

UNIVERZITET U BEOGRADU  
MATEMATIČKI FAKULTET



Luka Vujčić

UNAPREĐENJE API-JA MODERNIH VEB  
APLIKACIJA KROZ UPOTREBU GRAPHQL-A

master rad

Beograd, 2023.

**Mentor:**

dr Vladimir FILIPOVIĆ, redovni profesor  
Univerzitet u Beogradu, Matematički fakultet

**Članovi komisije:**

dr Mladen NIKOLIĆ, vanredni profesor  
Univerzitet u Beogradu, Matematički fakultet

dr Ivan ČUKIĆ, docent  
Univerzitet u Beogradu, Matematički fakultet

**Datum odbrane:** \_\_\_\_\_

**Naslov master rada:** Unapređenje API-ja modernih veb aplikacija kroz upotrebu GraphQL-a

**Rezime:** U okviru ovog rada dat je pregled REST arhitekture i prikazani su neki problemi koji se javljaju prilikom njenog korišćenja. Ovi problemi su sve prisutniji sa rastom kompleksnosti aplikacije. Prethodno uočeni problemi predstavljaju motivaciju za uvođenje GraphQL-a. Prikazan je GraphQL upitni jezik i njegova sintaksa i dati su primeri njegove upotrebe. Implementirana je jednostavna aplikacija iz realnog domena kroz REST arhitekturu i kroz GraphQL. Potom je napravljena komparativna analiza proučavanih arhitektonskih pristupa. Dati su osnovni saveti kada se opredeliti za jednu arhitekturu, a kada za drugu.

**Ključne reči:** REST, API, GraphQL, arhitektura

# Sadržaj

<b>1</b>	<b>REST arhitektura</b>	<b>1</b>
1.1	Nastanak REST arhitekture . . . . .	1
1.2	Pregled REST arhitekture . . . . .	1
1.3	Dizajn i funkcionalnost REST API-ja . . . . .	3
<b>2</b>	<b>GraphQL</b>	<b>7</b>
2.1	Nastanak GraphQL-a . . . . .	7
2.2	Motivacija stvaranja GraphQL-a . . . . .	7
2.3	GraphQL upitni jezik . . . . .	9
2.4	Razrešavači u GraphQL-u . . . . .	19
<b>3</b>	<b>Pregled modernih alata koji implementiraju GraphQL</b>	<b>20</b>
3.1	Server Apollo . . . . .	20
3.2	TypeGraphQL . . . . .	20
3.3	GraphQL Yoga . . . . .	21
<b>4</b>	<b>Rešavanje problema iz realnog sveta upotrebom GraphQL-a i REST arhitekture</b>	<b>22</b>
4.1	Prikaz entiteta koji se modeluje . . . . .	22
4.2	Pristup korišćenjem GraphQL-a . . . . .	23
4.3	Pristup korišćenjem REST arhitekture . . . . .	37
<b>5</b>	<b>Komparativna analiza GraphQL-a i REST API-ja</b>	<b>53</b>
5.1	Dizajn koda . . . . .	53
5.2	Performanse . . . . .	54
5.3	Skalabilnost . . . . .	55
5.4	Bezbednost . . . . .	56

*SADRŽAJ*

---

<b>6 Zaključak</b>	<b>58</b>
<b>Literatura</b>	<b>60</b>

# Glava 1

## REST arhitektura

U ovom poglavlju je prikazan kratak historijat nastanka REST arhitekture i njen uticaj na razvoj serverskih veb aplikacija. Pored historijata, prikazani su ključni elementi ove arhitekture, ukratko su opisani REST principi i prikazani su principi dizajna REST API-ja.

### 1.1 Nastanak REST arhitekture

REST arhitektura je ostvarila značajan uticaj na razvoj serverskih veb aplikacija i interneta u globalu. Ova arhitektura je stekla široku popularnost i postala skoro nezaobilazna u razvoju interfejsa serverskih aplikacija.

REST arhitekturu je predložio Roy Fielding u svojoj doktorskoj disertaciji iz 2000. godine pod nazivom „Architectural styles and the design of network-based software architectures” [15]. Motivacija za pisanje rada je bilo Fieldingovo nezadovoljstvo tadašnjim pristupima u razvoju mrežnih aplikacija.

Autor je prepoznao skalabilnost, performanse, jednostavnost i nadogradivost kao ključne aspekte kojima se treba voditi prilikom razvoja mrežnih aplikacija.

### 1.2 Pregled REST arhitekture

REST (REpresentational State Transfer) predstavlja arhitekturni stil koji je danas veoma popularan i zastupljen u razvoju modernih veb aplikacija. On predstavlja vid klijent-server arhitekture. Tri ključna elementa ove arhitekture: klijent, server i resurs.

Komunikacija između klijenta i servera je dvosmerna. Ona funkcioniše kao standardna klijent-server arhitektura gde se komunikacija zasniva na traženju i isporučivanju resursa.

U nastavku su navedeni i ukratko objašnjeni REST principi.

- **Slaba spregnutost klijenta i servera** je princip koji predlaže minimizovanje zavisnosti između klijenta i servera. Na ovaj način moguće je vršiti promene i na klijentskoj i na serverskoj strani nezavisno, bez potrebe da se menja druga strana. Ovim principom povećava se fleksibilnost, nezavisnost i proširivost sistema. Zahvaljujući ovom principu moguće je integrisati raznorodne komponente koje su saglasne sa njim.
- **Komunikacija bez čuvanja stanja** je princip koji predlaže da svaki zahtev od klijenta ka serveru sadrži sve neophodne informacije. Server ne čuva nikakve informacije o klijentu, odnosno svaki zahtev je samodovoljan i ne zavisi od prethodnih zahteva. Odgovornost čuvanja bilo kakvog stanja je na klijentu i kada je neophodno klijent ga šalje serveru.
- **Uniformnost interfejsa** je princip koji predlaže da klijent i server komuniciraju na uniforman način bez obzira na specifičnost uređaja ili aplikacije. Princip predlaže da se interfejs dizajnira na način da bude jednostavan i intuitivan za korišćenje.
- **Keširanje** je princip koji predlaže čuvanje odgovora sa servera na klijentskoj strani. Zahvaljujući ovom pristupu, smanjuje se broj zahteva ka serveru, povećavaju se performanse i smanjuje se opterećenost servera. Ova tehnika je posebno korisna kada se resursi ne menjaju ili se ne menjaju često.
- **Slojevitost sistema** je princip koji predlaže postavljanje više komponenti između klijenta i servera od kojih svaka obavlja svoj posao. Primeri takvih komponenti su: proksiji, mrežni prolazi i balanseri opterećenja. Neophodno je da svaka od ovih komponenti zna na koji način treba da komunicira sa komponentom ispred i iza sebe. Slojeviti sistemi su modularni, skalabilni, enkapsulirani, sigurni i pružaju dobre performanse

## 1.3 Dizajn i funkcionalnost REST API-ja

U ovom poglavlju je dat osnovni pregled HTTP-a, prikazan je odnos HTTP-a i REST arhitekture, predstavljeni su principi dizajna REST API-ja i dat je primer integracije HTTP-a i REST arhitekture.

### 1.3.1 HTTP

Pre osvrta na dizajn API-ja koji zadovoljava principe REST arhitekture, potrebno je osvrnuti se na neke ključne pojmove koji se odnose na specifičnosti HTTP-a.

**URL** (Uniform Resource Locator) je specifičan URI (Uniform Resource Identifier) koji identifikuje lokaciju resursa. Njegov oblik je **prokol://domen/putanja** [9]. Podrazumevani protokol za potrebe ovog rada je HTTP. U kontekstu dizajniranja API-ja, najznačajniji deo je putanja. Domen upućuje na adresu na kojoj će biti dostupan API i taj deo URL-a nije posebno zanimljiv u kontekstu dizajna API-ja [13].

**HTTP metode** su važan deo HTTP-a i u kontekstu REST arhitekture se koriste za rad sa resursima. Sledi navođenje najbitnijih metoda uz kratko objašnjenje o njihovoj upotrebi:

- GET - koristi se za dohvatanje resursa sa servera.
- POST - koristi se kao zahtev serveru za kreiranje novog resursa.
- DELETE - koristi se kao zahtev serveru da obriše određeni resurs.
- PATCH - koristi se kao zahtev serveru da modifikuje određeni resurs.
- PUT - koristi se kao zahtev serveru da zameni određeni resurs novim resursom.

U tabeli 1.1 su prikazane osobine HTTP zahteva. Analizirane su tri osobine: postojanje tela zahteva, zahtev je bezbedan i zahtev je idempotentan. Ove osobine su objašnjene u nastavku.

**Postojanje tela zahteva** pokazuje da li se podaci šalju kroz telo zahteva. Na primer, kod GET zahteva svi podaci se šalju kroz URL.

**Zahtev je bezbedan** Za zahtev se kaže da je bezbedan ako ne može da izvrši modifikaciju resursa.

**Zahtev je idempotentan** Za zahtev se kaže da je idempotentan ako višestruko izvršenje tog zahteva proizvodi isti rezultat kao i jedno izvršenje zahteva.



Tabela 1.1: Osobine HTTP zahteva

metod/osobina	Postojanje tela	Bezbednost	Idempotentnost
GET	-	x	x
POST	x	-	-
PUT	x	-	x
DELETE	-	-	x
PATCH	x	-	-

### 1.3.2 REST arhitektura i HTTP

Neretko, postoje nedoumice u vezi odnosa REST arhitekture i HTTP-a. REST arhitektura je agnostična i ne podrazumeva nijedan konkretan protokol, samim tim ne mora se nužno koristiti sa HTTP-om. Svakako, u praksi je kombinacija REST arhitekture i HTTP-a veoma česta.

### 1.3.3 Dizajn REST API-ja

Kao što je već spomenuto, u REST arhitekturi, centralno mesto zauzimaju resursi. Kada se identifikuju resursi od značaja, potrebno je odrediti njihovu hijerarhijsku vezu u poslovnom domenu i tu vezu treba preslikati na URL strukturu koja je hijerarhijski orijentisana. Pažljivo definisanje URL hijerarhije je veoma bitan deo dizajna API-ja.

API se opisuje kolekcijom krajnjih tačaka (eng. endpoints). Krajnja tačka predstavlja tačku komunikacije koju server obezbeđuje kako bi podržao HTTP komunikaciju sa drugim uređajima. Krajnja tačka je jedinstveno određena URL-om i HTTP metodom.

Primer krajnje tačke koja pruža funkcionalnost dohvaćanja korisnika je **GET** /v1/users. U primeru 1.3.3 data je lista krajnjih tačaka jednog API-ja.

Neki principi dizajna API-ja:

- HTTP metode se koriste na odgovarajući način kao akcije nad resursima
- URL treba da bude hijerarhijski orijentisan i intuitivan
- Ne preporučuje se velika dubina hijerarhije
- Poželjno je obezbediti verzionisanje API-ja kako bi bila omogućena kompatibilnost unazad i povećana otpornost API-ja na buduće promene

- Treba koristiti straničenje i filtriranje kada postoji mogućnost dohvaćanja velikog broja resursa
- Resurs u URL-u treba da bude predstavljen imenicom, a ne glagolom
- Imenica treba da bude u množini, a ne u jednini

Na primeru foruma prikazane su osnovne REST operacije.

Modeluju se dva tipa entiteta: Korisnik (User) i Članak (Article). U ovakvom jednostavnom sistemu obezbeđuju se CRUD (Create, Read, Update, Delete) operacije za korisnika i za članak.

Jedan članak može imati samo jednog korisnika kao autora. Korisnik može biti autor više različitih članaka. Na osnovu navednog odnosa članka i korisnika, može se uočiti određena hijerarhija koja se može preslikati na hijerarhiju u okviru URL-a.

U primeru je korišćeno i verzionisanje, koje je ostvareno dodavanjem prefiksa v1 na URL. Lista krajnjih tačaka jednog takvog API-ja prikazana je u nastavku.

- **GET** /v1/users - Dohvaćanje svih korisnika
- **POST** /v1/users - Pravljenje novog korisnika
- **PUT** /v1/users - Ažuriranje korisnika
- **DELETE** /v1/users - Brisanje svih korisnika
- **GET** /v1/users/1 - Dohvaćanje korisnika koji ima identifikator 1
- **PATCH** /v1/users/1 - Zamena korisnika sa identifikatorom 1, ako postoji
- **DELETE** /v1/users/1 - Brisanje korisnika sa identifikatorom 1, ako postoji
- **GET** /v1/users/1/articles - Dohvaćanje svih članaka za korisnika sa identifikatorom 1
- **POST** /v1/users/1/articles - Pravljenje članka za korisnika sa identifikatorom 1
- **PUT** /v1/users/1/articles - Zamena svih članaka za korisnika sa identifikatorom 1
- **DELETE** /v1/users/1/articles - Brisanje svih članaka za korisnika sa identifikatorom 1

Treba primetiti da su korišćene reči `users` i `articles` koje su imenice u množini. Ako se pogleda putanja `/users/1/articles` vidi se da je ona hijerarhijska i nije velike dubine. HTTP metode su na odgovarajući način pridružene akcijama (GET metoda je pridružena akciji dohvatanja resursa, a ne npr. ažuriranju resursa).

Prethodno dizajniran API je takođe jasan i intuitivan. Može se reći da je ovako prikazan API primer dobrog API-ja.

Treba imati u vidu da je ovo dosta jednostavan primer i da je u kompleksnijim sistemima dizajn API-ja daleko teži.

### 1.3.4 Primer integracije REST arhitekture i HTTP-a

U ovom poglavlju je dat prikaz integracije prethodno dizajniranog API-ja sa HTTP-om.

Za potrebe ovog primera se koristi testni domen **test.com**. Klijentu se pruža mogućnost da koristi usluge servera slanjem HTTP zahteva na odgovarajući URL. Kako bi klijent mogao da koristi usluge servera, neophodno je da krajnje tačke iz poglavlja 1.3.3 pretvori u URL-ove u obliku koji je prikazan u poglavlju 1.3.1. Pretvaranje se postiže dodavanjem prefiksa `http://www.test.com`.

Sledi prikaz nekih krajnjih tačaka iz primera 1.3.3 koji su svedeni na odgovarajući oblik. Za svaki URL je data HTTP metoda i opis funkcionalnosti tog URL-a:

- **POST** `http://www.test.com/v1/users` - pravljenje novog korisnika
- **GET** `http://www.test.com/v1/users/1` - dohvatanje specifičnog korisnika
- **DELETE** `http://www.test.com/v1/users/1/articles` - brisanje članaka za korisnika sa identifikatorom 1

# Glava 2

## GraphQL

U ovom poglavlju je opisan nastanak GraphQL-a i data je motivacija za njegovo stvaranje, pregledom nekih problema sa kojima se suočava REST arhitektura. Prikazan je GraphQL upitni jezik uz navođenje i opis osnovnih jezičkih konstrukcija. Prikazani su primeri upotrebe jezika.

### 2.1 Nastanak GraphQL-a

GraphQL je kreiran od strane inženjera Facebook-a, kao rešenje problema sa kojim se kompanija suočavala tokom razvoja aplikacija u svom poslovnom domenu. Specifikacija GraphQL-a je objavljena 2012. godine. Facebook je od 2012. godine pored upotrebe za veb aplikacije, GraphQL počeo da koristi i za mobilne aplikacije [17]. GraphQL je predstavljen je kao proizvod otvorenog koda 2015. godine [7].

### 2.2 Motivacija stvaranja GraphQL-a

Razvojem različitih veb aplikacija od nastanka REST API arhitekture, primećeni su određeni problemi sa rastom kompleksnosti aplikacija. Ovi problemi predstavljaju značajan ograničavajući faktor u eri sve zahtevnijih veb aplikacija. Pored uočenih problema, javila se inicijativa da se API serverskih aplikacija učini fleksibilnijim, efikasnijim za korišćenje i da se razvojnim timovima omogući jednostavnija upotreba API-ja. U ovom poglavlju su prikazani neki od problema koji opravdavaju potrebu uvođenja nečega poput GraphQL-a.

### 2.2.1 Prekomerno i nedovoljno preuzimanje

Kod REST arhitekture, server pruža određeni broj krajnjih tačaka klijentu, koje klijent koristi za potrebu komunikacije sa serverom. Na ovaj način, klijent je prinuđen da koristi već utvrđene krajnje tačke bez mogućnosti da ih prilagodi svojim potrebama. Neki primeri krajnjih tačaka prikazani su u primeru 1.3.3 u operacijama nad korisnicima i člancima.

Česta je situacija da server isporučuje klijentu veći obim podataka od potrebnih u jednom zahtevu. Ta situacija nastaje kao posledica nedovoljno dobrog dizajna i kao slabost REST arhitekture koja ne pruža fleksibilnost klijentu da dohvati tačno njemu potrebne podatke. Takav fenomen se naziva „prekomerno preuzimanje” (eng. „Over Fetching”). U mrežama malog protoka, ovo može biti veoma ozbiljan nedostatak.

Sa druge strane postoji i fenomen „nedovoljnog preuzimanja” (eng. „Under Fetching”). Ovaj fenomen se javlja u situaciji kada je API serverske aplikacije dizajniran na način da klijent ne može da dohvati sve njemu potrebne podatke u jednom zahtevu. To se obično dešava kada se u jednom zahtevu dohvata niz resursa, a potrebno je dohvatiti i dodatne detalje o tim resursima, koji po dizajnu API-ja nisu planirani da se isporuče zajedno sa ovim nizom kako bi se izbeglo „prekomerno preuzimanje”. Dohvatanje tih detalja obavlja se slanjem zahteva za svaki pojedinačni resurs iz već dohvaćenog niza. Ovo dovodi do povećanja broja zahteva ka serveru sto može dovesti i do povećanja opterećenja servera, a stoga i do degradacije performansi.

Ova dva fenomena predstavljaju dve krajnosti i prilikom dizajna API-ja treba pokušati napraviti balans.

Ove probleme moguće je efikasno rešiti upotrebom GraphQL-a. GraphQL pruža fleksibilnost u dohvatanju podataka. Zahvaljujući njemu, klijent može opisati strukturu njemu potrebnih podataka, dohvatiti samo neophodne podatke i to realizovati u tačno jednom zahtevu.

### 2.2.2 Broj krajnjih tačaka

U prethodnom poglavlju je pokazano da i jednostavna aplikacija može imati veliki broj krajnjih tačaka. Sa porastom kompleksnosti aplikacije, raste i broj krajnjih tačaka. To može biti teško za održavanje i takođe komunikacija između klijenta i servera postaje takođe kompleksija. Zahvaljujući GraphQL-u dovoljna je samo jedna krajnja tačka.

### 2.2.3 Verzionisanje

Tokom životnog ciklusa aplikacije dolazi do promene API-ja. Iz tog razloga neophodno je uvesti verzionisanje, kako bi se obezbedila kompatibilnost unazad. U REST arhitekturi, verzionisanje se obavlja dodavanjem verzije na početak URL-a. Često se koriste prefiksi poput **v1**, **v2**, **v3** za označavanje odgovarajuće verzije. Jedan primer takvog verzionisanja je prikazan u poglavlju 1.3.3.

GraphQL ne koristi modifikaciju URL-a za verzionisanje. Verzionisanje u GraphQL-u je moguće ostvariti kroz dodavanje novih polja ili tipova koja mogu podržati različite verzije API-ja. Više o poljima i tipovima je dato u poglavljima 2.3.1.1 i 2.3.4.

## 2.3 GraphQL upitni jezik

GraphQL upitni jezik je jezik koji se koristi da definiše strukturu i oblik zahteva koji se šalje ka serveru. Uz pomoć ovog jezika, klijent može da specificira koji su mu podaci potrebni i da ih dobije u jednom zahtevu, odnosno može obaviti modifikaciju podataka na serveru. U ovom poglavlju je dat osvrt na sintaksu GraphQL-a.

### 2.3.1 Upiti

Upiti su osnovni mehanizam koji koristi klijent za dohvaćanje podataka sa servera. Koristeći upit, klijentu se pruža mogućnost da dohvati njemu potrebne podatke sa servera koji su strukturirani na njemu potreban način. Upiti u kontekstu ovog poglavlja predstavljaju imutabilne strukture, odnosno mehanizam koji ne modifikuje podatke na serveru. Modifikacija resursa na serveru postiže se upotrebom mutacija koje su objašnjene u poglavlju 2.3.2. U ovom poglavlju su prikazane osnovne strukture jezika koje se koriste za kreiranje upita.

#### 2.3.1.1 Polja

Polja predstavljaju najjednostavniji deo sintakse i osnovu upita u GraphQL-u. Polja koja se navedu u upitu za određeni entitet su polja koja će server obezbediti u svom odgovoru.

U narednom primeru je prikazan upit sa poljima **name** i **email**.

```
1 {  
2   users {  
3     name
```

## GLAVA 2. GRAPHQL

---

```
4     email
5   }
6 }
```

### Odgovor servera

```
1 {
2   "data": {
3     "users": [
4       {
5         "name": "Petar Petrovic",
6         "email": "petarpetrovic@test.com"
7       },
8       {
9         "name": "Marko Markovic",
10        "email": "markomarkovic@test.com"
11      }
12    ]
13  }
14 }
```

Upiti takode mogu da sadrže ugnježdjene entitete.

U narednom primeru je prikazan upit sa ugnježđenim entitetima.

```
1 {
2   users {
3     name
4     email
5     articles {
6       title
7     }
8   }
9 }
```

### Odgovor servera

```
1 {
2   "data": {
3     "users": [
4       {
5         "name": "Petar Petrovic",
6         "email": "petarpetrovic@test.com",
7         "articles": [
8           {
9             "title": "Uvod u programiranje 1"
10          }
11        ]
12      },
13      {
14        "name": "Marko Markovic",
```

```
15     "email": "markomarkovic@test.com",
16     "articles": [
17       {
18         "title": "Uvod u programiranje 2"
19       }
20     ]
21   }
22 ]
23 }
24 }
```

U ovom primeru može se primetiti kako, veoma jednostavnim upitom, klijent može da zahteva tačno određena polja. Na ovaj način, klijent dobija tačno samo ona polja koja su mu potrebna i ne dolazi do prekomernog niti nedovoljnog preuzimanja.

### 2.3.1.2 Argumenti

GraphQL upiti podržavaju upotrebu argumenata.

U narednom primeru je prikazan upit sa argumentima i ugnježđenim entitetima kada identifikator ima vrednost 1.

```
1 {
2   user(id: "1") {
3     name
4   }
5 }
```

Odgovor servera

```
1 {
2   "data": {
3     "user": {
4       "name": "Petar Petrovic"
5     }
6   }
7 }
```

Korišćenjem odgovarajuće vrednosti argumenta moguće je dohvatiti i drugog korisnika.

U narednom primeru je prikazan upit sa argumentima i ugnježđenim entitetima kada identifikator ima vrednost 2.

```
1 {
2   user(id: "2") {
3     name
4   }
5 }
```



Odgovora servera

```
1 {
2   "data": {
3     "user": {
4       "name": "Marko Markovic"
5     }
6   }
7 }
```

### 2.3.1.3 Aliasi

Da bi bio izbegnut konflikt imena u rezultatu kada se dohvataju dva polja sa istim imenom, mogu se koristiti aliasi. U primeru koji sledi korišćeni su aliasi **user1** i **user2**.

U narednom primeru je prikazan upit sa aliasima, argumentima i ugnježđenim entitetima.

```
1 {
2   user1: user(id: 1) {
3     id
4     name
5   }
6   user2: user(id: 2) {
7     id
8     email
9   }
10 }
```

Odgovora servera

```
1 {
2   "data": {
3     "user1": {
4       "id": "1",
5       "name": "Petar Petrovic",
6     },
7     "user2": {
8       "id": "2",
9       "email": "markomarkovic@test.com"
10    }
11  }
12 }
```

U navedenom primeru potrebno je vratiti dva korisnika, pa s obzirom da su oni istog tipa, iskorišćeni su aliasi kako bi bio izbegnut konflikt. Ako ovaj upit ne bi

sadržao aliase, ovaj upit ne bi imao korektnu sintaksu, a i server ne bi mogao da razlikuje ova dva polja u fazi izvršenja upita.

### 2.3.1.4 Tip operacije

GraphQL podržava tri tipa operacija: upite, mutacije i pretplate. Do sada su bili prikazani samo upiti i korišćena je skraćena verzija za njihovo definisanje. Dobra je praksa da se, kada nije neophodno, naglasi tip operacije. U poglavlju 2.3.1.1 prikazan je jednostavan upit, ali je korišćena pojednostavljena (skraćena) sintaksa.

U narednom primeru je prikazan upit u kome nije korišćena skraćena sintaksa.

```
1 query UserNameEmail {
2   user {
3     name
4     email
5   }
6 }
```

### 2.3.1.5 Promenljive

Do sada je prikazano kako da, kao argument, bude prosleđena konstantna vrednost koja je fiksirana u upitu. Upitni jezik podržava i upotrebu promenljivih.

U narednom primeru je prikazan upit sa promenljivom `userId` koja sadrži vrednost identifikatora korisnika kojeg je potrebno dohvatiti i ona je skalarnog tipa ID.

Skalarni tipovi su prikazani u poglavlju 2.3.4.1.

```
1 query UserNameById($userId: ID) {
2   user(id: $userId) {
3     name
4   }
5 }
```

Da bi ovakvi upiti funkcionisali, potrebno je izvršiti tzv. vezivanje promenljivih. Vezivanje promenljive je moguće obaviti kroz polje `variables` koje se obezbeđuje zajedno sa GraphQL upitom u telu POST zahteva ka GraphQL krajnjoj tački.

Telo zahteva je dato u nastavku:

```
1 {
2   "query": "query User($userId: ID) {user(id: $userId) {name}}",
3   "variables": {
4     "userId": "1"
5   }
6 }
```

### 2.3.2 Mutacije

U poglavlju 2.3.1 su prikazani upiti kao deo imutabilne konstrukcije, odnosno mehanizma samo za dohvatanje podataka bez njihove modifikacije ili kreiranja. U ovom poglavlju su predstavljene mutacije kao mehanizam uz pomoću kojeg je moguće menjati podatke.

U tehničkom smislu moguće je modifikovati podatke i bez korišćenja mutacije samo upitom. Preporuka je da se za modifikaciju podataka koriste isključivo mutacije. Ovo je slično kao što je moguće koristiti i GET zahtev za modifikaciju podataka, ali to nije dobra praksa.

Upiti se izvršavaju u paraleli, dok se mutacije izvršavaju serijski. Ako bi se mutacije izvršavale paralelno, potencijalno bi se desila trka za resurse.

U narednom primeru je prikazana je mutacija kojom se kreira korisnik. Koristi se promenljiva **input** koja je definisana tipom **userInput**. Tipovi su prikazani u poglavlju 2.3.4.

```
1 mutation CreateUser($input: userInput) {
2   createUser(input: $input) {
3     id
4     name
5     email
6   }
7 }
```

Prikaz vezivanja promenljivih za prethodnu mutaciju.

```
1 "input": {
2   "id": 5
3   "name": "Lazar Lazarevic",
4   "email": "lazarlazarevic@test.com"
5 }
```

### 2.3.3 Pretplate

Pretplate omogućavaju komunikaciju između klijenta i servera u realnom vremenu. Pretplate su zasnovane na događajima. Klijent se pretplaćuje na odgovarajući događaj koji server emituje. Uspostavljanjem ovakvog mehanizma, klijent dobija od servera najnovije podatke u realnom vremenu. Više o pretplatama moguće je pročitati u knjizi [17].

U narednom primeru klijent se pretplaćuje na odgovarajući kanal i od servera dobija nove poruke u realnom vremenu. Poruke sadrže polja sadržaj poruke i identifikator pošiljaoca.

```
1 subscription {
2   newMessage(channelId: "1") {
3     content
4     senderId
5   }
6 }
```

Odgovor servera

```
1 {
2   "data": {
3     "newMessage": {
4       "content": "Sadržaj poruke",
5       "senderId": "Identifikator1"
6     }
7   }
8 }
```

### 2.3.4 Tipovi

Do sada su prikazani različiti upiti i mutacije. Može se primetiti da je svaki od navedenih upita mogao imati različitu strukturu. Može se uočiti i da, isključivo posmatranjem navede strukture, nije baš najjasnije kako izgledaju zahtevani podaci. Tu bi tipiziranje moglo biti od koristi. U ovom poglavlju prikazan je sistem tipova u GraphQL-u.

#### 2.3.4.1 Skalarni tipovi

Skalarni tipovi u GraphQL su primitivni tipovi i oni predstavljaju osnovu za konstrukciju drugih tipova. Primitivni tipovi u GraphQL-u su:

- **String** - UTF-8 enkodirana niska znakova
- **Int** - 32-bitni označeni ceo broj
- **Float** - označeni broj u pokretnom zarezu
- **Boolean** - true ili false
- **ID** - jedinstveni identifikator

Pored skalarnih, GraphQL podržava korisnički definisane skalarne tipove.

### 2.3.4.2 Objektni tipovi

Objektni tipovi predstavljaju osnovni način konstruisanja kompleksnih tipova. Omogućavaju da se polja grupišu zajedno i time definišu novi tip (objektni tip).

U narednom primeru je prikazana definicija jednog objektnog tipa. Ovaj objektni tip sadrži polje **articles** koje predstavlja kolekciju članaka u vidu liste objekata.

Više o listama dato je u poglavlju 2.3.4.6.

```
1 type User {
2   id: ID
3   name: String
4   email: String
5   articles: [Article]
6 }
```

### 2.3.4.3 Interfejsi

Poznato je da su interfejsi široko prisutni u svetu programiranja i da su jedan od veoma korišćenih konstrukcija u svetu objektno-orijentisanog programiranja. U ovom radu se interfejsi razmatraju samo u kontekstu GraphQL-a.

Interfejsi su apstraktan tip koji obećava da tipovi koji ga implementiraju moraju ispuniti obećanu strukturu. Treba napomenuti da jedan tip može implementirati više interfejsa.

U narednom primeru je prikazan je interfejs koji predstavlja osobu.

```
1 interface Person {
2   id: ID
3   fullName: string
4 }
```

U narednom primeru prikazana je definicije tipa koja implementira prethodno definisani interfejs.

```
1 type Employee implements Person {
2   id: ID
3   fullName: string
4   salary: Float
5   role: String
6 }
```

Uočava se da tip **Employee** mora imati **id** i **fullName** kao polja jer implementira interfejs **Person**.

### 2.3.4.4 Enumeracije

Enumeracije ili nabrojivi tipovi predstavljaju specifičan tip gde vrednost može biti iz ograničenog skupa vrednosti.

U narednom primeru je prikazan nabrojivi tip koji ima tri vrednosti i gde te vrednosti predstavljaju tri boje.

```
1 enum RGB {
2   BLUE
3   RED
4   GREEN
5 }
```

### 2.3.4.5 Unije

GraphQL podržava uniju kao tip. Tip unija se koristi kada mogu biti vraćeni objekti različitog tipa.

U narednom primeru su prikazane unije koje obuhvataju podatke o krugu i o kvadratu, a zatim je prikazan objektni tip koji koristi te unije.

```
1 type Circle {
2   id: ID
3   radius: Float
4 }
5
6 type Square {
7   id: ID
8   side: Float
9 }
10
11 union Shape = Circle | Square
12
13 type Query {
14   getShape(id: ID): Shape
15 }
```

### 2.3.4.6 Liste

GraphQL podržava kolekciju objekata i za to se koristi notacija u uglastim zagradama.

Treba napomenuti da naziv „Liste” **ne sugeriše** način implementacije u pogledu odabira apstraktne strukture podataka.

### 2.3.4.7 Null i Ne-Null vrednosti

Null predstavlja nepoznatu vrednost. GraphQL podržava postojanje Null vrednosti. Podrazumevano, svaka vrednost može biti Null, ako treba naglasiti da ne može biti Null vrednost, koristi se operator znak uzvika (!).

U narednom primeru je prikazana definicija tipa koji predstavlja korisnika sa Ne-Null poljima **id** i **email**.

```
1 type User {
2   id: ID!
3   name: String
4   email: String!
5   caption: String
6   articles: [Article]
7 }
```

### 2.3.5 Sheme

GraphQL shema predstavlja jednu od osnovnih komponenti GraphQL-a. Ona objašnjava koji tipovi su dostupni, njihovu strukturu i operacije koje je moguće izvoditi nad njima [16].

Schema se može posmatrati kao ugovor između klijenta i servera koji određuje koji **upiti**, **mutacije** i **pretplate** mogu biti korišćeni.

U narednom primeru je prikazana jednostavna shema sa tipom koji definiše korisnika, upitom za dohvaćanje svih korisnika i mutacijom za pravljenje novog korisnika.

```
1 type User {
2   id: ID!
3   name: String
4 }
5 type Query {
6   users: [User]
7 }
8 type Mutation{
9   createUser(name: String!): User
10 }
11 }
```

## 2.4 Razrešavači u GraphQL-u

**Razrešavači** (eng. resolvers) su funkcije koje opisuju mehanizam dohvaćanja podataka za određeno polje. Svako polje ima svoj razrešavač koji dohvata podatke za to polje. Razrešavači predstavljaju ključni deo GraphQL-a i njihova efikasnost igra glavnu ulogu u održavanju dobrih performansi klijentskih upita ka serveru. Detaljnije o razrešavačima moguće je pronaći u knjizi [16].

U narednom primeru je prikazana definicija funkcije razrešavač.

```
1 resolvers = {  
2   Query: {  
3     user: (_, { id }: any) => users.find((user) => user.id === +id)  
4   }  
5 }
```

Linija 3 definiše funkciju razrešavač koja prihvata identifikator korisnika i pronalazi odgovarajućeg korisnika u nizu.

Detaljniji prikaz upotrebe razrešavača moguće je pronaći na primeru aplikacije koja je razvijena za potrebe ovog rada u poglavlju 4.2.5.



## Glava 3

# Pregled modernih alata koji implementiraju GraphQL

Do sada je prikazana specifikacija GraphQL-a, kao i sama sintaksa jezika. U ovom poglavlju, dat je osvrt na moderne radne okvire (eng. framework) uz pomoć kojih je moguće koristiti GraphQL.

### 3.1 Server Apollo

Apollo je podržan u više različitih programskih jezika. Kompatibilan je sa bilo kojim GraphQL klijentom [2].

Podržava GraphQL pretplate, omogućavajući komunikaciju u realnom vremenu.

Apollo nudi optimizaciju performansi kroz keširanje i upotrebu paketne obrade (eng. batch). Na ovaj način smanjuje se opterećenost servera i smanjuje se vreme odgovora servera.

Ovaj server obezbeđuje mehanizam za rukovanje greškama i proverava da li su zahtevi u skladu sa definisanom shemom.

Apollo se lako integriše sa popularnim framework-ovima kao što su na primer Express i Fastify[3, 4].

### 3.2 TypeGraphQL

TypeGraphQL se prvenstveno fokusira na TypeScript. Njegova sintaksa je bazirana na dekoratorima, koji se postavljaju na metode i klase i zahvaljujući kojima se smanjuje potreba za dodatnim kodiranjem i poboljšava čitljivost programskog koda.

Dekoratori omogućavaju automatsko generisanje GraphQL sheme. TypeGraphQL ima ugrađenu podršku za keširanje i paketnu obradu i jednostavno se integriše sa Apollo serverom [1].

TypeGraphQL obezbeđuje mehanizam za rukovanje greškama i proverava da li su zahtevi u skladu sa definisanom shemom.

### **3.3 GraphQL Yoga**

GraphQL Yoga je biblioteka koja omogućava pokretanje GraphQL servera. Realizovan je kao sloj iznad Express-a. On omogućava definisanje GraphQL sheme korišćenjem GraphQL Schema Definition jezika. GraphQL Yoga omogućava i otpremanje datoteka od strane klijenta kao deo GraphQL mutacije. Pretplate su ugrađeni deo i one predstavljaju mehanizam za komunikaciju u realnom vremenu [8].

Ovaj okvir podržava upotrebu srednjeg sloja (eng. middleware) koji može biti lako korišćen za poslove: autentikacije, autorizacije, logovanja, itd.

GraphQL Yoga obezbeđuje mehanizam za rukovanje greškama i proverava da li su zahtevi u skladu sa definisanom shemom.

## Glava 4

# Rešavanje problema iz realnog sveta upotrebom GraphQL-a i REST arhitekture

U ovom poglavlju, prikazana je arhitektura, dizajn i implementacija razvijenih aplikacija koje koriste GraphQL odnosno tradicionalnu REST arhitekturu. Razvijene su dve aplikacije, jedna koja koristi REST arhitekturu i druga koja koristi GraphQL. Na ovom praktičnom primeru, dat je osvrt na teorijske prikaze sa početka ovog rada.

U implementaciji koristi se **Express** radni okvir (NodeJS). Baza podataka se ne koristi već se koriste fiksirani nizovi u kodu, kako se implementacija ne bi opterećivala komunikacijom sa bazom koja nije značajna u predmetu razmatranja ovog rada. Kao podrška tipiziranju biće korišćen programski jezik **Typescript** [12].

Klijentska strana je prikazana kroz **Postman**, kako bi se izbeglo razvijanje klijentskog dela. Postman je popularna platforma za razvijanje, testiranje i dokumentovanje API-ja [11].

Klijentski deo aplikacije može biti implementiran u većini modernih tehnologija, kao što je na primer **React**. Većina modernih klijentskih aplikacija ima podršku za korišćenje GraphQL-a.

### 4.1 Prikaz entiteta koji se modeluje

Poslovni domen serverske aplikacije koja se razvija u ovom radu je pojednostavljena verzija internet foruma.

Forum je veb aplikacija namenjena vodjenju diskusije između korisnika foruma. Sadrži članke koje kreiraju korisnici. Svaki korisnik može kreirati više članaka. Svaki članak ima: sadržaj, naslov, tagove, kategorije kojima pripada članak, broj sviđanja koje korisnici ostavljaju na članku i autora članka. Na svakom članak moguće je ostaviti više komentara. Svaki komentar ima: sadržaj, autora i broj sviđanja.

## 4.2 Pristup korišćenjem GraphQL-a

U ovom poglavlju, prikazano je rešenje upotrebom GraphQL-a. **Server Apollo** je korišćen kao radni okvir koji implementira GraphQL.

Cela aplikacija je dostupna na autorovom Github repozitorijumu kao softver otvorenog koda [5].

### 4.2.1 Inicijalizacija projekta i instalacija paketa

Za upravljanje paketima koristi se NPM [10] kao paket menadžer. Potrebno je inicijalizovati projekat. Inicijalizacija generiše potrebne datoteke u projektu. Pored inicijalizacije, dodaju se i potrebni paketi.

Inicijalizacija projekta i instalacija Typescript-a

```
# inicijalizacija projekta
npm init -y
# instalacija typescript paketa
npm install typescript
```

Instalacija ostalih potrebnih paketa

```
# instalacija graphql i express
npm install apollo-server-express express graphql
# instalacija odgovarajucih tipova
npm install -D @types/express @types/graphql @types/node
```

### 4.2.2 Skladištenje podataka

U poglavlju 4.2.2 su definisani podaci aplikacije. Podaci aplikacije su dati u samom kodu, bez upotrebe baze podataka.

Opis podataka je dat u programskom segmentu koji sledi.

```
1 export const users = [  
2   { id: 1, name: 'User 1' },  
3   { id: 2, name: 'User 2' }  
4 ];  
5  
6 export const articles = [  
7   { id: 1, title: 'Article 1',  
8     content: 'Content of Article1', tags: ['Tag1', 'Tag2'],  
9     categories: ['Category1'], likes: 0, authorId: 1  
10  },  
11  { id: 2, title: 'Article 2',  
12    content: 'Content of Article2', tags: ['Tag3', 'Tag4'],  
13    categories: ['Category2', 'Category3'], likes: 0, authorId: 2  
14  },  
15 ];  
16  
17 export const comments = [  
18   { id: 1, content: 'Comment 1', articleId: 1, likes: 0, userId: 2 },  
19   { id: 2, content: 'Comment 2', articleId: 2, likes: 0, userId: 1 },  
20 ];
```

### 4.2.3 Postavljanje Express servera i povezivanje sa serverom Apollo

U poglavlju 4.2.3 koristi se Express kao osnova HTTP servera. Server Apollo se nadograđuje na Express server. Serveru se prosleđuju **shema** i **razrešavači** (eng. **resolvers**).

Port 4000 je postavljen kao port na kome se pokreće aplikacija. Programski kod koji sledi opisuje podizanje servera.

```
1 import express from 'express';  
2 import { ApolloServer } from 'apollo-server-express';  
3 import { typeDefs } from './schema';  
4 import resolvers from './resolvers';  
5  
6 const app = express();
```

## GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

```
7
8 const server = new ApolloServer({ typeDefs, resolvers });
9
10 async function startApolloServer() {
11   await server.start();
12   server.applyMiddleware({ app });
13
14   const port = 4000;
15
16   app.listen(port, () => {
17     console.log(`Server started at http://localhost:${port}/graphql`);
18   });
19 }
20
21 startApolloServer();
```

Potrebni paketi su uključeni linijama 1-4. Express aplikacija je kreirana linijom 6. Serveru Apollo su prosleđeni rešavači 4.2.5 i GraphQL shema 4.2.4 linijom 8. Registracija srednjeg sloja urađena je linijom 12. Na ovaj način su povezani HTTP server i Apollo server. Linijom 16 otvoren je port na kome server prima zahteve od strane klijenata.

### 4.2.4 Definisane sheme

U ovom poglavlju definiše se shema u GraphQL jeziku.

Definisanje sheme za posmatrani primer

```
1 import { gql } from 'apollo-server-express';
2
3 export const typeDefs = gql`
4   type Query {
5     user(id: ID!): User
6     users: [User]
7     article(id: ID!): Article
8     articles: [Article]
9     comment(id: ID!): Comment
10    comments: [Comment]
11  }
```

#### GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

```
12
13 type User {
14     id: ID!
15     name: String
16     articles: [Article]
17 }
18
19 type Article {
20     id: ID!
21     title: String
22     content: String
23     tags: [String]
24     categories: [String]
25     likes: Int
26     authorId: ID!
27     author: User
28     comments: [Comment]
29 }
30
31 type Comment {
32     id: ID!
33     content: String
34     likes: Int
35     articleId: ID!
36     article: Article
37     userId: ID!
38     user: User
39 }
40
41 type Mutation {
42     createUser(name: String!): User
43     updateUser(id: ID!, name: String): User
44     deleteUser(id: ID!): User
45     createArticle (title: String, tags: [String], categories: [String],
46         content: String, authorId: String): Article
47     updateArticle(id: ID!, title: String!, content: String!): Article
48     deleteArticle(id: ID!): Article
```

## GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

```
48     createComment(content: String!, articleId: ID!, userId: ID!): Comment
49     updateComment(id: ID!, content: String!): Comment
50     deleteComment(id: ID!): Comment
51   }
52 ';
```

Upiti su definisani linijama 5-10. Linijama 13, 19, 31 su definisani redom tipovi User, Article i Comment. Mutacije su definisane linijama 42-50.

### 4.2.5 Definisanje razrešavača

Prilikom rada sa GraphQL-om neophodno je definisati funkcije rešavače. Naravno, pored običnog vraćanja podataka, funkcija razrešavač može višiti i modifikacije.

Programski kod koji sledi predstavlja razrešavač za problem koji se proučava.

```
1 import { users, articles, comments } from './data';
2
3 const resolvers = {
4   Query: {
5     user: (_, { id }: any) => users.find((user) => user.id === +id),
6     users: () => users,
7     article: (_, { id }: any) => articles.find((article) => article.id
8       === +id),
9     articles: () => articles,
10    comment: (_, any, { id }: any) => comments.find((comment) => comment.id
11      === +id),
12    comments: () => comments,
13  },
14  User: {
15    articles: (user: any) => articles.filter((article) => article.authorId
16      === +user.id),
17  },
18  Article: {
19    author: (article: any) => users.find((user) => user.id === +article.
20      authorId),
21    comments: (article: any) => comments.filter((comment) => comment.
22      articleId === +article.id),
23  },
24 }
```



#### GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

```
19   Comment: {
20     article: (comment: any) => articles.find((article) => article.id === +
21       comment.articleId),
22   },
23   Mutation: {
24     createUser: (parent: any, { name }: { name: string }) => {
25       const newUser = {
26         id: users.length + 1,
27         name,
28       };
29       users.push(newUser);
30
31       return newUser;
32     },
33     updateUser: (parent: any, { id, name }: { id: number; name: string })
34       => {
35       const user = users.find((user) => user.id === id);
36       if (!user) {
37         throw new Error('User not found');
38       }
39       user.name = name;
40
41       return user;
42     },
43     deleteUser: (parent: any, { id }: { id: number }) => {
44       const index = users.findIndex((user) => user.id === id);
45       if (index === -1) {
46         throw new Error('User not found');
47       }
48       const deletedUser = users.splice(index, 1)[0];
49
50       return deletedUser;
51     },
52     createArticle: (parent: any, { title, tags, categories, content,
53       authorId }: { title: string; content: string; tags: string[];
54       categories: string[]; authorId: number }) => {
```

#### GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

```
52     const newArticle = {
53       id: articles.length + 1,
54       title,
55       content,
56       tags,
57       categories,
58       likes: 0,
59       authorId,
60     };
61
62     articles.push(newArticle);
63
64     return newArticle;
65   },
66   updateArticle: (parent: any, { id, title, content, tags, categories }:
67     { id: number; title: string; content: string; tags:string[],
68     categories: string[]}) => {
69     const article = articles.find((article) => article.id === id);
70
71     if (!article) {
72       throw new Error('Article not found');
73     }
74
75     article.title = title;
76     article.content = content;
77     article.tags = tags;
78     article.categories = categories
79
80     return article;
81   },
82   deleteArticle: (parent: any, { id }: { id: number }) => {
83     const index = articles.findIndex((article) => article.id === id);
84
85     if (index === -1) {
86       throw new Error('Article not found');
87     }
88   }
```

#### GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

```
87     const deletedArticle = articles.splice(index, 1)[0];
88
89     return deletedArticle;
90 },
91 createComment: (parent: any, { content, articleId, userId }: { content:
92     string; articleId: number; userId: number }) => {
93     const newComment = {
94     id: comments.length + 1,
95     content,
96     likes: 0,
97     articleId,
98     userId,
99     };
100     comments.push(newComment);
101
102     return newComment;
103 },
104 updateComment: (parent: any, { id, content }: { id: number; content:
105     string }) => {
106     const comment = comments.find((comment) => comment.id === +id);
107     if (!comment) {
108     throw new Error('Comment not found');
109     }
110     comment.content = content;
111
112     return comment;
113 },
114 deleteComment: (parent: any, { id }: { id: number }) => {
115     const index = comments.findIndex((comment) => comment.id === +id);
116     if (index === -1) {
117     throw new Error('Comment not found');
118     }
119     const deletedComment = comments.splice(index, 1)[0];
120     return deletedComment;
121 },
```

## GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

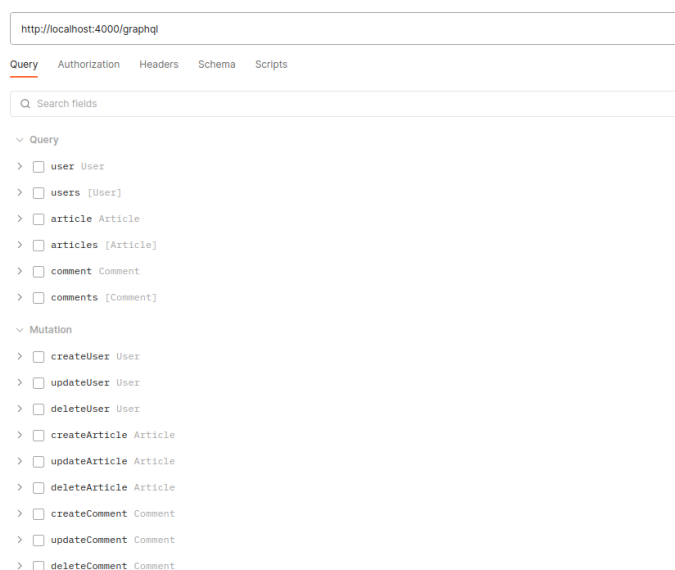
```
122   }
123 };
124
125 export default resolvers;
```

Razrešavači koji se odnose na upite definisani su linijama 5-20 i oni definišu način dohvaćanja podataka. Razrešavači koji se odnose na mutacije definisani su linijama 23-120 i predstavljaju način modifikovanja podataka.

### 4.2.6 Upotreba Postman-a za slanje zahteva ka serveru

Kako bi bilo izbegnuto razvijanje klijentske aplikacije, koja u ovom razmatranju ne bi imala značaj, koristi se Postman.

Otvaranjem Postman-a može se videti spisak svih mogućih upita i mutacija. Ovaj ispis nastaje iz čitanja odgovarajuće GraphQL sheme. Prikaz svih njih dat je na slici 4.1. Sva komunikacija sa serverom se odvija preko jedinstvene krajnje tačke kao što je prikazano u narednom primeru.



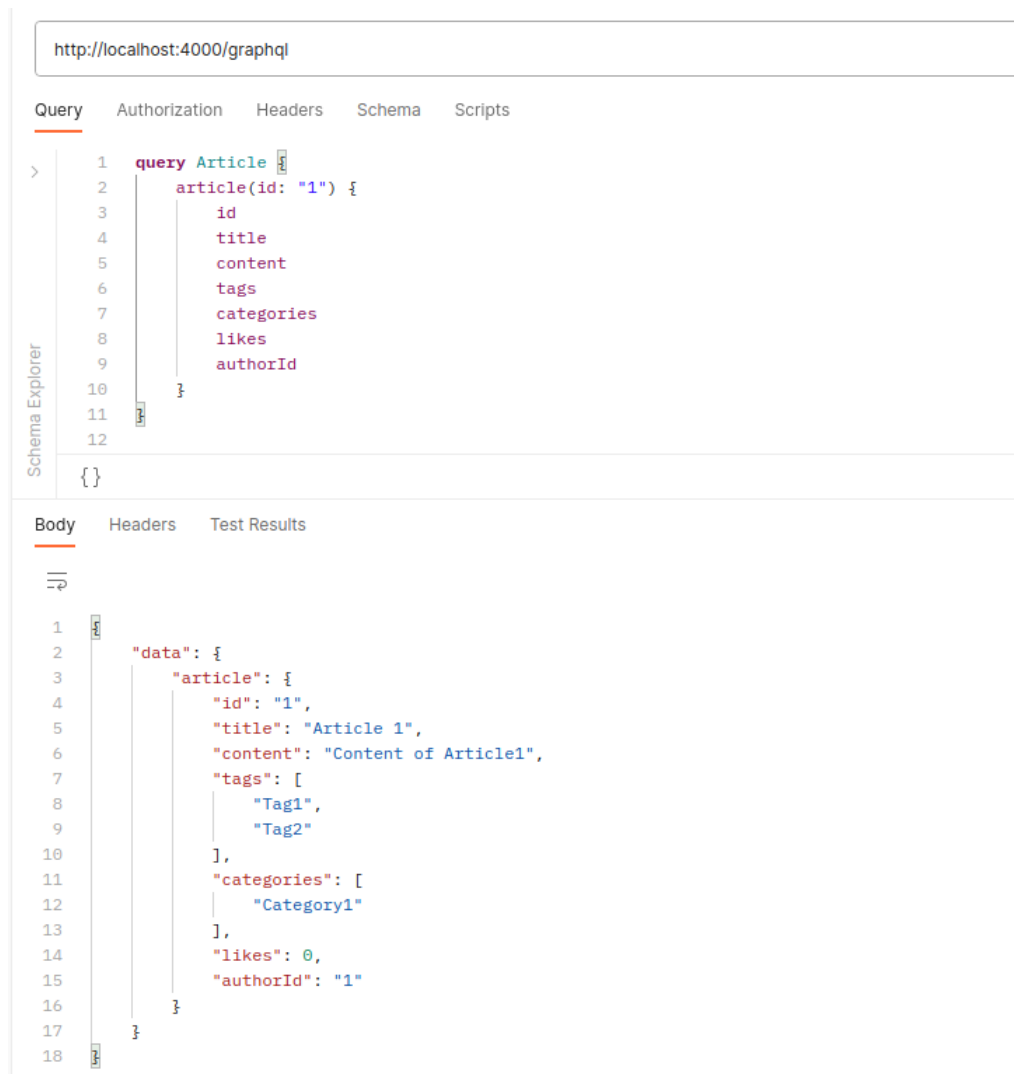
Slika 4.1: Prikaz upita i mutacija

#### 4.2.6.1 Dohvaćanje članka preko identifikatora

U primeru 4.2, napisan je upit za dohvaćanje članka čiji identifikator je 1. Dohvaćanje članka sa drugim identifikatorom se postiže jednostavnom promenom vrednosti

## GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

ovog parametra u upitu. U upitu su navedena polja koja se dohvataju. Struktura odgovora sa servera odgovara upitu koji je klijent poslao. Dohvatanje samo određenih polja članka, prikazano je u primeru 4.2.6.2.

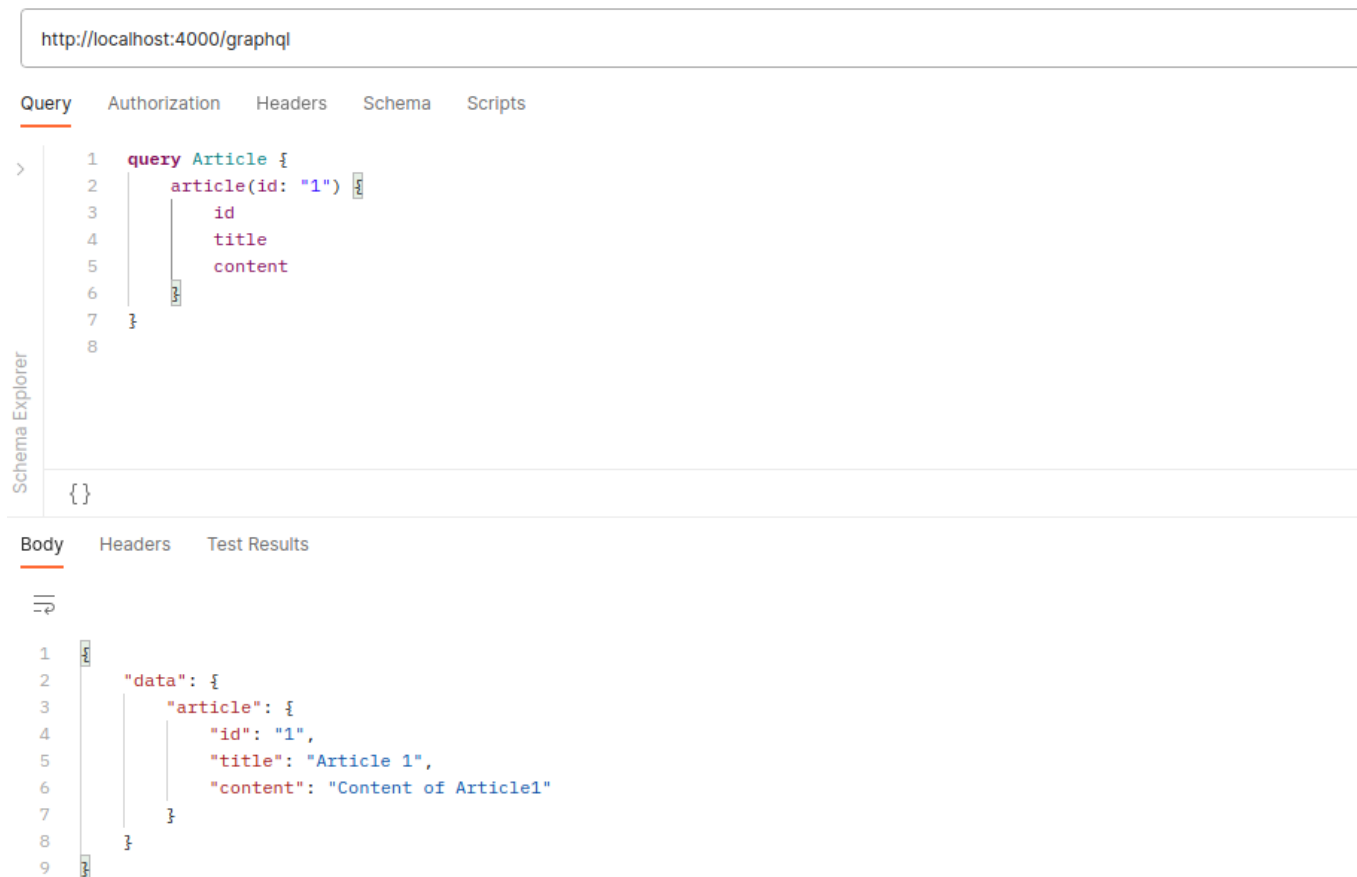


Slika 4.2: Upit dohvatanja članka sa parametrom id

### 4.2.6.2 Dohvatanje članka preko identifikatora sa specifičnim poljima

Zahvaljujući GraphQL upitu, mogu se zahtevati samo potrebna polja. Na ovaj način, izbegava se prekomerno preuzimanje. U primeru 4.3, dohvata se samo **id**, **naslov** i **sadržaj** određenog članka. Ako postoji potreba da se dohvati još neko polje, to je moguće uraditi vrlo jednostavno navođenjem datog polja u upitu. Server je u odgovoru dostavio tačno tražena polja.

## GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

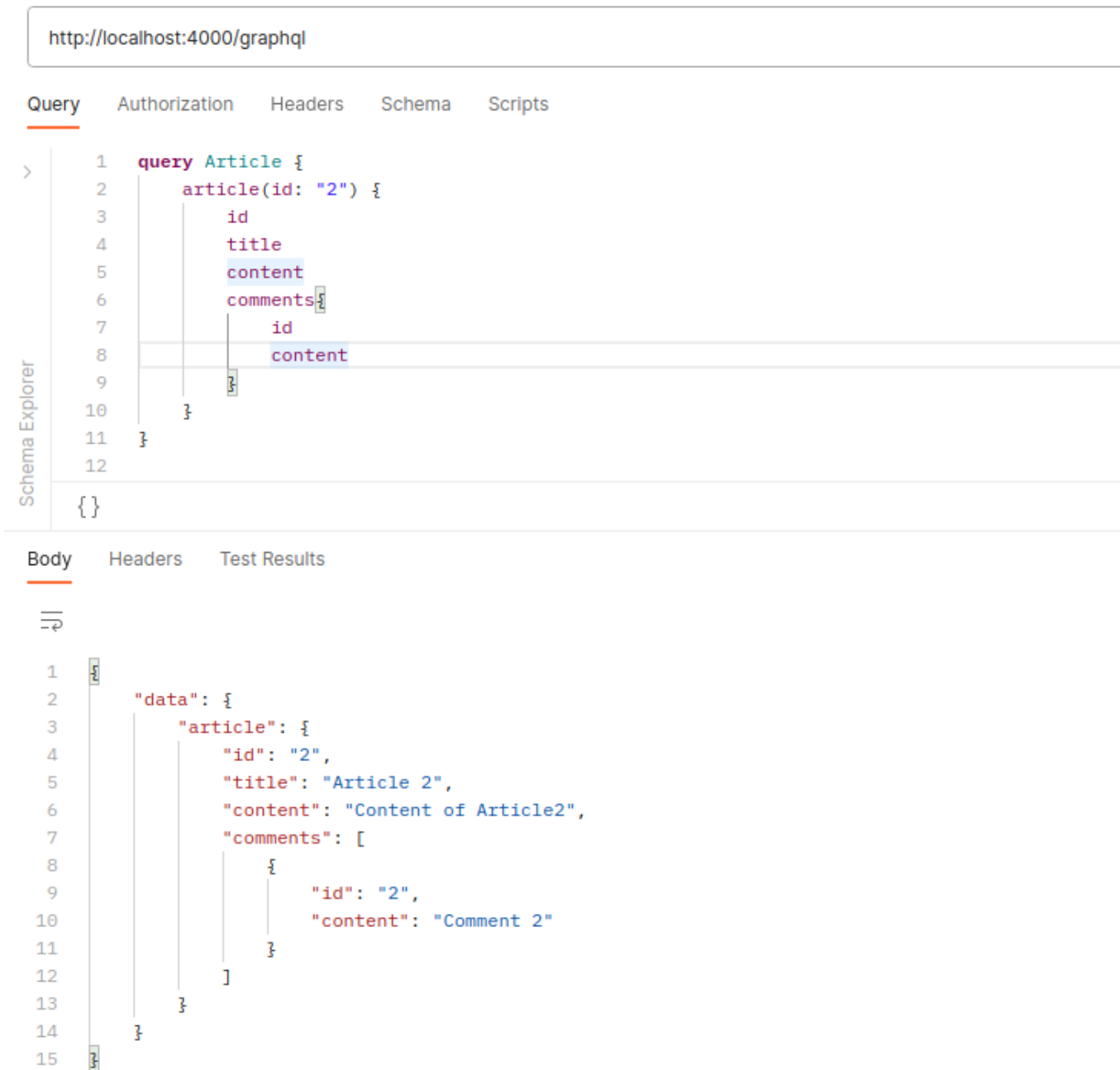


Slika 4.3: Upit dohvaćanja članka i određenih polja

### 4.2.6.3 Dohvaćanje članka i komentara

Ako postoji potreba za prikazivanjem komentara nekog članka, to je takođe moguće uraditi vrlo jednostavno. Potrebno je napraviti ugnježđen upit članka i komentara. U članku i komentaru navode se polja koja su potrebna klijentu i na ovaj način se izbegava prekomerno preuzimanje. Prikaz takvog upita i njegovog rezultata je dat na slici 4.4.

## GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE



The screenshot displays the GraphQL Playground interface. At the top, the URL is `http://localhost:4000/graphql`. The interface is divided into several tabs: **Query**, **Authorization**, **Headers**, **Schema**, and **Scripts**. The **Query** tab is active, showing the following query:

```
1 query Article {
2   article(id: "2") {
3     id
4     title
5     content
6     comments {
7       id
8       content
9     }
10  }
11 }
12
```

Below the query editor, the **Body** tab is active, showing the JSON response:

```
1 {
2   "data": {
3     "article": {
4       "id": "2",
5       "title": "Article 2",
6       "content": "Content of Article2",
7       "comments": [
8         {
9           "id": "2",
10          "content": "Comment 2"
11        }
12      ]
13    }
14  }
15 }
```

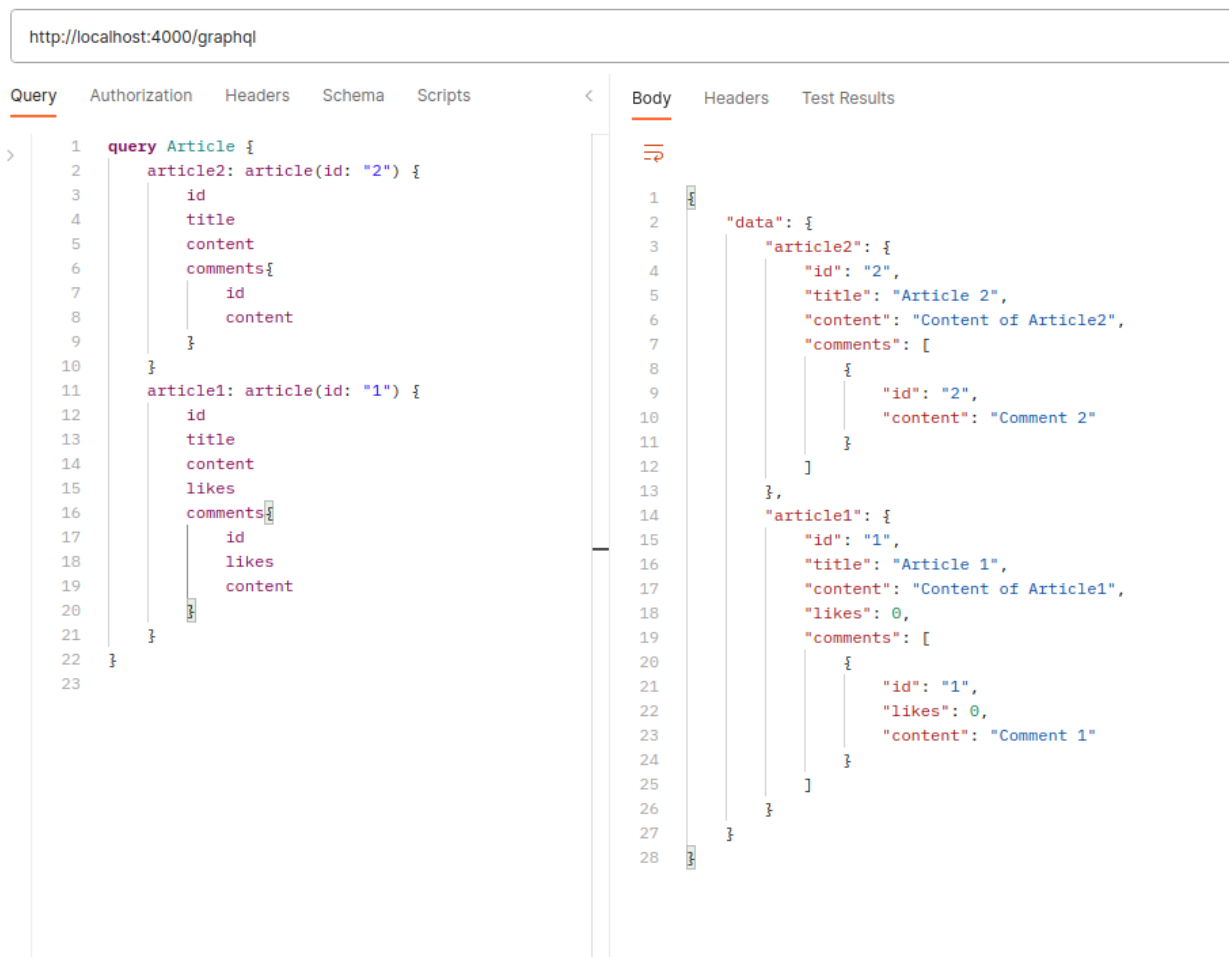
Slika 4.4: Upit dohvanjanja članka i njegovih komentara

### 4.2.6.4 Dohvatanje više članaka

Ako postoji potreba da se dohvati više članaka po identifikatoru to je moguće uraditi upotrebom aliasa. Prikaz takvog upita i njegovog rezultata dat je na slici 4.5. Polja koja se vraćaju prilikom dohvanjanja ova dva članka se razlikuju za polje

## GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

**likes**. Treba napomenuti da broj aliasa nije ograničen na dva u upitu i moguće je dohvatiti više od dva članka. Ovo demonstrira veoma veliku fleksibilnost GraphQL upita.



```
http://localhost:4000/graphql

Query Authorization Headers Schema Scripts Body Headers Test Results

1 query Article {
2   article2: article(id: "2") {
3     id
4     title
5     content
6     comments {
7       id
8       content
9     }
10  }
11  article1: article(id: "1") {
12    id
13    title
14    content
15    likes
16    comments {
17      id
18      likes
19      content
20    }
21  }
22 }
23

1 "data": {
2   "article2": {
3     "id": "2",
4     "title": "Article 2",
5     "content": "Content of Article2",
6     "comments": [
7       {
8         "id": "2",
9         "content": "Comment 2"
10      }
11    ]
12  },
13  "article1": {
14    "id": "1",
15    "title": "Article 1",
16    "content": "Content of Article1",
17    "likes": 0,
18    "comments": [
19      {
20        "id": "1",
21        "likes": 0,
22        "content": "Comment 1"
23      }
24    ]
25  }
26 }
27 }
```

Slika 4.5: Korišćenje aliasa

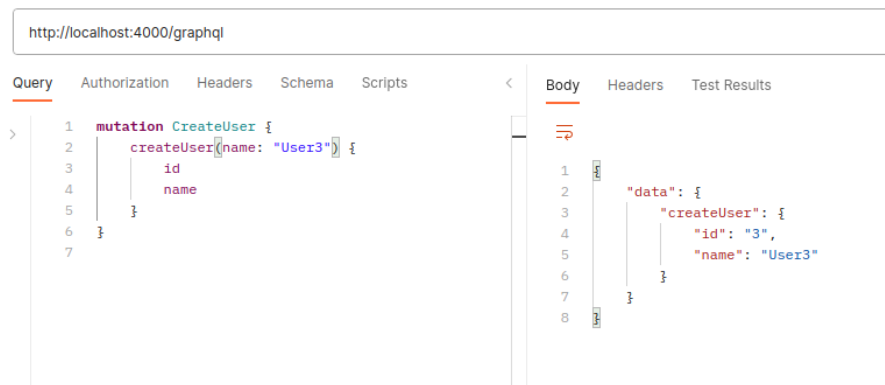
### 4.2.6.5 Dodavanje novog korisnika

Pored prikazanih upita za dohvaćanje podataka, moguće je napisati i mutacije za dodavanje i modifikaciju podataka. Ime novog korisnika se prosleđuje kao argument, a identifikator novog korisnika se generiše automatski od strane servera. Kada je novi korisnik uspešno dodat, server kao odgovor vraća korisnika sa poljima koja su prethodno navedena u upitu.

Primer 4.6 pokazuje kako je moguće dodati novog korisnika.



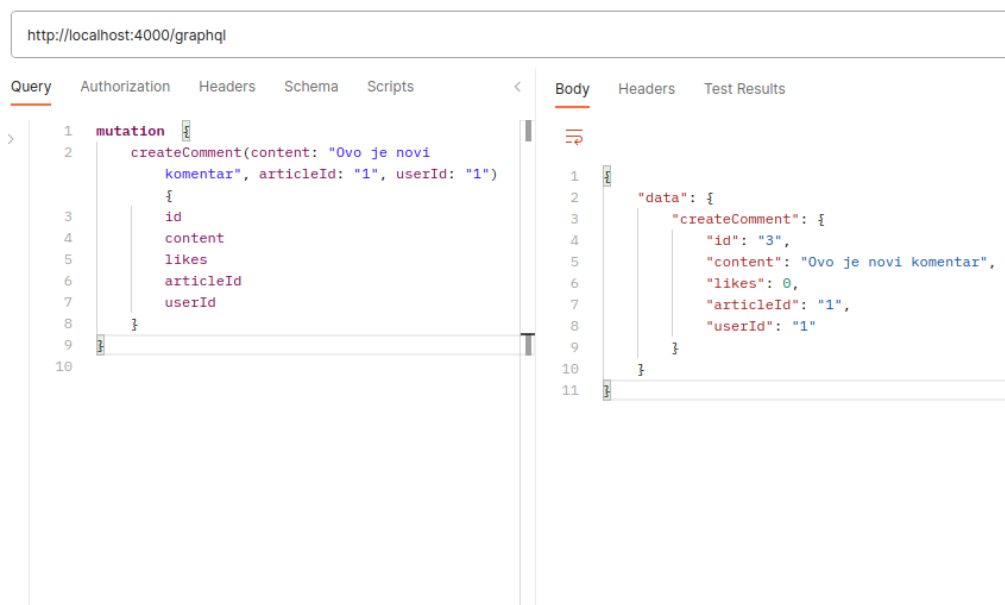
## GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE



Slika 4.6: Pravljenje korisnika uz pomoć mutacije `createUser`

### 4.2.6.6 Dodavanje novog komentara

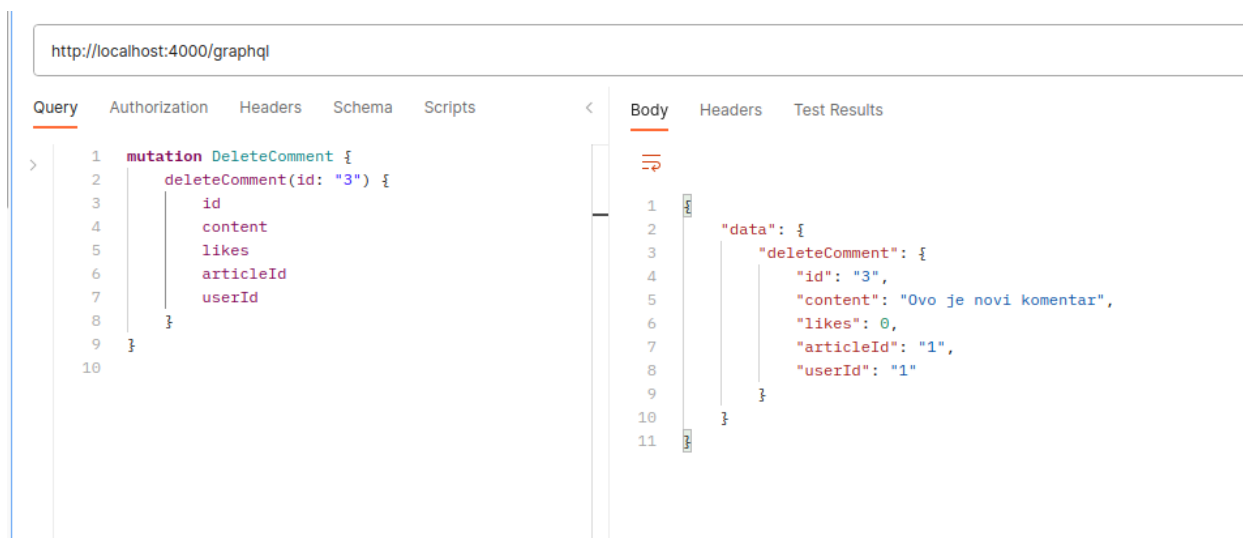
Vrlo je jednostavno dodati i komentar na članak. Prilikom dodavanja novog komentara, prosleđuje se sadržaj komentara, identifikator članka koji se komentariše kao i identifikator korisnika koji ostavlja komentar. Identifikator novog komentara se generiše automatski od strane servera. Kada je novi komentar uspešno dodan, server kao odgovor vraća komentar sa poljima koja su navedena u upitu. Dodavanje komentara je ostvareno narednom mutacijom.



Slika 4.7: Pravljenje komentara uz pomoć mutacije `createComment`

#### 4.2.6.7 Brisanje komentara

Prethodno dodati komentar je vrlo jednostavno i obrisati, koristeći **deleteComment** mutaciju. Kao argument mutacije se prosleđuje identifikator komentara koji je potrebno obrisati. Nakon brisanja komentara, server vraća obrisani komentar sa poljima koja su definisana u prethodno pokrenutoj mutaciji. Ovakva operacija je prikazana na slici 4.8.



Slika 4.8: Brisanje korisnika uz pomoć mutacije deleteComment

### 4.3 Pristup korišćenjem REST arhitekture

U ovom poglavlju je ranije prikazana aplikacija reimplementirana kao REST arhitektura. U prikazanom rešenju, koristi se Express za pokretanje HTTP servera. Posebno su razdvojeni sloj za rute i sloj servisa, u kome se nalazi poslovna logika.

Cela aplikacija je dostupna na autorovom Github repozitorijumu kao softver otvorenog koda [6].

#### 4.3.1 Inicijalizacija projekta i instalacija paketa

Za upravljanje paketima koristi se NPM kao paket menadžer.

Inicijalizacija projekta i instalacija TS

```
npm init -y #inicijalizacija projekta
npm install typescript #instalacija typescript paketa
```

Instalacija ostalih potrebnih biblioteka

```
npm install express #instalacija express
npm install -D @types/express @types/node #instalacija tipova
```

### 4.3.2 Postavljanje Express servera

U ovom poglavlju je dat opis podizanja Express servera, registrovane su ruta za korisnike, članke i komentare. Rute su registrovane kao funkcije srednjeg sloja prikazane aplikacije.

Podizanje servera

```
1 import express, { Application } from 'express';
2 import usersRouter from './routes/users';
3 import articlesRouter from './routes/articles';
4 import commentsRouter from './routes/comments';
5
6 const app: Application = express();
7 app.use(express.json());
8
9 app.use('/users', usersRouter);
10 app.use('/articles', articlesRouter);
11 app.use('/comments', commentsRouter);
12
13 const port = 4000;
14 app.listen(port, () => {
15   console.log('Server listening on port ${port}');
16 });
```

Potrebni paketi uključeni su linijama 1-4. Rute za korisnike, članke i komentare su registrovane redom linijama 9, 10, 11. Server je pokrenut da sluša zahteve na portu 4000 linijom 14.

### 4.3.3 Članci - rute

U ovom poglavlju prikazane su rute uz pomoć kojih je moguće obavljati operacije nad člancima.

```
1 import express, { Request, Response, Router } from 'express';
```

#### GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

```
2 import { getArticles, getArticleById, createArticle, updateArticle,
    deleteArticle, likeArticle } from '../services/articleService';
3 import { getCommentsByArticleId } from '../services/commentService';
4
5 const router: Router = express.Router();
6
7 router.get('/', (req: Request, res: Response) => {
8     const articles = getArticles();
9     res.json(articles);
10 });
11
12 router.get('/:articleId', (req: Request, res: Response) => {
13     const articleId: number = parseInt(req.params.articleId);
14     const article = getArticleById(articleId);
15     if (!article) {
16         return res.status(404).json({ message: 'Article not found' });
17     }
18     res.json(article);
19 });
20
21
22 router.get('/:articleId/comments', (req: Request, res: Response) => {
23     const articleId: number = parseInt(req.params.articleId);
24     const comments = getCommentsByArticleId(articleId);
25
26     res.json(comments);
27 });
28
29
30 router.post('/', (req: Request, res: Response) => {
31     const { title, content, tags, categories, userId } = req.body;
32     const newArticle = createArticle(title, content, tags || [], categories
        || [], userId);
33     res.status(201).json(newArticle);
34 });
35
36
```

#### GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

```
37
38 router.put('/:articleId', (req: Request, res: Response) => {
39   const articleId: number = parseInt(req.params.articleId);
40   const { title, content, tags, categories } = req.body;
41
42   const success = updateArticle(articleId, title, content, tags || [],
43     categories || []);
44
45   if (success) {
46     res.json({ message: 'Article updated successfully' });
47   } else {
48     res.status(404).json({ message: 'Article not found' });
49   }
50 });
51
52 router.delete('/:articleId', (req: Request, res: Response) => {
53   const articleId: number = parseInt(req.params.articleId);
54   const success = deleteArticle(articleId);
55   if (success) {
56     res.json({ message: 'Article deleted successfully' });
57   } else {
58     res.status(404).json({ message: 'Article not found' });
59   }
60 });
61
62
63 router.post('/articles/:articleId/like', (req: Request, res: Response) => {
64
65   const articleId: number = parseInt(req.params.articleId);
66   const success = likeArticle(articleId);
67   if (success) {
68     res.json({ message: 'Article liked successfully' });
69   } else {
70     res.status(404).json({ message: 'Article not found' });
71   }
72 });
```

## GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

```
72  
73  
74 export default router;
```

Ruta za dohvaćanje svih članaka definisana je linijom 7. Pored navedene rute, članak je moguće dohvatiti i po identifikatoru uz pomoć rute definisane linijom 12. Linijom 30 definisana je ruta za kreiranje novog članka. Dohvaćanje svih komentara određenog članka implementirano je u ruti linijom 22. Rute za brisanje i ažuriranje članka implementirane su linijama 38, odnosno 52. Linijom 63 implementirana je ruta za ostavljanje sviđanja na članak.

### 4.3.4 Korisnici - rute

U ovom poglavlju prikazane su rute uz pomoć kojih je moguće obavljati operacije nad korisnicima.

```
1 import express, { Request, Response, Router } from 'express';  
2 import { getUsers, getUserById, createUser, updateUser, deleteUser } from '  
  ../services/userService';  
3  
4 const router: Router = express.Router();  
5  
6  
7 router.get('/', (req: Request, res: Response) => {  
8   const users = getUsers();  
9   res.json(users);  
10 });  
11  
12  
13 router.get('/:userId', (req: Request, res: Response) => {  
14   const userId: number = parseInt(req.params.userId);  
15   const user = getUserById(userId);  
16   if (!user) {  
17     return res.status(404).json({ message: 'User not found' });  
18   }  
19   res.json(user);  
20 });  
21
```

#### GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

```
22 // Pravljenje novog korisnika
23 router.post('/', (req: Request, res: Response) => {
24     const { name } = req.body;
25     const newUser = createUser(name);
26     res.status(201).json(newUser);
27 });
28
29
30 router.put('/users/:userId', (req: Request, res: Response) => {
31     const userId: number = parseInt(req.params.userId);
32     const { name } = req.body;
33     const success = updateUser(userId, name);
34     if (success) {
35         res.json({ message: 'User updated successfully' });
36     } else {
37         res.status(404).json({ message: 'User not found' });
38     }
39 });
40
41
42 router.delete('/users/:userId', (req: Request, res: Response) => {
43     const userId: number = parseInt(req.params.userId);
44     const success = deleteUser(userId);
45     if (success) {
46         res.json({ message: 'User deleted successfully' });
47     } else {
48         res.status(404).json({ message: 'User not found' });
49     }
50 });
51
52 export default router;
```

Ruta za dohvaćanje svih korisnika implementirana je linijom 7. Dohvaćanje korisnika po identifikatoru implementirano je linijom 13. Kreiranje, ažuriranje i brisanje korisnika implementirano je u rutama linijama 23, 30, 42.

### 4.3.5 Komentari - rute

U ovom poglavlju prikazane su rute uz pomoć kojih je moguće obavljati operacije nad komentarima.

```
1 import express, { Request, Response, Router } from 'express';
2 import { createComment, deleteComment, getAllComments, getCommentById,
   getCommentsByArticleId, updateComment } from '../services/
   commentService';
3
4 const router: Router = express.Router();
5
6 router.get('/', (req: Request, res: Response) => {
7   const comments = getAllComments();
8
9   res.json(comments);
10 });
11
12
13 router.get('/:commentId', (req: Request, res: Response) => {
14   const commentId: number = parseInt(req.params.commentId);
15   const comment = getCommentById(commentId);
16   if (comment) {
17     res.json(comment);
18   } else {
19     res.status(404).json({ message: 'Comment not found' });
20   }
21 });
22
23
24 router.post('/', (req: Request, res: Response) => {
25   const { content, articleId, userId } = req.body;
26   const newComment = createComment(content, articleId, userId);
27
28   res.status(201).json(newComment);
29 });
30
31
32 router.put('/:commentId', (req: Request, res: Response) => {
```



## GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

```
33     const commentId: number = parseInt(req.params.commentId);
34     const { content } = req.body;
35     const success = updateComment(commentId, content);
36
37     if (success) {
38       res.json({ message: 'Comment updated successfully' });
39     } else {
40       res.status(404).json({ message: 'Comment not found' });
41     }
42   });
43
44
45   router.delete('/:commentId', (req: Request, res: Response) => {
46     const commentId: number = parseInt(req.params.commentId);
47     const success = deleteComment(commentId);
48     if (success) {
49       res.json({ message: 'Comment deleted successfully' });
50     } else {
51       res.status(404).json({ message: 'Comment not found' });
52     }
53   });
54
55   export default router;
```

Ruta za dohvaćanje svih komentara implementirana je linijom 6. Dohvaćanje komentara po identifikatoru implementirano je linijom 13. Kreiranje, ažuriranje i brisanje komentara implementirano je u rutama linijama 24, 32, 45.

### 4.3.6 Korisnici - servis

U ovom servisu se nalazi logika poslovnog domena za korisnike.

```
1 import { User } from '../models/user';
2
3 export const getUsers = (): User[] => {
4   return users;
5 };
6
7 export const getUserById = (id: number): User | undefined => {
```

#### GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

```
8     return users.find(user => user.id === id);
9 };
10
11 export const createUser = (name: string): User =>{
12     const newUser: User = {
13         id: users.length + 1, // id se dodeljuje po principu inkrementiranja
14         name,
15     };
16
17     users.push(newUser);
18
19     return newUser;
20 }
21
22 export const updateUser = (userId: number, name: string): boolean => {
23     const userIndex = users.findIndex((user) => user.id === userId);
24
25     if (userIndex === -1) {
26         return false;
27     }
28
29     users[userIndex].name = name;
30
31     return true;
32 }
33
34 export const deleteUser = (userId: number): boolean => {
35     const userIndex = users.findIndex((user) => user.id === userId);
36
37     if (userIndex === -1) {
38         return false;
39     }
40
41     users = users.filter((user) => user.id !== userId);
42
43     return true;
44 }
```

Funkcija za dohvatanje svih korisnika implementirana je linijom 3. Dohvatanje korisnika po identifikatoru implementirano je linijom 7. Kreiranje, ažuriranje i brisanje korisnika implementirano je u funkcijama linijama 11, 22, 34. Sve operacije se realizuju manipulacijom nad nizovima.

### 4.3.7 Članci - servis

U ovom servisu se nalazi logika poslovnog domena za članke.

```
1 import { Article } from '../models/article';
2
3 let articles: Article[] = [
4   { id: 1, title: 'Article 1', content: 'Content of Article1', tags: ['
5     Tag1', 'Tag2'], categories: ['Category1'], likes: 0, authorId: 1 },
6   { id: 2, title: 'Article 2', content: 'Content of Article2', tags: ['
7     Tag3', 'Tag4'], categories: ['Category2', 'Category3'], likes: 0,
8     authorId: 2 },
9 ];
10
11 export const getArticles = (): Article[] => {
12   return articles;
13 };
14
15 export const getArticleById = (id: number): Article | undefined => {
16   return articles.find(article => article.id === id);
17 };
18
19 export const createArticle = (title: string, content: string, tags: string
20   [], categories: string[], authorId: number): Article =>{
21   const newArticle: Article = {
22     id: articles.length + 1,
23     title,
24     content,
25     tags,
```

#### GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

```
26
27     articles.push(newArticle);
28
29     return newArticle;
30 }
31
32 export const updateArticle = (articleId: number, title: string, content:
33     string, tags: string[], categories: string[]): boolean => {
34
35     const articleIndex = articles.findIndex((article) => article.id ===
36         articleId);
37
38     if (articleIndex === -1) {
39         return false;
40     }
41
42     articles[articleIndex].title = title;
43     articles[articleIndex].content = content;
44     articles[articleIndex].tags = tags;
45     articles[articleIndex].categories = categories;
46
47     return true;
48 }
49
50 export const deleteArticle = (articleId: number): boolean => {
51     const articleIndex = articles.findIndex((article) => article.id ===
52         articleId);
53
54     if (articleIndex === -1) {
55         return false;
56     }
57
58     articles = articles.filter((article) => article.id !== articleId);
59
60     return true;
61 }
62
63 export const likeArticle = (articleId: number): boolean => {
```

## GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

```
60     const articleIndex = articles.findIndex((article) => article.id ===
        articleId);
61
62     if (articleIndex === -1) {
63         return false;
64     }
65
66     articles[articleIndex].likes++;
67
68     return true;
69 }
```

Funkcija za dohvatanje svih članaka implementirana je linijom 8. Dohvatanje članka po identifikatoru implementirano je linijom 12. Kreiranje, ažuriranje i brisanje članka implementirano je u funkcijama linijama 16, 32, 47. Funkcija za dodavanje sviđanja na članak implementirana je linijom 59. Sve operacije se realizuju manipulacijom nad nizovima.

### 4.3.8 Komentari - servis

U ovom servisu se nalazi logika poslovnog domena za komentare.

```
1 import { Comment } from '../models/comment';
2
3 let comments: Comment[] = [
4     { id: 1, content: 'Comment 1', articleId: 1, likes: 0, userId: 2 },
5     { id: 2, content: 'Comment 2', articleId: 2, likes: 0, userId: 1 },
6 ];
7
8 export const getAllComments = (): Comment[] => {
9     return comments;
10 }
11
12 export const getCommentById = (commentId: number): Comment[] => {
13     return comments.filter(comment => comment.id === commentId);
14 }
15
16 export const getCommentsByArticleId = (articleId: number): Comment[] => {
17     return comments.filter(comment => comment.articleId === articleId);
```

#### GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

```
18 };
19
20 export const createComment = (content: string, articleId: number, userId:
    number): Comment => {
21     const newComment: Comment = {
22         id: comments.length + 1,
23         content,
24         likes: 0,
25         articleId,
26         userId,
27     };
28
29     comments.push(newComment);
30
31     return newComment;
32 }
33
34 export const updateComment = (commentId: number, content: string): boolean
    => {
35     const commentIndex = comments.findIndex((comment) => comment.id ===
        commentId);
36     if (commentIndex === -1) {
37         return false;
38     }
39
40     comments[commentIndex].content = content;
41
42     return true;
43 }
44
45 export const deleteComment = (commentId: number): boolean => {
46     const commentIndex = comments.findIndex((comment) => comment.id ===
        commentId);
47
48     if (commentIndex === -1) {
49         return false;
50     }
```

## GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE

```
51  
52     comments = comments.filter((comment) => comment.id !== commentId);  
53  
54     return true;  
55 }
```

Funkcija za dohvatanje svih komentara implementirana je linijom 8. Dohvatanje komentara po identifikatoru implementirano je u funkciji linijom 12. Dohvatanje komentara sa jednog članka implementirano je u funkciji linijom 16. Kreiranje, ažuriranje i brisanje komentara implementirano je u funkcijama linijama 20, 34, 45. Sve operacije se realizuju manipulacijom nad nizovima.

### 4.3.9 Upotreba Postman-a za slanje zahteva ka serveru

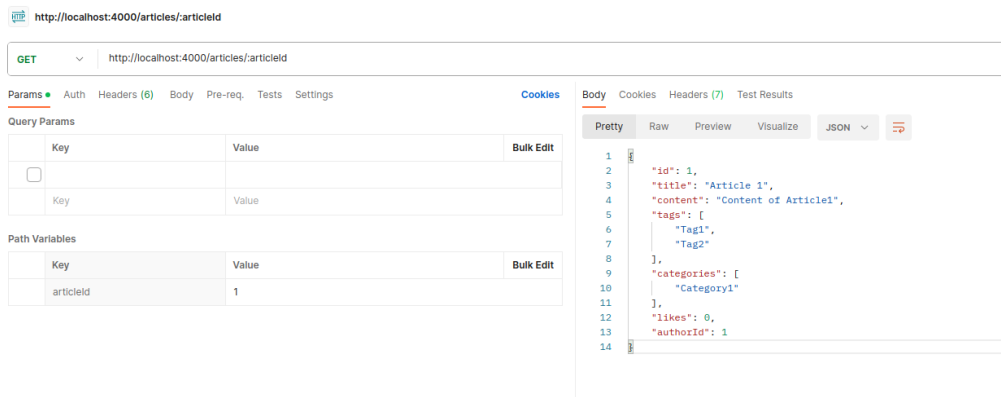
U poglavlju 4.2.6 prikazana je upotreba Postman-a prilikom slanja upita ka GraphQL-u, odnosno ka krajnjoj tački **/graphql**. U ovom poglavlju dat je prikaz kako se Postman može koristiti sa REST arhitekturom, kada aplikacija raspolaže sa više krajnjih tačaka, što je u duhu ove arhitekture.

#### 4.3.9.1 Dohvatanje članka po identifikatoru

U primeru 4.9 dohvata se članak sa odgovarajućim identifikatorom. Za ovo dohvatanje koristi se specifična krajnja tačka, za razliku od ranije prikazanog u pristupu sa GraphQL-om. Ranije je svaki zahtev išao preko unapred definisane krajnje tačke koja se koristila kao jedina krajnja tačka u komunikaciji servera i klijenta.

Treba primetiti da ova krajnja tačka vraća unapred predefinisani spisak polja članka i da klijent ne može da utiče na njih. Ovde se javlja problem prekomernog preuzimanja. Moguće je dodati nove krajnje tačke koje vraćaju neke podskupove ovih podataka i na taj način smanjiti prekomerno preuzimanje.

## GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE



Slika 4.9: Upit dohvanjanja članka po `articleId` u REST arhitekturi

### 4.3.9.2 Dohvatanje dva članka

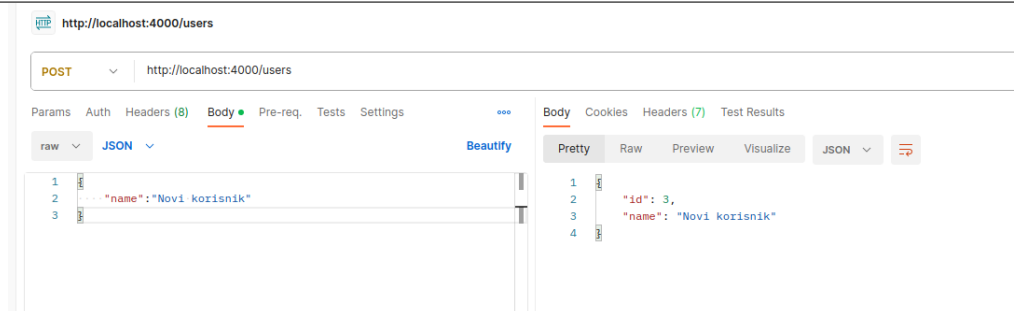
Ako postoji potreba da se dohvate dva članka, to je vrlo lako moguće uraditi u GraphQL-u, kao što je i prikazano u primeru 4.2.6.4. Međutim, u REST arhitekturi je potrebno dodati ili novu krajnju tačku koja prima niz identifikatora ili je potrebno zvati postojeću krajnju tačku dva puta sa različitim identifikatorima. U slučaju dodavanja nove krajnje tačke, broj krajnjih tačaka raste, što nije bio slučaj kod GraphQL arhitekture. U slučaju promene logike krajnje tačke da primi identifikator, logika na serveru postaje složenija. U navedenim slučajevima, u REST arhitekturi, promenu je neophodno napraviti na serveru. U slučaju GraphQL arhitekture, ovo se prevazilazi odgovarajućim upitom na klijentskoj strani.

### 4.3.9.3 Dodavanje novog korisnika

Dodavanje novog korisnika može se ostvariti slanjem zahteva na odgovarajuću krajnju tačku. U telu zahteva prosleđuje se ime za novog korisnika. Server kao odgovor vraća korisnika koji je napravljen. U ovom slučaju, u zahtevu ne postoji mogućnost navođenja koja polja server treba da vrati. Server vraća korisnika sa svim poljima. Prikaz zahteva i rezultata dat je na slici 4.10.



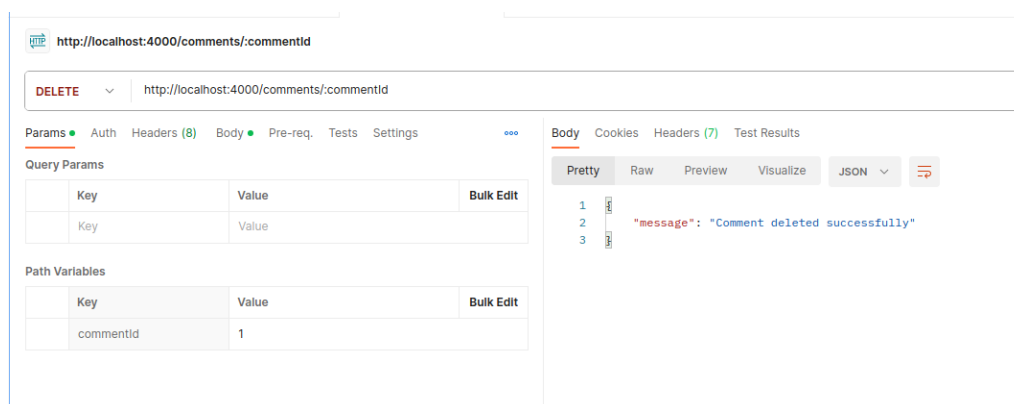
## GLAVA 4. REŠAVANJE PROBLEMA IZ REALNOG SVETA UPOTREBOM GRAPHQL-A I REST ARHITEKTURE



Slika 4.10: Dodavanje novog korisnika

### 4.3.9.4 Brisanje komentara

Brisanje komentara sa nekog članka može se ostvariti slanjem zahteva na odgovarajuću krajnju tačku. Identifikator komentara koji je potrebno obrisati se šalje kroz URL. Kao rezultat brisanja, server vraća poruku o uspešnom brisanju. Umesto poruke o uspešnom brisanju, ova funkcija na serveru je mogla biti implementirana na način da se vrati obrisani komentar. Prikaz zahteva i rezultata dat je na slici 4.11.



Slika 4.11: Brisanje komentara sa članka

## Glava 5

# Komparativna analiza GraphQL-a i REST API-ja

U ovom poglavlju prikazana je komparativna analiza GraphQL-a i REST API-ja kroz: **dizajn koda**, **performanse**, **skalabilnost** i **bezbednost**. Razmatranje ovih aspekata je ključno za razumevanje osobina i odnosa ova dva pristupa.

Komparativna analiza treba da pruži pomoć dizajneru API-ja prilikom izbora jednog od ova dva pristupa tokom razvoja API-ja serverske aplikacije.

### 5.1 Dizajn koda

Tokom razvoja softvera, neophodno je posebnu pažnju posvetiti dizajnu koda. Odabir GraphQL-a ili REST arhitekture ima značajan uticaj na dizajn koda.

Dizajn koda se u ovom poglavlju posmatra kroz sledeće aspekte: strukture krajnjih tačaka, strukture odgovora koje šalje server i verzionisanje.

#### 5.1.1 Struktura krajnjih tačaka

Broj krajnjih tačaka u ova dva pristupa je različit. U pristupu zasnovanom na GraphQL-u, cela komunikacija klijenta i servera odvija se preko jedne krajnje tačke. U pristupu zasnovanom na REST arhitekturi, postoji veliki broj krajnjih tačaka, od kojih svaka ima određenu funkcionalnost i namenu.

Slanje zahteva na različite krajnje tačke, u REST arhitekturi, prikazano je u poglavlju 4.3.9.

Primer korišćenja GraphQL krajnje tačke slanjem odgovarajućih upita, prikazan je u poglavlju 4.2.6.

### 5.1.2 Struktura odgovora sa servera

U REST arhitekturi, struktura odgovora definisana je logikom na serveru. Klijent nema uticaj na strukturu odgovora. Ako je potrebno promeniti strukturu odgovora, ta promena mora biti učinjena na serverskoj strani.

U GraphQL pristupu, struktura odgovora je definisana od strane klijenta. Klijent šalje upit i zahteva njemu potrebne podatke sa strukturom koja je odgovarajuća klijentu. U poglavlju 4.2.6 su dati primeri upita.

### 5.1.3 Verzionisanje

Verzionisanje je veoma bitan aspekt razvoja serverskog API-ja. API je podložan promenama i potrebno je očuvati kompatibilnost unazad.

U REST arhitekturi, verzionisanje se obavlja dodavanjem verzije u sam URL. Na ovaj način, obezbeđuje se da sve prethodne verzije mogu biti korišćene od strane klijenata koji nisu još uvek prešli na novije verzije.

U GraphQL-u se ne koristi eksplicitno verzionisanje, već se shema postepeno unapređuje tokom celog procesa prelaska. Neka polja mogu ostati deo sheme i postepeno se mogu izbacivati kako bi se održala kompatibilnost unazad.

## 5.2 Performanse

Performanse aplikacije predstavljaju veoma bitan aspekt. Bez obzira na izbor između REST-a i GraphQL-a, neophodno je da API ima dobre performanse. U ovom poglavlju navode se aspekti performansi u REST i GraphQL arhitekturi.

### 5.2.1 Performanse u REST-u

Kako bi se poboljšale performanse u REST arhitekturi mogu se probati neke od narednih stvari:

- Vraćanje samo neophodnih podataka - u ovom slučaju treba napraviti balans tako da ne dođe do prekomernog odnosno nedovoljnog preuzimanja i treba pokušati smanjiti broj zahteva. Ovo se može postići pravljenjem različitih

krajnjih tačaka, koje u zavisnosti od potrebe, vraćaju tačno neophodne rezultate.

- Kompresija - radi smanjnja odgovora koji se vraća sa servera.
- Implementacija stranicenja i upotreba filtera - na taj način moguće je smanjiti obim podataka koji se vraćaju.

### 5.2.2 Performanse u GraphQL-u

Kako bi se poboljšale performanse u GraphQL pristupu mogu se probati neke od narednih stvari:

- Optimizacija razrešavača - Efikasan razrešavač je ključni deo mehanizma izvršavanja GraphQL upita. Svaki pokrenuti upit se izvršava tako što se poziva razrešavač.
- Upotreba efikasnije biblioteke za GraphQL - Različite biblioteke mogu imati različite performanse pa samim tim posebnu pažnju treba posvetiti odabiru odgovarajuće biblioteke.
- Optimizacija sheme - Bitan aspekt je održavati shemu jednostavnom kako bi izvršenje upita od strane razrešavača bilo što efikasnije.

## 5.3 Skalabilnost

Skalabilnost u ovom kontekstu predstavlja sposobnost sistema da uspešno upravlja povećanim brojem zahteva i povećanim mrežnim protokom. Skalabilnost i performanse su dve spregnute stvari. Sistem koji ima loše performanse ne može imati dobru skalabilnost. Bez obzira na arhitekturu API-ja neki od principa povećanja skalabilnosti su: **keširanje**, **horizontalno skaliranje**, **asinhrono procesiranje** i drugi.

### 5.3.1 Skalabilnost u GraphQL-u

- Dohvatanje samo potrebnih podataka kroz upite.
- Grupisanje više upita u jedan zahtev kako bismo smanjili broj zahteva od strane klijenta.

- Optimizacija razrešavača takođe doprinosi skalabilnosti.
- Paralelno dohvaćanje podataka omogućava paralelno izvršavanje razrešavača, što omogućava konkurentno dohvaćanje podataka iz više izvora istovremeno.

## 5.4 Bezbednost

REST i GraphQL dele većinu bezbednosnih aspekata. Tokom rada sa GraphQL-om, potrebno je obratiti pažnju na neke stvari koje su specifične za GraphQL.

**Introspekcija sheme** omogućava klijentima da otkriju GraphQL shemu. Ovo može predstavljati bezbednosnu pretnju. Neophodno je postaviti mehanizam kontrole pristupa shemi, kako bi se izbeglo neovlašćeno istraživanje baze.

Bezbednost podataka koji se mogu dohvatiti sa servera je ključan aspekt bezbednosti. Određeni podaci ne treba da budu javno dostupni i neophodno je sprečiti neovlašćen pristup podacima. Ovo se postiže **autentikacijom** i **autorizacijom**. Korisnik kroz autentikaciju dokazuje svoj identitet, a autorizacijom se vrši provera prava pristupa podacima od strane tog korisnika.

Jedan od napada na koje je ranjiv GraphQL je takozvani napad iscrpljivanja resursa. Napad se realizuje pokretanjem upita koji uzimaju resurse servera. Neophodno je proceniti kompleksnost upita i uvesti ograničenja kako bi se izbeglo prekomerno uzimanje resursa servera.

Kompleksnost upita je mera koja pokazuje koliko je upit zahtevan za izvršavanje. Na kompleksnost upita utiču različiti aspekti:

- **Broj polja** - Što je veći broj polja u upitu, to je kompleksnost upita veća. Veliki broj polja može usporiti dohvaćanje podataka i može zauzeti previše računarskih resursa servera.
- **Dubina ugnježđenosti** - Upiti velike dubine mogu izazvati opterećenje servera. Njihovo izvršavanje može uzeti previše računarskih resursa. Ovaj problem je moguće rešiti uvođenjem ograničenja maksimalne dubine ugnježđenosti.
- **Složenost resursa** - Neki resursi mogu imati veliku složenost za dohvaćanje, nezavisno od broja polja ili dubine. Ova složenost može poticati od složenosti njihovog izračunavanja.

Moguće je uvesti i ograničenja po broju zahteva koje korisnik može poslati tokom jednog vremenskog intervala. Ovo se može postići na statički ili dinamički način. Statički način podrazumeva unapred definisanu vrednost broja zahteva tokom vremenskog intervala. Dinamički način podrazumeva da broj zahteva zavisi od trenutne opterećenosti sistema. U periodima visoke opterećenosti sistema, dozvoljeni broj zahteva je manji nego u periodima niske opterećenosti.

Ako se postavi ograničenje za upit, upit se odbija i u tom slučaju se ne obrađuje od strane servera. Više o bezbednosti u radu sa GraphQL-om moguće je pronaći u knjizi [14].

# Glava 6

## Zaključak

U radu je opisan nastanak REST arhitekture, dat je njen pregled i opisani su njeni principi. Prikazan je istorijat GraphQL-a, data je motivacija njegovog uvođenja i prikazani su GraphQL upitni jezik i razrešavači. Nakon prikaza GraphQL-a, dat je pregled modernih alata koji implementiraju GraphQL.

Ključni deo rada predstavlja implementacija rešenja problema iz realnog sveta upotrebom GraphQL-a i REST pristupa u poglavljima 4.2 i 4.3.

Aplikacija u GraphQL pristupu koristi samo **jednu krajnju tačku** za razliku od pristupa sa REST arhitekturom koja ima ne tako mali broj krajnjih tačaka za relativno jednostavnu aplikaciju. Broj ovih krajnjih tačaka raste sa povećanjem broja entiteta iz domena koji se modeluje. Korišćenje jedne krajnje tačke u GraphQL pristupu, odnosno više njih u REST pristupu prikazano je upotrebom Postman-a u poglavljima 4.2.6 i 4.3.9.

Prikazana je **fleksibilnost upita** dohvatanja podataka koju pruža GraphQL pristup u odnosu na REST. Fleksibilnost se opaža posmatranjem operacije dohvatanja članka po identifikatoru. U GraphQL pristupu, u primeru 4.2.6.1, omogućava se klijentu da tačno definiše potrebna polja, za razliku od pristupa 4.3.9.1 gde su na serveru definisana polja koja će biti isporučena klijentu i ovakva fleksibilnost nije omogućena klijentu. Značaj fleksibilnosti upita posebno se primećuje u kompleksnijim aplikacijama.

GraphQL omogućuje i **manji broj zahteva** u odnosu na REST pristup. GraphQL može dohvatiti potrebne podatke u jednom zahtevu dok je u REST pristupu neophodno slanje više zahteva ka serveru. U primeru 4.3.9.2 prikazan je način na koji GraphQL smanjuje broj zahteva ka serveru.

Kroz komparativnu analizu u poglavlju 5 rezimirane su: **performanse, skala-**

**bilnost, bezbednost i dizajn koda** u ova dva pristupa.

REST arhitektura je šire rasprostranjena u programerskoj zajednici nego GraphQL. Programeri su bolje upoznati sa REST arhitekturom i za razvoj javnih API-ja REST arhitektura predstavlja bolji izbor. Takođe, REST arhitektura kao tradicionalniji i stariji pristup ima rasprostranjeniju podršku u pogledu razvojnih alata i biblioteka.

GraphQL **nije univerzalno rešenje** i prilikom izbora jedne od ove dve arhitekture treba se voditi rezultatima ovog rada.



# Literatura

- [1] Apollo and typegraphql — <https://typegraphql.com/>. on-line at: <https://typegraphql.com/docs/bootstrap.html>.
- [2] Apollo server — [www.apollographql.com](http://www.apollographql.com). on-line at: <https://www.apollographql.com/docs/apollo-server/>.
- [3] Apollo server integrations — <https://www.apollographql.com/docs/apollo-server/integrations/integration-index/>. on-line at: <https://www.apollographql.com/docs/apollo-server/integrations/integration-index/>.
- [4] Apollo server with express — [www.apollographql.com](http://www.apollographql.com). on-line at: <https://www.apollographql.com/docs/apollo-server/api/express-middleware/>.
- [5] Application with graphql — luka vujčić github. on-line at: <https://github.com/LukaVujcic/REST-VS-GRAPHQL/tree/master/GraphQL/>.
- [6] Application with rest — luka vujčić github. on-line at: <https://github.com/LukaVujcic/REST-VS-GRAPHQL/tree/master/REST/>.
- [7] GraphQL — [spec.graphql.org](http://spec.graphql.org). on-line at: <https://spec.graphql.org/draft/>.
- [8] GraphQLyoga — <https://the-guild.dev/graphql/yoga-server/docs>. on-line at: <https://the-guild.dev/graphql/yoga-server/docs>.
- [9] Mozilla docs — [www.developer.mozilla.org/](http://www.developer.mozilla.org/). on-line at: [https://developer.mozilla.org/en-US/docs/Learn/Common\\_questions/Web\\_mechanics/What\\_is\\_a\\_URL/](https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Web_mechanics/What_is_a_URL/).
- [10] Npm — <https://www.npmjs.com/>. on-line at: <https://www.npmjs.com/>.

- [11] Postman — <https://www.postman.com/>. on-line at: <https://www.postman.com/>.
- [12] Typescript — <https://www.typescriptlang.org/>. on-line at: <https://www.typescriptlang.org/>.
- [13] John Belamaric and Cricket Liu. *Learning coredns configuring DNS for cloud native environments*. O'Reilly, 2019.
- [14] Samer Buna. *GraphQL in action*. Manning Publications Co., 2021.
- [15] Roy T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000.
- [16] Eve Porcello and Alex Banks. *Learning graphql: Declarative data fetching for modern web apps*. O'Reilly, 2018.
- [17] Robin Wieruch. *The road to graphql*. CreateSpace Independent Publishing Platform, 2019.