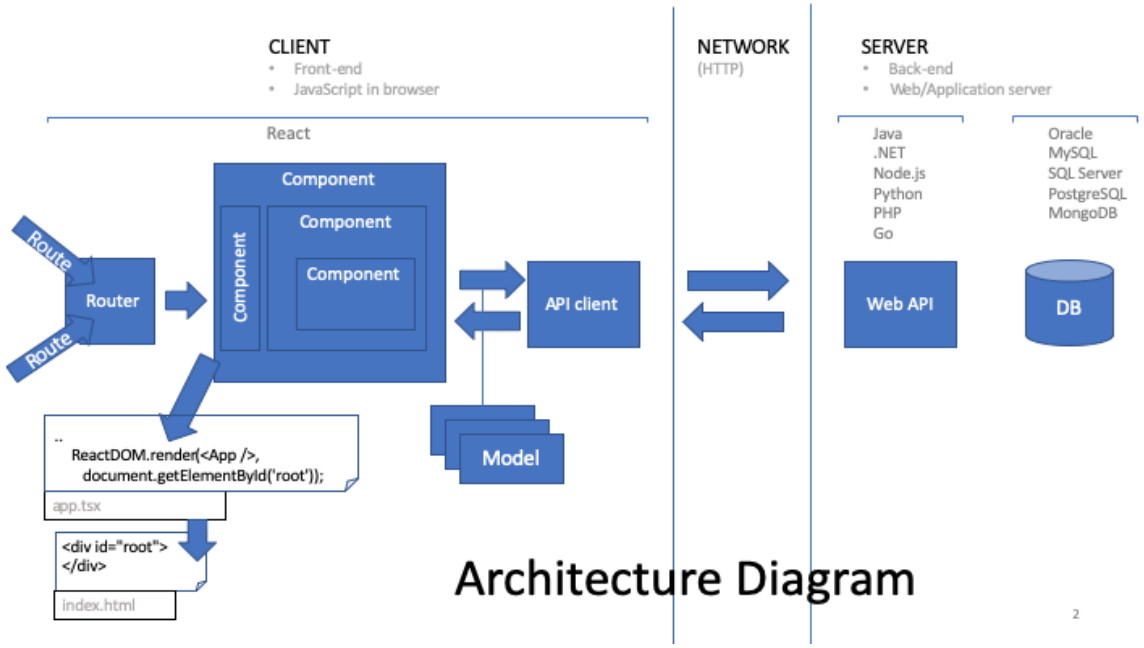
4. REACT

React je JavaScript biblioteka otvorenog koda namijenjena za razvoj korisničkih interfejsa, prvenstveno za razvoj jednostraničnih aplikacija (SPA) kojima se omogućava brzo i efikasno ažuriranje prikaza korisničkog interfejsa u skladu sa promjenama podataka. Prvobitno ga je razvio Joran Walke u okviru kompanije Facebook. Inspiraciju je pronašao u internom PHP okviru za dinamičko generisanje HTML-a. Prvo javno predstavljanje React-a bilo je 2013. godine i od tada React bilježi rast popularnosti i upotrebe kako u industriji, tako i u akademskim krugovima [33].

Razvoj React-a je od samog početka vođen potrebom za unapređenjem korisničkog doživljaja u velikim i složenim veb aplikacijama, gdje tradicionalne JavaScript biblioteke i okviri nisu pružali dovoljno dobre performanse, skalabilnost i modularnost. Facebook-ove aplikacije, kao što su News Feed i Instagram, bile su među prvim velikim sistemima koje su koristile React. Nakon inicijalnog izdanja, React je brzo postao jedan od često korištenih alata za razvoj veb aplikacija. Njegova popularnost dodatno je porasla pojavom alata i biblioteka koje proširuju njegove mogućnosti, uključujući React Router, Redux i Context API, React Native i druge [34]. Otvorenost ka zajednici, odlična dokumentacija i podrška velikog broja developera učinili su React jednom od najvažnijih i najuticajnijih tehnologija u veb razvoju.

4.1. Komponentna arhitektura

Razvoj korisničkih interfejsa zahtijeva modularizaciju i jasnu organizaciju koda, pri čemu se React ističe korištenjem komponentne arhitekture kao temeljnog pristupa razvoju. U React okruženju, aplikacije se ne grade kao jedinstvene i spregnute cjeline, već iz mnoštva nezavisnih, funkcionalno zaokruženih jedinica koje se nazivaju komponente (eng. *Components*). Komponente predstavljaju osnovne gradivne blokove React aplikacija, jer svaka od njih enkapsulira određenu funkcionalnost i dio vizuelnog prikaza interfejsa [35].



Slika 11: Dijagram arhitekture standardne React aplikacije [35]

Slika 11 prikazuje tipičnu arhitekturu modernih veb aplikacija zasnovanih na React biblioteci, gdje je poseban naglasak stavljen na razdvajanje klijentske i serverske strane te njihovu međusobnu komunikaciju putem mrežnih (HTTP) zahtjeva.

Ovakav koncept razvoja značajno doprinosi čitljivosti i održavanju izvornog koda. Jasno razgraničene, manje jedinice koda olakšavaju izolovano testiranje i brže otkrivanje grešaka, a njihova ponovna upotrebljivost omogućava značajne uštede vremena i napora u razvoju većih ili sličnih projekata. Modularnost podstiče i paralelizaciju razvoja, jer više inženjera ili timova može raditi na različitim komponentama istovremeno.

4.1.1. Definicija i vrste komponenti

U React-u svaka komponenta ima ulogu samostalnog funkcionalnog entiteta, čiji je zadatak da kreira određeni prikaz korisničkog interfejsa na osnovu proslijeđenih ulaznih podataka. Formalno gledano, komponenta je JavaScript funkcija ili klasa koja vraća elemente opisane putem JSX sintakse (JavaScript XML). Pravilno kreirane nezavisne komponente sadrže logiku i izgled dijela korisničkog interfejsa koji predstavljaju.

Tokom svog razvoja, React je definisao dvije osnovne vrste komponenti:

- **Funkcionalne komponente (eng. *Functional Components*)** - Funkcionalne komponente sve do pojave React hook-ova su bile ograničene na prikaz i nisu mogle imati vlastito stanje. To ih je činilo jednostavnim, ali pogodnim samo za manje segmente aplikacije. Razvojem hook-ova, funkcionalne komponente su dobile sposobnost upravljanja stanjem i bočnim efektima, čime su postale standard u savremenoj React praksi [36]. Njihova prednost ogleda se i u sintaksnoj jednostavnosti, lakoći testiranja i manjoj potrošnji resursa.

- **Komponente kao klase (eng. *Class Components*)** – Komponente u obliku klasa su dizajnirane kao JavaScript klase koje nasljeđuju React.Component klasu. Kao takve, imaju podršku za napredne funkcionalnosti poput metoda životnog ciklusa i manipulisanja lokalnim stanjem. Ove komponente su bile osnovni način implementacije kompleksnog ponašanja sve do popularizacije funkcionalnih komponenti. Danas se njihov udio u novim projektima značajno smanjuje, ali ostaju važan segment za održavanje naslijeđenih (eng. *Legacy*) React sistema [37].

4.1.2. Hijerarhija i kompozicija

Ključna karakteristika komponentne arhitekture u React-u je hijerarhijska organizacija komponenti, koja omogućava jasnu segmentaciju aplikacije u logičke cjeline. Svaka aplikacija ima svoju korijensku (eng. *Root*) komponentu koja dalje referencira potkomponente. Ove komponente mogu sadržavati dodatne potkomponente u skladu s potrebama interfejsa. Takva struktura najčešće se opisuje kao stablo komponenti (eng. *Component tree*).

Uz hijerarhiju, React podstiče koncept kompozicije. To znači da veće i kompleksnije funkcionalnosti aplikacije nastaju kombinovanjem manjih, elementarnih komponenti. Ovakva praksa pruža razvojnom timu fleksibilnost da prvo implementira generičke ili visokospecijalizovane komponente. Nakon toga, komponente se mogu modularno organizovati i povezivati, poput slagalice, kako bi se kreirao željeni korisnički interfejs.

4.1.3. Kontrola toka podataka kroz stanja i props-ove

U okviru hijerarhije i kompozicije React komponenti, komunikacija između njih se odvija putem dva osnovna mehanizma:

1. **Props -** Predstavlja podatke ili karakteristike koje roditeljska komponenta eksplicitno prosljeđuje svojim ugnježđenim komponentama (eng. *Children*). Props-ovi služe kao mehanizam za konfiguraciju i prilagođavanje ponašanja ugnježđene komponente na osnovu informacija dobijenih od nadređene (roditeljske) komponente. Važno je napomenuti da su props-ovi nepromjenljivi iz perspektive komponente koja ih prima. To znači da ugnježđena komponenta ne smije direktno mijenjati vrijednosti props-a koje je dobila, već ih isključivo koristiti za prikaz, internu logiku ili eventualno prosljeđivanje dalje drugim potkomponentama. Ova osobina omogućava konzistentnost aplikacije tokom njenog rada i minimizuje potencijalne greške.

2. **State** - Odnosi se na unutrašnje stanje pojedinačne komponente. State omogućava komponenti da pamti informacije između renderovanja, da reaguje na korisničke interakcije i promjene, te da dinamički ažurira svoj prikaz. Za razliku od props-a, state se mijenja unutar same komponente. Sposobnost upravljanja lokalnim stanjem čini komponente samostalnim i adaptivnim jedinicama korisničkog interfejsa.

Prethodno opisani mehanizmi upravljanja stanjima su srž koncepta jednosmjernog toka podataka (eng. *One-way data flow*) u React-u. Podaci se uvijek prenose s roditeljske komponente prema djetetu putem props-a („data down“), što osigurava transparentne tokove informacija unutar aplikacije. Kada je potrebno da dijete komponenta izvrši promjene ka roditelju, to se obično ostvaruje kroz funkcije koje roditelj prosljeđuje kao props. Alternativno, u složenijim aplikacijama se koriste napredni mehanizmi centralizovanog upravljanja stanjem poput Redux-a ili Context API-ja. Oni omogućavaju efikasnu i skalabilnu kontrolu nad protokom podataka i reakcijama na promjene u aplikaciji.

4.1.4. Životni ciklus React komponente

Životni ciklus komponente, u kontekstu React biblioteke, odnosi se na skup jasno definisanih faza kroz koje svaka komponenta prolazi od momenta kada se kreira, preko ažuriranja, pa sve do njenog uklanjanja iz prikaza aplikacije (eng. *mounting, updating i unmounting*). Ove faze omogućavaju kontrolisanje ponašanja komponenti, optimizaciju performansi i sinhronizovanje logike aplikacije. Postoje tri osnovne faze životnog ciklusa React komponente, a to su kreiranje, ažuriranje i uklanjanje. Svaka od ovih faza podrazumijeva izvođenje određenih metoda koje imaju uticaj na način na koji komponenta komunicira sa okolinom, upravlja sadržajem i reaguje na promjene stanja.

Kreiranje (eng. *Mounting*)

Kreiranje je prvi korak u životu svake React komponente. Tokom kreiranja, React instancira komponentu, povezuje je sa odgovarajućim ulaznim podacima (props) i stanjem (state), te renderuje njen prikaz.

U komponentama u obliku klase, proces kreiranja podrazumijeva izvršavanje nekoliko specifičnih metoda životnog ciklusa:

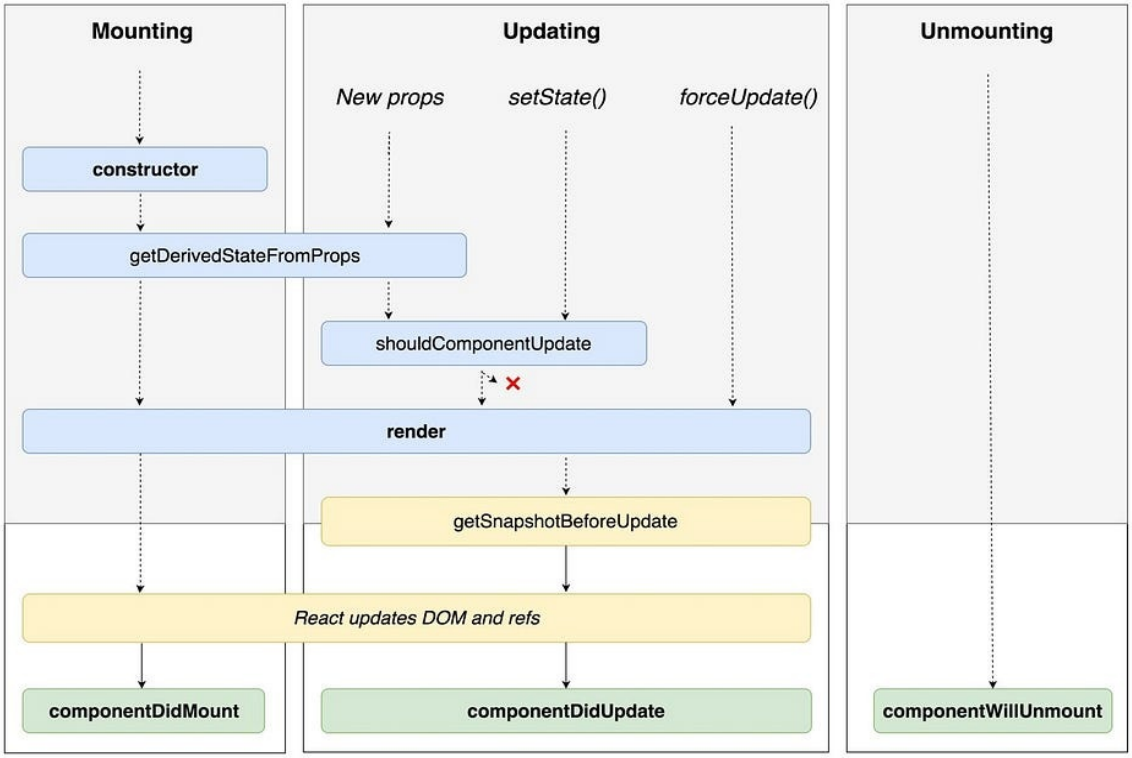
- *constructor(props)* - Konstruktor je prva metoda koja se poziva pri inicijalizaciji. Koristi se za postavljanje početnih vrijednosti stanja, vezivanje metoda za instancu i inicijalnu konfiguraciju komponente.

- *static getDerivedStateFromProps(props, state)* - Statička metoda koja omogućava sinhronizaciju stanja na osnovu promjena u props-ima, prije samog renderovanja.

- *render()* - Ova metoda je obavezna i njena svrha je da vraća elemente za prikaz. Render metoda ne smije imati bočne efekte, jer se poziva više puta tokom životnog ciklusa.

- *componentDidMount()* - Poziva se odmah nakon što je komponenta umetnuta u DOM. Pogodna je za inicijalizaciju eksternih resursa, AJAX poziva, manipulacije DOM-om ili podešavanje tajmera.

Pomenute metode se mogu uočiti u prvoj koloni dijagrama životnog ciklusa komponente kao klase, prikazanog na slici 12.



Slika 12: Dijagram životnog ciklusa komponente kao klase

U funkcionalnim komponentama, prilikom kreiranja se izvršavaju hook-ovi. Hook useState postavlja inicijalno stanje varijable. Hook useMemo se prvi put izvršava i vraća stanje referenci koja se nadalje koristi u komponenti. Hook useEffect se koristi za kontrolu različitih faza životnog ciklusa. Ako se ovim hook-ovima kao drugi argument proslijedi prazan niz („[]”), efekat će se izvršiti samo jednom, odmah nakon što se komponenta prvi put prikaže na ekranu.

Ažuriranje (eng. *Updating*)

Kao odgovor na promjene u props ili state vrijednostima, komponenta se ponovo renderuje. Ovaj proces se naziva ažuriranjem i ključan je za reaktivnost i dinamičan prikaz korisničkog interfejsa.

Komponente kao klase omogućavaju sljedeće metode koje su vezane za fazu ažuriranja:

- *static getDerivedStateFromProps(props, state)* - Metoda se poziva prilikom svake promjene props-a, omogućavajući sinhronizaciju stanja sa promijenjenim ulaznim vrijednostima.

- *shouldComponentUpdate(nextProps, nextState)* - Omogućeno je da se kroz povratnu vrijednost ove funkcije kontroliše da li će doći do ponovnog renderovanja komponente. Ova metoda služi za optimizaciju performansi sprečavanjem bespotrebnog renderovanja.

- *render()* - Funkcija render se izvršava svaki put kada dođe do promjene stanja ili ulaznih podataka i odgovorna je za prikaz.

- *getSnapshotBeforeUpdate(prevProps, prevState)* - Metoda koja omogućava čuvanje informacija o trenutno prikazanom sadržaju neposredno prije nadolazeće izmjene.

- *componentDidUpdate(prevProps, prevState, snapshot)* - Poziva se odmah nakon ažuriranja komponente i renderovanja u DOM. Ovdje se mogu vršiti operacije koje zavise od prethodnog i trenutnog stanja, ažurirati eksterne resurse ili sinhronizovati podatke sa serverom.

U funkcionalnim komponentama, svi efekti ažuriranja se realizuju korištenjem React hook-ova, gdje se kao niz zavisnosti navode vrijednosti na osnovu kojih će se hook-ovi ponovo izvršiti.

Uklanjanje (eng. *Unmounting*)

Posljednja faza životnog ciklusa komponente je uklanjanje iz DOM stabla. Do ovoga dolazi kada roditeljska komponenta odluči da više ne želi da prikazuje određenu komponentu.

Komponente u obliku klase imaju posebnu metodu *componentWillUnmount()*, koju izvršavaju neposredno prije samog uklanjanja komponente iz DOM-a. Tipične upotrebe uključuju čišćenje i oslobađanje resursa (npr. tajmera, WebSocket veza, gašenje osluškivača događaja i drugo) kako bi se spriječilo curenje memorije aplikacije.

Kod funkcionalnih komponenti, efekat čišćenja se postiže kroz povratnu funkciju unutar useEffect hook-a. Tako zadata funkcija se izvršava automatski čim dođe do uklanjanja ili promjene zavisnosti.

4.1.5. Principi dizajna i održavanja komponentne arhitekture

Komponentna arhitektura prirodno podstiče korištenje naprednih paradigmi dizajna, čime se unapređuje organizacija koda i olakšava održavanje. Ključne vrijednosti ove

arhitekture ogledaju se kroz nekoliko principa koji imaju direktan uticaj na efikasnost, čitljivost i ponovnu upotrebljivost koda, a to su:

- **Single Responsibility Principle (SRP) -** Princip jedinstvene odgovornosti

  Prema ovom principu, svaka komponenta treba da ima jasno definisanu funkciju i da bude odgovorna za jedan konkretan segment funkcionalnosti ili prikaza. Na taj način se pojednostavljuje razvoj i održavanje, jer izmjene u jednom dijelu aplikacije minimalno utiču na ostale segmente. U React-u to najčešće znači da jedna komponenta upravlja isključivo jednim dijelom korisničkog interfejsa ili logike (npr. prikaz tabele, unos forme, navigacija i drugo). Složenije funkcionalnosti nastaju kompozicijom više specijalizovanih komponenti. Ispravno implementiran SRP povećava modularnost sistema, smanjuje mogućnost grešaka i olakšava testiranje svake jedinice zasebno.

- **Separation of Concerns (SoC)** - Odvajanje odgovornosti

  Ovaj princip podrazumijeva razdvajanje raznih aspekata softvera u odvojene cjeline sa kojima se lako upravlja. React omogućava da „prezentacione" komponente budu fokusirane isključivo na izgled i prikaz, dok „pametne" komponente upravljaju logikom i stanjem. Ovako organizovan kod je intuitivniji, jednostavniji za navigaciju i minimizira međusobnu zavisnost različitih funkcionalnih segmenata. SoC je naročito važan u većim sistemima, gdje kombinovanje logičkih i prezentacionih slojeva može dovesti do nepreglednosti i poteškoća u održavanju.

- **Reusability** - Ponovna upotrebljivost

  Jedna od osnovnih prednosti komponentne arhitekture ogleda se u mogućnosti da se jednom napisana komponenta iskoristi u više različitih dijelova aplikacije, pa čak i u različitim projektima. Na primjer, generičke komponente kao što su dugmići, elementi forme ili liste, mogu se prilagoditi putem props-a bez potrebe za pisanjem novog koda. Time se direktno smanjuje vrijeme razvoja, povećava konzistentnost korisničkog interfejsa i olakšava dugoročna nadogradnja sistema. Osim toga, ovaj pristup omogućava formiranje biblioteka komponenti koje se mogu dijeliti unutar šireg razvojnog tima ili organizacije.

- **Testabilnost**

  Pažljivo razdvojene komponente znatno olakšavaju proces automatskog testiranja. S obzirom na to da su komponente u React-u projektovane da budu što izolovanije, lako ih je pojedinačno testirati kroz jedinične (eng. *unit*) testove, simulirajući različite ulazne vrijednosti i interakcije. Testabilnost omogućava bržu detekciju grešaka, stabilnost koda tokom refaktorisanja i sigurniju implementaciju novih funkcionalnosti.

- **Refaktorisanje i skalabilnost**

  Komponentna arhitektura pojednostavljuje proces prepravke i unapređenja postojećeg koda bez izmjene spoljašnjeg ponašanja. Dodavanje novih funkcionalnosti, ili proširenje postojećih, obavlja se dodavanjem novih ili ažuriranjem postojećih komponenti, bez narušavanja stabilnosti čitavog sistema.

Primjena navedenih principa dizajna donosi brojne pogodnosti: brže vrijeme razvoja, niže troškove održavanja, jednostavnije ispravljanje i prevenciju grešaka, kao i mogućnost daljeg širenja sistema bez eksponencijalnog povećanja kompleksnosti. Ovi principi su danas standard i osnova profesionalnog razvoja modernih veb aplikacija.

4.2. Virtualni DOM

Jedan od ključnih faktora koji je doprinio popularnosti i uspjehu React biblioteke je koncept virtualnog Document Object Modela (Virtual DOM, skraćeno VDOM). Ovaj koncept predstavlja fundamentalnu inovaciju u pristupu efikasnom manipulisanju korisničkim interfejsom unutar modernih veb aplikacija [36]. Razumijevanje principa, mehanizama i prednosti VDOM-a je od suštinske važnosti za razumijevanje performansi React aplikacija.

Ranije, manipulacija sadržajem veb stranice se realizovala kroz direktno upravljanje DOM-om. DOM je hijerarhijska struktura elemenata koja predstavlja cjelokupan prikaz stranice u veb pregledaču. Međutim, DOM je po svojoj prirodi kompleksan i relativno spor, naročito kod aplikacija sa mnogo interakcija i dinamičkih promjena. Svaka promjena u DOM-u, bilo da se radi o dodavanju, uklanjanju ili izmjeni elemenata, zahtijeva angažovanje pregledača u procesu ponovnog renderovanja i ponovnog izračunavanja stilova i rasporeda elemenata. Kod složenijih aplikacija, direktne i česte manipulacije utiču na performanse i rezultuju lošim korisničkim iskustvom [38].

Ove okolnosti su stvorile potrebu za apstraktnim slojem između aplikacije i stvarnog DOM-a. Osnovna funkcija takvog sloja je bila da omogući minimalan broj promjena na strani korisničkog interfejsa. Virtualni DOM može se definisati kao lagana, memorijski efikasna replika stvarnog DOM stabla, kreirana i održavana u memoriji [36]. U React-u, VDOM je objekat koji sadrži virtualnu reprezentaciju stvarne strukture komponenata i elemenata interfejsa. Svaki put kada se promijeni stanje aplikacije ili stignu novi podaci, React prvo ažurira VDOM, umjesto da odmah modifikuje stvarni DOM.

Ovakva strategija omogućava React-u da prikupi sve promjene unutar memorijske strukture, uporedi staru i novu verziju VDOM-a, analizira razlike između njih i optimizuje način na koji će promjene biti prenesene na stvarni DOM. Ključna prednost ovakve metode je mogućnost da se na stvarnom DOM-u izvrše samo one promjene koje su zaista neophodne, dok se redundantna ili suvišna ažuriranja izbjegavaju.

4.2.1. Usaglašavanje

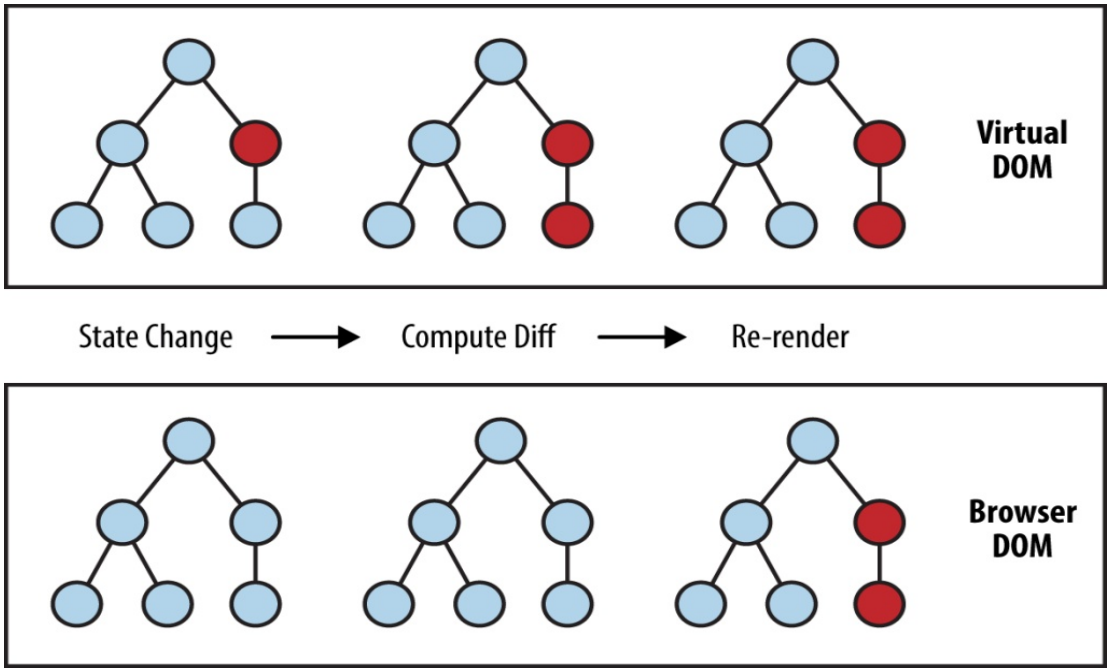
Usaglašavanje (eng. *Reconciliation*) predstavlja fundamentalni proces u React-u koji omogućava efikasno sinhronizovanje virtualnog prikaza aplikacije (VDOM) sa stvarnim prikazom u internet pregledaču. Ovaj proces dolazi do izražaja svaki put kada dođe do promjene stanja ili svojstava neke komponente, odnosno kada korisnička interakcija ili vanjski podaci zahtijevaju ažuriranje korisničkog interfejsa.

Diffing algoritam

Srž reconciliation procesa u React-u čini tzv. *diffing* algoritam. To je sofisticirani sistem za upoređivanje starog i novog VDOM stabla. React nije prvi softver koji koristi diffing tehnike, ali je među prvima značajno optimizovao ove procese za potrebe veb

interfejsa. Ključna pretpostavka je da se većina DOM stabla između dva renderovanja zapravo ne mijenja, već da su promjene lokalizovane na manje segmente [34].

Diffing algoritam operiše rekurzivnim prolaskom kroz odgovarajuće čvorove starog i novog VDOM-a, vršeći poređenje po tipu (npr. protiv) i po hijerarhijskoj poziciji unutar stabla. Ako se tip čvora promijeni, React će taj čvor smatrati potpuno novim, ukloniti postojeći DOM element i kreirati novu instancu. Ako tip ostane isti, React daljim poređenjem dječijih čvorova lokalizuje promjene na najniži mogući nivo i time minimizira broj stvarnih DOM operacija.



Slika 13: Ponašanje VDOM-a i DOM-a nakon promjene stanja komponente [39]

Na slici 13 je ilustrovana promjena stanja u jednom čvoru stabla. Lako se može uočiti da stvarni DOM ne prati međufaze VDOM-a, nego da se tek po završetku diffing algoritma promjene primjenjuju na stvarni DOM.

Optimizacija uz pomoć ključeva

Vrlo važan aspekt reconciliation procesa predstavlja upotreba jedinstvenih oznaka, tzv. ključeva (eng. *Keys*), prilikom renderovanja kolekcija elemenata. Ključ omogućava React-u da brzo i pouzdano mapira elemente iz prethodne i aktuelne verzije liste, te tako izbjegne nepotrebno uklanjanje i ponovno kreiranje čitavih sekcija DOM-a pri svakom ažuriranju. React koristi vrijednosti ključeva kao identifikatore i precizno određuje koje čvorove treba zadržati, premjestiti, izmijeniti ili ukloniti. Na ovaj način se drastično poboljšavaju performanse i stabilnost čitavog interfejsa, dok korisničko iskustvo ostaje dosljedno čak i kod čestih i kompleksnih promjena podataka. React uvodi dva osnovna pravila kako bi reconciliation bio optimalan:

- Komponente istog tipa i istih ključeva zadržavaju svoje stanje između renderovanja. To znači da, ukoliko se struktura komponenata i njihovi ključevi ne promijene, React neće rekonstruisati ili resetovati stanje komponente, već će je ažurirati sa minimalnim brojem operacija.

- Različiti tipovi čvorova ili promjena ključeva dovode do potpunog uništenja stare i kreiranja nove komponente. Ovo je važno razumjeti prilikom dizajna komponenata za očuvanje globalnog i lokalnog stanja aplikacije.

Uz ova pravila, React prepoznaje dva tipa komponenti: komponente koje sadrže vlastito stanje (eng. *Stateful*) i one koje su samo funkcije bez unutrašnjeg stanja (eng. *Stateless*). Pri svakom ažuriranju virtualnog stabla, React vodi računa o tome da li treba sačuvati, ažurirati ili izbaciti lokalno stanje u odgovarajućoj komponenti.

4.2.2. Prednosti i ograničenja pristupa zasnovanog na VDOM-u

Primjena VDOM-a donosi brojne prednosti u odnosu na dotadašnje pristupe:

- Optimizovane performanse - S obzirom na to da se analiziraju i primenjuju samo potrebne promjene na DOM-u, drastično se smanjuje broj skupih operacija.

- Deklarativni model razvoja - Programeri zadaju željeno stanje interfejsa, dok React, kroz VDOM i diffing algoritam, automatski obezbjeđuje efikasno ažuriranje prikaza.

- Dosljednost i predvidivost - Svaka promjena interfejsa prolazi kroz isti, deterministički proces iz VDOM-a do stvarnog DOM-a.

- Nezavisnost od internet pregledača - Koristeći principe rada VDOM-a, pisani React kod je nezavisan od specifičnosti različitih veb pregledača.

- Podrška za server-side rendering i testiranje - Zbog činjenice da je VDOM u potpunosti objekat u memoriji, lako je simulirati prikaz i testirati aplikaciju i izvan samog veb okruženja.

Iako VDOM donosi velike prednosti, njegova upotreba nije bez ograničenja. Kod velikih ili izuzetno kompleksnih aplikacija, sama izgradnja i poređenje VDOM stabla može predstavljati problem, posebno ako komponente nisu optimalno definisane ili su kompleksne za ponovnu kalkulaciju. U takvim situacijama napredne optimizacije (memoizacija komponenti, kontrola renderovanja, segmentacija stabla i slično) postaju neophodne za očuvanje performansi.

4.3. JSX

JSX, skraćenica za JavaScript XML, predstavlja jednu od inovacija koje su omogućile širenje React biblioteke u industriji. Osmišljen je kao sintaksna ekstenzija za JavaScript. Njome je omogućeno da se u logici aplikacije koristi deklarativni stil veoma sličan HTML-u, uz kombinovanje sa izrazima JavaScript jezika. Ova karakteristika ima veliki uticaj na način spajanja prikaza i logike. Takav stil doprinosi koherentnosti, čitljivosti i jednostavnijem održavanju koda [40].

Umjesto striktne podjele između HTML šablona i JavaScript logike, React putem JSX-a omogućava da sve što pripada jednoj komponenti ostaje objedinjeno u istoj fizičkoj datoteci i u okviru istih funkcija ili klasa. Drugim riječima, UI komponente u React-u se definišu kao "funkcije stanja", gdje korisnički interfejs zavisi od trenutnog stanja. JSX izrazi omogućavaju i interaktivnost, jer se svi JavaScript izrazi mogu direktno interpolirati unutar XML sintakse pomoću vitičastih zagrada („{}”).

Za razliku od HTML-a, JSX ima određena pravila:

- Svi elementi moraju imati jednog roditelja.
- Samostalni elementi moraju biti samostalno zatvoreni (poput img ili input tagova).
- Atributi se prilagođavaju, kao što se class i for mijenjaju sa className i htmlFor.
- Korištenje camelCase notacije za većinu atributa (npr. onClick, tabIndex).

JSX nije nativno podržan u veb pretraživačima. Umjesto toga, JSX koristi alat kao što je Babel koji vrši prevođenje JSX izraza u standardne JavaScript funkcijske pozive. Proces garantuje ispravno izvršavanje koda na različitim pretraživačima. Ovakav princip omogućava da se čitava struktura korisničkog interfejsa gradi deklarativno, dok React u pozadini generiše odgovarajuće virtualne čvorove koji se upoređuju i prenose stvarnom DOM-u.

4.4. React Hooks

React hook-ovi predstavljaju fundamentalni koncept u React-u, koji je značajno promijenio paradigmu razvoja funkcionalnih komponenti i način upravljanja stanjem i bočnim efektima unutar React aplikacija. Prije pojave hook-ova, klasne komponente su mogle da upravljaju lokalnim stanjem, prate životni ciklus i rukuju resursima, dok su funkcionalne komponente služile isključivo za jednostavne i prezentacione zadatke. Uvođenjem hook-ova, data je mogućnost funkcionalnim komponentama da postanu punopravni članovi React ekosistema, čime se postiže sintaksna jednostavnost, čitljivost i viši stepen modularnosti.

Osnovna filozofija hook-ova ogleda se u načelu ponovne upotrebe logike između komponenti, bez potrebe za kompleksnim hijerarhijama i obrascima kao što su komponente višeg reda (eng. *Higher-Order Components* - *HOC*) ili render props. Hook-ovi omogućavaju da se upravlja stanjem, reaguje na promjene i bočne efekte, pristupi kontekstu ili referencama, a da pri tome ne dođe do narušavanja organizacije koda ili dodatne složenosti.

React nudi skup hook-ova koji pokrivaju osnovne potrebe razvoja, ali se mogu definisati i vlastite hook funkcije za specifične zadatke. Osnovni i opšte poznati hook-ovi su:

- *useState* - Omogućava definisanje i upravljanje stanjem unutar funkcionalne komponente. Hook vraća aktuelnu vrijednost stanja i funkciju za ažuriranje.
- *useEffect* - Predstavlja ekvivalent metodama životnog ciklusa u komponentama u onliku klase. Hook omogućava reakciju na promjene podataka, izvršavanje asinhronih operacija i čišćenje resursa. Niz zavisnosti vrši kontrolu nad tim kada i koliko često efekat treba biti izvršen.
- *useContext* - Hook koji omogućava pristup globalno definisanom kontekstu unutar hijerarhije komponenti, bez potrebe za dubokim prosljeđivanjem props-ova. Takođe, pruža efikasan način za dijeljenje globalnih podataka između udaljenih komponenti.

- *useReducer* - Pruža naprednu kontrolu nad složenim stanjima koristeći princip sličan Redux šablonu, što je posebno korisno kod upravljanja objektima ili višestrukim povezanim vrijednostima stanja.

- *useRef* - Omogućava pristup i manipulaciju referencama na DOM elemente, ali i čuvanje promjenljivih vrijednosti koje ne pokreću ponovno iscrtavanje, što je veoma korisno za specifične slučajeve upotrebe.

Pored ugrađenih, moguće je razvijati i pozivati vlastite hook funkcije (eng. *Custom hooks*) kako bi enkapsulirali i dijelili logiku između različitih komponenti. *Custom hook* je funkcija sa prefiksom „*use*”. Obično poziva druge hook-ove unutar sebe i vraća podatke ili funkcije za korištenje u drugim komponentama.

Glavna prednost koncepta hook-ova je podizanje funkcionalnih komponenti na nivo gdje mogu u potpunosti zamijeniti klasne komponente. Međutim, uprkos brojnim prednostima, upotreba hook-ova dolazi sa određenim pravilima i ograničenjima. Hook-ovi se uvijek moraju pozivati na vrhu funkcionalne komponente ili u drugim hook-ovima, nikada unutar petlji, uslova ili ugnježđenih funkcija. Redoslijed hook poziva mora biti isti tokom svakog renderovanja. Nepoštovanje ovih pravila dovodi do nepredviđenih ponašanja aplikacije i komplikovanog otkrivanja grešaka.

5. PRIJEDLOG I OPIS RJEŠENJA ZA GENERISANJE APLIKACIJA

U savremenom razvoju softverskih sistema, jedan od najznačajnijih izazova jeste postizanje visoke produktivnosti i efikasnosti prilikom izrade aplikacija, a da se istovremeno očuvaju kvalitet, sigurnost i dosljednost. Većina poslovnih aplikacija, bez obzira na domensku oblast, dijeli određene tipične obrasce u pogledu arhitekture i načina manipulacije podacima. Često je to riječ o višeslojnom sistemu sa jasno definisanim modelima podataka, standardizovanim CRUD operacijama (kreiranje, čitanje, ažuriranje i brisanje), kao i kontrolom pristupa istim. Upravo zbog ovih ponavljajućih obrazaca, značajan dio programerskog rada svodi se na repetitivno implementiranje skoro identičnih segmenata koda.

Posljednjih godina, s porastom primjene agilnih metodologija razvoja i sve većom potrebom za brzim prototipima, značajna pažnja se posvećuje automatizaciji procesa razvoja softvera. U tom kontekstu uvode se alati koji omogućavaju generisanje dijelova softverskih rješenja uz minimalan ljudski angažman. Automatsko generisanje aplikacija ima višestruku vrijednost. Prije svega, smanjuje vrijeme potrebno za razvoj aplikacija, minimizuje rizik od nekonzistentnosti i olakšava kasnije održavanje sistema. Eliminacijom manuelnog pisanja koda, smanjuje se prostor za tipične greške, poboljšava se testabilnost i otklanjaju sigurnosne slabosti. Pored ovih neposrednih prednosti, automatizovano generisanje aplikacija otvara prostor za olakšano uvođenje novih tehnologija u razvojne timove.

Ovakvi trendovi u velikoj mjeri korespondiraju sa savremenim paradigmama razvoja softvera, posebno sa model-vođenim razvojem softvera (eng. *Model-Driven Development - MDD*) i niskokodnim razvojnim platformama (eng. *Low-Code Development Platforms - LCDP*). MDD podrazumijeva sistematsku upotrebu apstraktnih modela kao primarnih artefakata razvoja, koji se zatim transformišu u izvršni kod, čime se povećava nivo apstrakcije i smanjuje zavisnost od specifičnih programskih jezika. LCDP, s druge strane, omogućavaju razvoj aplikacija pomoću vizuelnih modela i minimalnog programiranja, čime se skraćuje vrijeme isporuke i otvara prostor za širu participaciju korisnika u procesu razvoja. Prema novijim istraživanjima, obje paradigme predstavljaju ključne pravce u savremenoj softverskoj industriji, jer kombinuju brzu isporuku, smanjenje troškova i unapređenu održivost sistema [41].

5.1. Analiza postojećih rješenja

Analiza postojećih alata za automatsko generisanje aplikacija ukazuje na to da je automatizacija razvoja prepoznata kao bitan faktor unapređenja produktivnosti. Međutim, trenutna rješenja još uvijek posjeduju određena ograničenja koja je potrebno prevazići kako bi ovakav pristup postao univerzalno prihvaćen i efikasno primjenjiv u najširem spektru upotrebe. U okviru ove analize fokus je bio na generatorima koji omogućavaju automatsko generisanje Spring i React aplikacija.

5.1.1. Spring Roo

Spring Roo predstavlja alat za generisanje aplikacija unutar Spring ekosistema, posebno namijenjen razvoju Java aplikacija sa podrškom za osnovne CRUD operacije. Njegova ključna prednost leži u automatizaciji procesa kreiranja entiteta, repozitorijuma,

servisa i kontrolera, uz integraciju sa alatima poput Spring Boot-a, Maven-a i Hibernate-a. Spring Roo omogućava programerima da definišu konfiguraciju aplikacije, kreiranje entiteta, polja i relacija putem komandne linije (CLI) [42].

Specifičnost Spring Roo-a ogleda se u upotrebi AspectJ fajlova (.aj) i anotacija, kao što su *@RooEntity* i *@RooService*. Ove anotacije i aspekti razdvajaju automatski generisani kod od koda koji programer ručno implementira, čime se obezbjeđuje da buduća ažuriranja generisanog koda ne utiču na poslovnu logiku aplikacije. Arhitektura generisanih aplikacija uključuje sljedeće komponente:

- Entiteti - Java klase koje predstavljaju modele podataka sa definisanim poljima i relacijama;
- Repozitorijumi - interfejsi za pristup podacima;
- Servisi - poslovna logika aplikacije;
- Kontroleri - upravljanje HTTP zahtjevima i interakcijom sa klijentskim slojem;
- AspectJ fajlovi - sadrže generisani kod za AOP funkcionalnosti i omogućavaju Spring Roo-u da prati i ažurira entitete i servise bez konflikta sa ručno napisanim kodom;
- Konfiguracioni fajlovi - uključuju application.properties i druge konfiguracije potrebne za pokretanje aplikacije;
- Klijentska aplikacija - jednostavna JSP aplikacija ukoliko su uključene opcije za korisnički interfejs.

Arhitektura generisanih aplikacija prati slojevitu strukturu, što olakšava održavanje, testiranje i buduće proširenje sistema. Prednost Spring Roo-a je u tome što kroz automatizaciju i upotrebu AOP-a omogućava razdvajanje generisanog i ručno pisanog koda, čime se smanjuje rizik od grešaka i povećava efikasnost razvoja.

Međutim, upotreba Spring Roo-a zahtjeva dobro poznavanje komandnog okruženja i specifičnosti Spring razvojnog okvira. Ova karakteristika može predstavljati izazov za manje iskusne programere ili korisnike koji nisu dovoljno upoznati sa Java ekosistemom. Efikasno korišćenje alata zahtjeva razumijevanje kako generisani kod funkcioniše i kako se integriše u postojeću arhitekturu aplikacije.

5.1.2. OpenAPI Generator

OpenAPI Generator predstavlja alat za automatsko generisanje koda i dokumentacije na osnovu OpenAPI specifikacije. OpenAPI specifikacija detaljno opisuje API operacije, uključujući REST pristupne tačke, HTTP metode, parametre, odgovore, šeme podataka, validacije, kao i dodatne informacije poput autentifikacije i autorizacije (npr. OAuth, JWT). Specifikacija može biti u JSON ili YAML formatu [43].

OpenAPI Generator koristi unaprijed definisane šablone (poput Mustache-a), kako bi korisnicima omogućio generisanje serverskog i klijentskog koda sa dokumentacijom, konfiguracionim fajlovima i testovima. Podržava širok spektar programskih jezika i

razvojnih okvira, uključujući Java, C#, Python, PHP, JavaScript, TypeScript, Swift, Kotlin, kao i baze podataka kao što je MySQL. Korištenje alata može se realizovati putem komandne linije, grafičkog interfejsa ili konfiguracionog fajla, što omogućava fleksibilnost u odabiru jezika, šablona i načina generisanja.

Generisani rezultati uključuju:

- Serverski kod - implementacija API-ja prilagođena odabranom jeziku i razvojnom okviru (npr. Spring Boot za Java, Express.js za Node.js, Flask ili FastAPI za Python);
- Klijentski kod - implementacija komunikacije klijentske strane sa API-jem u jezicima kao što su JavaScript, TypeScript, Swift ili Kotlin;
- Dokumentacija - automatski generisani prikazi u HTML, Markdown ili Confluence formatu;
- Konfiguracioni fajlovi - uključuju pripremljene konfiguracije za Docker, testiranje ili CI/CD integracije;
- Testovi - skripte za automatsku validaciju API funkcionalnosti.

Prednost OpenAPI Generatora ogleda se u brzini i preciznosti generisanja koda, smanjenju ručnog rada i mogućnosti održavanja konzistentnog API-ja kroz sve razvojne faze. Ipak, rad sa ovim alatom zahtjeva napredno razumijevanje dizajna API-ja i OpenAPI standarda, što može predstavljati izazov za manje iskusne programere ili timove.

5.1.3. CodeSmith

CodeSmith generator predstavlja alat za automatizovano generisanje koda koji omogućava visok nivo prilagodljivosti kroz korištenje unaprijed definisanih šablona kao ulaznih podataka. Alat omogućava kreiranje različitih softverskih artefakata, uključujući baze podataka, poslovnu logiku, korisnički interfejs, konfiguracione fajlove i dokumentaciju [44].

Proces generisanja u CodeSmith-u započinje definisanjem šablona, koji određuju strukturu i format koda koji će biti generisan. Šabloni se mogu kreirati korištenjem različitih jezika i tehnologija, uključujući T4, Razor ili obične tekstualne fajlove. Nakon definisanja šablona, korisnik unosi potrebne informacije, kao što su podaci o bazama podataka, modeli podataka ili konfiguracije sistema. Ovi ulazi mogu dolaziti iz različitih izvora, uključujući DDL skripte, JSON fajlove, konfiguracione fajlove ili direktni unos putem grafičkog korisničkog interfejsa (GUI).

Na osnovu kombinacije šablona i korisničkih ulaznih podataka, CodeSmith Generator kreira izlazne fajlove koji mogu uključivati:

- SQL skripte - za kreiranje i modifikaciju baza podataka;
- Serverski kod - modele, kontrolere, servise i repozitorijume prilagođene odabranoj arhitekturi;
- Klijentski kod - HTML, CSS, JavaScript fajlove ili komponente određenog razvojnog okvira;

- Konfiguracione fajlove - uključujući pripremu za Docker, Webpack ili druge alate za integraciju;
- Dokumentaciju - API dokumentaciju i druge prateće materijale.

Prednost CodeSmith Generator-a ogleda se u visokom stepenu fleksibilnosti i mogućnosti prilagođavanja različitim projektima i arhitekturama. Istovremeno, izmjene i prilagođavanja šablona zahtjevaju dodatni napor, što može usporiti razvoj kada se korisnički zahtjevi značajno razlikuju od tipične strukture predviđene šablonom.

5.1.4. Yeamon generatori

Yeoman je platforma za automatizaciju generisanja softverskih projekata na osnovu predefinisanih šablona, poznatih kao Yeoman generatori. Ovi generatori omogućavaju kreiranje kompletne strukture projekta, koja obuhvata konfiguracione fajlove, osnovni kod, skripte i druge neophodne komponente. Generatori su implementirani kao Node.js moduli koji sadrže šablone za određene tipove projekata ili komponenti, kao i logiku za prilagođavanje generisanih fajlova prema unosima korisnika [45].

Interakcija sa Yeoman platformom realizuje se putem komandne linije, gdje korisnik sam pokreće željeni generator. Tokom ovog procesa, generator koristi biblioteke poput Inquirer.js za interaktivno postavljanje pitanja korisniku. Korisnik unosi informacije o projektu, kao što su naziv, tehnologije i preferencije u vezi sa konfiguracijom koda. Na osnovu ovih ulaznih podataka, generator bira odgovarajuće šablone i prilagođava ih tako da generisani projekat odgovara specifičnim zahtjevima korisnika.

Generisana aplikacija obuhvata:

- strukturu direktorijuma,
- osnovne skripte za pokretanje i testiranje,
- konfiguracione fajlove za razvojno okruženje,
- osnovne implementacije serverskih i klijentskih komponenti (ukoliko generator podržava više slojeva aplikacije),
- pripremljene šablone za dalje proširenje sistema.

Ovaj pristup omogućava ubrzanje početnog razvoja, standardizaciju strukture projekata i smanjenje rizika od grešaka pri inicijalnom kreiranju aplikacije. Njegova prednost se ogleda se u fleksibilnosti i modularnosti, jer korisnici mogu birati generator koji najbolje odgovara tipu projekta ili kombinovati više generatora kako bi kreirali kompleksne aplikacije. S druge strane, efektivno korištenje alata zahtjeva dobro poznavanje rada u CLI okruženju, kao i razumijevanje strukture šablona i logike generisanja.

5.1.5. JHipster

JHipster je alat otvorenog koda i omogućava brzo kreiranje modernih veb aplikacija i mikroservisnih sistema. Jedna generisana aplikacija se sastoje od serverskog i klijentskog dijela. Serverski dio je implementiran kao Spring Boot aplikacija, sa podrškom za Javu ili Kotlin, dok se klijentski dio generiše kao SPA korištenjem Angular, React ili Vue razvojnog

okvira. Pored veb aplikacija, JHipster omogućava generisanje i mobilnih klijenata kroz tehnologije kao što su Ionic i React Native [46].

Proces generisanja aplikacije započinje interaktivnim pitanjima na komandnoj liniji, koja omogućavaju konfiguraciju tipa aplikacije (monolitna ili mikroservisna), naziv aplikacije, tip autentifikacije (JWT, OAuth 2.0, sesija), bazu podataka, klijentski razvojni okvir, alat za pokretanje (Maven ili Gradle) i dodatne opcije poput internacionalizacije, Docker konfiguracije, CI/CD skripti i test okruženja. Ovaj pristup omogućava korisnicima da precizno definišu arhitekturu i funkcionalnosti generisane aplikacije.

Pored konfiguracije putem komande linije, JHipster omogućava i generisanje koda na osnovu modela definisanih u specijalizovanom JDL jeziku. JDL fajlovi definišu entitete, njihove atribute, odnose između entiteta, kao i druge aspekte aplikacije. Za vizuelno kreiranje i uređivanje modela dostupan je alat JDL Studio, koji olakšava definisanje i pregled strukture entiteta.

Nakon inicijalne konfiguracije ili odabira JDL modela, generator kreira kompletan projekat sa svim neophodnim fajlovima i slojevima. Serverska strana sadrži Spring Boot aplikaciju sa podrazumijevanom konfiguracijom i implementacijom poslovne logike kroz REST kontrolere, servise, entitete i repozitorijume. Klijentska strana obuhvata generisanu SPA aplikaciju sa odgovarajućim komponentama, servisima, stilovima i pripremljenim modulima za interakciju sa serverskom stranom.

JHipster ističu modularnost i fleksibilnost, jer je omogućena detaljna konfiguracija aplikacije, generisanje složenih modela i integracija različitih tehnologija u jedinstvenu arhitekturu. Istovremeno, efikasno korištenje alata zahtjeva upoznavanje sa CLI okruženjem i razumijevanje JDL jezika.

5.1.6. Veliki jezički modeli

Pored specijalizovanih generatora koda, na tržištu su sve prisutnija rješenja zasnovana na velikim jezičkim modelima (eng. *Large Language Models - LLM*), kao što su ChatGPT [47], GitHub Copilot [48] i slični alati. Ovi modeli koriste napredne tehnike obrade prirodnog jezika i omogućavaju generisanje programskog koda u gotovo svim poznatim programskim jezicima. Njihova primjena značajno ubrzava proces razvoja i doprinosi automatizaciji zadataka koji se kreću od pisanja jednostavnih skripti, preko kreiranja složenih funkcionalnosti, do izrade dokumentacije i testova.

Međutim, upotreba velikih jezičkih modela nosi određene rizike i ograničenja. Automatski generisan kod ne prolazi uvijek kroz standardizovane procese verifikacije i validacije, pa nije zagarantovano da će biti ispravan, siguran ili optimalan za zadatu primjenu. Dodatni izazov jeste činjenica da je za generisanje koda često potrebno dijeliti domensko znanje i interne logike sistema sa jezičkim modelima, čime se osjetljivi podaci i vlastite informacije mogu izložiti eksternim servisima. Postoji šansa da povjerljive informacije, koje model koristi za učenje, budu kasnije otkrivene drugim korisnicima ili upotrijebljene u neželjenim kontekstima, što je veliki rizik za zaštitu podataka i intelektualne svojine.

Uprkos ovim izazovima, alati zasnovani na velikim jezičkim modelima predstavljaju važan pravac razvoja u oblasti automatizacije programiranja. Njihova efikasnost i svestranost otvaraju mogućnost značajnog unaprijeđenja produktivnosti u softverskom inženjerstvu, ali

njihova primjena mora biti pažljivo regulisana, uz obavezno uvođenje mehanizama provjere kvaliteta i zaštite podataka.

5.2. Prijedlog rješenja

Nakon sprovedene analize postojećih generatora aplikacija, kao i detaljne identifikacije njihovih prednosti i ograničenja, definisani su zahtjevi i smjernice za razvoj novog sistema za automatsko generisanje softverskih rješenja. Osnovni cilj ovih smjernica jeste obezbjeđivanje pristupačnosti, intuitivnosti i fleksibilnosti u procesu generisanja aplikacija. Poseban akcenat stavljen je na unapređenje korisničkog iskustva i lakoći korištenja, s ciljem da sistem bude upotrebljiv širokom spektru korisnika, nezavisno od njihovog tehničkog predznanja.

Prije svega, zamišljeno je da početak razvoja ne zahtijeva opširno poznavanje specifičnih programskih jezika, kompleksnih komandnih linija ili korištenje naprednih razvojnih alata. Fokus je na podršci za standardizovane i široko prihvaćene formate kao što su SQL DDL skripte, ili jasno definisane JSON modele. Na ovaj način, značajno se snižava ulazna barijera za upotrebu sistema, što omogućava korištenje alata ne samo od strane programera različitih nivoa stručnosti, već i od analitičara, IT administratora i ostalih stručnjaka izvan uže oblasti softverskog razvoja.

Druga bitna stavka podrazumijeva razvoj interaktivnog korisničkog interfejsa umjesto rada preko komandne linije. To može biti korisnički interfejs koje vodi korisnika korak po korak kroz čitav proces: od unosa šeme i inicijalnih podataka, preko detaljne konfiguracije, pa sve do preuzimanja kompletno generisanih aplikacija. Time se povećava pristupačnost, ali i smanjuje vjerovatnoća greške u procesu konfiguracije.

Treća smjernica se odnosi na samu generisanu aplikaciju, a to je podrška višeslojnoj arhitekturi i savremenim tehnološkim standardima. Predloženo rješenje treba omogućiti generisanje korespodentnog serverskog i klijentskog dijela, uz mogućnost zasebnog funkcionisanja. Na taj način se korisnicima pruža fleksibilnost da izaberu koje slojeve žele generisati. Oba dijela sistema moraju odgovarati zadatoj šemi baze podataka, pri čemu su u generisanim aplikacijama implementirane standardne CRUD funkcionalnosti za sve entitete iz te šeme.

Uz CRUD funkcionalnosti, potrebno je da generisane aplikacije implementiraju osnovne sigurnosne i administrativne mehanizme. To podrazumijeva automatsko uključivanje autentifikacionih i autorizacionih procedura, bez dodatnog ručnog programiranja. Ova integracija garantuje dosljedno sprovođenje osnovnih sigurnosnih politika još pri samom generisanju softverskog rješenja, čime se eliminišu česte greške i sigurnosni propusti.

Na kraju, dizajn treba da objedini najbolje prakse i tehnološke prednosti postojećih sistema, ali pri tome izbjegavajući kompleksnost. Fokus treba biti na maksimalnoj automatizaciji, ali istovremeno i na mogućnosti jednostavnog prilagođavanja svakog segmenta sistema konkretnim poslovnim potrebama i preferencijama korisnika. Sve opcije konfiguracije, proširenja i nadogradnje sistema koncipirane su tako da generisani sistem dugoročno ostane upotrebljiv, skalabilan i spreman za brzu implementaciju novih zahtjeva.

Implementacijom ovih smjernica, predloženi sistem ima za cilj da postane pouzdan alat za ubrzan razvoj softverskih rješenja sa osnovnim funkcionalnostima. Takođe, uvijek može

poslužiti kao dobar temelj koji se lako nadograđuje domenskom logikom i funkcionalnim zahtjevima.

5.3. Opis implementiranog rješenja

U nastavku rada, biće predstavljen praktični dio istraživanja sa fokusom na tehničku realizaciju predloženog sistema za automatsko generisanje aplikacija na osnovu šeme baze podataka. Detaljno će biti opisani ključni koraci implementacije, korištene tehnologije i struktura sistema, kao i sam proces transformacije ulaznih podataka u funkcionalnu aplikaciju.

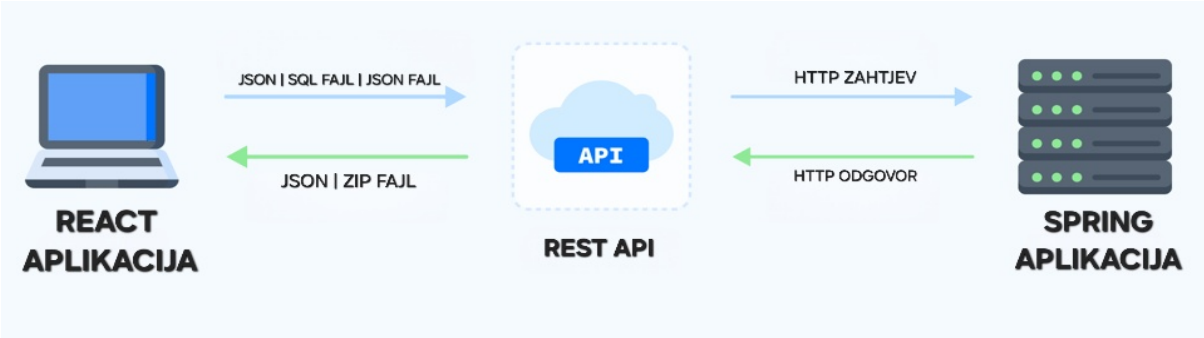
5.3.1. Opšti pregled arhitekture i organizacije sistema

Razvijeni sistem za automatsko generisanje aplikacija zasnovan je na višeslojnoj arhitekturi, koja se odlikuje potpunim razdvajanjem odgovornosti i modularnim pristupom. Osnovni cilj ovakve arhitekture je da omogući korisnicima intuitivan, efikasan i siguran proces od samog početka, pa sve do preuzimanja funkcionalnog izvornog koda.

Arhitekturu sistema čine:

1. Serverska aplikacija – Implementirana upotrebom Spring Boot okvira i zadužena je za parsiranje i validaciju ulaznih podataka, transformaciju SQL šeme u univerzalni model, kao i samo generisanje koda. Posebna pažnja posvećena je bezbjednosti. Server ne čuva trajno nikakva stanja niti izvorni kod aplikacije, već sve informacije obrađuje i transportuje isključivo putem REST API zahtjeva. Generisana aplikacija se pakuje u ZIP arhivu i vraća korisniku kao HTTP odgovor. Na ovaj način, samo korisnik ima pristup generisanom kodu, čime se povećava privatnost i sigurnost podataka.

2. Korisnički interfejs – Aplikacija koja je razvijena u React-u. Pruža pregledan i interaktivan vizuelni interfejs, koji korisniku omogućava unos šeme baze podataka, podešavanje željenih parametara aplikacije i preuzimanje generisanih rezultata. Sve konfiguracije se realizuju kroz grafički interfejs, čime je pojednostavljena upotreba alata.



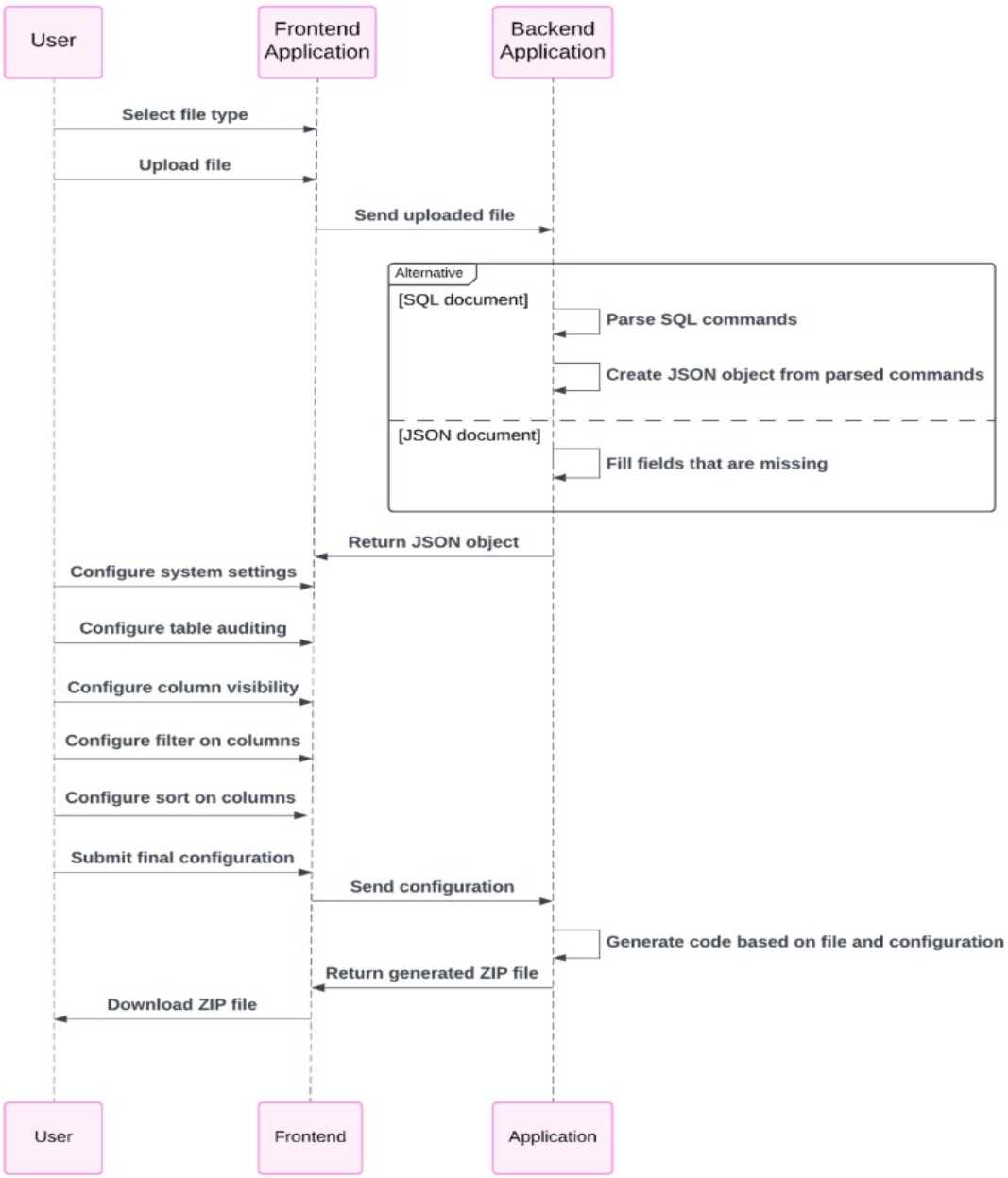
Slika 14: Dijagram komunikacije razvijenog sistema za generisanje aplikacija na osnovu šeme baze podatka

Na slici 14 dat je prikaz osnovnog modela komunikacije unutar sistema. Komunikacija između korisničkog i serverskog dijela u potpunosti je zasnovana na RESTful API principima, pri čemu se podaci razmjenjuju u JSON formatu ili kao fajlovi. Modularna

struktura aplikacije omogućava jednostavnu zamjenu ili proširenje pojedinih komponenti sistema, bez rizika po stabilnost i korisnički doživljaj cijelog sistema.

5.3.2. Tok podataka i interakcija između komponenti

Proces generisanja aplikacije može se predstaviti kroz sekvencijalni tok prikazan na slici 15.



Slika 15: Sekvencijalni dijagram toka podataka implementiranog rješenja

Tok akcija obuhvata sve faze korisničke interakcije i obradu na strani serverske aplikacije:

- Korisnik putem veb interfejsa bira tip ulaznih podataka (DDL skripta ili JSON fajl), te prilaže izabrani fajl.

- Klijent prosljeđuje priloženi fajl serverskoj aplikaciji putem HTTP POST zahtjeva.
- Obrada na serverskoj strani zavisi od tipa fajla:
  - Ako je priložena DDL skripta, sistem je parsira i transformiše u standardizovani JSON format na osnovu unaprijed definisane šeme.
  - Ako je priložen JSON fajl, vrši se validacija i eventualno dopunjavanje izostavljenih podataka (npr. mapiranja zavisnosti među tabelama, tipovi polja i slično).
- Rezultujući JSON objekat se šalje nazad klijentskoj aplikaciji, gdje ga korisnik dalje može prilagođavati kroz interaktivne interfejse (npr. podešavanje nadzora, vidljivosti polja, polja za filtriranje i sortiranje).
- Nakon što korisnik potvrdi sva podešavanja, šema i njena konfiguracija se šalju serverskoj aplikaciji, koja na osnovu primljenih informacija generiše kod aplikacije.
- Generisana aplikacija se kompresuje u ZIP arhivu i isporučuje korisniku putem veb interfejsa.

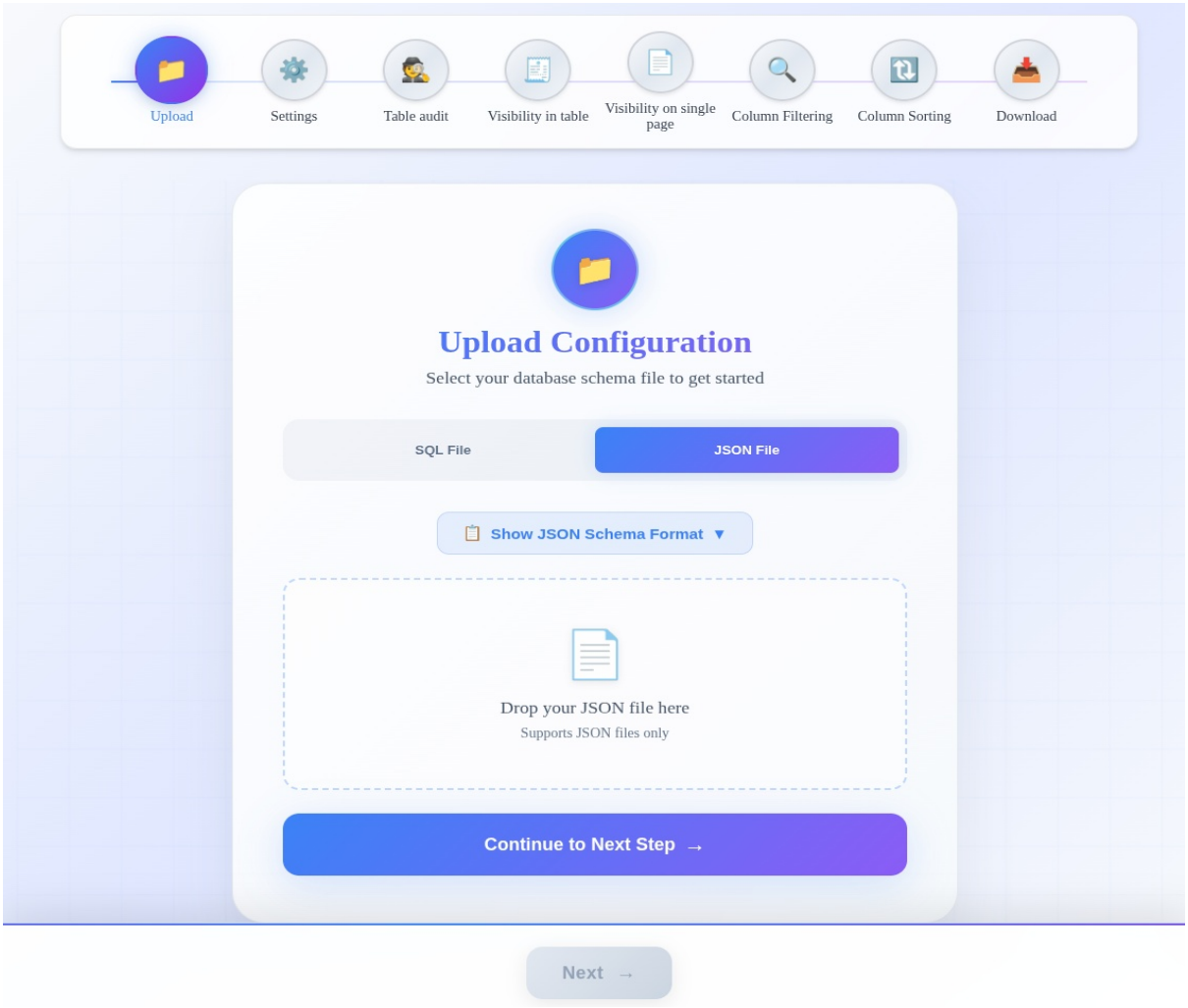
5.3.3. Ulazni podaci sistema

U implementiranom sistemu, ulazni podaci predstavljaju polaznu tačku cjelokupnog sistema za generisanje aplikacija. Obezbjeđivanjem podrške za različite oblike definicija šema, poput DDL skripti i strukturisanih JSON dokumenata po propisanoj šemi, sistem omogućava fleksibilnost i pristupačnost različitim kategorijama korisnika, od administratora baza do programera i analitičara.

Proces generisanja aplikacije započinje odabirom i unosom ulaznih podataka na korisničkom interfejsu generatora. Sistem podržava dva formata za predstavljanje šeme baze podataka, a to su MySQL DDL skripta i JSON fajl definisan prema zadatoj šemi.

Prvi tip ulaznih podataka je DDL skripta koja se koristi za definisanje strukture baze podataka unutar MySQL okruženja. Korištenjem naredbi kao što su CREATE DATABASE, CREATE TABLE i ALTER TABLE, korisniku je omogućeno da kroz postojeće ili nove skripte definiše tabele, njihove atribute, odnose među tabelama, kao i ograničenja na nivou baze (npr. primarni ključevi, strani ključevi, restrikcije jedinstvenosti i slično). Ovaj pristup posebno je važan za korisnike koji već posjeduju razvijene baze podataka i žele da ih automatski prenesu u novu aplikaciju uz minimalan manuelni napor.

Drugi podržani format ulaznih podataka je JSON fajl, koji mora biti usklađen sa unaprijed definisanom JSON šemom. Ova šema je kreirana da eksplicitno i transparentno opiše sve tehničke karakteristike baze: nazivi tabela i kolona, njihovi tipovi, veličine, vrijednosti za enumeracije, null ograničenja, jedinstvenost, primarne i strane ključeve, te druge bitne karakteristike. Takav pristup je posebno pogodan za automatizovanu obradu i omogućava veću kontrolu nad generisanjem aplikacije.



Slika 16: Prikaz ekrana za importovanje ulaznih podataka (SQL ili JSON fajl)

Na slici 16 je prikazan izgled korisničkog interfejsa u inicijalnom koraku, gdje korisnik bira način unosa, prilaže odgovarajuću datoteku (SQL ili JSON) i pristupa sljedećim koracima dalje konfiguracije. Intuitivan vizuelni prikaz jasno razlikuje podržane formate i omogućava korisniku da vrlo brzo započne proces generisanja.

Po izvršenom odabiru datoteke, klijentska aplikacija šalje zahtjev na odgovarajuću pristupnu tačku na serveru. Ukoliko korisnik uploaduje SQL skriptu, klijent poziva specifičnu pristupnu tačku čija je svrha parsiranje DDL-a. U slučaju da je učitan JSON fajl, klijent inicira poziv ka pristupnoj tački gdje se vrši validacija i dorada ulaznog JSON modela. Oba pristupa omogućavaju korisniku visok nivo fleksibilnosti, transparentnosti i brzine, što je dodatno unaprijeđeno intuitivnim korisničkim interfejsom.

Struktura i opis JSON šeme baze podataka

Kako je prethodno rečeno, jedan od tipova ulaznih podataka predloženog sistema za generisanje aplikacija jeste unaprijed definisana JSON šema, koja služi kao strog i formatiran opis modela baze podataka. Ova šema dizajnirana je tako da precizno i transparentno artikuliše sve relevantne karakteristike baza podataka, olakšavajući automatizaciju daljeg procesa generisanja aplikacija.

```typescript
interface Column {
    columnName: string;
    columnType: string;
    nullable: boolean;
    foreignKey: boolean;
    foreignTableName: string;
    foreignColumnName: string;
    autoIncrement: boolean;
    unique: boolean;
    displayName: string;
    defaultValue: string | number | boolean | null;
    visible: boolean;
    primaryKey: boolean;
    enumTypeValues: string[];
    columnSize: number;
    hasFilter: boolean;
    hasSort: boolean;
}

interface Table {
    tableName: string;
    hasAudit: boolean;
    columns: Column[];
}

interface Schema {
    databaseName: string;
    enabledBackend: boolean;
    enabledFrontend: boolean;
    tables: Record<string, Table>;
}
```

Slika 17: Definicija tipa JSON šeme u TypeScript programskom jeziku

JSON šema se sastoji od nekoliko glavnih sekcija i polja, gdje svako ima jasno definisanu svrhu i semantiku. Na slici 17 se vidi kako izgleda šema, kao i koji su tipovi njenih polja.

U nastavku su detaljno opisani osnovni elementi JSON šeme. Svako polje ima definisanu namjenu i pravila upotrebe, pa se od korisnika očekuje da poštuje zadatu šemu i na taj način definiše strukturu svoje baze. Očekivano ponašanje generisane aplikacije može se dobiti samo uz pomoć validnog JSON objekta i pravilnim popunjavanjem sljedećih polja:

- **databaseName** - Naziv baze podataka.
- **enabledBackend / enabledFrontend** - Vrijednosti koje označavaju da li je potrebno generisati serverski, odnosno klijentski, dio aplikacije.
- **tables** - Glavni objekat koji obuhvata sve tabele koje su uključene u šemu, gdje svaka tabela ima svoje ime kao ključ. Tabela, kao objekat, ima sljedeća polja:
    - **tableName** - Naziv tabele.

- **hasAudit** - Označava da li će tabela imati automatski generisana polja za nadzor kreiranja i izmjene podataka u bazi (npr. createdAt, updatedAt, createdBy, editedBy).
- **columns** - Lista objekata, gdje svaki predstavlja jednu kolonu date tabele sa svim relevantnim opisnim informacijama:
  - **columnName** - Originalno ime kolone u bazi podataka.
  - **columnType** - MySQL tip podataka kolone (npr. VARCHAR, INT, ENUM).
  - **nullable** - Vrijednost koja označava da li kolona može primati NULL vrijednosti.
  - **foreignKey**, **foreignTableName**, **foreignColumnName** - Navedena polja opisuju da li je kolona strani ključ i ako jeste, na koju tabelu i kolonu se referencira.
  - **autoIncrement** - Polje ima vrijednost „true” ako je kolona auto-increment.
  - **unique** - Atribut koji nosi informaciju da li kolona ima jedinstvenost u tabeli.
  - **displayName** - Naziv kolone koji će biti prikazan na korisničkom interfejsu.
  - **defaultValue** - Podrazumijevana vrijednost kolone.
  - **visible** - Da li je kolona vidljiva u tabelarnom prikazu.
  - **primaryKey** - Oznaka da li je kolona primarni ključ.
  - **enumTypeValues** - Lista vrijednosti ukoliko je columnType polje ENUM.
  - **columnSize** - Veličina polja ukoliko je polje columnType VARCHAR.
  - **hasFilter/hasSort** - Da li je moguće filtrirati/sortirati po toj koloni.

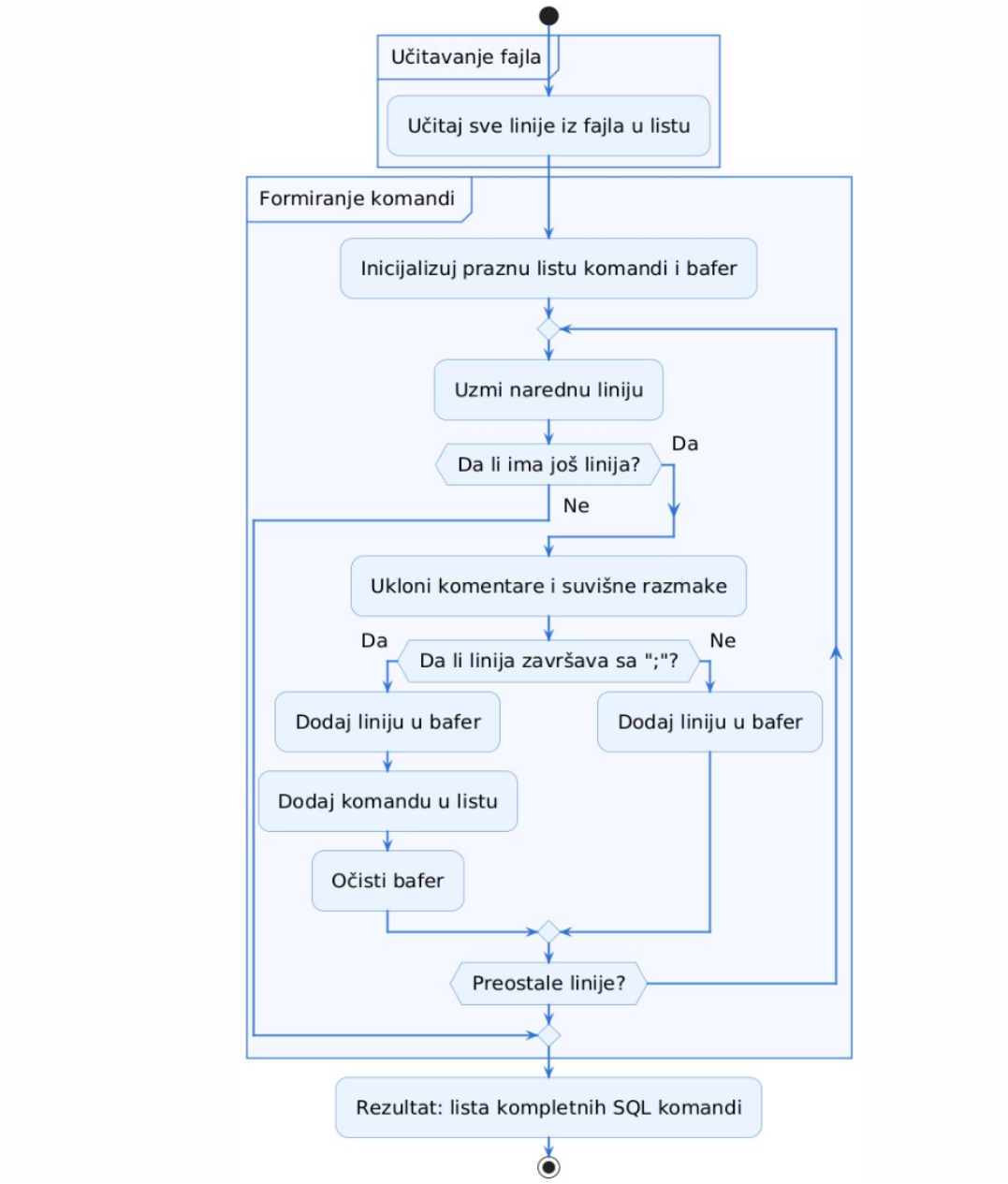
Uvođenjem ovako strukturisane šeme, postiže se visok nivo formalizma i kompatibilnosti, te se osigurava tačna transformacija modela baze podataka u programski kod bez rizika od nepreciznosti ili pogrešne interpretacije podataka. Ovaj pristup je lakši za dalju obradu, ali i za konfigurisanje budućeg sistema.

5.3.4. Parsiranje i transformacija ulaznih podataka

Jedan od najkritičnijih koraka u procesu generisanja je pouzdano, determinističko i precizno parsiranje ulaznih podataka. Primarni cilj jeste transformacija standardnog modela podataka u strogo definisan, programski obradiv model. Rezultat tog procesa je univerzalni JSON objekat koji postaje referentna tačka za sve naredne faze konfiguracije i generisanja aplikacije.

Parsiranje MySQL DDL skripte

Kada korisnik pošalje DDL skriptu, serverska aplikacija inicira razrađen parserski mehanizam. Prvi korak uključuje sekvencijalno čitanje fajla, pri čemu se linije analiziraju radi detekcije kompletnih SQL komandi. Pri tome se vrši filtriranje komentara, kao i ignorisanje praznih ili irelevantnih linija. Važan element segmentacije je prepoznavanje kraja komandi na osnovu karaktera ";". Već u ovoj fazi, sistem vodi računa da doslovno prati sintaksu DDL fajlova kako bi time izbjegao nejasnoće i smanjio rizik od pogrešne interpretacije DDL izraza. Svako odstupanje od standardne sintakse i podržanog formata može rezultovati ignorisanjem dijela izraza ili generisanjem greške.



Slika 18: Dijagram aktivnosti čitanja i objedinjavanja SQL DDL komandi

Na slici 18 je prikazan dijagram aktivnosti koji reprezentuje funkcionalnost čitanja fajla i njegove transformacije u komande. Prva faza čita fajl liniju po liniju, dok druga faza sklapa višelinijske komande.

Nakon detektovanja komandi, svaka od njih ide na drugi korak parsiranja. Na osnovu identifikovanog tipa SQL operacije, pokreću se parserske procedure. Komanda CREATE DATABASE je opciona. Ukoliko parser naiđe na ovu komandu, iz nje ekstrahuje naziv baze podataka, te ga kasnije aplicira pri imenovanju generisanih projekata.

U slučaju komande CREATE TABLE, parser najprije izdvaja naziv tabele, a zatim, koristeći prepoznavanje otvorene i zatvorene zagrade, izdvaja blok sa deklarisanim kolonama i eventualnim ograničenjima (eng. *Constraints*). Sadržaj ovog bloka se dalje segmentira na pojedinačne dijelove koristeći znak zapeta, kao separator, čime se dobija lista definicija. Svaka od njih može predstavljati deklaraciju kolone ili ograničenja. U slučaju definicije kolone, vrši se detaljna ekstrakcija sljedećih podataka:

- Naziv kolone - String koji je uvijek na početku komande.
- Tip podatka - String koji se nalazi neposredno iza naziva. Parser izdvaja glavni tip i dodatne informacije o veličini (npr. za VARCHAR) ili dopuštenim vrijednostima (npr. za ENUM).
- Auto-increment - Provjerava se prisustvo ključne riječi AUTO_INCREMENT.
- Nullability - Analizira se prisustvo ili izostanak NOT NULL, čime se određuje da li je vrijednost u koloni obavezna.
- Ograničenja - Dodatno se identifikuju ograničenja kao što su UNIQUE ili PRIMARY KEY.
- Strani ključevi - Ako je definicija kolone dio stranog ključa, parser detektuje ključne riječi REFERENCES, te izdvaja referenciranu tabelu i kolonu, ažurirajući internu mapu relacija između entiteta.

Na ovaj način, za svaku kolonu iz CREATE TABLE izraza, sistem instancira objekat koji uključuje sve tehničke i meta-informacije potrebne za kasnije faze generisanja i konfiguracije aplikacije. Ovakav pristup osigurava determinističu transformaciju složenih SQL izraza u precizan i programski obradiv JSON model baze podataka.

Prilikom obrade ALTER TABLE komandi, parser identifikuje i primjenjuje eventualne naknadne izmjene nad prethodno definisanim tabelama i kolonama, čime je omogućena visoka fleksibilnost definicija šeme.

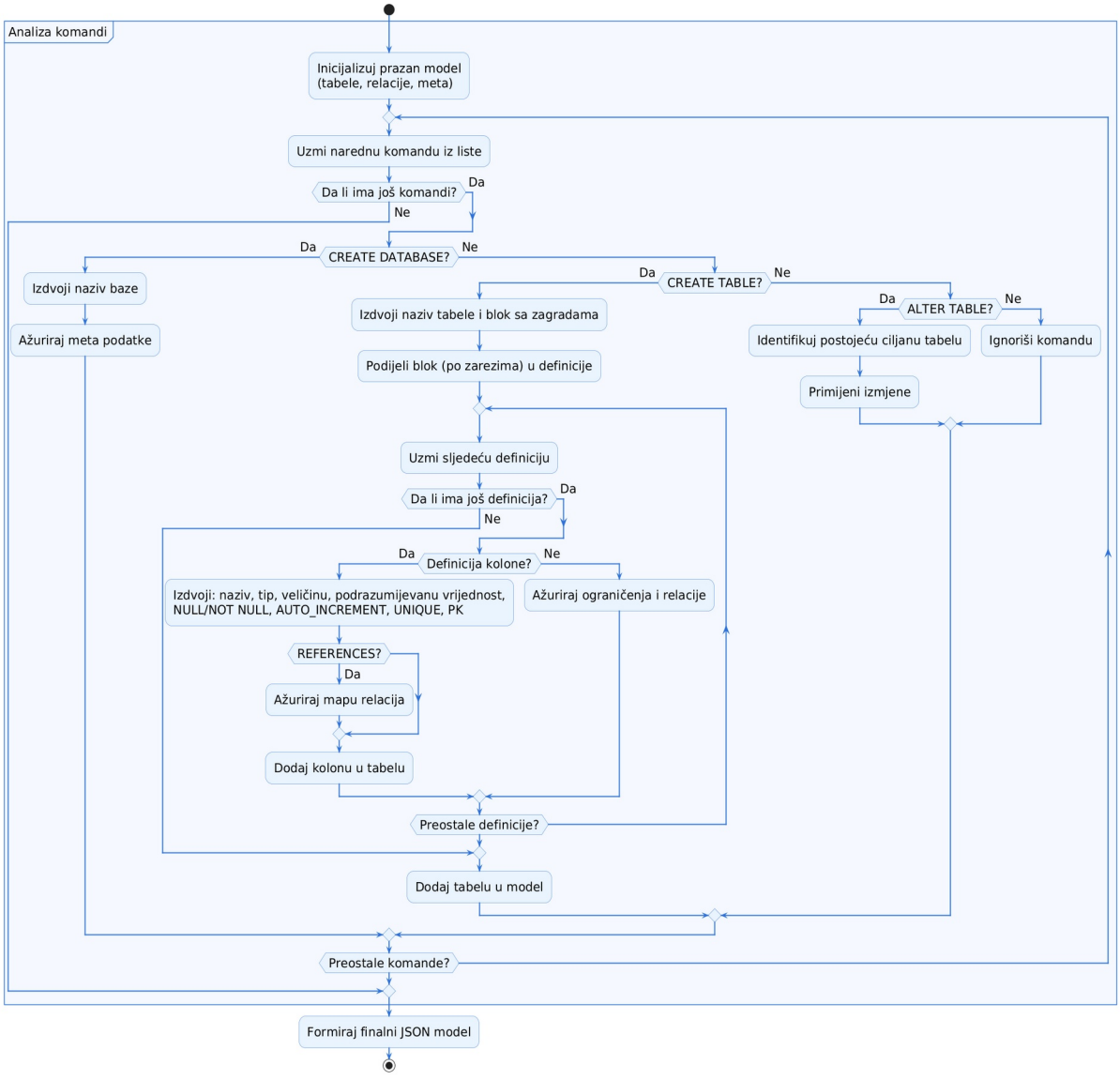
Pri procesu parsiranja, podržani su samo jasno definisani i standardizovani SQL izrazi. Sve nestandardne konstrukcije, napredna ograničenja, optimizacijski indeksi, trigeri, složene funkcije i izrazi mimo prepoznatljivih ključnih riječi, su ignorisani. Ključan zahtjev je poštovanje poretka ključnih riječi i standardizovanog opisa kolona. Svako ozbiljnije odstupanje može rezultovati djelimičnim modelom ili pogrešnim podacima.

Rezultat pravilnog parsiranja je struktura podataka koja obuhvata:

- Listu svih tabela sa nazivima i pripadajućim kolonama. Sve kolone sadrže potrebne informacije o tipovima, veličinama, podrazumijevanim vrijednostima i ograničenjima.

- Mapu relacija - Za svaku tabelu evidentirana su polja koja su strani ključevi, kao i tabele/kolone na koje pokazuju. Konkretno, ovi podaci su korisni za automatsko kreiranje relacija na aplikacionom sloju.

- Dodatne interne atribute - Na primjer, interni camelCase naziv kolone, inicijalna vidljivost, meta polja za audit, mogućnost filtriranja/sortiranja i drugo.



Slika 19: Dijagram aktivnosti parsiranja DDL komandi u objekte

Na slici 19 je prikazan dijagrama aktivnosti prethodno opisanog transformisanja DDL komandi u JSON objekat.

Obrada JSON objekta

Ukoliko se kao ulaz dostavi JSON fajl, proces je lakši i efikasniji. Glavna prednost ovog pristupa leži u tome što je korisnik već eksplicitno kreirao i konfigurisao model koji odgovara ciljevima aplikacije. To čini parserski zadatak manje zahtjevnim u poređenju sa DDL pristupom.

Obrada JSON objekta obuhvata dvije ključne operacije:

- Validacija - Svaka tabela mora sadržati naziv, opcione audit atribute i listu kolona. Svaka kolona mora imati osnovna polja (naziv, tip, naziv koji služi za prikaz, primarni ključ), a za strane ključeve posebno se provjerava ispravnost referenciranja. Nevalidna ili izostavljena polja izazivaju grešku koja se korisniku šalje kao odgovor.

- Dopunjavanje meta-podataka - Ako su pojedine sekcije nepotpune (npr. nije eksplicitno naveden atribut kolone), sistem ih automatski popunjava podrazumijevanim vrijednostima prema zadatim pravilima: kolona prihvata NULL vrijednosti ako nije drugačije definisano, strani ključevi su podrazumijevano nevidljivi, kao i primarni ključevi koji su samouvećavajući. Posebna pažnja posvećuje se automatskom generisanju interne mape veza među tabelama.

Glavni cilj je da se omogući korisniku da može ubrzati unos minimalnom deskripcijom, a da sistem sam dopuni sve dodatne elemente neophodne za generisanje aplikacija.

Na kraju ove faze, generisani JSON objekat ima proširena polja tako da sadrži sve što i parsirana ekvivalentna DDL skripta, te je potpuno pripremljen za prikaz na klijentskoj aplikaciji. Takav model postaje univerzalna tačka dalje komunikacije između serverskog i klijentskog sloja, gdje korisnik na klijentskoj aplikaciji dobija mogućnost pregleda i dorade konfiguracije.

5.3.5. Interaktivno podešavanje generisane aplikacije

Nakon što je proces parsiranja završen, a ulazni podaci uspješno konvertovani u standardizovani JSON model spreman za dalju obradu, sistem prelazi u fazu interaktivnog podešavanje budućih aplikacija. Ova etapa se ističe mogućnošću detaljne personalizacije. Korisnicima je omogućeno da iz programerski obradivog modela generišu aplikaciju potpuno usklađenu s poslovnim zahtjevima i operativnim specifičnostima.

Motivacija za ovu fazu je želja da se izbjegnu ograničenja postojećih generatora koda, koji korisniku generišu samo ono što je već definisano u šablonima. Interaktivno podešavanje uvodi sloj fleksibilnosti gdje korisnik bez ikakve ručne izmjene koda upravlja određenim aspektima budućeg sistema. Prelaskom u ovu fazu, korisniku se kroz grafički interfejs prikazuje vodič sa jasno definisanim koracima. Svaka faza vodi korisnika kroz niz odluka i podešavanja, čime mu omogućava da na intuitivan način prilagodi izgled, sigurnost i ponašanje generisane aplikacije. Ove faze su osmišljene kako bi pokrile specifične segmente konfiguracije, koje će biti detaljno opisane u nastavku.

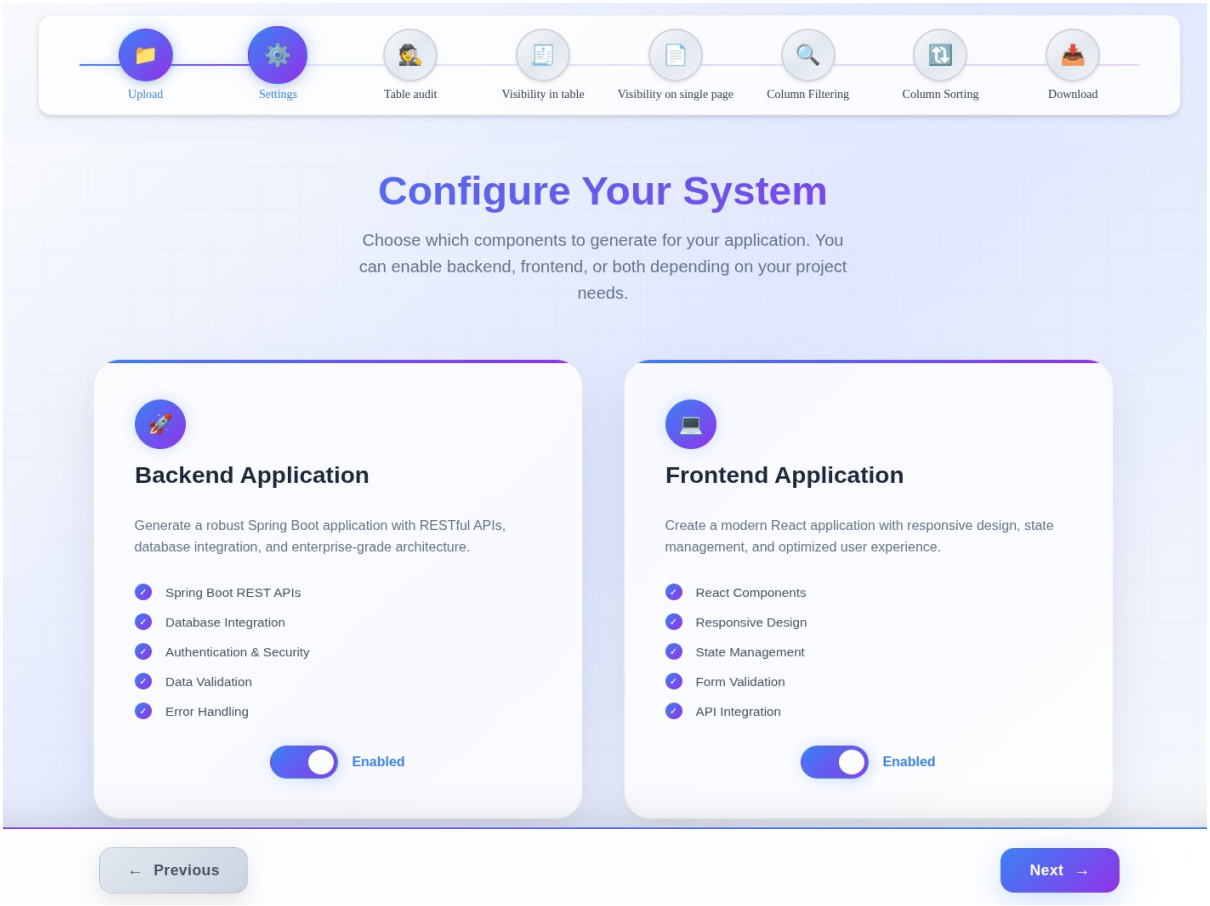
Odabir tipa aplikacije za generisanje

Prva faza konfiguracije se odnosi na osnovnu arhitektonsku odluku gdje korisnik bira da li želi generisati:

- Spring Boot kao serversku aplikaciju,
- React kao klijentsku aplikaciju.

Ovakav izbor omogućava da rješenje bude primjenjivo kako u ranim fazama razvoja, tako i u postojećim projektima gdje je potrebno zamijeniti ili izgraditi samo jedan sloj.

Na slici 20 je prikazan interfejs gdje korisnik može omogućiti generisanje dijelova sistema. Ukoliko je u prvom koraku priložena DDL skripta, njome se nije moglo ranije odrediti koji dio sistema će biti generisan, tako da se postavlja podrazumijevana vrijednost da se generiše i klijentska i serverska aplikacija. Ukoliko je priložena JSON datoteka, kao inicijalno stanje, interfejs prikazuje zadato podešavanje. Zaglavlje reprezentuje progres gdje korisnik vidi do kog koraka je stigao. Na dnu ekrana je omogućena navigacija ka sljedećem koraku, ili povratak na prethodni. Upotrebom Redux-a, korisnički interfejs ne gubi prethodna stanja, tako da se korisnik uvijek može vratiti i izmijeniti neku od postavljenih konfiguracija.



Slika 20: Korisnički interfejs gdje korisnik bira dijelove sistema koji će se generisati

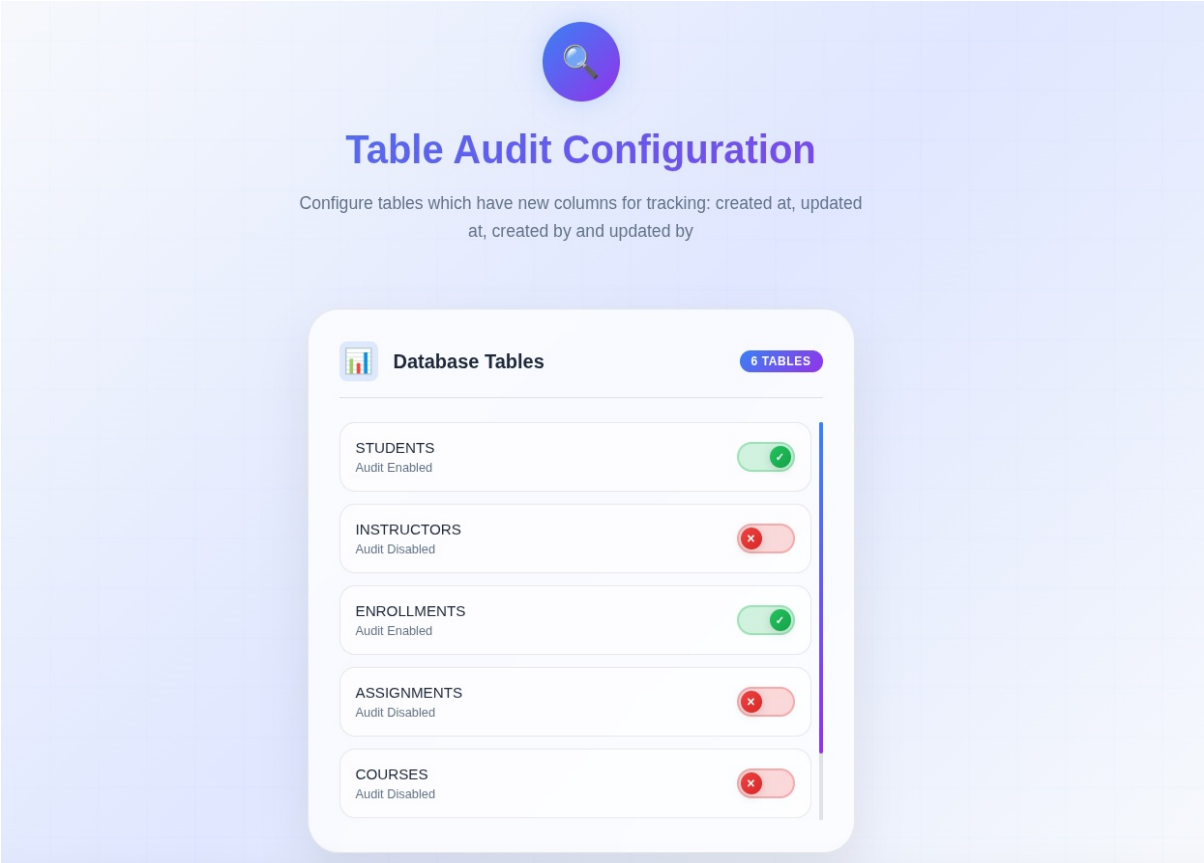
Odabir i podešavanje audit polja

Nakon inicijalnog tehničkog izbora, korisnik ulazi u fazu konfigurisanja tzv. audit polja. U profesionalnim i sigurnosno osnaženim okruženjima vrlo je važno imati automatski zapis o nastanku i izmjenama zapisa u bazi. To su podaci koji nose informaciju ko je, kada i sa kog naloga izvršio promjenu.

Interfejs aplikacije omogućava korisniku pregled svih pronađenih tabela u modelu, te jednostavnim biranjem se određuje koje tabele treba proširiti audit kolonama:

- createdAt - Datum i vrijeme kreiranja zapisa.
- updatedAt - Datum i vrijeme posljednje izmjene.
- createdBy - Primarni ključ korisnika koji je kreirao zapis.
- updatedBy - Primarni ključ korisnika koji je zadnji izmijenio podatak.

Ovaj korak je potpuno opcionalan. Korisnik može odabrati sve, pojedine ili nijednu tabelu, u zavisnosti od poslovnih potreba i interne politike nadzora. Dio korisničkog interfejsa je prikazan na slici 21, gdje korisnik može omogućiti ili onemogućiti automatsko dodavanje audit polja. Po podrazumijevanoj vrijednosti su isključena polja za sve tabele.

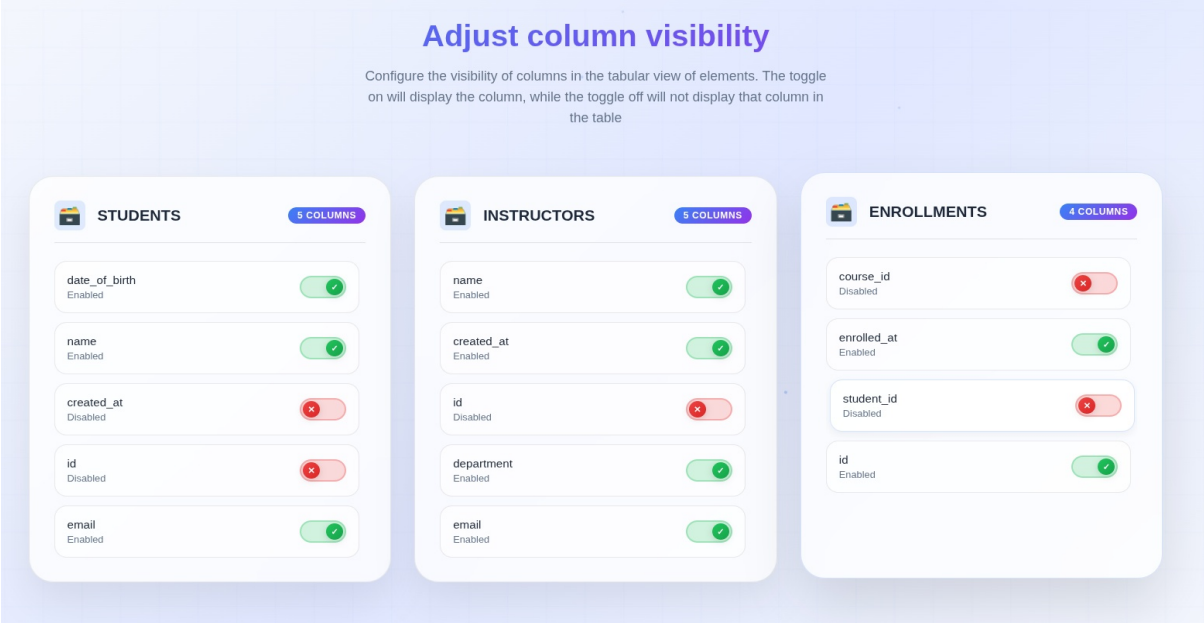


Slika 21: Dio korisničkog interfejsa za selektovanje tabela kojima je potrebno dodati audit polja

Konfigurisanje vidljivosti kolona

Sljedeća sekvenca u procesu vodi korisnika da precizno definiše koje kolone će biti vidljive krajnjim korisnicima u različitim dijelovima aplikacije, a to su tabelarni prikaz i detaljni prikaz pojedinačnog objekta. Interfejs (kao na slici 22) automatski predlaže optimalnu vidljivost, gdje podrazumijevano uklanja vidljivost za kolone koje su primarni samouvećavajući ključevi, kao i za kolone koje su strani ključevi.

Korisnik može dodati ili sakriti kolone, čime podešava vizuelnu kompleksnost korisničkog interfejsa. Preporučeno je da se u ovoj fazi podesi optimalan broj kolona za tabelarni prikaz.



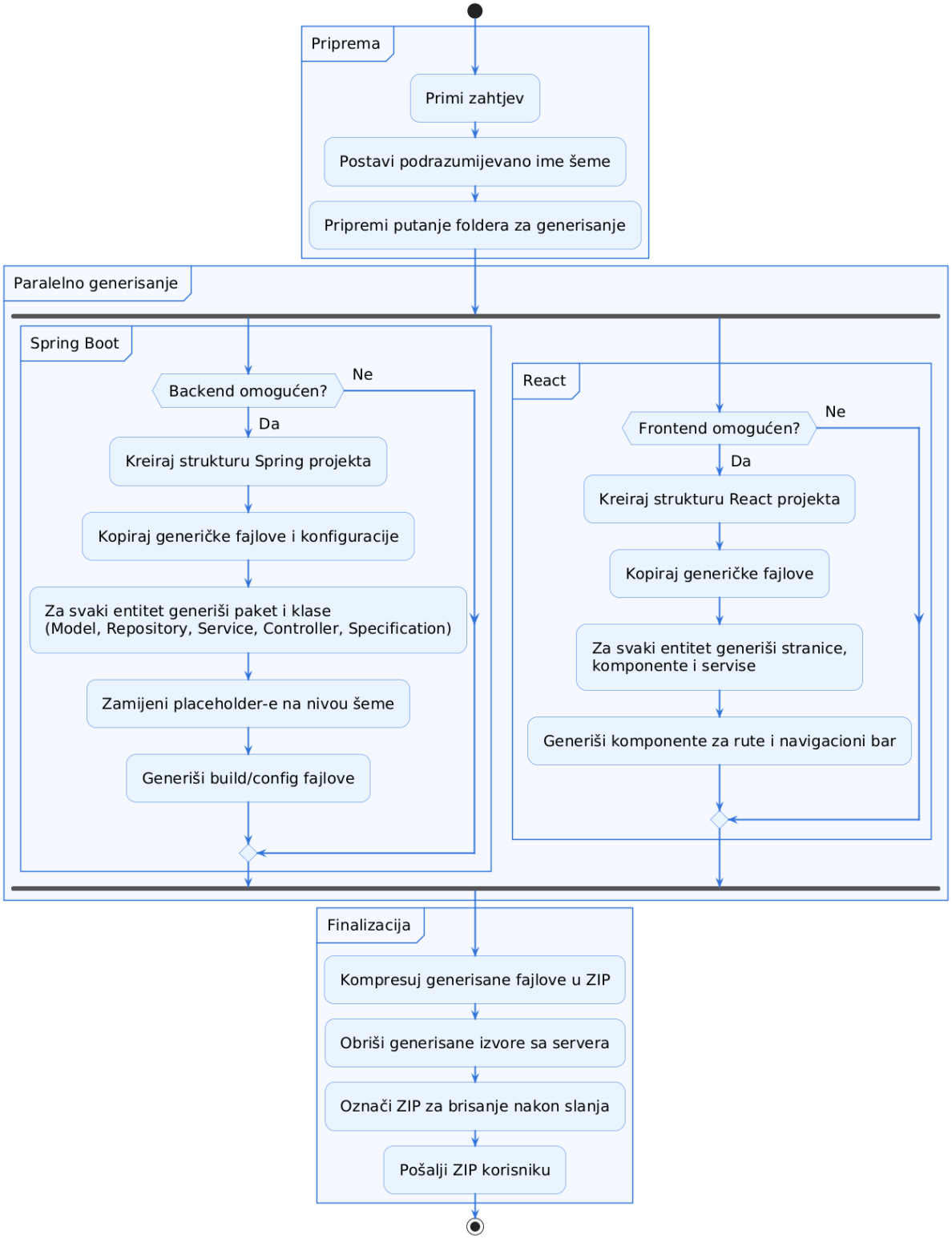
Slika 22: Konfigurisanje vidljivosti kolona po tabeli (za tabelarne preglede)

Odabir polja za filtriranje i sortiranje

Nakon što su definisane vidljivosti kolona, interfejs vodi korisnika kroz konfiguraciju polja za filtriranje i sortiranje. Ova funkcionalnost je ključna za rad s velikim količinama podataka, gdje je brz pristup i pretraga često najvažnija operacija. Korisniku se nudi lista svih tabela sa pripadajućim kolonama, gdje korisnik bira kolone koje će biti dio filtriranja i sortiranja podataka.

Finalna konfiguracija i pokretanje procesa generisanja

Po završetku svih koraka konfiguracije, korisnik dobija sumarizovani pregled svojih odabira. Tek nakon što potvrdi generisanje, svi parametrizovani meta-podaci se šalju putem REST API poziva na serversku aplikaciju. Serverska aplikacija tada koristi kompletne korisničke postavke kako bi, u zavisnosti od izabranog tipa aplikacije, automatski generisao izvorni kod nove serverske i/ili klijentske aplikacije. Ova automatizovana procedura osigurava da se pravila, sigurnosne politike i korisničke preferencije precizno i vjerodostojno reflektuju u konačnom softverskom rješenju.



Slika 23: Dijagram aktivnosti generisanja Spring i React aplikacija

Prije samog procesa generisanja, bilo je neophodno izraditi reprezentativne projekte za oba sloja (serverski i klijentski) koji pokrivaju širok spektar mogućih veza među entitetima, kao i različite tipove polja sa pripadajućim ograničenjima. Na osnovu ovih projekata izdvojene su šablonski (eng. *template*) fajlovi univerzalne strukture, koji služe kao osnova za

sve buduće generisane aplikacije. Dijelovi koda koji zavise od konkretne šeme baze podataka zamijenjeni su posebno definisanim oznakama (eng. placeholders), u obliku konstanti #{KONSTANTA}#, čime su šabloni pripremljeni za dinamičku adaptaciju. Ovakav pristup omogućava generatoru da u procesu izgradnje nove aplikacije precizno personalizuje svaki dio koda u skladu sa stvarnim entitetima i zahtjevima korisnika.

Sam proces generisanja odvija se kroz nekoliko definisanih faza. Na početku se, u skladu sa korisničkim izborom, automatski kreira osnovna struktura direktorijuma aplikacije, bilo da je riječ o serverskom, klijentskom ili oba sloja sistema. Nakon toga, sve zajedničke datoteke koje nisu zavisne od konkretnog poslovnog modela kopiraju se direktno u odgovarajuće direktorijume. Zatim, za svaki entitet iz korisnički definisane šeme baze podataka, generator automatski formira zaseban poddirektorijum, čime se osigurava modularnost, preglednost i lako održavanje generisanog koda.

Poseban korak ovog procesa predstavlja obrada šablonskih datoteka koje sadrže predefinisane oznake za zamjenu (placeholder). U toku ovog koraka, generator sukcesivno prolazi kroz svaki od ovih fajlova i zamjenjuje sve pronađene placeholder-e. Placeholder se dinamički mijenja sa dijelovima koda koji je generisan na osnovu meta-podataka konkretnog entiteta, njegovih kolona, kao i unaprijed definisanih sigurnosnih i funkcionalnih opcija. Kada je proces generisanja u potpunosti završen, kompletan izvorni kod se automatski kompresuje u ZIP arhivu i prosljeđuje korisniku putem aplikacije. Na taj način, generisani materijal se odmah uklanja sa serverske strane, što dodatno garantuje privatnost i sigurnost svih korisničkih podataka.

Prethodno opisani proces je prikazan u obliku dijagrama aktivnosti na slici 23.

Proces generisanja Spring Boot aplikacije kao serverskog dijela sistema uključuje:

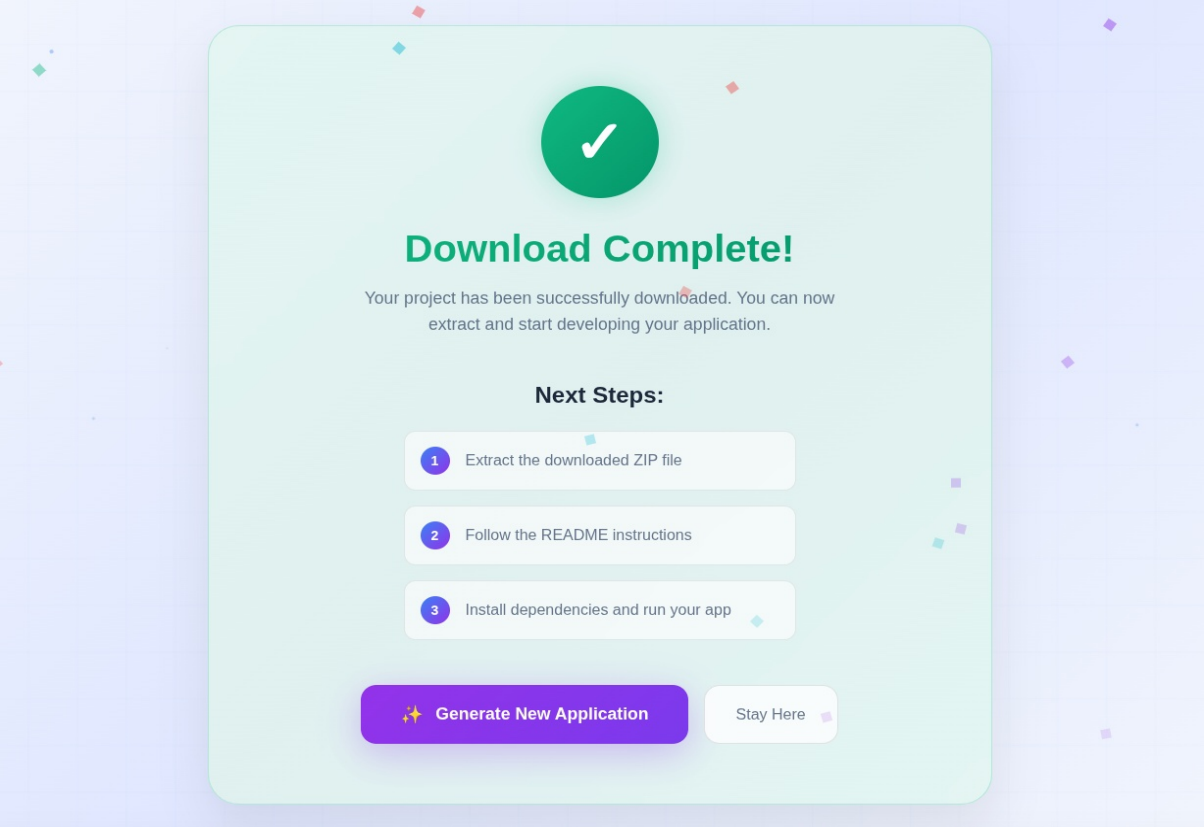
- Izgradnju standardizovane strukture direktorijuma za Spring Boot projekat, pripremljenu za rad sa Java 17 okruženjem.
- Automatsko kreiranje odvojenih paketa za svaki entitet iz šeme sa pripadajućim klasama:
    - Entitetska Java klasa sa validacionim anotacijama, relacionim aspektima i, po potrebi, audit atributima.
    - Repository klasa sa osnovnim CRUD metodama, što omogućava standardizovano upravljanje podacima na nivou baze.
    - Specification klasa za napredno filtriranje podataka, čime se omogućava dinamička pretraga prema korisničkim kriterijumima.
    - Servisni sloj u kojem se nalaze sve poslovne funkcije vezane za manipulaciju podacima.
    - REST API kontroler, sa zaštićenim pristupnim tačkama za osnovne CRUD operacije.
- Podešavanje sigurnosti sistema kroz Spring Security, sa integrisanom autentifikacijom putem JWT tokena i mehanizmom za osvježavanje tokena.

- Generisanje konfiguracionih i build fajlova (poput build.gradle) sa svim potrebnim zavisnostima, uključujući alatke za izradu API dokumentacije (OpenAPI).

Ukoliko korisnik izabere generisanje klijentskog sloja, tj. React aplikacije, generator prolazi kroz sljedeće korake:

- Formira strukturu React projekta, organizovanu po entitetima, sa definisanim direktorijumima za zajedničke komponente, servise i konfiguracije.
- Dinamički generiše korisničke interfejse za pregled, kreiranje, izmjenu i brisanje podataka, pri čemu su svi prikazi i dostupnost operacija usklađeni sa privilegijama definisanim tokom konfiguracije.
- Implementira React Router sa višestrukim rutama i stranicama, uključujući module za autentifikaciju (login, promjena lozinke), autorizaciju i sigurnosna pravila na nivou korisničkog interfejsa.
- Generiše početni administratorski interfejs, te strane i forme za upravljanje korisnicima, što omogućava kontrolu pristupa aplikaciji.

Na ovaj način, korisniku je omogućeno da, putem potpuno automatizovanog i skalabilnog procesa, za samo nekoliko trenutaka dobije sistem prilagođen svojim poslovnim potrebama. Nakon uspješnog generisanja, korisnik preuzima ZIP fajl sa kompletnim izvornim kodom (slika 24).



Slika 24: Dio korisničkog interfejsa nakon preuzimanja generisanog sistema

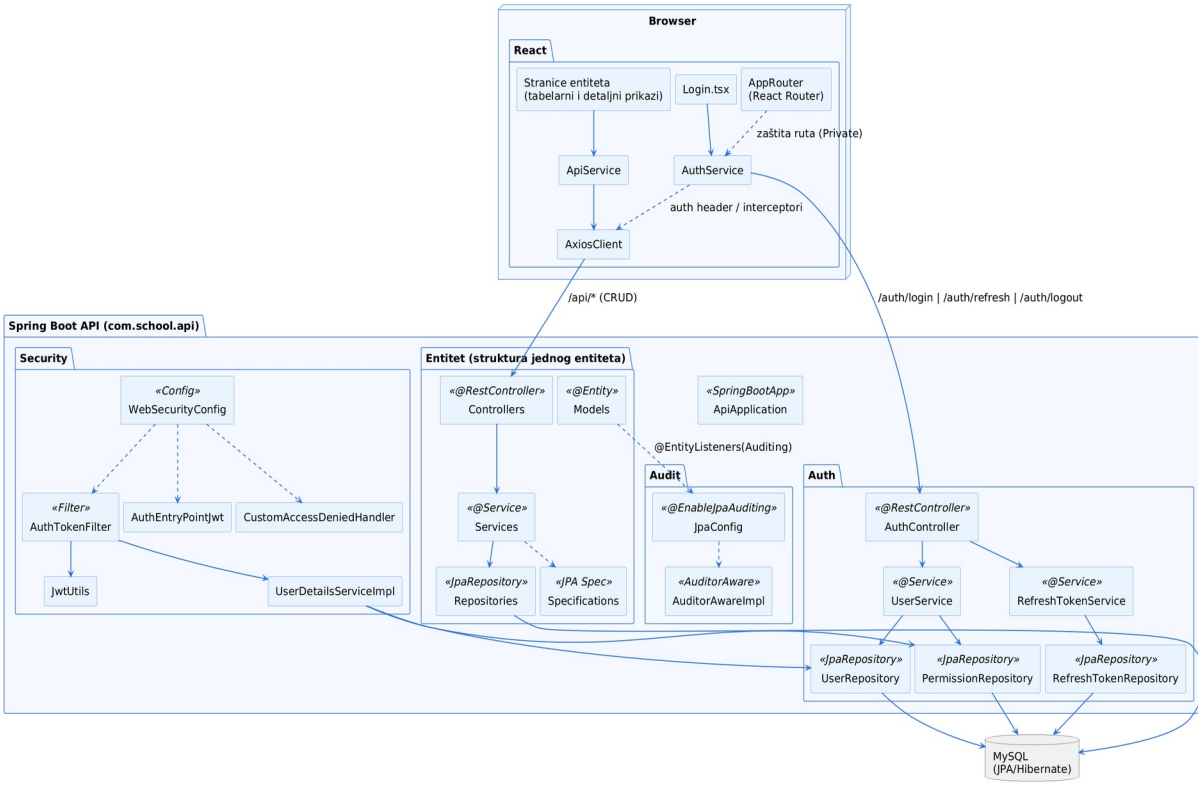
5.4. Opis generisane aplikacije

Rezultat prethodno opisanog procesa je generisana aplikacija koja obuhvata serversku i klijentsku aplikaciju. Takav sistem, osim osnovnog rukovanja podacima, sadrži i funkcionalnosti poput autentifikacije, autorizacije, nadzora i fleksibilnog korisničkog interfejsa. Ovakav automatizovani pristup donosi višestruke benefite: skraćuje vrijeme razvoja, smanjuje mogućnost grešaka, podiže sigurnosne standarde i omogućava brzu adaptaciju softverskog rješenja novim poslovnim potrebama.

5.4.1. Arhitektura generisanog sistema

Generisani sistem zasniva se na jasno razdvojenoj dvoslojnoj arhitekturi: serverski dio i klijentski dio komuniciraju putem REST API-ja. Serverska aplikacija je organizovana u standardizovane pakete sa slojevima (Controller, Service, Repository, Specification, Model) po svakom entitetu, pri čemu JPA/Hibernate obezbjeđuje ORM mapiranje ka MySQL bazi. Sigurnosni mehanizmi primjenjuju Spring Security sa JWT autentifikacijom i rotacijom *access/refresh* tokena, dok autorizacija počiva na granularnim privilegijama po resursu. Dijeljene funkcionalnosti uključuju JPA auditing (AuditorAware) i zajedničke *util* module, čime se obezbjeđuju konzistentnost i ponovna upotrebljivost koda. Ovakva modularna i slojevita organizacija olakšava održavanje, testiranje i nadogradnju sistema.

Klijentski dio je React aplikacija sa modulima organizovanim po entitetima. Svaki entitet sadrži stranicu za tabelarni prikaz, stranicu za prikaz jednog objekta, te gradivne komponente (forme, tabele, modali) i servisni sloj. Navigacija je realizovana React Router-om, sa zaštićenim rutama i provjerom privilegija. Komunikacijski sloj koristi Axios i generisani ApiService, dok zajedničke UI komponente i prilagođeni hook-ovi (paginacija, filtriranje, osvježavanje tokena) obezbjeđuju dosljedan korisnički interfejs i efikasno rukovanje podacima.



Slika 25: Dijagram komponenti generisanog sistema

Opisana arhitektura primjenjuje princip razdvajanja odgovornosti, omogućava konzistentno širenje na nove entitete bez strukturnih izmjena. Dijagram komponenti generisanog sistema je prikazan na slici 25.

5.4.2. Pokretanje aplikacije

Za korištenje i testiranje generisane aplikacije, neophodno je sprovesti nekoliko inicijalnih koraka prilagođavanja i instalacije zavisnosti. Serverska i klijentska aplikacija dolaze sa definisanim konfiguracionim fajlovima i paketima zavisnosti, što pojednostavljuje proces pokretanja sistema.

Generisana serverska aplikacija koristi Java 17 i Spring Boot framework (verzija 3.2.3), te zavisnosti kao što su JPA, Spring Security, JWT i MySQL konektor. Kompletan sistem je spreman za upotrebu nakon što korisnik:

1. prilagodi parametre za bazu u konfiguracionom fajlu (application.properties);
2. instalira potrebne biblioteke putem Gradle build sistema;
3. pokrene aplikaciju.

Ostali aspekti, uključujući sigurnosne mehanizme i REST API sloj, dolaze konfigurisani prema izboru i unosu tokom faze interaktivnog podešavanja.

Klijentska aplikacija je razvijena korištenjem React-a, TypeScript-a i biblioteka za korisnički interfejs. Prije inicijalnog pokretanja, potrebno je:

- instalirati Node.js (preporučena verzija 18.x ili novija);
- izvršiti instalaciju svih podataka putem komande *npm install;*
- formatirati kod komandom *npm run format;*
- postaviti varijablu za domen serverske aplikacije u .env fajl;
- pokrenuti server aplikacije komandom *npm run start*.

Nakon ovih koraka, aplikaciji je moguće pristupiti putem veb pregledača, sa svim funkcionalnostima vidljivim i dostupnim u zavisnosti od korisničkih privilegija.

5.4.3. Autentifikacija korisnika

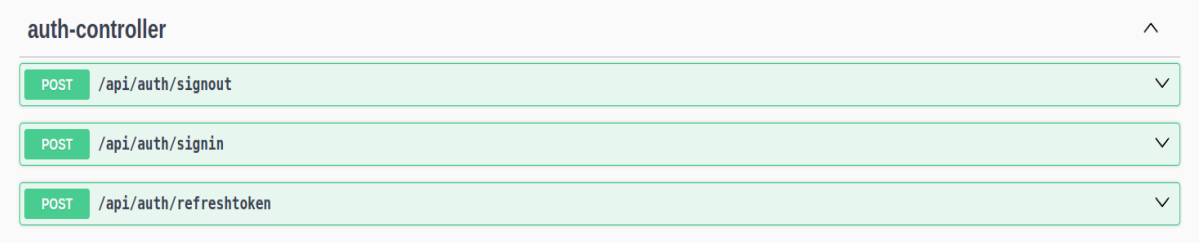
Autentifikacija u generisanom sistemu je realizovana kroz mehanizam JWT tokena, sa podrškom za osvježavanje (eng. *refresh*) sesije. Ova arhitektura omogućava stabilno razdvajanje odgovornosti između serverskog i klijentskog sloja, kao i potpunu kontrolu nad sistemom sesija korisnika.

Autentifikacija sa strane servera

Na serverskoj strani, ključne funkcionalnosti autentifikacije implementirane su kroz REST API unutar posebnog kontrolera za autentifikaciju. Tri osnovne pristupne tačke su prikazane na slici 26 i imaju sljedeće funkcije:

- POST */api/auth/signin* - Klijent u zahtjevu šalje korisničko ime i lozinku. Na osnovu zadatih kredencijala, vrši se validacija, a zatim se generišu *access* i *refresh* tokeni koje server vraća klijentu. *Access* je JWT token koji se koristi za autentifikaciju svih daljih zahtjeva korisnika prema serveru. *Refresh* omogućava dobijanje novog *access* tokena bez nove prijave, čime se produžava korisnička sesija ne narušavajući sigurnosnu politiku.
- POST */api/auth/refreshtoken* - Kada *access* token istekne, klijent može, korištenjem važećeg *refresh* tokena, zatražiti novi par tokena. Serverska aplikacija validira *refresh* token i ukoliko je validan izdaje nove tokene, a stari *refresh* token poništava (tzv. "rotate" mehanizam).
- POST */api/auth/signout* - Pristupna tačka za odjavu korisnika.

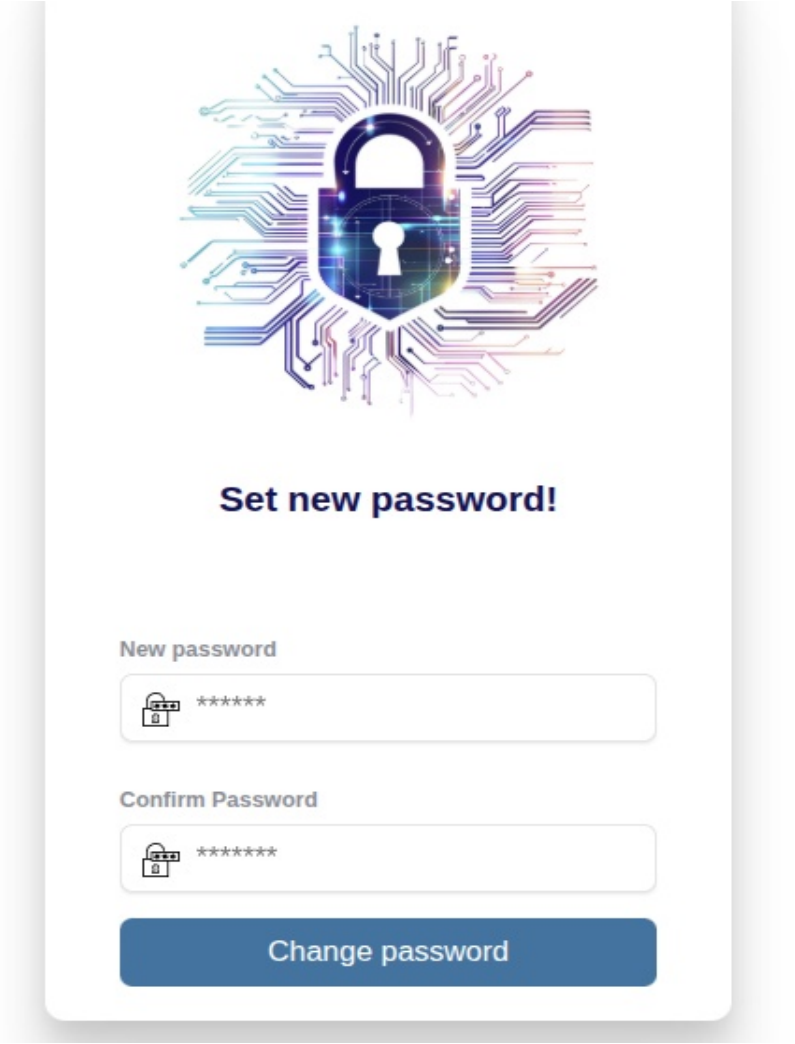
Vrijeme važenja *access* i *refresh* tokena se podešava kroz parametre u konfiguracionom *application.properties* fajlu Spring Boot aplikacije.



Slika 26: Swagger interfejs sa prikazom pristupnih tačaka kontrolera za autentifikaciju

Autentifikacija sa strane klijenta

Na klijentskoj strani, autentifikacija je usklađena sa savremenim korisničkim iskustvom i sigurnosnim standardima veb aplikacija. Pristup sistemu započinje kroz login formu, gdje korisnik unosi svoje pristupne podatke, odnosno korisničko ime i lozinku. Uneseni kredencijali se prilikom prijave šalju POST zahtjevom prema */api/auth/signin* pristupnoj tački. Ukoliko je u pitanju prva prijava korisnika, odnosno ako je korisniku inicijalno dodijeljena lozinka od strane administratora, aplikacija ga automatski preusmjerava na formu za promjenu lozinke, kao što je ilustrovano na slici 27. Ova sigurnosna mjera obavezna je za nastavak rada na aplikaciji.



Slika 27: Korisnički interfejs prilikom promjene dodijeljene lozinke

Nakon uspješne autentifikacije, klijentska aplikacija pamti dobijene access i refresh tokene. Access token je pohranjen u memoriji aplikacije, dok se refresh token smješta kao HttpOnly cookie. Tokom dalje upotrebe, svaki zahtjev prema zaštićenim resursima automatski uključuje access token, te korisnik zadržava potpun pristup bez potrebe za višestrukim prijavljivanjima.

Kada access token istekne server signalizira odgovorom sa HTTP status kodom 401. U tom slučaju, klijentska aplikacija automatski koristi refresh token kako bi, putem POST zahtjeva na pristupnu tačku */api/auth/refreshtoken*, dobila nove važeće tokene. Ova funkcionalnost osigurava nesmetanu i sigurnu sesiju korisnika, dokle god je refresh token validan. Kada korisnik odluči da se odjavi iz sistema, aplikacija poziva pristupnu tačku na putanji */api/auth/signout*, čime se poništavaju svi važeći tokeni i sesija se prekida.

Ovaj tok pruža optimalan balans između sigurnosti i korisničkog komfora, osiguravajući da svaka sesija bude zaštićena uz minimalno narušavanje korisničkog iskustva.

5.4.4. Autorizacija korisnika

Nakon uspješne autentifikacije, svaki pristup aplikaciji se obrađuje po definisanim pravilima autorizacije. Riječ je o sistemu provjere prava pristupa koji osigurava da korisnici mogu pristupati, modifikovati ili brisati podatke isključivo u skladu s privilegijama koje su im dodijeljene. Ovaj model garantuje zaštitu podataka, usklađenost s poslovnim pravilima, kao i sigurnost sistema od neovlaštenih manipulacija.

Model privilegija i administracija korisnika

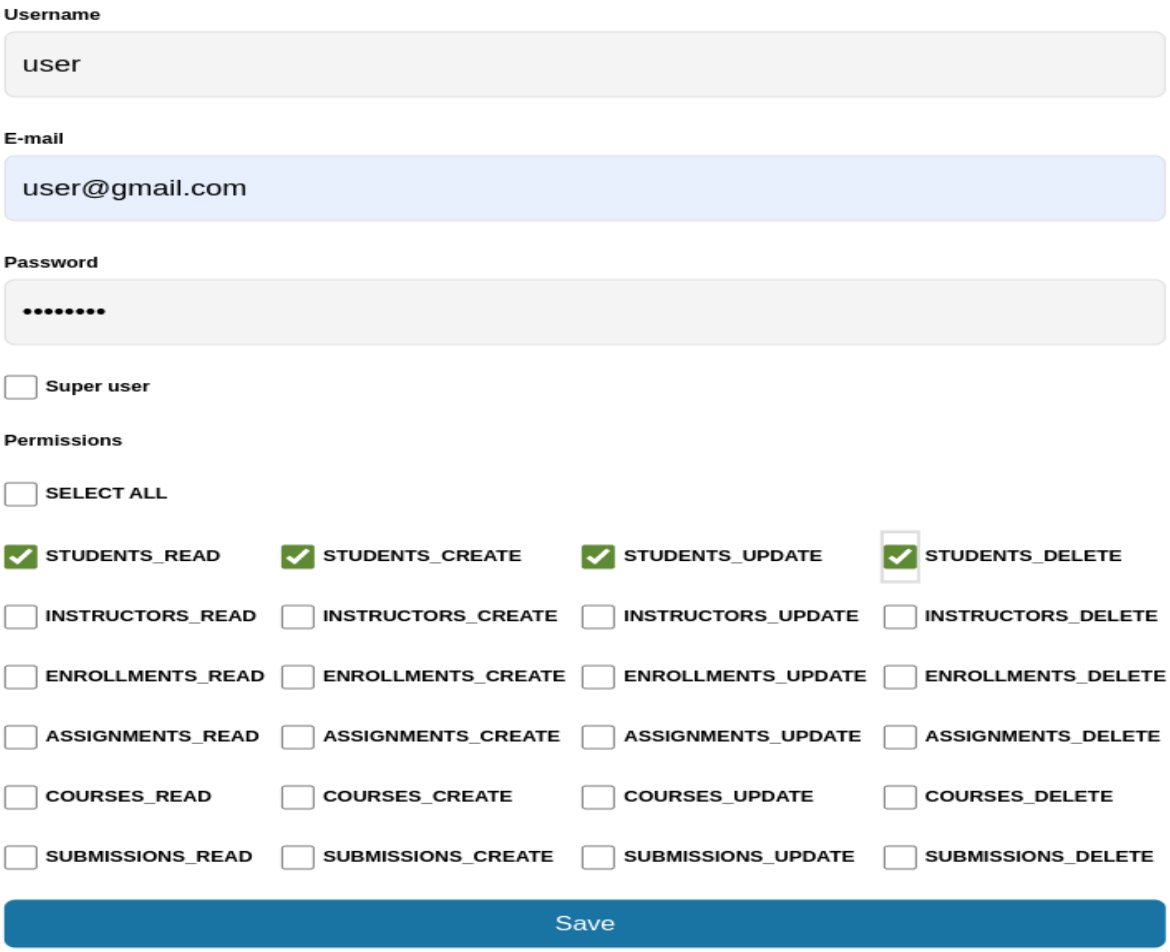
U generisanom sistemu postoje dvije vrste korisničkih naloga:

- Super user (administrator) - Korisnik sa potpunim pristupom korisnicima sistema. Samo super user može upravljati korisnicima: kreirati nove naloge, uređivati ih, dodjeljivati im privilegije i brisati. Pored toga, super user može da postavi sebi i drugim korisnicima privilegije, tako da može da ima nalog koji upravlja određenim dijelom sistema.

- Običan korisnik - Korisnik kome su dodijeljene privilegije na nivou pojedinačnih tabela i tipova operacija (čitanje, kreiranje, izmjena, brisanje). Svaka privilegija definisana je granularno po svakoj tabeli i svakoj CRUD operaciji (slika 25).

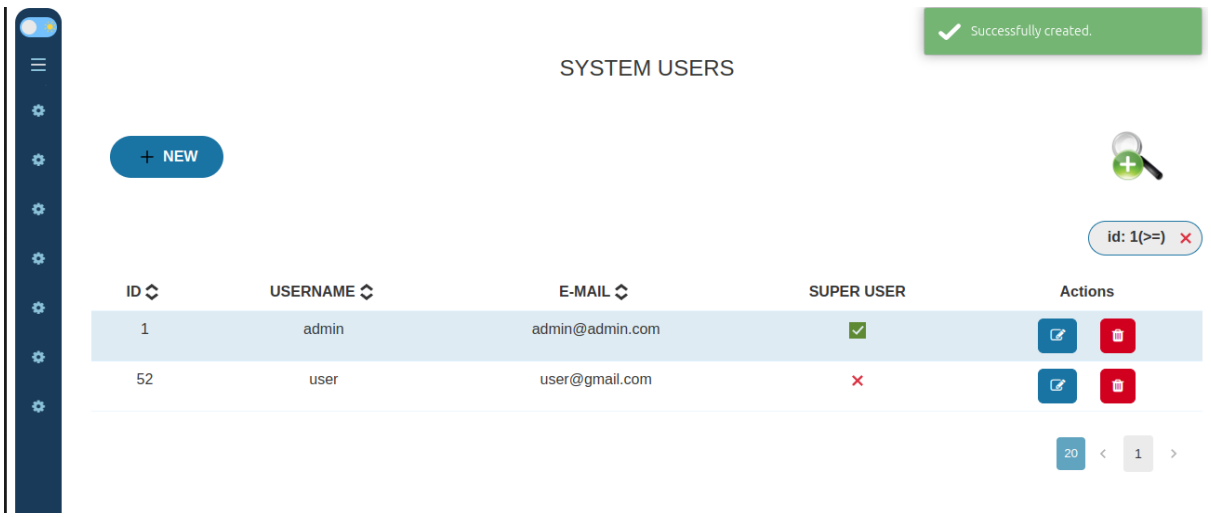
Kompletan životni ciklus korisnika, od kreiranja do brisanja, zavisi samo od super user-a i njegovog upravljanja. Novi korisnici ne mogu samoinicijativno kreirati naloge, niti mijenjati vlastite privilegije, što sprečava eskalaciju prava i povećava sigurnost sistema.

Pri prvom pokretanju sistema, sistem automatski dodaje prvog korisnika sa korisničkim imenom „admin” i lozinkom „admin”. To je nalog koji služi kao ulazna tačka u sistem.



Slika 28: Korisnički interfejs sa formom za kreiranje novog korisnika

Slika 28 prikazuje dio korisničkog interfejsa sa formom za kreiranje novog korisnika. Klikom na dugme „Save”, korisnički nalog se čuva u sistemu i na ekranu se prikazuje poruka o uspješnosti akcije (interfejs sa slike 29).



Slika 29: Korisnički interfejs koji prikazuje listu korisnika sistema

Autorizacija na strani servera

Na serverskoj strani, striktna kontrola pristupa realizovana je kroz kombinaciju Spring Security mehanizama, te kontrole JWT tokena. Svaki dolazni zahtjev nosi JWT token (dobijen pri autentifikaciji) koji u korisničkim podacima uključuje informaciju o njegovim privilegijama. To je lista dozvola sa vrijednostima poput STUDENTS_READ, COURSES_UPDATE i druge.

Za svaku pristupnu tačku serverske aplikacije, implementirane su provjere na dva nivoa: provjera autentifikacije i provjera autorizacije. Prije izvršenja poslovne logike servis provjerava da li korisnik ima eksplicitnu dozvolu za traženu akciju nad konkretnim resursom. Ove provjere su implementirane kroz filtere i sigurnosne anotacije (*@PreAuthorize*). Time je omogućena granularna kontrola pristupa dijelovima API-ja, minimizujući rizik od sigurnosnih propusta i grešaka u autorizacionoj logici.

Na primjer, za pristupnu tačku */api/students* sa GET HTTP metodom, aplikacija traži privilegiju STUDENTS_READ u tokenu korisnika. Izvorni kod pristupne tačke je prikazan na slici 30.

```
@PreAuthorize("hasAuthority('STUDENTS_READ')")
public ResponseEntity<Students> getStudentsById(@PathVariable Integer id) {
    Students students = studentsService.findById(id);
    if (students != null) {
        return ResponseEntity.ok().body(students);
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

Slika 30: REST API pristupna tačka za dohvatanje liste studenata

Autorizacija na strani klijenta

Na nivou klijentske aplikacije, autorizacija je dosljedno mapirana prema privilegijama korisnika. Šta će korisnik da vidi na ekranu zavisi od njegovih dozvola i pravila:

- Navigacija - Po prijavi, korisnički meni prikazuje isključivo one resurse (tabele) za koje korisnik ima pravo čitanja. Slika 31 daje primjer u kom je korisnik ograničen samo na tabelu "STUDENTS".

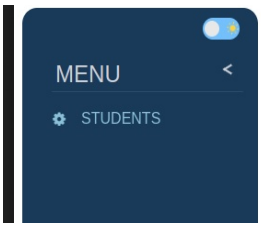
- Dugmad i akcije - Za svaku tabelarnu ili detaljnu komponentu, vidljivost dugmadi za kreiranje, izmjenu ili brisanje objekta zavisi od privilegija iz korisničkog tokena:

    - Ako korisnik nema pravo na CREATE akciju za dati entitet, dugme „New” nije vidljivo.

    - Ako nema pravo na UPDATE akciju, nema ni opciju za editovanje objekata.

    - Ako nema pravo na DELETE, dugme za brisanje nije dostupno.

Sva ova pravila implementirana su pozadinskom logikom i reflektuju aktuelne dozvole iz tokena.



Slika 31: Dio korisničkog interfejsa kada je prijavljen korisnik koji ima privilegiju čitanja zamo za entitet student

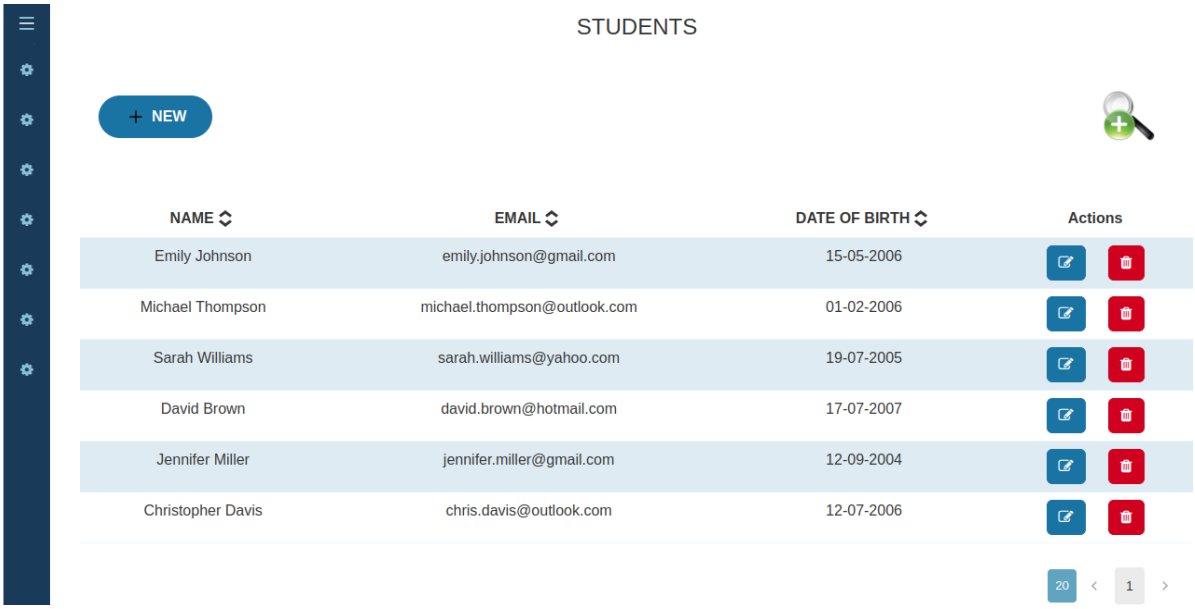
Iako klijent sakriva elemente interfejsa korisnicima, serverska aplikacija sprovodi nezavisnu provjeru. Nijedan zahtjev ne može biti odobren ako korisnik nema odgovarajuću privilegiju, čak i u slučaju pokušaja slanja zahtjeva direktno prema API-ju (npr. preko Postman alata).

5.4.5. CRUD operacije nad podacima

Centralni dio svake poslovne aplikacije jeste mogućnost upravljanja podacima, što u osnovi obuhvata CRUD operacije: čitanje (eng. *Read/Retrieve*), kreiranje (eng. *Create*), ažuriranje (eng. *Update*) i brisanje (eng. *Delete*) zapisa. U generisanoj aplikaciji, ove operacije su automatizovane za svaki entitet u modelu podataka, uz potpuno očuvanje sigurnosti i granulisane autorizacije. Svaka operacija je izložena kroz odgovarajuće REST pristupne tačke na serverskoj strani i jasno strukturisane vizuelne komponente na klijentskoj strani.

Tabelarni prikaz podataka

Za pregled većeg broja objekata, aplikacija koristi tabelarni prikaz podataka, koji omogućava jasno sagledavanje, jednostavno pretraživanje i selekciju željenih zapisa (slika 32). Kolone, u prikazanoj tabeli, odgovaraju onim kolonama koje je korisnik prethodno označio kao vidljive tokom faze konfiguracije.



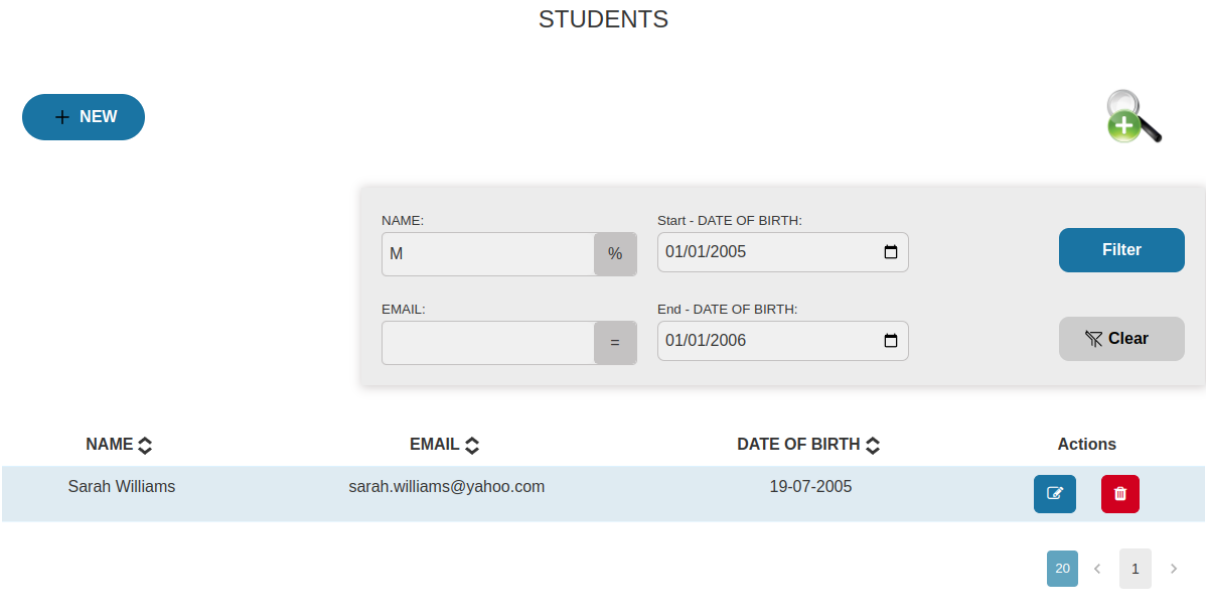
Slika 32: Dio korisničkog interfejsa sa tabelarnim prikazom studenata

Da bi se omogućilo korisnicima da efikasno upravljaju velikim skupovima podataka bez opterećivanja preglednosti ili performansi aplikacije, implementirana je paginacija. Na dnu tabele korisnicima je omogućeno biranje broja redova u tabeli, kao i navigacija kroz stranice.

Pristupna tačka */api/data/students/filter* prima parametre za straničenje (veličina stranice i broj stranice), filtriranje i sortiranje. Na osnovu tih parametara, vraća podatke za traženu stranicu i broj zapisa po stranici, omogućavajući optimizovan rad i sa velikim brojem zapisa.

Filtriranje podataka

Još jedan od važnih zahtjeva za ovakve aplikacije jeste fleksibilno filtriranje podataka. U generisanoj aplikaciji, korisnicima su dostupne višestruke filter opcije koje omogućavaju vrlo precizno i kombinovano filtriranje po više polja istovremeno. Na primjer, postoji mogućnost da korisnik pretraži sve studente čije ime sadrži određeno slovo, a datum rođenja je u zadatom opsegu (slika 33).



Slika 33: Apliciranje filtera nad entitetom Students

Filteri generisanih aplikacija su u potpunosti tipizirani. Tip filtera i njegove mogućnosti zavise od tipa podatka kolone:

- String polja - Omogućeno je pretraživanje potpuno istih ili sličnih vrijednosti. Operator „=” označava potpuno poklapanje, dok operator „%” traži djelimično poklapanje.
- Numerička polja - Dozvoljeno je filtriranje prema vrijednosti i operacijama:
  - jednako (=),
  - veće (>),

- veće ili jednako (>=),
- manje (=),
- manje ili jednako (<=),
- interval ([]).

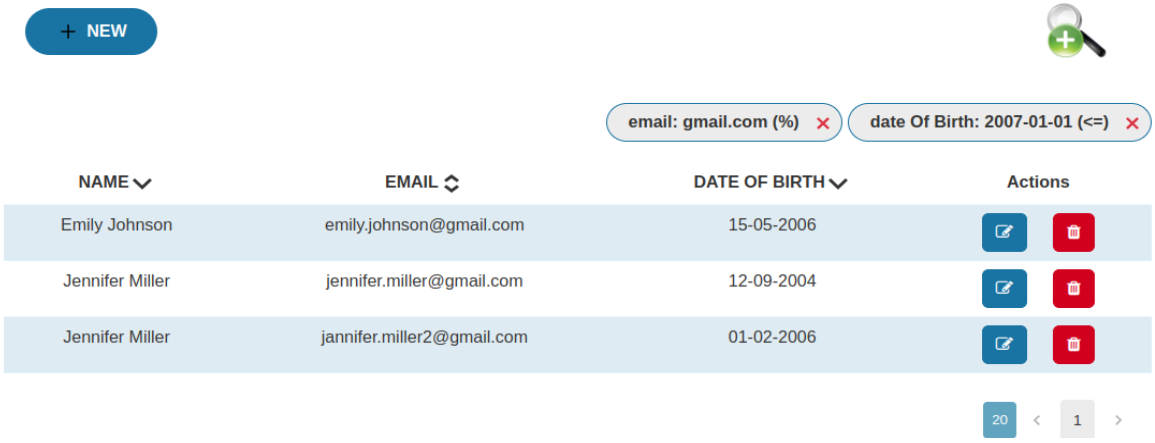
- Datumi - Za ove tipove polja omogućena su filtriranja po opsegu (od-do), uz korištenje grafičkih elemenata interfejsa. Korisnik može izabrati početni datum te filtrirati polja koja su mlađa od njega. Isto tako može izabrati samo krajnji datum i dobiti polja koja imaju stariji datum. Ukoliko su popunjeni i početni i krajnji datum, biraju se polja u tom intervalu (prikazano na slici 30).
- Enumeracije - Izbor je moguć samo po tačnoj vrijednosti koja se bira iz predefinisane padajuće liste.
- Boolean - Forma iscrtava checkbox polja za boolean tipove, te se tako mogu filtrirati tačne i netačne vrijednosti.

Korisničko iskustvo je tako dizajnirano da elementi forme i logički operatori prate tipove polja.

Na serverskoj strani, sve kombinacije filtera transformišu se u dinamički formiran SQL upit gdje se precizno obrađuju svi tipovi i operacije. Aplikaciona logika automatski pravi odgovarajuće Predicate objekte za svaku stavku filtera, omogućavajući AND logičku kompoziciju. Sveukupno, model omogućava napredne pretrage, izvještaje i analizu, bez potrebe za dodatnim kodiranjem na strani korisnika.

Sortiranje podataka

Pored naprednog filtriranja, korisnicima je omogućeno i sortiranje tabele po jednom ili više polja. Izbor kolona za sortiranje obavlja se klikom na zaglavlje kolone (ikona strelice). Višestrukim klikovima se mijenja pravac - prvi klik sortira rastuće, drugi opadajuće i treći klik poništava sortiranje po toj koloni. Zajedno sa filterima, ka serveru se šalje lista kolona i pravaca sortiranja, gdje serverska aplikacija generiše Sort.Order[] objekat za JPA query.



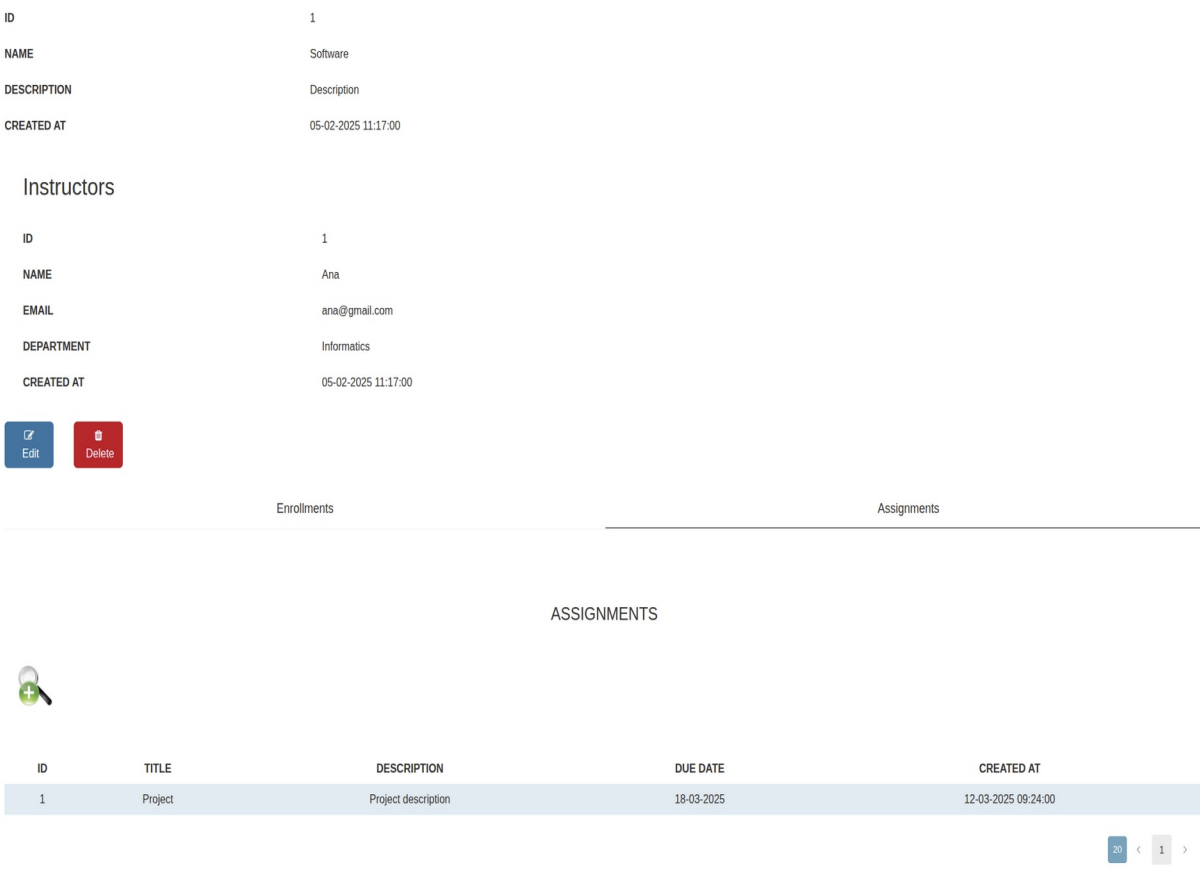
Slika 34: Prikaz sortiranja studenata po imenu i po datumu rođenja

Sortiranje po više kolona funkcioniše po prioritetu: sistem sortira najprije prema prvoj izabranoj koloni, zatim, ako postoje isti rezultati, prema sljedećoj u listi itd. Primjer takvog sortiranja je prikazan na slici 34. Dva studenta imaju isto ime pa je potrebno uzeti drugu karakteristiku po kojoj će se sortirati. To je u ovom primjeru datum rođenja.

Kombinovanjem višestrukih filtera i sortiranja, korisnički prikaz postaje dovoljno dobar alat za rad sa kompleksnim podacima i analitikom, što ga čini vrijednim za krajnje korisnike.

Detaljan prikaz pojedinačnog objekta

Pored tabelarnog, postoji mogućnost detaljnog pregleda pojedinačnog objekta, odnosno svih njegovih atributa i povezanih informacija. U generisanim aplikacijama, komponente za ovaj prikaz su automatski generisane i u potpunosti usklađene sa unaprijed definisanim pravilima vidljivosti. Detaljan prikaz objekta dostupan je klikom na red u tabelarnom prikazu entiteta. Svakom objektu se pristupa putem jedinstvene rute u aplikaciji, u formatu /entityName/{id}. Posjetom stranice, korisniku se prikazuje ekran na kojem su u vrhu prikazane sve kolone koje su, tokom faze konfiguracije, označene kao vidljive za detaljan pregled. Odmah nakon osnovnih informacija o izabranom entitetu, prikazuju se svi objekti koji su referencirani stranim ključevima.



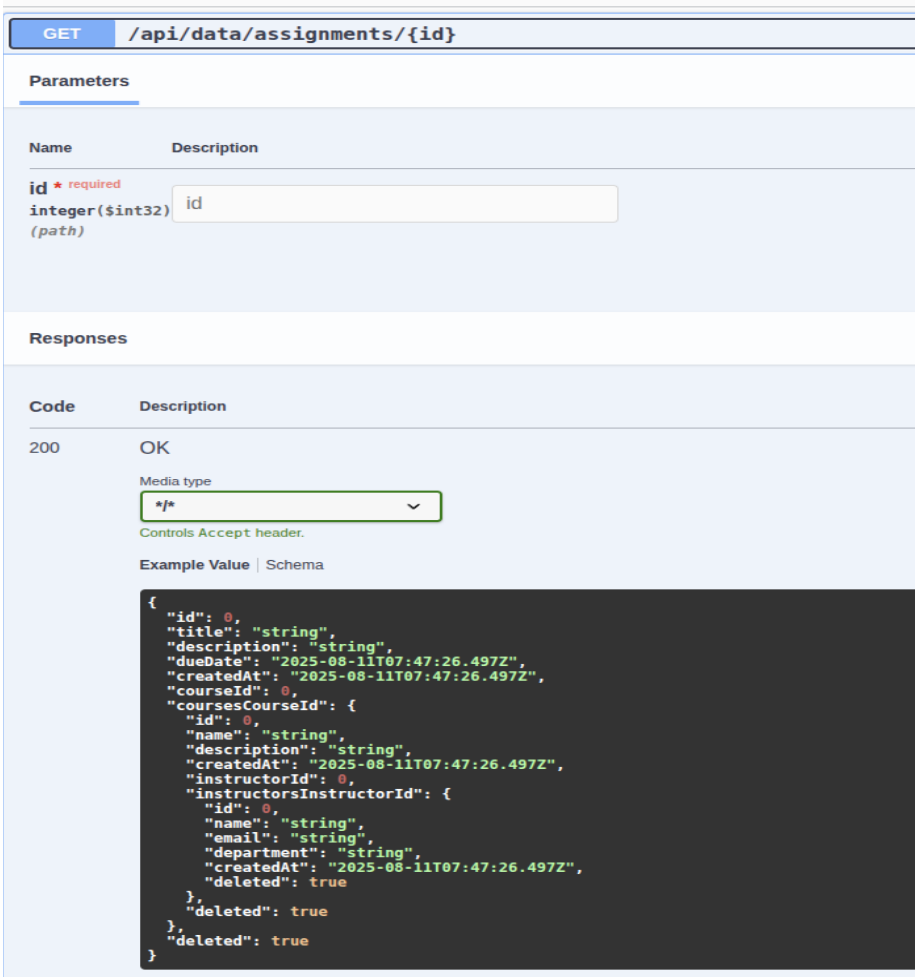
Slika 35: Detaljan prikaza jednog kursa sa instruktorom i listom zadataka

Na primjer, u prikazu kursa, automatski su prikazani i svi detalji vezanog instruktora, uključujući njegovo ime, e-mail, odjel i datum kreiranja, kao što je ilustrovano na slici 35. Pored direktno vezanih objekata, generisana aplikacija omogućava tabelarni prikaz svih

entiteta koji referenciraju izabrani objekat. U slučaju kursa, ispod glavnih podataka nalaze se tabovi u kojima je moguće pregledati sve zadatke („Assignments”) ili upisane studente („Enrollments”), koji dati kurs imaju za strani ključ. Svaki od ovih međuzavisnih objekata prikazan je u zasebnoj tabeli, sa paginacijom i opcijama filtriranja/sortiranja.

Ovako organizovan interfejs omogućava korisnicima da na jednom mjestu dobiju potpun pregled svih podataka i veza koje su povezane sa izabranim entitetom, čime se poboljšava produktivnost u radu, bez potrebe za ručnom navigacijom ili dodatnim pretragama.

Sa strane servera, detaljan prikaz pojedinačnog objekta omogućen je putem specifične REST pristupne tačke. Primjer pristupne tačke za entitet „Assignments“ je GET metoda, sa putanjom */api/data/assignments/{id}*, koja prima id objekta kao parametar putanje, provjerava da li korisnik ima privilegiju „ASSIGNMENTS_READ” i vraća potpunu reprezentaciju objekta, uključujući sve relevantne atribute kao i ugnježđene objekte koji su strani ključevi. Povratna vrijednost je JSON objekat, što omogućava klijentu da prikaže sve potrebne podatke. Na slici 36 je prikazan dio Swagger interfejsa za opisanu pristupnu tačku.



Slika 36: Dio Swagger interfejsa za GET /api/data/assignments/{id}

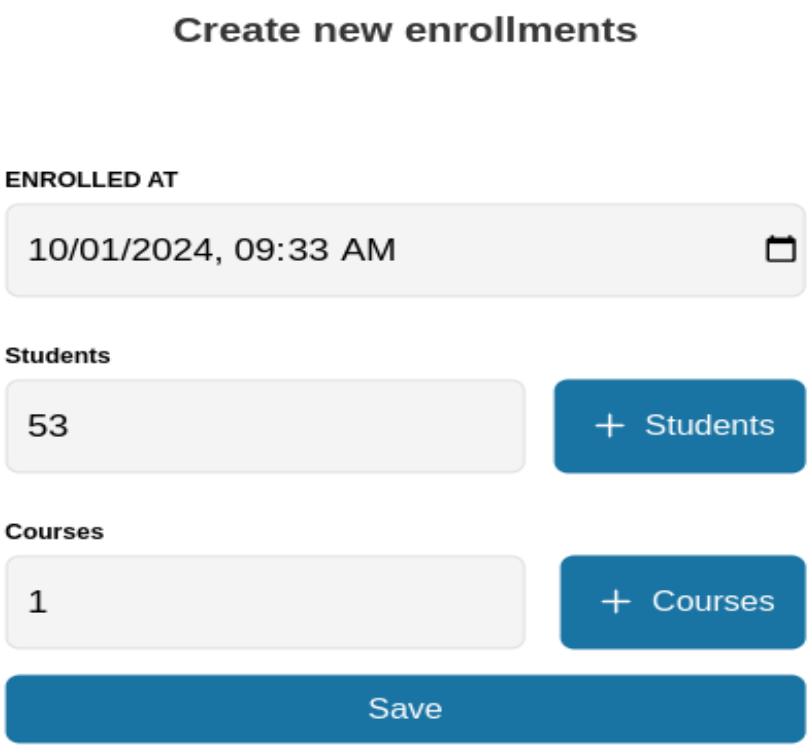
Kreiranje novih objekata

U generisanoj aplikaciji, funkcionalnost kreiranja je potpuno prilagođena potrebama sigurnosti, preciznosti i efikasnosti. Mogućnost kreiranja novih objekata dostupna je korisnicima kojima su dodijeljene neophodne privilegije (npr. STUDENTS_CREATE, COURSES_CREATE). U sklopu tabelarnog prikaza, nalazi se dugme „New”, koje otvara modal sa formom za unos novog objekta.

Forma za kreiranje objekata se generiše dinamički prema šemi entiteta i meta-podacima iz baze. Oblik input polja biran je prema tipu podatka i ograničenjima definisanim u bazi:

- Tekstualna input polja odgovaraju tekstualnim tipovima u bazi (npr. VARCHAR, CHAR, TEXT).
- Numerička input polja odgovaraju numeričkim tipovima baze (npr. INT, DECIMAL, BIGINT, FLOAT).
- Enum se reprezentuje kao padajuća lista sa unaprijed definisanim opcijama.
- Boolean vrijednosti su prikazane u obliku checkbox polja.
- Datum i vrijeme se prikazuju kao date picker, time picker ili kombinovani kalendar.
- Strani ključevi se biraju iz tabelarnih prikaza postojećih podataka.

Na slici 37 je prikazana forma za kreiranje upisa na kurs. Za jedan novi upis potrebno je odabrati datum i vrijeme, kao i studenta i kurs za koji je upis vezan.



Slika 37: Snimak dijela ekrana sa formom za kreiranje novog upisa studenta na kurs

Da bi podaci bili konzistentni sa šemom u bazi, generisane aplikacije imaju validaciju forme sa strane forntenda:

- Polja sa atributom NOT NULL su obavezna.
- Za string polja (VARCHAR) postavlja se ograničenje na maksimalan broj karaktera.
- Numerička polja dozvoljavaju samo brojeve.
- Vrijednost enumeracije je moguće birati samo iz unaprijed ponuđenih opcija.

Kada su svi podaci ispravno uneseni, klikom na „Save" šalje se POST zahtjev na REST pristupnu tačku. Serverska aplikacija prima zahtjev te provjerava privilegiju sa *@PreAuthorize* anotacijom. Ako korisnik ima privilegiju i ako su podaci validni, novi objekat se kreira, a korisnik dobija obavještenje o uspješnom kreiranju (vizuelni toast modal). Tabela (ili detaljan prikaz) automatski se ažurira novim podacima bez potrebe za ponovnim učitavanjem stranice. Eventualne greške se prikazuju kroz jasne poruke o uzroku.

Sloj nadzora osigurava praćenje novih unosa te se tako na svako kreiranje objekta automatski zapisuju podaci o korisniku (createdBy) i vremenu (createdAt).

Izmjena postojećih objekata

Upravljanje već postojećim podacima podrazumijeva mogućnost njihovog ažuriranja, odnosno izmjene. Pristup formi za izmjenu objekta dostupan je samo onim korisnicima koji posjeduju odgovarajuću privilegiju (npr. ASSIGNMENTS_UPDATE, COURSES_UPDATE). Mogućnost izmjene je intuitivna i proces se pokreće preko „Edit" dugmeta na više mjesta:

- U tabelarnom prikazu - Dugme za izmjenu nalazi se u svakom redu tabele. Klikom na dugme otvara se forma sa već popunjenim podacima trenutnog objekta.
- Na detaljnom prikazu objekta - Dugme za izmjenu pozicionirano je ispod osnovnih informacija o objektu. Klikom na njega korisnik ulazi u režim izmjene.

Interaktivna forma za izmjenu je ista kao i forma za kreiranje, samo sa popunjenim poljima. Forma sadrži sva polja koja su dozvoljena za izmjenu. Polja koja nisu izmjenjiva, ili se računaju automatski (npr. id, audit polja kao što su createDate, createdBy i slično), prikazuju se kao polja koja su dozvoljena samo za čitanje. Validacija podataka je podržana isto kao i kod kreiranja objekata.

Sa strane servera, pristupna tačka za ažuriranje entiteta je PUT metoda na putanji */api/data/{Entity}/{id}*. Pristupanje ovoj metodi zahtijeva validan JWT token i korisničku privilegiju za izmjenu. Ako je validacija zahtjeva uspješna i korisnik ima pravo, server ažurira objekat, upisuje audit podatke (updatedAt, updatedBy) i vraća novi, ažurirani model objekta. U slučaju neautorizovanog pokušaja izmjene, server vraća odgovarajuću grešku (403 Forbidden).

Kada poruka o uspješnosti akcije dođe na stranu klijenta, korisnik biva obaviješten vizuelnom toast notifikacijom o uspješnoj izmjeni. Podaci na tabelarnom ili detaljnom prikazu objekta se odmah ažuriraju, odražavajući sve promjene bez potrebe za dodatnim

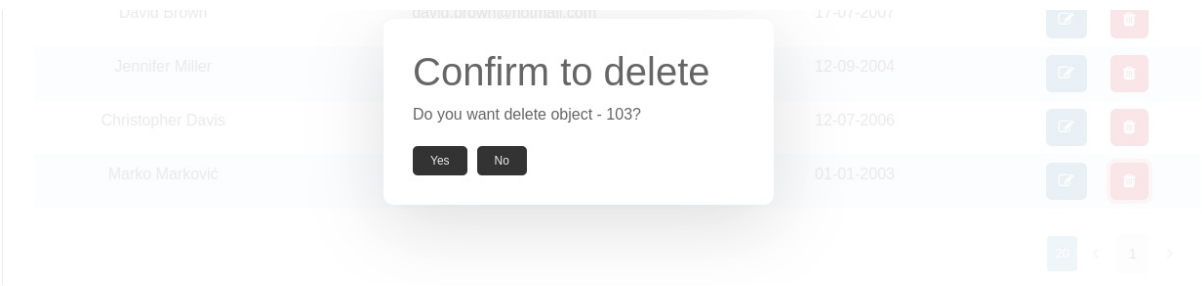
osvježavanjem stranice. Ukoliko dođe do greške, korisniku se prikazuje razlog i ostavlja mogućnost da ispravi unos.

Brisanje objekata

Brisanje podataka je kritična operacija u svakom informacionom sistemu, zahtijeva posebnu pažnju, upravo zbog mogućih posljedica gubitka informacija. U generisanoj aplikaciji, ovaj proces je pažljivo kontrolisan i višeslojno zaštićen.

Brisanje objekata dostupno je samo korisnicima sa eksplicitnom DELETE privilegijom. Proces je moguće inicirati i sa tabelarnog i sa prikaza pojedinačnog objekta. Klikom na dugme „Delete", korisniku se prikazuje modal za potvrdu (eng. *Confirm modal*), gdje se eksplicitno pita da li zaista želi trajno obrisati izabrani objekat, uz navođenje njegovog primarnog ključa (slika 38).

Na strani servera, brisanje entiteta se izvršava putem DELETE pristupne tačke na putanji */api/data/Entity/{id}*. Serverska metoda prepoznaje ID objekta iz parametra putanje. Prije samog brisanja, aplikacija provjerava da li korisnik ima odgovarajuću privilegiju. Ako korisnik zadovoljava uslove, sistem koristi „soft delete" pristup, čime se fizičko brisanje izbjegava zbog mogućnosti nadzora ili vraćanja podataka. U slučaju uspjeha vraća se HTTP odgovor 204 (No Content), a UI automatski ažurira pogled. Ako je ID neispravan ili objekat ne postoji, korisniku se prikazuje poruka o grešci.



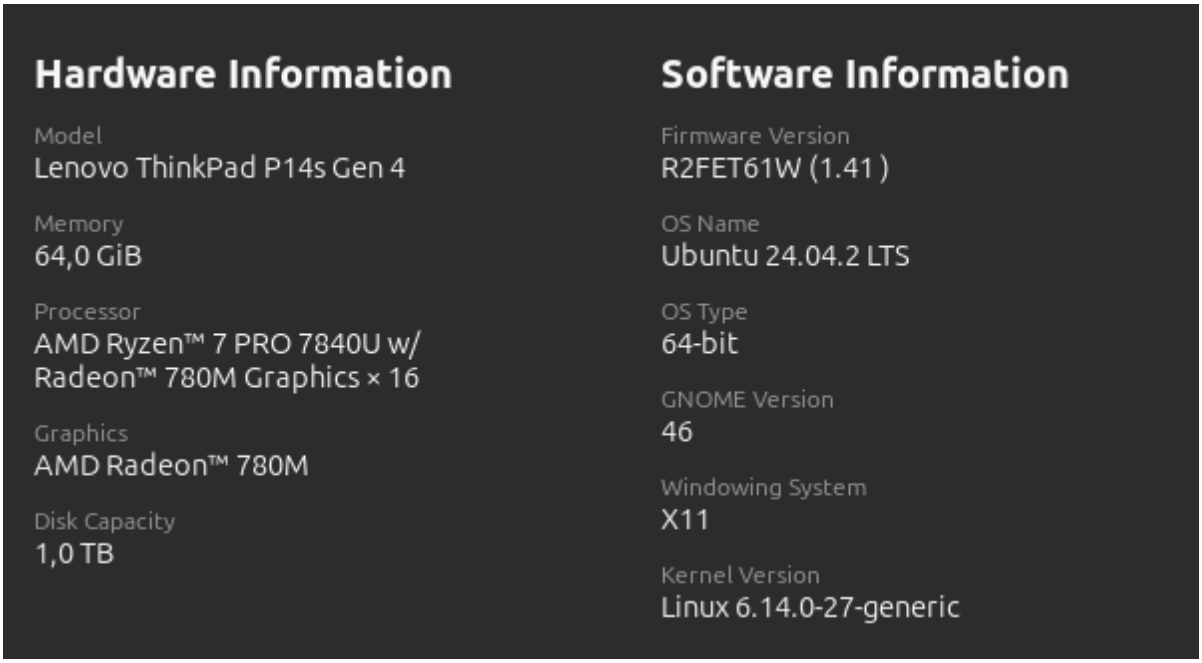
Slika 38: Modal za potvrdu brisanja objekta sa primarnim ključem 103

5.4.6. Dostupnost implementiranog rješenja

Prethodno opisani generator dostupan je kao otvoreni GitHub repozitorijum na linku: https://github.com/DanijelaVukosav/code-crafter.

6. EVALUACIJA IMPLEMENTIRANOG RJEŠENJA

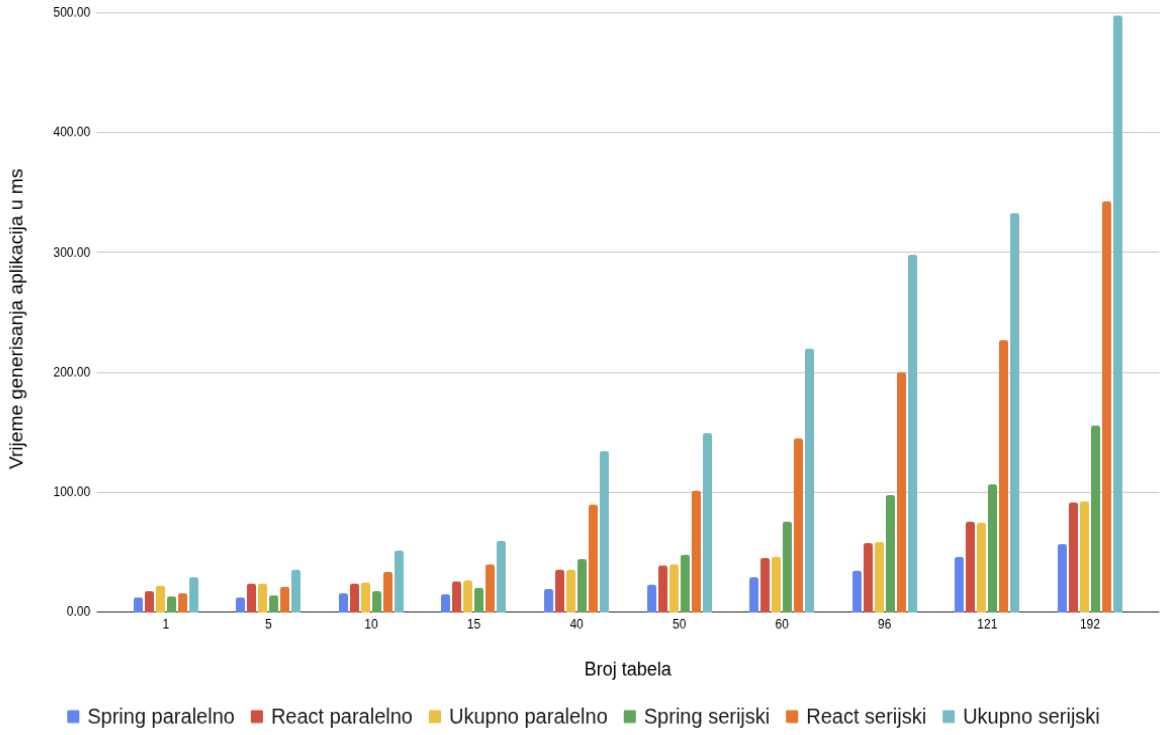
U sklopu eksperimenta vršeno je mjerenje vremena generisanja Spring i React aplikacija za različit broj entiteta. Specifikacije sistema korištenog za izvođenje mjerenja date su na slici 39.



Slika 39: Karakteristike sistema na kom je vršena evaluacija implementiranog rješenja

Generisanje je vršeno serijski i paralelno. Serijsko generisanje podrazumijeva sekvencijalno kreiranje neophodnih fajlova za svaki entitet pojedinačno, pri čemu se prvo generiše jedna, a zatim druga aplikacija. S obzirom na to da se za svaki entitet generišu isti fajlovi, može se izvršiti paralelizacija ovog procesa. Takođe, zahvaljujući dobro definisanom API-ju, moguće je izvršiti paralelno generisanje samih aplikacija. Dakle, paralelno generisanje obuhvata istovremeno generisanje aplikacija, kao i entiteta u okviru istih.

Dobijeni rezultati mjerenja su reda veličine milisekundi, što sugeriše visoku efikasnost i dobre performanse generatora. Na slici 40 su prikazana su pojedinačna vremena generisanja aplikacija, kao i sistema u cjelini. U slučaju manjeg broja entiteta ne postoji značajna razlika u performansama sekvencijalnog i paralelnog načina generisanja. Sa porastom broja entiteta sve je izraženija efikasnost koju donosi paralelno generisanje koda. Lako je uočiti da sistem može generisati složenije strukture, čak i one sa 200 entiteta, za manje od 100 ms. Takođe, rezultati mjerenja pokazuju da je ukupno vrijeme sekvencijalnog generisanja čitavog sistema, očekivano, jednako sumi vremena potrebnih za generisanje pojedinačnih aplikacija. Sa druge strane, vrijeme potrebno za paralelno generisanje sistema odgovara vremenu generisanja React aplikacije jer njeno generisanje traje duže od generisanja Spring aplikacije. Razlog za ovo je činjenica da se za React aplikaciju generiše veći broj fajlova u odnosu na Spring aplikaciju.



Slika 40: Analiza vremena generisanja sistema u zavisnosti od broja entiteta

Kada se gleda iz ugla efektivnosti, trenutna verzija rješenja generiše isključivo CRUD funkcionalnosti zasnovane na akcijama nad entitetima, bez generisanja poslovne logike. Poslovnu logiku nije moguće zaključiti na osnovu šeme baze podatka, te se taj dio sistema ostavlja korisniku da sam uradi. Shodno tome, kompletnost i tačnost odnosi se na CRUD opseg, u kojem je pokrivenost koda visoka, jer se serverski i klijentski artefakti generišu u potpunosti. Osnovne operacije su funkcionalne i pri prvom pokretanju aplikacije, dok je za specifičnu domensku logiku potrebna ručna intervencija.

Na kraju, važno je napomenuti da je u sklopu eksperimenta izvršena statička analiza generisanog koda pomoću Qodana alata. Tokom analize, alat je pregledao kod sa ciljem identifikovanja potencijalnih grešaka, sigurnosnih propusta i drugih nepravilnosti. Rezultat analize pokazuje da nema prisutnih grešaka niti upozorenja, čime je potvrđeno da generisani kod ispunjava standarde kvaliteta i sigurnosti.

6.1. Komparativna analiza implementiranog generatora i postojećih sistema za generisanje aplikacija

Poređenjem implementiranog rješenja sa široko korištenim generatorima aplikacija, uočavaju se razlike u pristupu, nivou automatizacije i pristupačnosti krajnjem korisniku. Klasični alati kao što su Spring Roo, CodeSmith Generator, OpenAPI Generator i JHipster obično zahtijevaju napredno tehničko znanje, rad kroz komandnu liniju ili upravljanje specijalizovanim meta-modelima, što može predstavljati prepreku za korisnike sa manje iskustva. Većina postojećih rješenja oslanja se na strogo ograničen rad sa šablonima, manuelno prilagođavanje ili formalizovane specifikacije, što iziskuje dodatno vrijeme za savladavanje alata i održavanje integracija.

Implementirani generator, nasuprot tome, naglasak stavlja na jednostavnost ulaznih podataka i intuitivno korisničko iskustvo, omogućujući rad sa standardnim DDL skriptama ili strogo definisanim JSON modelima. Za razliku od drugih sistema, nije potrebno prethodno poznavanje specijalizovanih jezika ili okruženja. Kompleksni sistemi mogu biti generisani u izuzetno kratkom vremenskom periodu, bez gubitka na kvalitetu i pouzdanosti koda.

Dodatno, implementirani sistem uvodi rješenja kao što su automatski nadzor, kontrola vidljivosti polja, modularna autorizacija po entitetu i tipu akcije, kao i napredno podešavanje filtriranja i sortiranja. Ovo su funkcionalnosti koje konkurentskim generatorima često nedostaju ili zahtijevaju dodatnu ručnu intervenciju. Pristup generisanju aplikacija bez očuvanja generisanog koda na serverskoj strani dodatno podiže nivo sigurnosti i privatnosti korisnika.

Kao rezultat, implementirani generator odlikuje se većim stepenom automatizacije, kraćim vremenom obuke korisnika, većom fleksibilnošću u konfiguraciji i znatno širim spektrom primjenjivosti, naročito u okruženjima gdje je brzina razvoja, sigurnost i lakoća upotrebe od izuzetnog značaja.

U tabeli 7 je prikazano poređenje implementiranog sistema sa drugim generatorima po određenim i često bitnim odlikama.

Tabela 7: Poređenje implementirane aplikacije sa drugim generatorima

Karakteristika	**Spring Roo**	**CodeSmith**	**OpenAPI Generator**	**JHipster**	**CodeCrafter**
Generisanje serverske aplikacije	Da	Da	Da	Da	Da
Generisanje klijentske aplikacije	Ograničeno	Ne	Ne	Da	Da
Generisanje na osnovu OpenApi specifikacije	Ne	Ne	Da	Ograničeno	Ne
Generisanje testova	Da	Ograničeno	Ograničeno	Da	Ne
Bezbjednost i autentifikacija	Da	Ne	Ne	Da	Da
Autorizacija korisnika	Ne	Ne	Ne	Djelimična	Da
Automatski nadzor	Ne	Ne	Ne	Ne	Da
Konfiguracija interfejsa, filtriranja i sortiranja	Ne	Ne	Ne	Ne	Da
Internacionalizacija	Ne	Ograničeno	Ograničeno	Da	Ne
CI/CD integracija	Ne	Ograničeno	Da	Da	Ne
Upotreba u poslovnim sistemima	Ograničena	Da	Da	Da	Da
Ekstenzibilnost	Ograničena	Da	Ograničena	Da	Da

6.2. Ograničenja sistema

Uprkos do sada pokazanoj fleksibilnosti, trenutna verzija generatora ima i ograničenja:

- Podrška za primarne ključeve ograničena je na jednostavne ključeve.
- Napredni kompleksni tipovi podataka nisu uzeti u obzir.
- Tehnološka osnova je fokusirana na Spring Boot i React.
- Ulazi su standardizovani, ali se opet od korisnika očekuje bar osnovno poznavanje rada baza podataka.

Ova ograničenja su uvedena zbog očuvanja stabilnosti sistema i jednostavnog održavanja postojećih šablona, pri čemu je arhitektura unaprijed dizajnirana tako da omogućava jednostavno dodavanje novih šablona i proširenja u skladu sa budućim potrebama.

7. ZAKLJUČAK

U radu je predstavljeno, teorijski utemeljeno i praktično verifikovano rješenje za automatsko generisanje veb aplikacija na osnovu šeme baze podataka, sa fokusom na savremene tehnologije: Spring Boot i React. Polazeći od motivacije da se ubrza inicijalna faza razvoja i istovremeno podigne konzistentnost i sigurnost izlaznog koda, implementiran je generator koji iz standardizovanih ulaza (SQL DDL i strogo definisan JSON model) deterministički proizvodi funkcionalne aplikacije sa ugrađenim mehanizmima autentifikacije, autorizacije, nadzora, paginacije, filtriranja i sortiranja. Metodološki, rad kombinuje sistematski pregled postojećih rješenja, modularni dizajn i implementaciju, te evaluaciju kroz funkcionalna testiranja, mjerenja performansi i komparativnu analizu.

Ključni doprinos ogleda se u ideji i rješenju kojim se formalizuje ulaz i standardizuje proces generisanja aplikacija, kao i u integraciji sigurnosnih i administrativnih politika u inicijalnom kodu. Time se repetitivni i rizični segmenti razvoja čine determinističkim i provjerljivim, što neposredno smanjuje vjerovatnoću grešaka, skraćuje vrijeme isporuke i podiže ukupni kvalitet. Posebnu vrijednost ima interaktivna konfiguracija koja korisniku omogućava da bez izmjene koda definiše audit polja, vidljivost kolona, te polja za filtriranje i sortiranje, čime se postiže balans između automatizacije i prilagodljivosti.

Rezultati evaluacije ukazuju na visoku efikasnost sistema: vrijeme generisanja ostaje u rasponu milisekundi i za kompleksnije šeme sa većim brojem entiteta, a statička analiza pronalazi potencijalne propuste u generisanom kodu. Komparativno posmatrano, u odnosu na često korištene alate kao što su JHipster, Spring Roo, OpenAPI Generator i drugi, primijenjeni pristup smanjuje ulaznu barijeru, isporučuje uniformnu osnovu i uvodi korisničku konfigurabilnost korisničkog interfejsa i podataka. Time se jasnije definiše kao rješenje pogodno za brzu prototipizaciju, početne domenske implementacije i obrazovne scenarije, ali i za modernizaciju i izlaganje postojećih podataka savremenim arhitekturama.

Ograničenja trenutne verzije proističu iz određenih projektantskih izbora u korist stabilnosti i determinističnosti: podržani su jednostavni primarni ključevi, napredni tipovi podataka (npr. JSON, BLOB, geotipovi) nisu automatski obrađeni i generisani, a tehnološki aspekt je fokusiran na Spring Boot i React. Format ulaza je ograničen na DDL i na JSON fajl koji zadovoljava striktno definisanu JSON šemu. Ove granice ne umanjuju vrijednost rješenja u tipičnim obrađenim scenarijima upotrebe.

Ovaj alat je namijenjen upotrebi od strane, prije svega, tri primarne grupe korisnika: profesionalnih timova koji žele skratiti inicijalnu fazu razvoja i postaviti standardizovanu osnovu projekta, administratora i IT stručnjaka kojima je potrebna brza modernizacija i izlaganje podataka savremenim API-jem, te studenata i početnika za koje je alat može biti odlična polazna tačka jer iz modela baze podataka vodi do potpuno funkcionalnog koda. Praktično posmatrano, alat je koristan u ranim fazama razvoja kada se želi što brže isporučiti radna osnova sa kritičnim funkcionalnostima, kao i u migracijama gdje postojeće DDL skripte treba transformisati u moderni sistem. Vrijednost je izražena i u edukaciji, jer studenti mogu neposredno vidjeti vezu između modela i aplikacije. Pored toga, uniformnost izlaza smanjuje rizik od grešaka i sigurnosnih propusta. Prednost modularnog dizajna je u tome što omogućava selektivno generisanje slojeva i njihovo uklapanje u postojeće kodove.

Unapređenjem ovako postavljene osnove, generator može prerasti u kompleksan sistem sa velikim brojem funkcionalnosti. Jedan značajan pravac unapređenja može biti

uvođenje automatizovanog testiranja koji se generiše zajedno sa kodom. Moguće je generisati jedinične i integracione testove za REST API pristupne tačke sa svim sigurnosnim aspektima, dok je za klijentsku aplikaciju moguće generisati jedinične i komponentne testove. Generisani testovi se mogu objediniti u CI/CD, tako da generator ne isporučuje samo kod, već i gotovu testnu infrastrukturu i skripte koje timovi odmah uključuju u svoj razvojni tok.

Uz testove, prijedlog unapređenja je i poboljšanje mehanizma prava kroz model uloga. Uloge bi bile kompozicije skupova dozvola (npr. STUDENTS_READ, COURSES_UPDATE), sa mogućnošću definisanja podrazumijevanih (Administrator, Editor, Viewer) i prilagođenih uloga na osnovu potreba organizacije. U administrativnom interfejsu, uloge bi se dodjeljivale korisnicima, uz pregled i obrazloženje privilegija. Kao naredni korak, sistem može postepeno podržati i elemente ABAC pristupa, gdje bi određene radnje zavisile od atributa subjekta, objekta i konteksta (npr. „instruktor može uređivati samo kurseve svog odjela" ili „korisnik može vidjeti samo zapise koje je kreirao"). Ovakva evolucija politike pristupa zadržava jednostavnost za većinu potreba, a otvara vrata sofisticiranijim scenarijima.

Distribucija alata i generisanih aplikacija kroz kontejnersku paradigmu je još jedan važan korak ka privatnosti, prenosivosti i jednostavnosti. Ideja da korisnik ne mora slati svoju šemu baze na mrežu može se riješiti korištenjem Docker slike generatora koja se pokreće lokalno, bez ikakvog slanja podataka van servera. U tom režimu, veb interfejs generatora radi na lokalnom računaru, a sav rad se odvija u kontejneru. Pored toga, ima smisla generisati i Docker fajlove i docker-compose primjere za generisane aplikacije: serversku i klijentsku aplikaciju kao samostalne slike te referentni servis baze. Ovakav pristup je kvalitetno rješenje za privatnost podataka, jer sve ostaje kod korisnika.

Automatizacija lokalizacije predstavlja još jedan značajan korak ka tome da generator bude spreman za realne, višeregionalne primjene, bez dodatnog ručnog rada. U praksi to znači da se već u fazi generisanja obrade osnovni resursi za višejezičnost, kako na serveru, tako i na klijentu. Na strani servera, generator može kreirati fajlove sa porukama i opisima validacija, te unaprijed postaviti mehanizam za izbor jezika putem HTTP zaglavlja ili korisničkog podešavanja profila. Na klijentu, inicijalno se pripremaju datoteke sa porukama i ključevima koji su izvedeni iz šeme (nazivi entiteta, kolona, grešaka i formi), a komponente se automatski parametrizuju bibliotekama koje rješavaju razlike među jezicima. Time se dobija dosljedna internacionalizacija korisničkog interfejsa i poruka o greškama, uz mogućnost da se prevodi kasnije dopunjavaju. Za naprednije potrebe, generator može da pripremi integraciju sa servisima za prevođenje. Rezultat je sistem koji je spreman za više tržišta, a lokalizacija postaje kontinuirani proces umjesto jednokratnog napora.

Sintezom prethodno navedenih prijedloga dobija se jasna slika kako generator evoluira iz alata za brzo podizanje projekata u platformu koja isporučuje i kvalitet i sigurnost i održivost. Prethodno opisani pravci čine logičan nastavak započete arhitekture - sistem ostaje modularan i proširiv, a svaka nova funkcionalnost dolazi kao dodatni sloj koji ne narušava tok akcija od šeme do koda.

8. LITERATURA

[1] R. Elmasri and S. Navathe, Fundamentals of database systems, Seventh edition. Boston Munich: Pearson, 2016.
[2] C. J. Date, An introduction to database systems, 8. ed. Boston, Mass.: Pearson, Addison-Wesley, 2004.
[3] A. Silberschatz, H. F. Korth, and S. Sudarshan, Database system concepts, 6th ed. New York: McGraw-Hill, 2011.
[4] E. F. Codd, "A relational model of data for large shared data banks," Commun. ACM, vol. 13, no. 6, pp. 377–387, Jun. 1970, doi: 10.1145/362384.362685.
[5] R. Elmasri and S. Navathe, Fundamentals of database systems, Seventh, Global edition. in Always learning. Boston: Pearson, 2017.
[6] D. Brđanin, S. Marić, Relacione baze podataka, Elektrotehnički fakultet Banja Luka 2012.
[7] "Overview of MySQL Storage Engine Architecture," Posjećeno: avgust, 2025. [Online]. Dostupno: https://dev.mysql.com/doc/refman/8.4/en/pluggable-storage-overview.html
[8] R. Ramakrishnan and J. Gehrke, Database management systems, 3. ed., Internat. ed. in McGraw-Hill higher education. Boston: McGraw-Hill, 2003.
[9] R. Johnson, Expert one-on-one J2EE design and development. Indianapolis, IN: Wrox, 2003.
[10] C. Walls, Spring Boot in Action. New York: Manning Publications Co. LLC, 2016.
[11] R. Malhotra, Rapid Java Persistence and Microservices: Persistence Made Easy Using Java EE8, JPA and Spring. Berkeley, CA: Apress L. P, 2019.
[12] C. Walls, Spring in action, Fifth edition. Shelter Island, NY: Manning Publications, 2019.
[13] R. Laddad, AspectJ in action: practical aspect-oriented programming. Greenwich, CT: Manning, 2003.
[14] "2. Introduction to the Spring Framework." Posjećeno: avgust, 2025. [Online]. Dostupno: https://docs.spring.io/spring-framework/docs/4.2.8.RELEASE/spring-framework-reference/html/overview.html
[15] G. L. Turnquist, Learning Spring Boot 3.0: Simplify the development of production-grade applications using Java and Spring, 1st ed. Birmingham: Packt Publishing Limited, 2022.
[16] R. Johnson, Ed., Professional Java development with the Spring Framework. Indianapolis, Ind: Wiley Pub, 2005.
[17] I. R. Forman and N. Forman, Java reflection in action. Greenwich, Conn: Manning, 2005.
[18] R. R. Karanam, Mastering Spring 5: an effective guide to build enterprise applications using Java Spring and Spring Boot framework, 2nd edition, 2nd ed. Erscheinungsort nicht ermittelbar: Packt Publishing, 2019.
[19] Avicsebooks, "Basic of Spring," Medium. Posjećeno: avgust, 2025. [Online]. Dostupno: https://medium.com/@avicsebooks/basic-of-spring-b99fb70456ad
[20] V. Rajput and E. Martin, "Difference Between BeanFactory and ApplicationContext." Posjećeno: avgust, 2025. [Online]. Dostupno: https://www.baeldung.com/spring-beanfactory-vs-applicationcontext
[21] "Inversion of Control Containers and the Dependency Injection pattern," martinfowler.com. Posjećeno: avgust, 2025. [Online]. Dostupno: https://martinfowler.com/articles/injection.html
[22] "Understanding the Spring Bean Life Cycle - BootcampToProd." Posjećeno: avgust, 2025. [Online]. Dostupno: https://bootcamptoprod.com/spring-bean-life-cycle-explained/

[23] “The Java Persistence API - A Simpler Programming Model for Entity Persistence.” Posjećeno: avgust, 2025. [Online]. Dostupno: https://www.oracle.com/technical-resources/articles/java/jpa.html
[24] C. Bauer, G. King, G. Gregory, and C. Bauer, Java persistence with Hibernate, Second edition. Shelter Island, NY: Manning, 2016.
[25] “Hibernate Architecture – Decodejava.com.” Posjećeno: avgust, 2025. [Online]. Dostupno: https://www.decodejava.com/hibernate-architecture.htm
[26] “2.2. Overview of Spring MVC Architecture — TERASOLUNA Global Framework Development Guideline 1.0.1.RELEASE documentation.” Posjećeno: avgust, 2025. [Online]. Dostupno: https://terasolunaorg.github.io/guideline/1.0.1.RELEASE/en/Overview/SpringMVCOverview.html
[27] D. Ferraiolo, D. R. Kuhn, and R. Chandramouli, Role-based access control. in Artech House computer security series. Boston London: Artech House, 2003.
[28] V. C. Hu, D. R. Kuhn, D. F. Ferraiolo, and J. Voas, “Attribute-Based Access Control,” Computer, vol. 48, no. 2, pp. 85–88, Feb. 2015, doi: 10.1109/MC.2015.33.
[29] D. Hardt, “The OAuth 2.0 Authorization Framework,” Internet Engineering Task Force, Request for Comments RFC 6749, Oct. 2012. Posjećeno: avgust, 2025. [Online]. Dostupno: https://datatracker.ietf.org/doc/rfc6749/
[30] bezkoder, “Spring Boot Token based Authentication with Spring Security & JWT,” BezKoder. Posjećeno: avgust, 2025. [Online]. Dostupno: https://www.bezkoder.com/spring-boot-jwt-authentication/
[31] M. Jones, J. Bradley, and N. Sakimura, “JSON Web Token (JWT),” RFC Editor, RFC7519, May 2015. doi: 10.17487/RFC7519.
[32] T. Bui, “JWT Authentication and Authorization with Spring Boot 3 and Spring Security 6,” Medium.Posjećeno: avgust, 2025. [Online]. Dostupno: https://medium.com/@truongbui95/jwt-authentication-and-authorization-with-spring-boot-3-and-spring-security-6-2f90f9337421
[33] T. Occhino and J. Walke, “Introduction to React.js,” Tech talk, Facebook Seattle, 2013. Posjećeno: avgust, 2025. [Online]. Dostupno: https://www.youtube.com/watch?v=XxVg_s8xAms.
[34] D. Abramov, “Presentational and Container Components,” Medium. Posjećeno: avgust, 2025. [Online]. Dostupno: https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0
[35] “Architecture | Hands on React.” Posjećeno: avgust, 2025. [Online]. Dostupno: https://handsonreact.com/docs/architecture
[36] “React.” Posjećeno: avgust, 2025. [Online]. Dostupno: https://react.dev/
[37] A. Freeman, Pro React 16. Berkeley, CA: Apress L. P, 2019.
[38] “Vue.js.” Posjećeno: avgust, 2025. [Online]. Dostupno: https://vuejs.org/
[39] B. Eisenman, Learning react native: building mobile mobile applications with JavaScript, First edition. Beijing Boston Farnham Sebastopol Tokyo: O’Reilly, 2015.
[40] A. Banks and E. Porcello, Learning React: modern patterns for developing React Apps, Second edition. Beijing Boston Farnham Sebastopol Tokyo: O’Reilly, 2020.
[41] M. D. L. Tosi, J. L. C. Izquierdo, and J. Cabot, “A Metascience Study of the Low-Code Scientific Field,” Jun. 30, 2025, arXiv: arXiv:2408.05975. doi: 10.48550/arXiv.2408.05975.
[42] “Spring Roo - Reference Documentation.” Posjećeno: avgust, 2025. [Online]. Dostupno: https://docs.spring.io/spring-roo/docs/current/reference/html/
[43] “Hello from OpenAPI Generator | OpenAPI Generator.” Posjećeno: avgust, 2025. [Online]. Dostupno: https://openapi-generator.tech/

[44] “CodeSmith Tools.” Posjećeno: avgust, 2025. [Online]. Dostupno: https://www.codesmithtools.com/
[45] “The web’s scaffolding tool for modern webapps | Yeoman.” Posjećeno: avgust, 2025. [Online]. Dostupno: https://yeoman.io/
[46] Jh. Team, “JHipster - Full Stack Platform for the Modern Developer! | JHipster.” Posjećeno: avgust, 2025. [Online]. Dostupno: https://www.jhipster.tech/
[47] “OpenAI Platform.” Posjećeno: avgust, 2025. [Online]. Dostupno: https://platform.openai.com
[48] “GitHub Copilot · Your AI pair programmer,” GitHub. Posjećeno: avgust, 2025. [Online]. Dostupno: https://github.com/features/copilot

Prilog 1.

Biografija autora

Lični podaci

Rođena 08.10.1999. godine u Nevesinju.

Obrazovanje

Završila Gimnaziju “Aleksa Šantić” u Nevesinju kao đak generacije.
Prvi ciklus studija upisala 2018. godine na Elektrotehničkom fakultetu Univerziteta u Banjoj Luci, studijski program Računarstvo i informatika, smjer Softversko inženjerstvo. Diplomirala je 2022. godine sa završnim radom prvog ciklusa pod nazivom “Sistem za generisanje React aplikacija na osnovu šeme baze podataka” i prosječnom ocjenom u toku studija od 9,00.
Drugi ciklus studija je upisala 2023. godine, na Elektrotehničkom fakultetu Univerziteta u Banjoj Luci, studijski program Računarstvo i informatika.

Radno iskustvo

Radno iskustvo je započela 2022. godine kao Front-end softverski inženjer u “Alea Partners” kompaniji, gdje je bila angažovana na razvoju i održavanju pet *betting* projekata. 2024. godine prelazi u “FatCat” kao Full-stack inženjer, a od 2025. godine radi za “IT House”, takođe, kao Full-stack inženjer. Primarne tehnologije u radu su JavaScript/Typescript razvojni okviri i biblioteke.

Прилог 2.

УНИВЕРЗИТЕТ У БАЊОЈ ЛУЦИ
ПОДАЦИ О АУТОРУ ОДБРАЊЕНОГ МАСТЕР/МАГИСТАРСКОГ РАДА

Име и презиме аутора мастер/магистарског рада: **Данијела Вукосав**

Датум, мјесто и држава рођења аутора: **08.10.1999, Невесиње, Босна и Херцеговина**

Назив завршеног факултета/Академије аутора и година дипломирања:
Електротехнички факултет Универзитета у Бањој Луци, 2022. године

Датум одбране завршног/дипломског рада аутора: **14.10.2022. године**

Наслов завршног/дипломског рада аутора:
Систем за генерисање React апликација на основу шеме базе података

Академско звање коју је аутор стекао одбраном завршног/дипломског рада:
Дипломирани инжењер електротехнике (240 ECTS)

Академско звање које је аутор стекао одбраном мастер/магистарског рада:
Мастер електротехнике – 300 ECTS – Рачунарство и информатика

Назив факултета/Академије на коме је мастер/магистарски рад одбрањен:
Електротехнички факултет Универзитета у Бањој Луци

Наслов мастер/магистарског рада и датум одбране:
Генерисање веб апликација на основу шеме базе података, 30. 9. 2025.

Научна област мастер/магистарског рада према CERIF шифрарнику: **T 120**

Имена ментора и чланова комисије за одбрану мастер/магистарског рада:
проф. др Зоран Ђурић, предсједник
доц. др Михајло Савић, ментор
проф. др Дражен Брђанин, члан

У Бањој Луци, дана 11.09.2025. год.

Декан

Изјава 1

ИЗЈАВА О АУТОРСТВУ

Изјављујем да је
мастер/магистарски рад

Наслов рада **Генерисање веб апликација на основу шеме базе података**

Наслов рада на енглеском језику: **Generation of web application based on a database scheme**

- ☒ резултат сопственог истраживачког рада,
- ☒ да мастер/магистарски рад, у цјелини или у дијеловима, није био предложен за добијање било које дипломе према студијским програмима других високошколских установа,
- ☒ да су резултати коректно наведени и
- ☒ да нисам кршио/ла ауторска права и користио интелектуалну својину других лица.

У Бањој Луци 11.09.2025. год

Потпис кандидата

Јанкула Вукосав

Изјава 2

Изјава којом се овлашћује Електротехнички факултет/ Академија умјетности Универзитета у Бањој Луци да мастер/магистарски рад учини јавно доступним

Овлашћујем Електротехнички факултет/ Академију умјетности Универзитета у Бањој Луци да мој мастер/магистарски рад, под насловом

Генерисање веб апликација на основу шеме базе података

који је моје ауторско дјело, учини јавно доступним.

Мастер/магистарски рад са свим прилозима предао/ла сам у електронском формату, погодном за трајно архивирање.

Мој мастер/магистарски рад, похрањен у дигитални репозиторијум Универзитета у Бањој Луци, могу да користе сви који поштују одредбе садржане у одабраном типу лиценце Креативне заједнице *(Creative Commons)*, за коју сам се одлучио/ла.

1. Ауторство
2. Ауторство - некомерцијално
3. Ауторство - некомерцијално - без прераде
4. Ауторство - некомерцијално - дијелити под истим условима
5. Ауторство - без прераде
6. Ауторство - дијелити под истим условима

(Молимо да заокружите само једну од шест понуђених лиценци, кратак опис лиценци дат је на полеђини листа).

У Бањој Луци 11.09.2025. год.

Потпис кандидата

Јанкуша Вукосав

Изјава 3

Изјава о идентичности штампане и електронске верзије мастер/магистарског рада

Име и презиме аутора: Данијела Вукосав

Наслов рада: Генерисање веб апликација на основу шеме базе података

Ментор: доц. др Михајло Савић

Изјављујем да је штампана верзија мог мастер/магистарског рада идентична електронској верзији коју сам предао/ла за дигитални репозиторијум Универзитета у Бањој Луци.

У Бањој Луци 11.09.2025. год.

Потпис кандидата

Данијела Вукосав

УНИВЕРЗИТЕТ У БАЊОЈ ЛУЦИ
ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ
Патре 5
78000 Бања Лука

Проф. др Зоран Ђурић
Електротехнички факултет Универзитета у Бањој Луци

Доц. др Михајло Савић
Електротехнички факултет Универзитета у Бањој Луци

Проф. др Дражен Брђанин
Електротехнички факултет Универзитета у Бањој Луци

НАУЧНО-НАСТАВНОМ ВИЈЕЋУ

ЕЛЕКТРОТЕХНИЧКОГ ФАКУЛТЕТА УНИВЕРЗИТЕТА У БАЊОЈ ЛУЦИ

Одлуком Научно-наставног вијећа Електротехничког факултета Универзитета у Бањој Луци број 20/3.296-11/25 од 19.06.2025. године, именовани смо за чланове Комисије за завршни рад II циклуса, под називом "**Генерисање веб апликација на основу шеме базе података**", кандидата Данијеле Вукосав. Након прегледа приложеног рада подносимо сљедећи

ИЗВЈЕШТАЈ

1. БИОГРАФСКИ ПОДАЦИ КАНДИДАТА

Данијела Вукосав, дипл. инж. електротехнике, рођена је 08.10.1999. године у Невесињу, гдје је стекла основно и средњошколско образовање. Дипломирала је на Електротехничком факултету Универзитета у Бањој Луци 2022. године (оцјеном 10) са завршним радом I циклуса под називом "Систем за генерисање React апликација на основу шеме базе података" и просјечном оцјеном у току студија од 9,00.

Студије II циклуса студија је уписала 2023. године на Електротехничком факултету Универзитета у Бањој Луци и положила све испите просјечном оцјеном 10. Од 2023. године је запослена на радном мјесту "софтверски инжењер" у предузећима IT House, Бања Лука, FatCat Coders, Београд и Alea Partners, Бања Лука.

Кандидаткиња је објавила један рад у научном часопису националног значаја:

- D. Vukosav, D. Banjac, M. Ljubojević, and M. Savić, "CodeCrafter – Efficient Code Generator for Modern Single-page Web Applications," International Journal of Electrical Engineering and Computing, vol. 9, no. 1, pp. 1–9, Jun. 2025, doi: 10.7251/IJEEC2501001V.

2. ОСНОВНИ ПОДАЦИ О РАДУ

Завршни рад II циклуса студија кандидата Данијеле Вукосав, под називом "Генерисање веб апликација на основу шеме базе података", садржи 90 нумерисаних страница са укупно 40 слика и 7 табела, а организован је у седам поглавља:

1. Увод
2. Базе података
3. Spring Boot
4. React
5. Приједлог и опис рјешења за генерисање апликација
6. Евалуација имплементираног рјешења
7. Закључак

Списак коришћене литературе садржи 48 цитираних извора.

3. АНАЛИЗА РАДА

У уводном дијелу рада прво су описани мотивација за израду рада и предмет истраживања, а затим су наведени циљеви, методологија и остварени резултати истраживања. Потом је укратко приказан садржај рада по главама и наведен је научни рад у ком су објављени резултати истраживања. Основни мотив за истраживање и израду рада представља недоступност рјешења на тржишту које обједињује и имплементира све неопходне функционалности система за генерисање савремених веб апликација на основу шеме базе података, према захтјевима постављеним у овом раду.

Друго поглавље даје теоријску основу база података уз кратки опис процеса моделовања и управљања подацима. Дат је преглед кратак историјата развоја база података, као и кратак опис основних врста база података. Посебна пажња је посвећена релационим базама података и нормалним формама.

Треће поглавље даје детаљан опис Spring Boot развојног оквира. Описане су основне карактеристике и концепти Spring оквира, као и технологије које представљају основу, како оквира, тако и апликација које су реализоване употребом Spring оквира. Дат је преглед основних модула оквира уз кратак опис сваког од наведених модула. Описан је Spring Boot развојни оквир и извршено је поређење Spring и Spring Boot оквира. Поглавље садржи и опис стандардних начина комуникације са базама података у апликацијама развијеним употребом Spring Boot развојног оквира, као и преглед основних сигурносних механизама доступних у овом развојном оквиру.

Четврто поглавље је посвећено React библиотеци за изградњу корисничких интерфејса у склопу једностраничних веб апликација. Поглавље детаљно обрађује архитектуру и начин функционисања React библиотеке, али и апликација развијених употребом ове библиотеке. Посебна пажња је посвећена компонентама и животном циклусу компоненти, као и основним концептима и технологијама које чине React (Virtual DOM, JSX и Hooks).

Пето поглавље садржи опис приједлога система за генерисање веб апликација употребом претходно описаних технологија на основу шеме базе података. Анализирана су тренутно доступна рјешења, те је представљен основни приједлог система који отклања недостатке уочене у постојећим системима. Детаљно је описан комплетан процес од обраде улазних података до генерисања кода веб апликације, а посебна пажња је посвећена проблему аутентификације и ауторизације корисника у генерисаној апликацији. Представљена је и функционалност генерисане апликације на конкретним примјерима и уз одговарајуће графичке приказе рада апликације.

Шесто поглавље је посвећено евалуацији рада генератора кода. Систем је тестиран на већем броју улазних шема база података, при чему се број табела у шеми кретао од једне до 192 табеле. Мјерена су времена генерисања кода у серијском и паралелном режиму рада и у свим посматраним случајевима је вријеме генерисања било испод 500 ms, док су времена извршавања у паралелном режиму рада била мања од 100 ms. Након генерисања кода извршена је и статичка провјера генерисаног кода употребом аутоматизованог алата при чему нису детектовани пропусти или друге неправилности. Извршено је и сумарно поређење имплементираног рјешења и тренутно доступних рјешења на тржишту, те су анализирана ограничења имплементираног система.

Рад завршава седмим поглављем које садржи закључни приказ резултата истраживања, као и завршна разматрања остварених циљева и основних доприноса рада, те приједлог могућих будућих праваца истраживања и развоја система.

4. НАЈВАЖНИЈИ ДОПРИНОСИ

Комисија сматра да је кандидат, кроз спроведено истраживање, реализовао завршни рад II циклуса студија који садржи више значајних доприноса, од којих су најважнији :

1. Извршена је анализа тренутно доступних рјешења на тржишту.
2. Анализиране су функционалности генератора кода, као и подршка за сваку од њих од стране претходно анализираних рјешења.
3. Развијено је функционално рјешење за генерисање савремених једностраничних веб апликација на основу шеме базе података.
4. Генерисана апликација подржава механизме контроле приступа засноване на улогама и дозволама, као и ограничења приступу подацима дефинисана од стране корисника.
5. Имплементирани систем има високе перформансе генерисања кода што је потврђено серијом мјерења времена генерисања апликација различитог нивоа комплексности.
6. Генерисане апликације задовољавају статичко тестирање кода на сигурносне пропусте и друге неправилности.

5. ЗАКЉУЧАК И ПРИЈЕДЛОГ

Комисија сматра да завршни рад II циклуса под називом "Генерисање веб апликација на основу шеме базе података", кандидата Данијеле Вукосав, садржи све потребне елементе и резултате којима су остварени постављени циљеви истраживања, те са задовољством предлаже Научно-наставном вијећу Електротехничког факултета Универзитета у Бањој Луци да усвоји извјештај Комисије и одобри заказивање усмене јавне одбране.

Бања Лука, 28.08.2025. године

Проф. др Зоран Ђурић, предсједник

Доц. др Михајло Савић, ментор

Проф. др Дражен Брђанин, члан