Korištenje GraphQL-a za razvoj mrežnih aplikacija

Pedišić, Mario

SVEUČILIŠTE U SPLITU
PRIRODOSLOVNO MATEMATIČKI FAKULTET

DIPLOMSKI RAD

KORIŠTENJE GRAPHQL-a ZA RAZVOJ MREŽNIH APLIKACIJA

Mario Pedišić

Split, rujan 2022.

Temeljna dokumentacijska kartica

Diplomski rad

Sveučilište u Splitu
Prirodoslovno-matematički fakultet
Odjel za informatiku
Ruđera Boškovića 33, 21000 Split, Hrvatska

KORIŠTENJE GRAPHQL-A ZA RAZVOJ MREŽNIH APLIKACIJA

Mario Pedišić

SAŽETAK

Cilj ovog rada je napraviti pregled dostupnih alata i tehnologija koje se koriste za izradu mrežnih aplikacija sa fokusom na komunikaciju između različitih slojeva aplikacije. Najveći dio rada odnosi se na implementaciju GraphQL-a, odnosno jezika za slanje upita koji klijentima omogućava zahtijevanje točno onih podatka koji su im potrebni. Kako bi prikazao tu implementaciju u radu ću opisati izradu GraphQL poslužitelja i klijentske aplikacije koja koristi podatke sa tog poslužitelja. Između ostalog prikazat ću funkcionalnosti GraphQL upita, mutacija i pretplata, te napraviti usporedbu tih operacija sa operacijama koje podržava REST.

Ključne riječi: GraphQL, REST, komunikacijski protokoli, arhitektura mrežnih aplikacija

Rad je pohranjen u knjižnici Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu

Rad sadrži: 66 stranica, 72 grafička prikaza, 2 tablice i 32 literaturna navoda. Izvornik je na hrvatskom jeziku.

Mentor: **Doc. dr. sc. Goran Zaharija,** *docent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Ocjenjivači: **Doc. dr. sc. Goran Zaharija,** *docent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Prof. dr. sc. Andrina Granić, *redoviti profesor u trajnom zvanju Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Doc. dr. sc. Monika Mladenović, *docent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Rad prihvaćen: **Rujan**, **2022.**

Basic documentation card

Graduate thesis

University of Split
Faculty of Science
Department of Informatics
Ruđera Boškovića 33, 21000 Split, Croatia

USING GRAPHQL FOR NETWORK APPLICATION DEVELOPMENT

Mario Pedišić

ABSTRACT

The aim of this paper is to make an overview of the available tools and technologies that are used to create network applications with a focus on the communication between different layers of the application. Most of the paper is related to the implementation of GraphQL, a language for sending queries that enables clients to request exactly the data they need. In order to show that implementation, in this paper I will describe the creation of a GraphQL server and a client application that uses data from that server. Among other things, I will show the functionality of GraphQL queries, mutations and subscriptions, and make a comparison of these operations with the operations supported by REST.

Key words: GraphQL, REST, communication protocols, network application architecture

Thesis deposited in library of Faculty of science, University of Split

Thesis consists of 66 pages, 72 figures, 2 tables and 32 references,
Original language: Croatian

Mentor: **Goran Zaharija, Ph.D.** *Assistant Professor of Faculty of Science, University of Split*

Reviewers: **Goran Zaharija, Ph.D.** *Assistant Professor of Faculty of Science, University of Split*

Andrina Granić, Ph.D. *Full Professor of Faculty of Science, University of Split*

Monika Mladenović, Ph.D. *Assistant Professor of Faculty of Science, University of Split*

Thesis accepted: **September, 2022.**

IZJAVA

kojom izjavljujem s punom materijalnom i moralnom odgovornošću da sam diplomski rad s naslovom KORIŠTENJE GRAPHQL-A ZA RAZVOJ MREŽNIH APLIKACIJA izradio samostalno pod voditeljstvom Doc. dr. sc., Goran Zaharija. U radu sam primijenio metodologiju znanstvenoistraživačkog rada i koristio literaturu koja je navedena na kraju diplomskog rada. Tuđe spoznaje, stavove, zaključke, teorije i zakonitosti koje sam izravno ili parafrazirajući naveo u diplomskom radu na uobičajen, standardan način citirao sam i povezao s fusnotama s korištenim bibliografskim jedinicama. Rad je pisan u duhu hrvatskog jezika.

Student

Mario Pedišić

1 Sadržaj

1. Uvod

Količina informacija koju prosječna osoba obradi u jednom danu je zapanjujuća i raste iz dana u dan. Danas zahvaljujući internetu i mobilnim uređajima u trenutku možemo pronaći informacije o onome što nas zanima. Zbog velike količine podataka i potrebe korisnika za najnovijim informacijama aplikacije koje svakodnevno koristimo ne bi mogle obavljati svoju funkciju bez korištenja udaljenih baza podataka.

Osim zbog količine podataka baze podatka su potrebne kako bi mrežne aplikacije omogućile uslugu milijunima korisnika istovremeno. Kako bi maksimalno iskoristili prednosti koje nam nudi baza podatka, potrebno je bilo osmisliti određena pravila ili standarde kako bi omogućili korisnicima učinkovitiji pristup podacima.

Osim što tim standardima želimo povećati zadovoljstvo korisnika, također želimo povećati sigurnost komunikacije između korisnika i baze, te sigurnost samih podataka. Drugim riječima potrebno je osigurati da korisnici mogu pristupiti podacima u bilo kojem trenutku.

Danas postoje različite arhitekture koje koristimo za izradu mrežnih aplikacija i protokoli koji klijentima omogućavaju pristup podacima. Postoje različiti razlozi zbog kojih se pojedina arhitektura koristila u prošlosti, odnosno razlozi zašto su neke od tih arhitektura zastarjele i zamijenjene naprednijima [1].

U sljedećim poglavljima prikazat ću neke od najčešćih arhitektura mrežnih aplikacija, te načine na koje možemo ostvariti vezu između baze podataka i klijenta. Ostvarivanje te veze pri izradi mrežnih aplikacija može se implementirati na više načina.

Neki od najpopularnijih načina su temeljeni na REST arhitekturi ili u novije vrijeme GraphQL-u. U slijedećim poglavljima navesti ću što je to REST i GraphQL, te koje su im prednosti i mane. U zadnjem poglavlju ovog rada prikazati ću izradu mrežne aplikacije koja se temelji upravo na GraphQL-u što je i tema ovog rada.

2. Razvoj Mrežnih aplikacija

Da bi shvatili što je to GraphQL, te kako ga koristiti u razvoju mrežnih aplikacija potrebno je krenuti od početka, te shvatiti osnovne pojmove kao što su računalna mreža, klijent i poslužitelj. Razvoj mrežne aplikacije uključuje izradu programa koji na neki način dohvaća i koristi podatke sa mreže odnosno sa drugih računala, pa bi bilo smisleno krenuti od toga.

2.1 Računalna mreža

Računalnu mrežu predstavlja dva ili više međusobno povezana računala u svrhu razmjene resursa. Ti resursi mogu predstavljati podatke, sklopovlje ili programe. Dva ili više računala možemo smatrati povezanima ako ona imaju sposobnost međusobne razmjene informacija. Kako bi se ostvarila takva komunikacija računala u računalnoj mreži koriste komunikacijske protokole [2].

U današnje vrijeme većina uređaja kao što su osobna računala, pametni telefoni, satovi i televizori komuniciraju na neki način sa drugim računalima ili poslužiteljima, pa možemo reći da i oni predstavljaju dio računalne mreže. Računalne mreže su danas široko zastupljene, a ponajviše u tome što nam omogućavaju pristup *World Wide Web-u*. Osim toga podržavaju različite primjene, između ostalog zajedničko korištenje programa i poslužitelja za pohranu, korištenje elektroničke pošte, zajedničko korištenje pisača i pristup udaljenim računalima.

2.2 Distribuirani računalni sustavi

Sustavi koji se sastoje od skupa računala povezanih računalnim mrežama nazivamo distribuiranim sustavom. Računala u takvim sustavima predstavljaju različite komponente tog sustava. Ona su opremljena pripadajućom programskom podrškom koja im omogućava da koordiniraju svoje aktivnosti i dijele resurse. Kao što sam već naveo kod računalnih mreža, ti resursi mogu biti sklopovlje, programska podrška ili podaci. Resurse u distribuiranim sustavima nazivamo sustavnim resursima.

Računala u distribuiranim sustavima komuniciraju i koordiniraju svoje akcije preko određenih poruka. Različita računala međusobno komuniciraju i dijele resurse kako bi postigle neki zajednički cilj. Postoji više načina odnosno mehanizama pomoću kojih računala međusobno šalju poruke. Poruke se najčešće šalju čistim HTTP-om ili slanjem upita (engl. *Query*), što ću kasnije u radu detaljnije opisati.

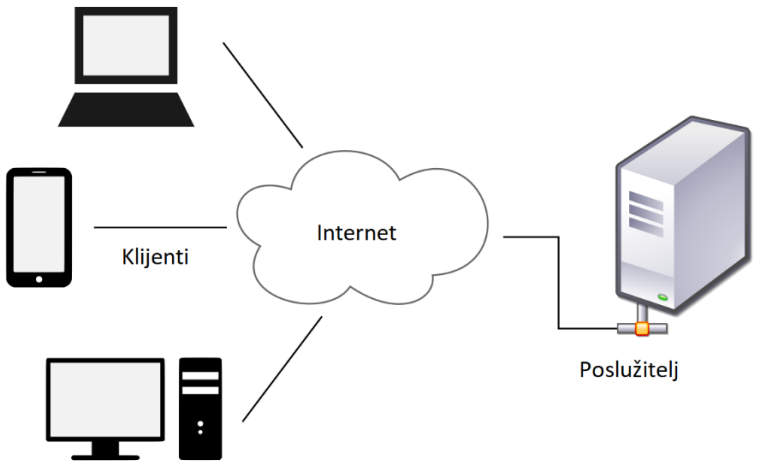
2.3 Arhitekture distribuiranih računalnih sustava

Za povezivanje računala u distribuiranim sustavima, te njihovu međusobnu komunikaciju osmišljeno je više različitih hardverskih i softverskih arhitektura. Hardverske arhitekture, odnosno hardverska rješenja kod distribuiranih sustava su potrebna da bi se računala fizički umrežila nekom vrstom mreže. Softverska rješenja su u ovom kontekstu potrebna da bi se omogućila komunikacija između računala u sustavu.

Unutar distribuiranih računalnih sustava softverska rješenja predstavljaju distribuirani programi. Proces pisanja takvih programa naziva se distribuirano programiranje. Neke od osnovnih arhitektura distribuiranog programiranja su: klijent-poslužitelj, troslojna, n-slojna ili P2P (engl. *Peer-to-peer*) [1].

2.3.1 Klijent-poslužitelj

Klijent-poslužitelj arhitektura distribuiranih računalnih sustava je arhitektura u kojoj se resursi ili zadaci dijele između klijenta i poslužitelja. U ovakvom modelu klijent (engl. *Client*) je onaj koji zahtjeva, a poslužitelj (engl. *Server*) pruža usluge ili resurse.



Slika 1 – grafički prikaz arhitekture klijent-poslužitelj

World wide web i e-pošta su samo neki od usluga koji koriste klijent-poslužitelj arhitekturu. Karakteristično je za ovu arhitekturu da se klijent i poslužitelj nalaze na odvojenom sklopovlju (iako se mogu nalaziti i na istom). Grafički prikaz takve arhitekture je prikazan na Slici 1. Na Slici 1 osobno računalo, prijenosno računalo i pametni telefon predstavljaju klijente, dok računalo koje nazivamo domaćin (engl. *Web Host*) predstavlja poslužitelja. Na ovom prikazu klijent i poslužitelj su umreženi preko interneta. U kontekstu priče o računalnim mrežama Internet predstavlja javno dostupnu podatkovnu mrežu koja povezuje računala i računalne mreže. Komunikacija između pojedinih računalnih mreža na internetu omogućena je uz pomoć

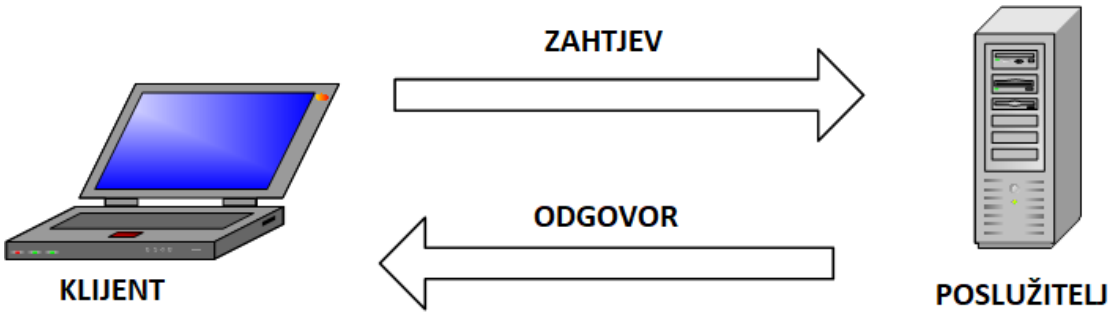
internetskog protokola. Iz razloga što Internet povezuje više računalnih mreža često ga nazivamo mrežom svih mreža [3].

U modelu klijent-poslužitelj poslužitelj je taj koji dijeli podatke ili usluge, stoga klijent obično samo prima podatke ili usluge bez da dijeli svoje. Nakon što je ostvarena veza između klijenta i poslužitelja, da bi klijent pristupio resursima poslužitelja prvo je potrebno poslati zahtjev. Kako je na jedan poslužitelj uobičajeno spojeno više klijenata istovremeno, poslužitelj čeka dolazne zahtjeve.

Model klijent-poslužitelj je najčešće primijenjen model distribuiranih sustava. Dijeljenje resursa predstavlja uslugu poslužitelja (engl. *Service*). Tipovi poslužitelja se dijele prema tipu usluge koji pružaju. Vezano za ovaj rad, tip poslužitelja koji je nama najzanimljiviji je web poslužitelj (engl. *Web Server*). To je tip poslužitelja koji dijeli svoje resurse, odnosno pruža usluge, web stranicama.

Komunikacija klijenta i poslužitelja

Pojedino računalo može imati obje uloge istovremeno, odnosno biti i klijent i poslužitelj. Pojedini poslužitelji mogu tražiti neke resurse koji dijele drugi poslužitelji, te na taj način imati obje uloge. Poslužitelji najčešće komuniciraju da bi sinkronizirali podatke, u tom slučaju taj tip komunikacije nazivamo komunikacija poslužitelj-poslužitelj.



Slika 2 - komunikacija u modelu klijent-poslužitelj

Komunikacija između klijenata i poslužitelja se odvija porukama u stilu zahtjev-odgovor kao što je prikazano na slici 2. Na grafičkom prikazu, klijent šalje zahtjev, a poslužitelj vrača odgovor [1].

Usluga koju poslužitelj pruža je apstrakcija resursa tog računala. Drugim riječima klijent se ne mora „brinuti“ što poslužitelj radi i kako radi. Od strane klijenta se zahtjeva samo da može protumačiti odgovor. Razumijevanje odgovora poslužitelja od strane klijenta se odnosi na

razumijevanje sadržaja i formata podataka. Isto vrijedi i za poslužitelj koji se ne mora brinuti na koji način klijent koristi podatke, bitno je da razumije zahtjev i dostavi valjani odgovor.

Kako bi komunikacija bila uspješna klijent i poslužitelj se moraju razumjeti odnosno imati zajednički jezik i slijediti pravila definirana komunikacijskim protokolom. Ti protokoli definiraju kako će se komunikacija odvijati između klijenta i poslužitelja. Kao još jedan dodatan sloj apstrakcije poslužitelj može implementirati aplikacijsko programsko sučelje (engl. *Application programming interface, API*), što će biti detaljnije objašnjeno u radu. Za sad je dovoljno reći da API dodatno formalizira razmjenu podataka odnosno olakšava pristup usluzi poslužitelja.

Tipično je da je jedan poslužitelj služi više klijenata. Količina zahtjeva koju poslužitelj može obraditi je u nekom vremenskom intervalu je ograničena te da ne bi došlo do zastoja u opskrbi klijenata podacima, poslužitelj koristi sustav raspoređivanja kako bi odredio prioritete između različitih zahtjeva klijenata.

Iako je klijent-model dosta zastupljen i često se koristi, ne zadovoljava sve uvjete. Obrada zahtjeva od strane poslužitelja se može iskoristiti kontraproduktivno, odnosno ako želimo „srušiti“ poslužitelja, ponekad je dovoljno preopteretiti ga sa više zahtjeva nego on može podnijeti. Kako bi dodatno zaštitili podatke prilikom razmjene, poželjno je koristiti enkripciju.

Primjer modela klijent-poslužitelj

Kako bi prikazao korištenje Graphql-a za razvoj mrežnih aplikacija u radu ću prikazati pristupanje bazi podataka i dohvaćanje informacija vezanih za poslani upit. Primjer kojim ću to opisati je dohvaćanje knjiga koje je uneseni autor napisao. Isti primjer mogu iskoristi kako bi prikazao rad modela klijent-poslužitelj i pristupanju usluzi poslužitelja.

Pristupimo li usluzi dohvaćanja knjiga koje je uneseni autor napisao, mi kao klijent pokrećemo zahtjev prema poslužitelju. Poslužitelj zatim interpretira zahtjev i pronalazi podatke. Nakon toga poslužitelj šalje odgovor korisniku odnosno dijeli tražene podatke. Na kraju klijent računalo interpretira podatke te prikazuje rezultate korisniku odnosno knjige koje je napisao uneseni autor.

Svaki poslani zahtjev računalo poslužitelj obrađuje te vrača podatke. Kada su svi zahtjevi ispunjeni, komunikacija je završena i klijent računalo prikazuje dobivene podatke korisniku.

2.3.2 Troslojna arhitektura

Kad pričamo o podvrstama arhitekture klijent-poslužitelj moramo spomenuti troslojnu arhitekturu. Ona predstavlja najdominantniju podvrstu te arhitekture. Primjer dohvaćanja knjiga koje je napisao uneseni autor, koji je objašnjen u prošlom poglavlju, predstavlja takozvanu dvoslojnu arhitekturu. Takva arhitektura predstavlja originalnu klijent-poslužitelj arhitekturu [32].

Za razliku od troslojne arhitekture koja je prikazana na slici 3, dvoslojna nema aplikacijski sloj, odnosno ima samo prezentacijski i podatkovni sloj. To znači da korisnik ima direktan pristup podatkovnom sloju. Kako se ova arhitektura sastoji od samo dva sloja, logika dohvaćanja i obrade podataka se nalazi u prezentacijskom sloju, podatkovnom sloju ili oboje.

Zajednički naziv za sve aplikacijske arhitekture s više od jednog sloja je N-slojna arhitektura. Iako su dvoslojna i troslojna arhitektura često koriste, ostale N-slojne arhitekture su rijetke. Razlog tome je nedostatak prednosti takvih arhitektura, sporiji rad aplikacije, teže upravljanje razvojem i skuplje održavanje i rad aplikacije. Iz tog razloga naziv n-slojna arhitektura, ili pod drugim imenom višeslojna arhitektura najčešće označuju troslojnu arhitekturu.

Slika 3- troslojna arhitektura

Kao što sam naziv govori, troslojna arhitektura se sastoji od 3 logička i fizička sloja. Prvi sloj se naziva prezentacijski i uglavnom uključuje korisničko sučelje. Drugi se naziva aplikacijski i u tom sloju se podaci obrađuju. Treći sloj se naziva podatkovni, a služi za pohranu i upravljanje podacima koji se koriste unutar aplikacije.

Korištenje troslojne arhitekture ima mnoge prednosti. Gledano sa strane razvoja softvera, ovakav pristup može omogućiti da pojedine slojeve razvijaju odvojeni timovi, te se mogu vršiti popravci i nadogradnje bez utjecaja na druge spojeve. Nadogradnja može značiti i zamjenu bilo koji od 3 sloja u postupnosti kao odgovor na promjene u zahtjevima korisnika ili napredak

tehnologije. Primjer toga je zamjena operacijskog sustava na prezentacijskom sloju. U troslojnoj arhitekturi to ne bi trebalo utjecati na ostale slojeve.

Ostale prednosti ovakve arhitekture u odnosu na dvoslojnu su brži razvoj slojeva, poboljšana skalabilnost i pouzdanost te poboljšana sigurnost. Zato što svaki sloj ima svoju ulogu u aplikaciji, za razvoj pojedinog sloja mogu se koristiti različiti alati i programski jezici. Iz istog razloga više timova mogu istovremeno raditi na različitim slojevima aplikacije što znatno ubrzava proizvodnju.

Pojedini sloj se može izmijeniti neovisno o drugima. Zato što su slojevi logički odvojeni manja je vjerojatnost da će pogreška na jednom sloju izazvati prestanak funkcioniranja drugih slojeva. To znači da su aplikacije koje koriste troslojnu arhitekturu u pravilu pouzdanije od onih koje koriste dvoslojnu. U troslojnoj arhitekturi ne postoji mogućnost izravne komunikacije između prezentacijskog i podatkovnog sloja. Znači da aplikacijski sloj na neki način štiti podatkovni od izravnog kontakta sa klijentom što znači bolju sigurnost podataka.

Prezentacijski, aplikacijski i podatkovni sloj

Prezentacijski sloj predstavlja sloj koji je vidljiv krajnjem korisniku. Informacije se korisniku mogu prikazati u web pregledniku ili u grafičkom korisničkom sučelju (engl. *Graphic user interface, GUI*). Osim što prikazuje podatke korisniku, prezentacijski sloj prikuplja podatke od korisnika te mu omogućava komunikaciju sa aplikacijom. Samim time što je ovaj sloj vidljiv krajnjem korisniku on predstavlja najviši sloj aplikacije. Programeri koji rade na razvoju prezentacijskog sloja neke aplikacije mogu pisati kod u različitim jezicima, ovisno o platformi. Na primjer ako se podaci korisniku prikazuju u web pregledniku, prezentacijski sloj je najčešće razvijen koristeći HTML, CSS i JavaScript [31].

Aplikacijski sloj aplikacije obrađuje podatke i kontrolira funkcionalnost aplikacije. Jedini sloj koji izravno komunicira sa ostala dva sloja, iz tog razloga također se naziva srednji sloj. Aplikacijski sloj obrađuje informacije prikupljene u prezentacijskom sloju, ali i dodaje, briše ili mijenja podatke u podatkovnom sloju. Tu je bitno naglasiti da aplikacijski sloj komunicira s podatkovnim slojem pomoću API poziva, o čemu će biti govora kasnije u radu.

Sloj u kojem se nalazi baza podataka, odnosno u kojem aplikacija pohranjuje podatke naziva se podatkovni sloj. U troslojnoj arhitekturi on komunicira samo sa podatkovnim slojem.

Troslojna arhitektura u razvoju mrežnih aplikacija

U razvoju mrežnih aplikacija pojedini slojevi imaju druga imena, ali slične funkcije. Nazivi slojeva su web poslužitelj, aplikacijski poslužitelj i poslužitelj baze podataka.

Web poslužitelj predstavlja prezentacijski sloj. Kod mrežnih aplikacija to može biti web-stranica ili web-mjesto. Kod različitih web-stranica kao što su društvene mreže ili e-trgovine web poslužitelj predstavlja korisničko sučelje gdje korisnik unosi svoje podatke ili dodaje proizvode u košaricu. Sadržaj koji je prikazan korisniku može biti statičan ili dinamičan.

Aplikacijskom sloju kod razvoja mrežnih aplikacija odgovara aplikacijski poslužitelj. Taj dio mrežne aplikacije je zadužen za obradu korisničkih unosa. Na primjeru društvenih mreža, aplikacijski poslužitelj je zadužen za dohvaćanje podataka vezanih za upit korisnika, na primjer informacije o odabranom profilu iz baze podataka.

Poslužitelj baze podataka predstavlja posljednji sloj. Služi kao podatkovni sloj mrežne aplikacije. Poslužitelj baze podataka sastoji se od skupova podataka i softvera za upravljanje bazom podataka. On upravlja podacima unutar baze i omogućuje pristup podacima.

2.3.3 Servisno orijentirana arhitektura

Arhitektura koja se na neki način gradi na n-slojnoj arhitekturi je servisno orijentirana arhitektura, odnosno SOA. U n-slojnoj arhitekturi aplikaciju dijelimo na zasebne cjeline koje međusobno komuniciraju, a svaka ima svoju funkciju. SOA je metoda razvoja softvera koja koristi zasebne komponente za razvoj aplikacije. Te komponente nazivamo usluge, a svaka usluga ima svoju funkcionalnost i može komunicirati sa ostalim uslugama. SOA pristup omogućava programeru izradu usluge jednom koju zatim može koristiti u više aplikacija ili korištenje usluga treće strane [4].

Jedan od čestih primjera korištenja usluga u aplikaciji je unos podataka bankovne kartice prilikom Internet kupovine. Nakon odabira predmeta koje želimo kupiti i unosa osobnih podataka Internet stranica, korisnika preusmjerava na stranice banke gdje on unosi podatke vezane za način plaćanja i podatke za to plaćanje. Po završetku unosa informacija o plaćanju usluga vraća korisnika na stranicu Internet trgovine.

Servisno orijentirana arhitektura uključuje korištenje četiri komponente:

Usluga - osnovna komponenta SOA-e, koja može biti privatna ili javna, sastoji se od tri glavne značajke. Značajka implementacije označava kod koji se koristi za implementaciju određene usluge. Značajka ugovora o usluzi označava uvjete koje treba zadovoljiti prije korištenja

određene usluge, te cijenu usluge. Posljednja značajka je servisno sučelje koja definira kako pozvati uslugu za obavljanje zadatka ili razmjenu podataka

Pružatelj usluga – davatelj usluga koji pruža jednu ili više usluga. Organizacije mogu kreirati vlastite usluge ili ih kupiti od dobavljača trećih strana.

Korisnik usluga – potrošač koji zahtjeva pristup određenoj usluzi.

Registar usluga – repozitorij usluga, odnosno mrežno dostupan imenik usluga. Sadrži dokumentaciju koja sadrži informacije o usluzi i načinu implementacije. Registar usluga korisnik koristi kako bi pronašao uslugu koja mu odgovara.

U servisno orijentiranoj arhitekturi usluge pružaju funkcionalnost korisnicima, te funkcioniraju neovisno jedna o drugoj. Potrošač traži uslugu i šalje ulazne podatke. Usluga zatim obrađuje podatke, izvršava zadatak i vraća odgovor. Aplikacija za komunikaciju sa uslugama koristi komunikacijske protokole. Standardni protokoli za implementaciju servisno orijentirane arhitekture uključuju SOAP i RESTful HTTP o kojima će biti govora kasnije.

Ova arhitektura ima mnoge prednosti u odnosu na prethodno opisane arhitekture. Samo korištenje usluga treće strane i korištenje iste usluge u različitim aplikacijama znači brži razvoj aplikacije. Također aplikacije temeljene na servisno orijentiranoj arhitekturi lakše je održavati, odnosno ažurirati i otklanjati pogreške. Održavanje je jednostavnije iz razloga što promjene je potrebno napraviti samo na pojedinim uslugama, a ne na cijeloj aplikaciji. Osim što je moguće ažurirati pojedinu komponentu, moguće je i zamijeniti zastarjelu. Na taj način učinkovito i isplativo možemo modernizirati aplikaciju koja koristi zastarjele usluge.

2.3.4 Mikroservisi

Arhitektura mikroservisi sastoji se od vrlo malih i potpuno neovisnih softverskih komponenti koje su fokusirane na samo jedan zadatak. Mikrousluge komuniciraju putem pravila koja programeri stvaraju kako bi drugim softverskim sustavima omogućili komunikaciju s njihovom mikrouslugom, a ta pravila nazivamo API.

Sličnosti između arhitekture mikroservisa i servisno orijentirane arhitekture su velike iz razloga što mikroservisi predstavljaju evoluciju SOA-e. U arhitekturi mikroservisa različite usluge ne dijele podatke kao kod SOA-e nego favoriziraju kopiranje podataka. To ih čini potpuno neovisnim i kompatibilnim s modernim poslovnim okruženjima temeljenim na oblaku (engl. *Cloud Computing*). Kopiranje podatka na koje se može gledati kao nedostatak mikroservisi nadoknađuju brzinom izvedbe [4].

3 Aplikacijsko programsko sučelje

Računalo za komunikaciju sa korisnikom koristi grafičko korisničko sučelje. Ono korisniku pruža uvid u tražene informacije i također uzima podatke koje korisnik unosi. U radu programa ili web stanice aplikacijsko programsko sučelje ima sličnu ulogu. Dok GUI omogućava razmjenu podataka između korisnika i računala, API omogućava isto između dva ili više računalna ili dijelova aplikacije. Za razliku od korisničkog sučelja, nije namijenjen da ga koristi krajnji korisnik.

API koristi programer u kodu aplikacije. On se sastoji od različitih dijelova koji su dostupni programeru, a predstavljaju alate i usluge koje programer ima na raspolaganju. Ako aplikacija koristi neku od usluga aplikacijskog programskog sučelja kažemo da poziva tu API uslugu.

Kod API-a je bitno spomenuti API specifikaciju. Ona predstavlja smjernice ili standard kako koristiti aplikacijsko programsko sučelje unutar aplikacije. To znači da definira pozive koje programer može koristiti ili implementirati.

Iako postoje aplikacijska programska sučelja za programske jezike, operacijske sustave, računalno sklopovlje i ostalo, najčešće poistovjećujemo taj izraz sa mrežnim API-em.

3.1 Mrežni API

Između ostalog mrežno aplikacijsko programsko sučelje omogućava komunikaciju između računala koji su povezani putem interneta. Svaka vrsta API-a predstavlja skup definiranih pravila koja navode kako računala međusobno komuniciraju. Mrežni API se nalazi između aplikacije i web poslužitelja. Drugim riječima mrežni API nam omogućuje pristup s klijentskih uređaja bili to mobilni telefoni, osobna računala i slično na web poslužitelj korištenjem određenih protokola.

Da bi koristili neke od usluga API-ja kao korisnik moramo pokrenuti već spomenuti API poziv. Taj poziv poznat kao i zahtjev (engl. *Request*) služi za dohvaćanje informacija. Zahtjev se zatim obrađuje na putu od klijenta ka web poslužitelju putem API-ja. API zatim upućuje daljnji poziv web poslužitelju ili programu. Da bi se ovaj korak izvršio, API Zahtjev mora biti ispravan. Po primitku poziva poslužitelj šalje API-ju tražene podatke (engl. *Response*). Na kraju API dijeli podatke aplikaciji koja ih je tražila.

Sa strane poslužitelja važno je spomenuti krajnju točku (engl. *Endpoint*). Ona predstavlja poziciju gdje API mogu pristupiti podacima, te pomažu osigurati ispravno funkcioniranje aplikacije. Učinkovitost mrežnog API-ja ovisi o tome koliko dobro komunicira sa krajnjim

točkama. Krajnjim točkama se obično pristupa putem URI-a (engl. *Uniform resource identifier*). URI predstavlja jedinstveni niz znakova koji indicira neki resurs.

Kranje točke API-ja moraju biti statične. To znači da se lokacija resursa ne bi smjela mijenjati. Izmjenom lokacije resursa mijenjamo i krajnju točku. U tom slučaju će aplikacije koje pristupaju toj krajnjoj točci prestati funkcionirati, jer resurs kojem žele pristupiti više ne postoji na tom mjestu [5].

Prema mogućnosti pristupa korisnika podacima, mrežne API-je dijelimo na četiri kategorije:

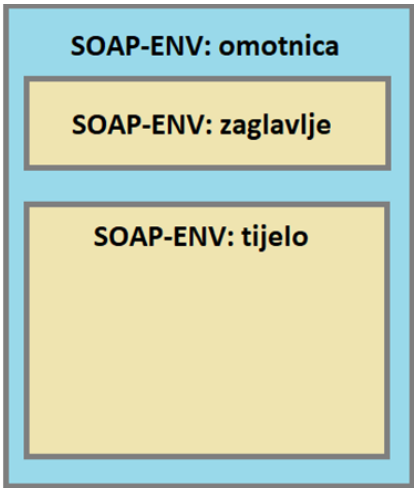
- **API otvorenog tipa** – može ih koristiti bilo tko, a za pristup se koristi HTTP protokol. Predstavljaju aplikacijsko programsko sučelje otvorenog koda (engl. *Open source*) poznati kao i javni API. Imaju jasno definirane krajnje točke, te format odgovora i zahtjeva.
- **Partnerski API** - aplikacijsko programsko sučelje dostupno za korištenje unutar određene udruge ili tvrtke, te njihovim suradnicima.
- **Interni API** – osmišljeniji samo za interno korištenje unutar tvrtke. Nije javno dostupan ostalim korisnicima.
- **Kompozitni API** – kombiniraju podatke i usluge više različitih API-ja. Na taj način programerima je omogućen pristup različitim krajnjim točkama u jednom API pozivu. Ovakva vrsta API-ja se koristi kad je za izvođenje jednog zadatka (primjer prikaz informacija na web stranici) potrebno dohvatiti informacije iz nekoliko izvora [6].

3.2 SOAP

U prethodnom poglavlju sam naveo da API predstavlja skup definiranih pravila kako računala ili dijelovi aplikacije međusobno komuniciraju. Kako bi se računala međusobno razumjela, odnosno razumjela poruke koje šalju i primaju, uvedeni su skupovi pravila kako pisati te poruke. Ta pravila nazivaju se protokoli. Jedan od takvih protokola je SOAP (engl. *Simple Object Access Protocol*).

SOAP predstavlja protokol za poruke koji omogućuje komunikaciju između računala ili različitih dijelova aplikacije. SOAP poruka predstavlja informacije koje se razmjenjuju između 2 računala. Koristi se kod prijenosa podataka unutar računalnih mreža, implementacijom mrežnih usluga. Mrežna usluga (engl. *Web service*) ustvari predstavlja mrežni API, a označava uslugu koju jedno računalo omogućava drugom putem interneta.

Kao format za poruke SOAP koristi XML, te se prenosi preko niza standardnih protokola, najčešće preko HTTP-a. Idejno SOAP je razvijen kao posrednik za aplikacije koje komuniciraju preko interneta pisane u različitim programskim jezicima. Drugim riječima SOAP omogućava programerima pozivanje mrežnih usluga, te primanje odgovora neovisno o programskom jeziku i platformi. To je moguće jer SOAP radi preko već spomenutog HTTP-a koji predstavlja mrežni protokol pred instaliran na svim operativnim sustavima (Windows, Linux, macOS) [7].



Slika 4 - struktura SOAP poruke

SOAP kao protokol za razmjenu poruka se sastoji od 3 dijela:

- Omotnica - prikazana na slici 4, definira strukturu poruke i inkapsulira sve podatke u poruci
- Skup pravila kodiranja za izražavanje instanci aplikacijski definiranih tipova podataka
- Skup pravila za predstavljane procedura poziva i odgovora

SOAP poruka je XML dokument, a sastoji se od sljedećih elemenata prikazanih na slici 4:

- **Omotnica** (engl. *Envelope*) – osim što definira strukturu poruke identificira XML dokument kao SOAP poruku
- **Zaglavlje** (engl. *Header*) – Sadrži dodatne informacije o poruci. Na primjer informacije za provjeru autentičnosti aplikacije koja poziva
- **Tijelo** (engl. *Body*) – sadrži informacije o pozivu i odgovoru, odnosno pojedinosti o tome što je potrebno poslati s mrežne usluge aplikaciji koja poziva
- **Greška** (engl. *Fault*) - daje izvještaj korisniku o pogreškama koje su se dogodile tijekom slanja poruke ili dohvaćanja informacija

Kod slanja SOAP zahtjeva potrebno je navesti samo omotnicu i tijelo poruke. S razlogom što je XML dokument u velikoj mjeri čitljiv čak i korisnicima koji se prvi put susreću s njim SOAP zahtjeve je jednostavno napraviti. Kako bi ostvarili komunikaciju računalo klijent prvo šalje zahtjev za uslugu. SOAP klijent šalje taj isti XML dokument na SOAP poslužitelj. Po primitku poruke SOAP poslužitelj šalje poruku vezano za određenu uslugu koju zahtjeva korisnik. Poruka koja sadrži ono što je korisnik tražio prvo se šalje rukovoditelju zahtjeva (engl. *SOAP* request handler), a zatim samom klijentu. Za slanje SOAP zahtjeva i odgovora koriste se protokoli kao što su HTTPS ili HTTP.

SOAP je prvi protokol koji je služio za povezivanje mrežnih servisa u servisno orijentiranoj arhitekturi (engl. *Service oriented architecture, SOA*). Iako je u današnjici zamijenjen suvremenijim protokolima, još uvijek se u uporabi zbog prednosti koje sam naveo u radu prvenstveno korištenje HTTP protokola i neovisnost o operativnom sustavu ili programskom jeziku u kojem su aplikacije koje povezuje pisane.

Neki od razloga zbog kojih ovaj protokol više nije zastupljen u istoj mjeri u odnosu na druge protokole ili arhitekture je manjak fleksibilnosti i lošija brzina izvođenja. Novije metode kao što je RESTful arhitektura, nude veću fleksibilnost jer koriste druge formate za razmjenu podataka osim XML-a, poput JSON-a. Također XML čini SOAP poruke relativno velikima što usporava izvršavanje zahtjeva [30].

3.3 REST

Kao odgovor na nedostatke SOAP protokola nastala je REST arhitektura. Veća fleksibilnost, jednostavnost i više načina za razmjenu informacija u usporedbi sa prijašnjim arhitekturama, učinila su REST najdominantnijom arhitekturom u području komunikacije komponenata distribuiranih aplikacija.

REST je kratica za *REpresentational State Transfer*, a označava stil arhitekture softvera. Važno je naglasiti da REST nije programski jezik ili dio softvera koji se može izvršiti. REST je prvenstveno pronašao primjenu u klijent-poslužitelj arhitekturi distribuiranih računalnih sustava. Omogućava komunikaciju različitih komponenti aplikacije preko interneta.

3.3.1 REST načela

REST arhitektura se temelji na šest načela. Ona opisuju kako su mrežni resursi definirani i adresirani. Sustav koji je u skladu sa nekim ili svim od načela REST-a naziva se RESTfull sustav. Šest REST načela su:

Arhitektura klijent poslužitelj

Korištenjem ovog načela aplikaciju dijelimo na korisnički dio i dio za pohranu podataka. To načelo je u skladu sa načelom SoC (*Separation of concerns*) odnosno takozvano „odvajanje briga“. Odvajanjem korisničkog sučelja i pohrane podataka omogućena je neovisnost razvoja pojedinih komponenti. To je bitno jer omogućuje razvoj komponente korisničkog sučelja bez prethodnog poznavanja kako radi poslužitelj i obratno.

Neovisnost između zahtjeva (engl. Statelessness)

Ovo načelo označava da svaki zahtjev klijenta prema poslužitelju mora sadržavati sve ono što je potrebno da bi poslužitelj razumio i ispunio zahtjev. Neovisnost između zahtjeva drugim riječima znači da poslužitelju nisu dostupne informacije o prijašnjoj komunikaciji sa pojedinim klijentom. Cilj ovog načela je stvoriti neovisnost između pojedinih zahtjeva. Klijentska strana ima sve informacije o komunikaciji, te ih šalje poslužitelju. Ovo načelo povećava performanse sustava je smanjuje opterećenje na poslužitelju te povećava sigurnost jer ako je za pristup resursu potrebna autentifikacija, klijent se mora autentificirati pri svakom zahtjevu.

Priručna memorija (engl. Caching)

Klijenti mogu privremeno spremiti resurse u pred memoriju što dovodi do povećanja performansi i smanjenja broja interakcija klijent-poslužitelj. Resursi se mogu klasificirati kako kao one što se mogu predmemorirati (engl. *Cacheable*) ili ne mogu predmemorirati (engl. Non-cacheable). Ako je resurs označen kao onaj što se može predmemorirati klijent ga može ponovno koristiti za kasniji odgovarajući zahtjev. Podaci koji se ne mogu predmemorirati su podaci koji se mijenjanju tijekom vremena.

Jedinstveno sučelje (engl. Uniform interface)

Sve komponente aplikacije temeljene na REST načelima moraju slijediti ista pravila za komunikaciju. Ovim načelom znatno je pojednostavljena cjelokupna arhitektura sustava i omogućeno bolje razumijevanje interakcija pojedinih komponenti, te omogućen njihov neovisan razvoj. Ovo načelo ima dodatna četiri pod načela, a to su:

- Identifikacija resursa u zahtjevima – mora postojati mogućnost identifikacije svakog resursa koji se dijeli između klijenta i poslužitelja. Na primjeru korištenja RESTful web API-a pojedini resurs je identificiran URI-jem unutar zahtjeva.

- Manipulacija resursima kroz reprezentaciju – korisnici bi morali moći raditi izmjene nad resursima makar imali samo reprezentaciju resursa
- Samo opisne poruke – svaka poruka mora biti dovoljno informativna kako bi omogućila obradu zahtjeva. Također poruka bi trebala pružiti dodatne informacije o radnjama koje klijent može izvesti nad resursom
- Hipermedija kao motor aplikacije – korištenjem samo URI-ja klijentska aplikacija bi trebala dinamički pokretati sve ostale resurse i interakcije korištenjem hiperveza.

Slojeviti sustav (engl. Layered system)

Unutar slojevitih sustava svaka komponenta ne može vidjeti dublje od onog sloja aplikacije s kojim izravno komunicira. To znači da klijent nema informacija o povezanosti, odnosno ne može znati je li komunicira izravno sa poslužiteljem ili nekim od komponenata između. Ovo je još jedno načelo koje dodatno omogućuje neovisan razvoj ili izmjenu komponenata unutar aplikacije.

Kod na zahtjev (engl. Code on demand)

Kod na zahtjev nije obavezno načelo. Poslužitelj može dati klijentu dodatni kod kako bi privremeno proširio njegove funkcionalnosti [8].

3.3.2 RESTful API

Dok SOAP predstavlja protokol, REST predstavlja opće smjernice za izradu aplikacijskog programskog sučelja. API-je koje slijede gore navedene smjernice nazivamo RESTful API. Ne postoji službeni standard za izradu RESTful API-ja. Mnogi programeri svoje API-je nazivaju RESTful, iako ne slijede sva ograničenja REST arhitekture prilikom izrade.

Kako bi klijent prestupio resursu koji se nalazi na poslužitelju mora poslati već spomenuti REST zahtjev. Zahtjev se općenito sastoji od:

- HTTP metode – ovaj dio zahtjeva definira koji vrstu operacije klijent želi izvršiti
- Zaglavlje – omogućuje klijentu slanje informacija o zahtjevu
- Putanja do resursa
- Tijelo poruke koje sadrži podatke (neobavezno pri slanju zahtjeva)

3.3.3 CRUD

Prilikom izrade aplikacijskog programskog sučelja želimo osmisliti model koji nam pruža četiri osnovne funkcionalnosti, a to su stvaraj, čitaj, ažuriraj i briši resurse. Model koji zadovoljava te četiri osnovne funkcionalnosti nazivamo CRUD. Ovaj akronim dolazi od prvih slova četiri osnovne funkcionalnosti na engleskom jeziku (*Create, Read, Update, Delete*). CRUD odgovara osnovnim HTTP metodama koje koristi REST, a to su GET, POST, PUT i DELETE [9].

HTTP POST

Ovu metodu koristimo za izradu novih resursa u REST arhitekturi. Drugim riječima POST stvara novi resurs navedene vrste. Na slici 5 je prikazan primjer dodavanja novog autora u bazu podataka. U ovom primjeru slanjem zahtjeva POST stvaramo novog autora sa nazivom „Steven Pressfield“ unutar resursa autori.

```
POST http://www.knjiznica.hr/autori/
Tijelo:
{
  "autor": {
    "ime": “Steven Pressfield”,
  }
}
```

Slika 5 – POST zahtjev

Ako je stvaranje novog autora bilo uspješno poslužitelj bi nam trebao vratiti odgovor u kojem navodi da je resurs izrađen. Resursu je dodana oznaka (ID) koju možemo koristiti za ostale HTTP metode.

HTTP GET

Za dohvaćanje resursa u REST arhitekturi koristimo HTTP GET metodu. Ova metoda ne mijenja resurse, samo ih dohvaća. GET možemo koristi za dohvaćanje svih podataka unutar resursa ili pojedinih stavki koristeći ID u zahtjevu.

```
Zahtjev:
GET http://www.knjiznica.hr/autori/1
Odgovor:
Response: Status Code - 200 (OK)
Tijelo:
{
    "autor": {
        "ime": “Steven Pressfield”,
    }
}
```

Slika 6 - GET zahtjev

U primjeru koji je prikazan na slici 6 dohvaćamo objekt autor koji ima ID jedan unutar resursa autori. Ako je zahtjev uspješno izvršen u odgovoru bi trebali dobiti poruku o tome u obliku HTTP status koda i tražene podatke kao što je prikazano na slici 6.

HTTP PUT

Za ažuriranje podataka pojedinih resursa koristimo HTTP PUT metodu. Ona koristi ID kako bi dohvatila resurs i napravila tražene izmjene. Primjer ovog zahtjeva je prikazan na slici 7. Na ovom primjeru dohvaćamo objekt resursa autori pod ID-em 1 i mijenjamo ime autora.

```
Zahtjev:
PUT http://www.knjiznica.hr/autori/1
Tijelo:
{
    "autor": {
        "ime": “Stephen King”,
    }
}
```

Slika 7 - PUT zahtjev

PUT zahtjev je jako koristan u slučaju izmjene postojećih podataka. Izmjene na resursu radimo najčešće zbog pogrešnog unosa ili promjene vrijednosti. Kao odgovor na zahtjev poslužitelj šalje povratnu informaciju, ovisno o tome jeli zahtjev izvršen.

HTTP DELETE

Za uklanjanje resursa iz sustava koristimo HTTP Delete metodu. Kao i kod drugih metoda resursu pristupamo preko ID-a.

```
DELETE http://www.knjiznica.hr/autori/1
```

Slika 8 - DELETE zahtjev

Ako želimo ukloniti stavku koju smo dodali u primjeru GET metode pozivamo zahtjev prikazan na slici 8. Ovaj zahtjev će ukloniti objekt autor koji ima vrijednost ID-a jedan iz resursa autori. Po izvršenju zahtjeva poslužitelj vrača potvrdnu informaciju.

Ako se želimo uvjeriti da je podatak stvarno izbrisan pozivamo zahtjev GET nad cijelim resursom. Na taj način vidimo da stavka sa ID-em jedan više nije unutar resursa autori. Isti efekt bi postigli pozivanjem GET zahtjeva nad objektom autor koji ima vrijednost ID-a jedan. U tom slučaju poslužitelj šalje poruku da stavka koja ima ID jedan nije pronađena.

HTTP Status kodovi

Kao što sam prikazao u prethodnim primjerima nakon izvršavanja pojedinih poziva klijent dobiva povratnu informaciju vezanu za pojedini zahtjev. Te informacije su poslane klijentu u obliku kodova. Takve kodove nazivamo HTTP statusni kodovi, te svaki ima specifično značenje. Najčešći kodovi i njihova značenja:

- 200 (OK) – standardni odgovor za uspješne HTTP zahtjeve
- 201 (CREATED) – standardni odgovor ako je uspješno kreiran resurs
- 204 (NO CONTENT) - standardni odgovor za uspješne HTTP zahtjeve, bez povratnih informacija u tijelu odgovora
- 400 (BAD REQUEST) – zahtjev nije izvršen zbog loše sintakse zahtjeva, prevelike veličine ili druge pogreške od strane klijenta
- 403 (FORBIDDEN) – klijent nema dopuštenje za pristup ovom resursu
- 404 (NOT FOUND) - resurs nije moguće pronaći
- 500 (INTERNAL SERVER ERROR) – generički odgovor u slučaju greške od strane poslužitelja

U slučaju uspješne obrade poziva GET i PUT zahtjevi bi trebali vratiti kod 200 (OK), POST zahtjev kod 201 (CREATED), a DELETE zahtjev kod 204(NO CONTENT) [9].

3.4 GraphQL

Iako je REST i dalje najdominantnija arhitektura vezano za izradu aplikacijskog programskog sučelja posljednjih godina pojavio se moderniji pristup koji donosi poboljšanja i nove mogućnosti. Broj API-a neprestano raste te postaju sve složeniji. Načela REST-a u velikoj mjeri nisu dovoljna za izradu učinkovitog API-ja. Noviji pristup izradi API-ja uzima dobre karakteristike REST-a kao bazu, te ga nadograđuje da odgovori na zahtjeve novog doba.

Način dohvaćanja podataka sa poslužitelja, odnosno količina podatka koje dobivamo u jednom pozivu nije prihvatljiva za korisnike koji pristupljuju podacima sa mobilnog uređaja. Smanjena je učinkovitost jer u većini slučajeva u RESTful API-jima korisnik dohvaća više podataka nego mu je potrebno [29].

Kada je nastao REST, aplikacije klijenata su bile relativno jednostavne. Broj aplikacija koje bi istovremeno pristupale poslužitelju je također bio manji i zahtijevale bi isti skup podataka. REST ne zadovoljava zahtjeve modernih API-ja jer vrača fiksnu strukturu podataka. Moderne aplikacije zahtijevaju dinamički odabir podataka kojima žele pristupiti.

Kod REST arhitekture promjene u zahtjevima korisnika najčešće zahtijevaju promjene na strani poslužitelja. Primjer, ako klijentska aplikacija zahtjeva dodatan podatak unutar resursa, jedno od rješenja je da na strani poslužitelja napravimo promjene na krajnjoj točci. To nije moguće ako više klijentskih aplikacija pristupa istoj krajnjoj točci. Kako bi izbjegli takvu situaciju moramo uvesti različite verzije API-ja što znatno usporava iteracije proizvoda.

Moderna alternativa REST arhitekturi je GraphQL. Ona ima za zadatak riješiti nedostatke REST-a na način da korisniku omogućuje traženje specifičnih podataka koje on zahtjeva. GraphQL predstavlja jezik za postavljanje upita bazama podataka iz klijentskih aplikacija. Najveća prednost GraphQL-a leži u tome što omogućuje korisniku izradu preciznijih zahtjeva za podatke, odnosno da dobije samo ono što mu je potrebno.

Razvoj GraphQL je krenuo od strane Facebook-a 2012 godine, prvenstveno za interne potrebe. Cilj je bio osmisliti način koji će dovesti do smanjenja upotrebe mreže prilikom dohvaćanja podataka od strane klijenta. GraphQL je danas podržan od strane mnogih programskih jezika kao što su Java, Python, C# i Node.js. Osim Facebook-a koriste ga mnoge druge organizacije poput Netflix-a, Github-a i PayPal-a [10].

3.4.1 Osnovne strukture GraphQL-a

U GraphQL-u način na koji pristupamo podacima definiran je uz pomoć modela podataka koji nazivamo shema (engl. *Schema*). Ona definira kakvi se upiti mogu uputiti poslužitelju, koju vrstu podataka možemo dohvatiti i odnose između pojedinih vrsta. Najosnovnija komponenta GraphQL sheme je tip (engl. *Type*). On opisuje vrste pojedinih objekata koje se možemo zahtijevati od poslužitelja te polja koja imaju. Polje (engl. *Field*) predstavlja atribut pojedinog objekta koje sadrži neku vrijednost [14].

GraphQL klijentu omogućava tri primarne operacije:

- Upit za čitanje podataka (engl. *Query*)
- Mutacije za izmjene podataka (engl. *Mutation*)
- Pretplata za primanje podataka u stvarnom vremenu (engl. *Subscription*)

Usporedno sa REST-om QUERY bi odgovarao GET metodi, a MUTATION PUT, POST i DELETE metodi.

3.4.2 Usporedba GraphQL-a i REST-a

S obzirom da GraphQL i REST imaju istu ulogu u izradi aplikacije imaju mnoge sličnosti. Za dohvaćanje resursa potrebno je poslati upit. Klijent odgovor može primiti u JSON formatu i oboje kao osnovu koriste HTTP protokol.

Međutim značajne razlike počinju već u samoj arhitekturi. Kao što sam već naveo GraphQL je klijentski orijentiran dok jer REST poslužiteljski. Drugim riječima REST sa strane poslužitelja ima definirane odgovore odnosno koje podatke će vratiti. Klijent nema pravo izravno birati što točno želi dohvatiti. Kod GraphQL-a odabir podataka je prepušten klijentu, on odlučuje što točno želi dohvatiti [13].

Pristup podacima se također razlikuje. Dohvaćanje podataka u REST arhitekturi se obavlja preko krajnjih točki, dok je kod GraphQL-a način pristupanja podacima definiran korištenjem shema i tipova. Razlika u pristupu podacima također definira broj poziva koje klijent mora uputiti da bi dobio sve potrebne podatke. Iz razloga što REST ima fiksnu strukturu podataka koje klijent može dohvatiti, teško će dohvatiti sve podatke samo jednim pozivom. Suprotno tome odabirom podataka koje klijent želi dohvatiti omogućeno mu je dohvaćanje svih potrebnih podataka jednim pozivom.

Za razliku od REST arhitekture, korištenjem GraphQL-a postižemo dohvaćanje svih potrebnih podataka slanjem samo jednog zahtjeva. Dohvaćanje podataka koristeći RESTful API najčešće

rezultira dohvaćanjem nepotrebno velikog seta podataka ili nepotpunog. Problem dohvaćanja više podataka nego klijent zahtjeva nazivamo *over-fetching*, a problem dohvaćanja seta podataka koji ne pruža sve informacije koje klijent zahtjeva u jednom pozivu, *under-fetching*.

Sam broj poziva znatno utječe na brzinu izvršavanja zahtjeva, odnosno učinkovitost. Iz razloga što dohvaća samo potrebne podatke i upućuje samo jedan poziv GraphQL je općenito brži od REST-a. Jedna od mogućnosti REST-a koje GraphQL ne podržava je pred memoriranje resursa. To može pozitivno utjecati na brzinu izvršavanja, ali dodatno opterećuje klijenta, što se želi izbjeći upotrebom GraphQL-a.

Samim time što se radi o relativno novijoj tehnologiji GraphQL je kompleksniji za početnike od REST-a. REST je jako dobro podržan i dokumentiran iz razloga što predstavlja najdominantniju arhitekturu današnjice te se koristi dugi niz godina. S druge strane GraphQL pronalazi sve veći broj primjena, što donosi ubrzanom razvoju biblioteka i alata koji ga podržavaju. Zbog popularnosti modernijih arhitektura poput GraphQL-a i vremena koje je REST dostupan na tržištu razvoj biblioteka i alata koji ga podržavaju lagano opada.

Zbog manjeg broja potrebnih upita i mogućnosti dohvaćanja samo potrebnih podataka najbolja primjena GraphQL-a je u mobilnim aplikacijama. S druge strane najbolja primjena REST-a je u jednostavnim aplikacijama i aplikacijama koje dohvaćaju veliku količinu podataka odjednom [11].

3.4.3 GraphQL prednosti

Postoje mnoge prednosti GraphQL-a u odnosu na protokole i arhitekture koje su mu prethodile. Najveća prednost je već spomenuto dohvaćanje svih potrebnih podataka jednim API pozivom. U usporedbi sa REST-om u GraphQL-u nema potrebe za kombiniranjem više izvora informacija da bi dobili željene podatke.

Odabirom podataka koje klijent želi dohvatiti uklanja se problem dohvaćanja premalog ili prevelikog broja podataka. Za razliku REST odgovori sadrže ili previše ili premalo podataka, što stvara potrebu za slanjem dodatnih zahtjeva.

Klijent prilagođava zahtjeve svojim potrebama. GraphQL API omogućuje prilagodbu zahtjeva kako bi klijent dobio samo informacije koje su mu potrebne.

GraphQL IDE prikazan na slici 9 između ostalog omogućuje programerima uvid u sheme kako bi se osiguralo da aplikacije traže samo ono što je moguće u odgovarajućem formatu. Uvidom u shemu programerima je olakšano postavljanje upita, te omogućen uvid u strukture podataka.

Postavljanje upita je dodatno olakšano jer GraphQL IDE prikazuje usporedno upit i odgovor na taj upit. Na taj način programer može jednostavno dodati nova polja postojećim upitima i usporediti rezultat.

Slika 9 - GraphQL IDE

Jedno od dodatnih mogućnosti GraphQL API-ja je automatsko generiranje dokumentacije za API. To omogućuje programerima da se više posvete pisanju koda, a manje izradi popratnih sadržaja. Sinkroniziranjem API-ja i dokumentacije, sve promjene u kodu se prenose na dokument.

Jedan od najvećih problema RESTful API-ja je što teško napraviti promjene na poslužitelju bez promjena na klijentu. Kako bi omogućili izmjenu krajnjih točaka potrebno je napraviti više verzija API-ja. Na taj način omogućavamo prijašnjim korisnicima produženu funkcionalnost, a novim korisnicima poboljšanju verziju API-ja. Kod GraphQL API-ja ne postoji potreba za različitim verzijama. Postoji samo jedna verzija API-ja koja se mijenja na razini polja. Polja je moguće ukloniti iz sheme bez utjecaja na postojeće upite [14].

GraphQL omogućuje zajedničko korištenje polja koja se koriste u više upita. To polje se može prenositi između povezanih komponenata. Ta funkcionalnost dodatno povećava učinkovitost jer uklanja potrebu za ponovnim dohvaćanjem već dostupnih podataka.

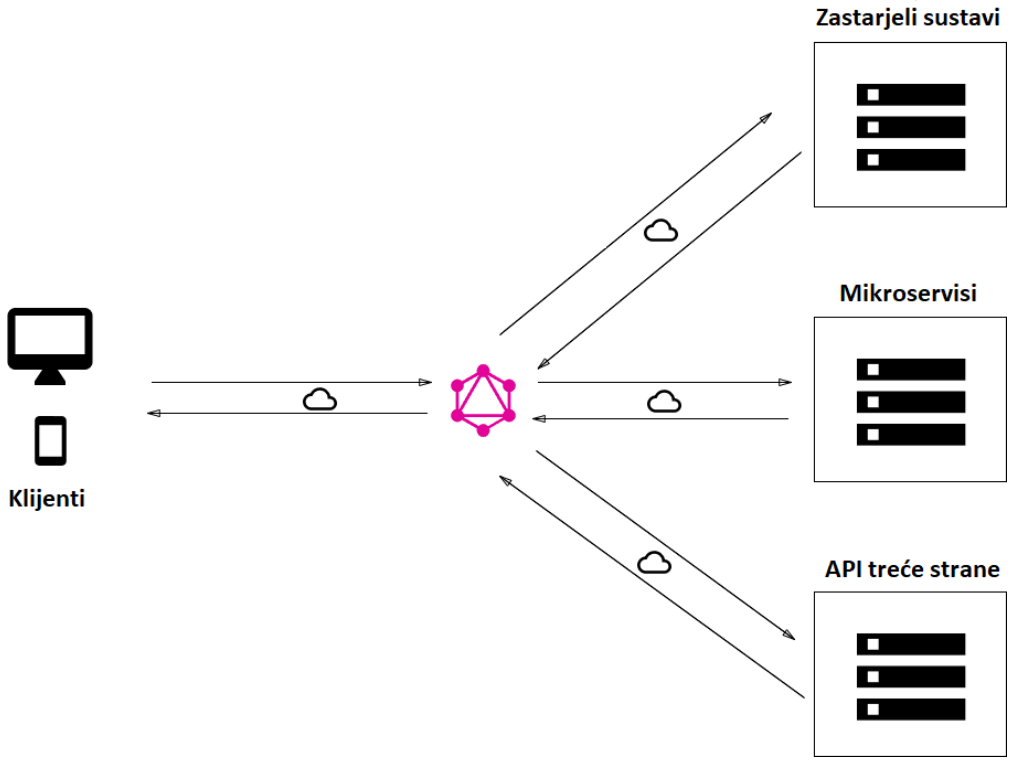
Općenito GraphQL nudi više informacija kad se dogodi pogreška u izvršavanju koda. RESTful API nudi ograničeni set informacija vezanih za pogrešku, primarno samo HTTP status kod.

GraphQL API nudi detaljniji uvid u pogrešku na način da prikaže točno onaj dio koda koji je prouzročio tu pogrešku.

Izradom GraphQL sheme korisniku se nudi mogućnost odabira funkcionalnosti koje želi prikazati, te kako one rade. U REST arhitekturi to nije slučaj, to jest možemo prikazati sve ili ništa. Ako se upit sastoji od nekih privatnih informacija, REST neće prikazati ni javne dijelove tog upita [11].

Usporedno sa RESTful API-jima GraphQL nudi još jednu dodatnu mogućnost, a to su pretplate (engl. *Subscriptions*). Ova funkcionalnost korisniku omogućava primanje poruka sa poslužitelja u stvarnom vremenu. Pretplate se unutar aplikacije mogu koristiti na različite načine, prvenstveno za automatsko ažuriranje prikaza podataka kad dođe do izmjene nad njima.

Možda jedna od najboljih primjena GraphQL-a je mogućnost ujedinjenja različitih sustava kao što je prikazano na slici 10. Koristeći GraphQL API kao posrednik možemo integrirati više različitih sustava i sakriti njihovu složenost. GraphQL poslužitelj će u takvom sustavu dohvaćati podatke i pretvoriti ih u GraphQL format odgovora. Ovo je jako bitno za zastarjele sustave i API-je (engl. *Legacy systems*) kako bi omogućili njihovo daljnje korištenje.



Slika 10 - GraphQL kao posrednik

Kod izrada aplikacija GraphQL API nam može omogućiti komunikaciju sa različitim mrežnim uslugama te njihovo spajanje u jednu GraphQL shemu. Svaka od tih usluga ima vlastitu GraphQL krajnju točku i shemu, ali postoji i globalna shema koja objedinjuje sve ostale.

Sve prethodno navedene prednosti kao što su zajedničko korištenje polja, korištenje GraphQL IDE-a i pretplata programerima omogućuju bržu i bezbolniju izradu aplikacija, a korisnicima bolju uslugu. Iz svega navedenog može se vidjeti zašto je GraphQL postao dobra alternativa za REST za određene primjene.

Važno je napomenuti da se funkcionalnosti GraphQL-a kao što je slanje specifičnih upita mogu replicirati kod REST-a korištenjem različitih biblioteka. Korištenje GraphQL-a je i dalje preporučljivo jer je implementacija takvih biblioteka zahtjevna.

3.4.4 GraphQL API nedostaci

S obzirom sa se radi o relativno novoj tehnologiji, koja je uz to složenija od REST-a, potrebno je neko vrijeme da se uđe u korak s njom. Upoznavanje sa novim alatima i jezicima poput SDL-a (engl. *Schema Definition Language*) zahtjeva dosta vremena koje svaki projekt ili programer nema. Iz tog razloga i dobre uhodanosti REST arhitekture razvojni timovi se ipak odluče ne iskoristi GraphQL i mnoge prednosti koje on nudi.

Drugi razlog zašto bi programeri odabrali REST ispred GraphQL-a odnosi se na složenost aplikacije koju razvijaju. REST je bolji izbor za jednostavne aplikacije koje ne dohvaćaju puno podataka sa poslužitelja, prvenstveno zbog složenosti GraphQL-a. On je također bolji izbor za aplikacije kojima nije potreban fleksibilan pristup podacima ili aplikacijama čiji zahtjevi za podacima dobro odgovaraju krajnjim točkama.

Korištenjem GraphQL-a postoji opasnost od rušenja poslužitelja slanjem više upita vezanih za ugniježđena polja odjednom. Slanje takvih zahtjeva može ugroziti sigurnost sustava, pa je za složenije upite REST bolja opcija. Korištenjem REST-a dohvaćanje podataka možda neće biti jednako brzo kao korištenjem GraphQL-a, ali će biti sigurnije iz perspektive održavanja funkcionalnosti poslužitelja.

Potrebno je još spomenuti da GraphQL podržava mehanizme obrane protiv takvih upita, potrebno ih je samo znati implementirati. Ti mehanizmi su između ostalog pridodavanje vremenskog ograničenja pojedinog zahtjeva, ograničavanje maksimalne dubine upita, određivanje složenosti pojedinih upita, izbjegavanje rekurzije i upornih upita.

GraphQL je ograničen u radu sa datotekama, te ne podržava slanje datoteka. To ograničenje nije prisutno u REST arhitekturi. Ako ipak poželimo omogućiti slanje datoteka korištenjem GraphQL-a to možemo učiniti implementacijom biblioteka kao što je Apollo koja omogućuje slanje posebnih vrsta zahtjeva [12].

4 Korištenje GraphQL-a u mrežnim aplikacijama

U poglavlju prije naveo sam prednosti i nedostatke GraphQL-a u odnosu na REST arhitekturu. Iz svega navedenog vidljivo je da GraphQL omogućava dodatne funkcionalnosti i poboljšava performanse sustava u određenim primjenama. Korištenje dodatnih funkcionalnosti najbolje je prikazati kroz praktičan primjer. U tu svrhu prikazati ću primjer izrade jednostavne mrežne aplikacije koja se sastoji od klijentske aplikacije i GraphQL poslužitelja. U primjeru ću navesti koje su prednosti implementacije GraphQL-a, te napraviti usporedbu sa REST-om.

Kao što sam već naveo GraphQL najveću primjenu trenutno pronalazi u aplikacijama za mobilne uređaje koji su ograničeni količinom podataka koje mogu preuzeti. Ta ograničenja mogu biti zbog brzine izvedbe ili ograničenja količine podatkovnog prometa. Društvene mreže kao što su Facebook i Instagram, na korisnikovom uređaju učitavaju velike količine podataka u relativno kratkom vremenskom periodu. Da bi izbjegli navedena ograničenja upravo je Facebook osmislio GraphQL.

Aplikacija koju ću u ovom primjeru prikazati funkcionira kao knjižnica. Ona omogućuje korisniku uvid u različite autore, te knjige koje su napisali. Uvid u knjige pojedinog autora ostvarujem odabirom tog autora. Odabirom pojedine knjige otvara se novi prozor koji nam prikazuje dodatne informacije o knjizi kao što su žanr i godina prvog izdanja.

Iz razloga što GraphQL omogućava dodatne funkcionalnosti osim samo dohvaćanja sadržaja, korisniku će u primjeru biti omogućeno da doda nove autore i knjige. Također moći će ukloniti knjige i autore koji mu ne odgovaraju i ažurirati postojeće u slučaju da postoji pogreška u podacima.

4.1 Podaci

Svi podaci biti će spremljeni na strani poslužitelja, a klijentska aplikacija će ih dohvaćati preko GraphQL upita. Klijent će imati mogućnost izmjene podataka unutar baze podataka koristeći GraphQL mutacije. Na taj način je osigurana dovoljna razina apstrakcije koja omogućava korisničkoj aplikaciji dohvaćanje podataka bez direktnog znanja o tome kako radi poslužitelj.

Za izradu aplikacije prvo što je potrebno su podaci koji će se prikazati korisniku u aplikaciji. U tu svrhu mogu se iskoristi različiti API-ji koji postoje kao što su GoodReads API ili Google Books API [15]. Klijentskoj aplikaciji oni mogu dati podatke, ali joj ne mogu omogućiti izmjene. Iz tog razloga u ovom primjeru ću izradit manji skup knjiga i autora nad kojim ću moći raditi izmjene kao što su dodaj, ukloni i ažuriraj.

S obzirom da aplikacija predstavlja knjižnicu koja sadrži knjige grupirane po nazivu autora, prvi skup podataka predstavljaju autori. Podatke koje pojedini autor ima su naziv i identifikacijska oznaka. U tablici 1 su prikazani neki od autora odnosno njihovi nazivi i oznake. Naziv autora predstavlja podatak koji će aplikacija prikazati korisniku, a *id* će se između ostalog koristiti u kodu aplikacije za identificiranje pojedinog autora. Naziv autora predstavlja tekstualni tip podataka, a *id* cjelobrojnu vrijednost.

Tablica 1 - autori

name	id
Hans Christian Andersen	1
J. K. Rowling	2
F. Scott Fitzgerald	3
Dan Brown	4
Fjodor Dostojevski	5

Osim što baza podataka mora sadržavati autore, mora sadržavati i knjige pojedinih autora. U tablici 2 prikazane su neke od knjiga koje se nalaze u bazi podataka. Podaci koji čine pojedinu knjigu su *name, id, authorId, year i genre*. Podatak name je tekstualni tip podataka koji predstavlja naziv pojedine knjige. Taj podatak će se najčešće koristiti za prikaz pojedine knjige korisniku u klijentskoj aplikaciji. Slično kao i kod pojedinog autora, id predstavlja cjelobrojni tip podataka koji između ostalog služi za identifikaciju pojedine knjige u kodu aplikacije. Podaci year i genre daju dodatne informacije vezane za pojedinu knjigu. Ti podaci su čisto informativnog karaktera, tekstualnog tipa.

Tablica 2 – knjige

name	id	authorId	year	genre
Djevojčica sa šibicama	1	1	1845	bajka
Carevo novo ruho	2	1	1837	bajka
Snježna kraljica	3	1	1844	bajka
Harry Potter i kamen mudraca	4	2	1997	fantastika
Veliki Gatsby	9	3	1925	Modernistički roman

Podatak *authorId* ima jako bitnu ulogu, a to je povezivanje pojedinog autora i knjiga koje je on napisao. Id u tablici 1 i *authorId* u tablici 2 predstavljaju isti podatak, koji ovisno o vrijednosti povezuje autora i knjige. Iz tablica 1 i 2 je vidljivo da su autori „Hans Christian Andersen“ i knjige „Djevojčica sa šibicama“, „Carevo novo ruho“ i „Snježna kraljica“ međusobno povezani

na taj način jer id autora i *authorId* pojedinih knjiga imaju istu vrijednost, odnosno vrijednost jedan. Isto vrijedi i za sve ostale autore i knjige.

4.2 Visual Studio Code

Za izradu mrežnih aplikacija možemo koristiti različite alate i programe. Jedan od popularnijih uređivača izvornog koda je i Visual Studio Code, koji ću koristiti za izradu aplikacije u ovom primjeru. Osmišljen od strane Microsofta 2015. godine VS Code predstavlja najpopularniji alat za razvoj softvera. U anketi iz 2021. 70% od 82 000 ispitanika je izjavilo da koristi VS Code [16].

Najveća prednost ovog uređivača koda je mogućnost korištenja sa različitim programskim jezicima kao što su Java, JavaScript, Node.js i Python. VS Code omogućuje korisnicima otvaranje jednog ili više direktorija koji se mogu spremiti za buduću ponovnu upotrebu. Ta funkcionalnost omogućuje VS Code-u da radi kao uređivač koda koji ne ovisi o jeziku.

VS Code omogućava korištenje različitih programskih jezika i rješenja unutar jednog uređivača što je upravo ono što je potrebno za izradu mrežnih aplikacija. VS Code će znatno olakšati izradu aplikacije u ovom primjeru jer ću za izradu koristiti različite programske jezike kao što su JavaScript, HTML i CSS.

4.3 Struktura mrežne aplikacije

Nakon definiranja baze podataka i strukture podataka unutar nje, te podatke je moguće koristiti unutar aplikacije. Za dohvaćanje podataka iz baze koristi se REST ili GraphQL API ovisno o potrebama i složenosti aplikacije. U ovom primjeru izradit ću GraphQL API jer želimo iskoristiti mogućnost dohvaćanja samo onih podataka koje želimo, a tu funkcionalnost RESTful API ne omogućava.

U ovom primjeru za ispravno funkcioniranje aplikacije, GraphQL je potrebno implementirati na razini poslužitelja i klijenta. Osnovna struktura aplikacije se sastoji od baze podataka, poslužitelja i klijenta kao što je prikazano na slici 11. Postoje različiti alati za izradu GraphQL klijenta i poslužitelja, a sljedećim poglavljima navest ću koji su alati najpopularniji i zašto.

4.3.1 GraphQL poslužitelj

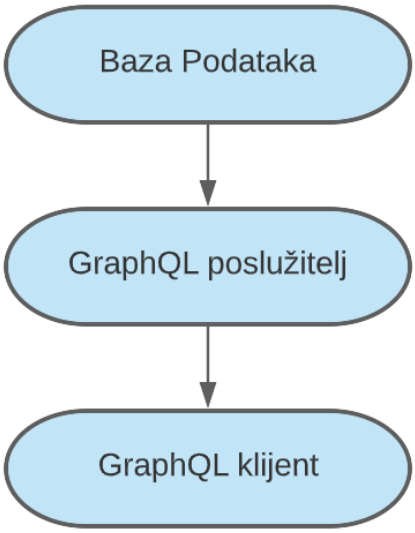
Za ispravan rad aplikacije odnosno, dohvaćanje i izmjenu podataka potrebno je implementirati GraphQL poslužitelj. Na slici 11 on predstavlja dio koji komunicira sa bazom podataka i klijentskom aplikacijom. Neki od najpopularnijih alata za izradu GraphQL poslužitelja su Express GraphQL, Apollo GraphQL poslužitelj i GraphQL Yoga.

Apollo GraphQL poslužitelj

Trenutno najpopularniji GraphQL poslužitelj otvorenog koda koji je kompatibilan sa bilo kojim GraphQL klijentom. Također može koristiti podatke iz bilo kojeg izvora te je jednostavan za korištenje [17].

Express GraphQL

Express GraphQL API omogućuje najjednostavniju izradu GraphQL API poslužitelja. Nije svestran kao Apollo poslužitelj, odnosno nema neke od dodatnih mogućnosti, ali to nije velika mana. Može se dobro iskoristiti za učenje i izradu osnovnih upita. Iz tog razloga u aplikaciji koju ću prikazati u ovom radu ću koristiti upravo ovaj alat za izradu GraphQL poslužitelja.



Slika 11 - struktura mrežne aplikacije

4.3.2 GraphQL klijent

Predstavlja dio aplikacije koji osim što komunicira sa korisnikom, šalje upite poslužitelju vezano za dohvaćanje podataka i koristi mutacije za izmjenu podataka.

Reley

Facebook-ova JavaScript biblioteka za implementaciju GraphQL-a sa klijentske strane. Najčešće se koristi za izradu aplikacija u React-u, te predstavlja alternativu Apollo klijentu.

Apollo klijent

Najpopularnija biblioteka za upravljanje stanjem pomoću GraphQL-a unutar JavaScript aplikacija. Osim što podržava dohvaćanje i izmjenu podataka na poslužitelju, dodaje mogućnost predmemoriranja zahtjeva kako bi se ostvarila dodatna ušteda podatkovnog

prometa. U okviru projekta u ovom radu koristiti ću upravo ovaj alat za izradu GraphQL klijenta iz razloga što je korištenje dosta intuitivno i moguće je slati zahtjeve na različite poslužitelje pa tako i na Express GraphQL.

4.4 Izrada GraphQL poslužitelja koristeći Express GraphQL

Nakon izrade baze podataka potrebno je implementirati GraphQL poslužitelj kako bi klijentskoj aplikaciji omogućili dohvaćanje podataka iz nje. Primjer korištenja GraphQL-a koji ću prikazati u ovom radu uključuje dva zasebna djela. Prvi dio aplikacije predstavlja GraphQL poslužitelj. U radi je već spomenuto da ću za tu svrhu koristiti Express GraphQL. On predstavlja najjednostavniji način za pokretanje GraphQL API poslužitelja, sa osnovnim mogućnostima [18].

Za korištenje Express GraphQL-a potrebno je instalirati dva dodatna paketa unutar aplikacije. Te dodatke instaliramo korištenjem naredbe „*npm i express-graphql graphql*" unutar terminala u direktoriju aplikacije. Za korištenje funkcionalnosti ovih paketa potrebno ih je prvo uvesti u kodu aplikacije kao što je prikazano u kodu 1.

```
const express = require('express')
const expressGraphQL = require('express-graphql')
```

Kod 1- Express GraphQL

Express je minimalan i fleksibilan okvir web aplikacije Node.js koji omogućava različite mogućnosti kod izrade web i mobilnih aplikacija. Omogućuje brzo i jednostavno stvaranje API-ja korištenjem HTTP uslužnih metoda i među programa. Postoji mnogo mrežnih okvira temeljenih na Express-u među kojima je i Express GraphQL.

Korištenje modula *express-graphql* kao što je prikazano u kodu 1 omogućuje staranje Express poslužitelja koji pokreče GraphQL API, odnosno izradu poslužitelja na temelju GraphQL sheme [18].

```
const {GraphQLSchema, GraphQLObjectType, GraphQLString, GraphQLList,
      GraphQLInt, GraphQLNonNull} = require('graphql')
```

Kod 2 - GraphQL objekti

Za izradu GraphQL poslužitelja potrebno je uvesti dodatne module iz root modula *graphql* kao što je prikazano u kodu 2. Moduli kao što su *GraphQLSchema* i *GraphQLObjectType* koriste se definiranje GraphQL shema i tipova. Postoji više modula osim koji su navedeni u kodu 2. Navedeni predstavljaju neke od osnovnih modula za izradu GraphQL poslužitelja. Unutar

primjera izrade GraphQL poslužitelja koji ću prikazati u ovom radu ovi moduli se često koriste, a detaljnije ću pisati o njima kod implementacije pojedinih modula.

4.4.1 GraphQL shema

Shema predstavlja osnovni model podataka GraphQL poslužitelja. Poslužitelj koristi shemu za opisivanje oblika svih tipova podataka čija polja predstavljaju vrijednosti baze podataka. Osim tipova podataka shema definira dostupne upite, mutacije i pretplate. GraphQL shema pisana je u SDL-u (engl. *Schema Definition Language*). Shema korištena za izradu GraphQL poslužitelja vezano za projekt koji ću prikazati u ovom radu prikazana je u kodu 3.

```
const schema = new GraphQLSchema({
  query: RootQueryType,
  mutation: RootMutationType,
  subscription: RootSubscriptionType
})
```

Kod 3 - GraphQL shema

Kod izrade GraphQL poslužitelja korištenjem Express GraphQL-a shema predstavlja instancu modula *GraphQLSchema* koji smo prethodno uvezli kao što je prikazano u kodu 2. Ta shema definira 3 osnovne vrste operacija. *Query* predstavlja upite koje koristimo za dohvaćanje podataka sa poslužitelja. U ovom primjeru sve vrste upita koje se koriste unutar aplikacije definirane su unutar *RootQueryType-a*. Mutacije odnosno operacije izmjene podataka u bazi definirane su unutar *RootMutationType-a*. Ako aplikacija koristi dohvaćanje podataka u stvarnom vremenu potrebno je definirati i *RootSubscriptionType*.

Osim tipova vezanih za korištenje upita, mutacija i pretplata potrebno je definirati i tipove podataka. U ovom primjeru za korištenje podataka koji se nalaze u bazi potrebno je definirati samo dva tipa podataka. Ti tipovi su *BookType* i *AuthorType* tip sa pripadajućim poljima. Shema osim što definira tipove definira i veze između njih. Različiti tipovi mogu biti međusobno povezani što ću i prikazati u kodu aplikacije.

4.4.2 GraphQL tipovi podataka

Osim definiranja operacija unutar GraphQL sheme potrebno je definirati i tipove podataka kako bi poslužitelj znao koje podatke vratiti klijentu. Kao što sam već spomenuo unutar aplikacije ću koristi dva tipa podataka. Implementacija *BookType* tipa sa pripadajućim poljima prikazana je u kodu 4.

Svaki tip podataka predstavlja instancu modula *GraphQLObjectType* unutar koje definiramo polja i ostale potrebne podatke. Potrebno je definirati naziv i opis tipa kao što je prikazano u kodu 4, a zatim polja pojedinog tipa. Polja moraju odgovarati vrijednostima unutar baze podataka koja je prikazana u tablici 2.

```
const BookType = new GraphQLObjectType({
name: 'Book',
description: 'Knjiga tip',
fields: () => ({
id: { type: GraphQLNonNull(GraphQLInt) },
name: { type: GraphQLNonNull(GraphQLString) },
authorId: { type: GraphQLNonNull(GraphQLInt) },
year: { type: GraphQLNonNull(GraphQLInt) },
genre: { type: GraphQLNonNull(GraphQLString) },
author: {
type: AuthorType,
resolve: (book) => {
return authors.find(author => author.id === book.authorId)
        }
      }
   })
})
```

Kod 4 – BookType

Svako polje sadrži podatke navedene vrste. Tipovi polja unutar GraphQL tipa mogu biti skalari, objekti, enum, unija ili apstraktni tipovi. Unutar pojedinog tipa podataka potrebno je navesti koji tipovi podataka predstavljaju pojedina polja, a to u okviru ovog projekta postižemo korištenjem modula *GraphQLInt* i *GraphQLString*. Polja *id*, *authorId i year* predstavljaju cjelobrojne vrijednosti što odgovara tipu *GraphQLInt*.

Polja tipa *BookType* odnosno polja *name* i *genre* predstavljaju tekstualne vrijednosti što odgovara modulu *GraphQLString*. Osim skalarnih tipova podataka *String* i *Int* za definiranje tipa polja možemo koristiti *Float*, *Boolean* i *ID*. *ID* je također cjelobrojna vrijednost, a predstavlja jedinstveni identifikator objekta. Također moguće je definirati i vlastiti skalarni tip podatka [19]

Kako bi osigurali da sva polja moraju poprimiti neku vrijednost kod definiranja polja koristimo *GraphQLNonNull* modul. U slučaju da poslužitelj pokuša vratiti praznu vrijednost na polje koji je definirano korištenjem *GraphQLNonNull* modula, korisniku se javlja greška.

U okviru aplikacije prije nego definiramo GraphQL operacije potrebno je definirati i drugi tip podataka, odnosno *AuthorType* tip. Slično kao i *BookType* tip podataka, *AuthorType* mora

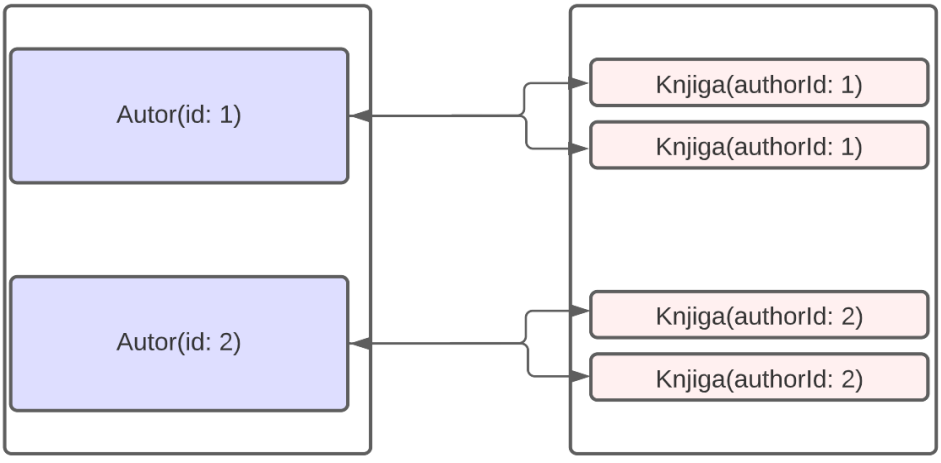
odgovarati podacima iz baze prikazanim u tablici 1. Implementacija *AuthorType* tipa prikazana je u kodu 5.

```
const AuthorType = new GraphQLObjectType({
  name: 'Author',
  description: 'autor tip',
  fields: () => ({
    id: { type: GraphQLNonNull(GraphQLInt) },
    name: { type: GraphQLNonNull(GraphQLString) },
    books: {
      type: new GraphQLList(BookType),
      resolve: (author) => {
        return books.filter(book => book.authorId === author.id)
      }
    }
  })
})
```

Kod 5 - AuthorType

Svi tipovi podataka unutar Express GraphQL poslužitelja definirani su kao instanca *GraphQLObjectType*-a pa tako i *AuthorType* tip. Kao i ostali tipovi podataka potrebno je navesti naziv, opis i polja kao što je prikazano u kodu 5. Polje *id* predstavlja cjelobrojnu vrijednost koja ne smije biti prazna. Polje ime također ne smije biti prazno, a predstavlja tekstualnu vrijednost. Definiranjem ovog tipa implementirani su svi tipovi koji su potrebni za izradu GraphQL poslužitelja. Sva polja pojedinih tipova odgovaraju podacima u tablici 1 i tablici 2.

Jedino što je sad potrebno je napraviti vezu između *AuthorType* i *BookType* tipova. Ta veza predstavlja vezu jedan prema više, odnosno jedan autor je napisao jednu ili više knjiga, a jedna knjiga ima samo jednog autora kao što je prikazano na slici 12. Na ovaj način klijentu je



Slika 12 - veza između autora i knjiga

omogućeno ne samo dohvaćanje podataka o pojedinoj knjizi koristeći samo jedan upit nego i dohvaćanje podataka vezanih za autora te knjige koristeći isti upit, što slični protokoli i arhitekture ne omogućavaju na ovakav način.

U ovom primjeru podaci unutar tablice 1 i tablice 2 u bazi podataka nisu povezani, što je potrebno napraviti u GraphQL poslužitelju. U slučaju da GraphQL poslužitelj omogućava dohvaćanje polja povezanih tipova takve tipove nazivamo ugniježđeni tipovi. To postižemo na način da svakom povezanom tipu dodamo polje koje će predstavljati ugniježđeni tip.

Na primjeru *BookType* tipa, kao što je prikazano u kodu 4, kako bi omogućili korisniku dohvaćanje podataka o autoru knjige zajedno sa podacima o pojedinoj knjizi dodajemo novo polje koje se naziva *author*. Tip tog polja će biti *AuthorType* što će povezati tipove podataka i na taj način omogućiti dohvaćanje polja autora pojedine knjige. U tu svrhu koristimo metodu *resolve()*, o kojoj će biti govora kasnije. Za sad je dovoljno reći da će ta metoda prilikom upita vezanih za pojedinu knjigu pronaći odgovarajućeg autora uspoređivanjem *authorId-a* autora i *id-a* knjige, te vratiti ona polja *AuthorType* tipa koja klijent traži.

Slično tome, unutar aplikacije moramo omogućiti klijentskoj strani da prilikom slanja upita vezanih za pojedinog autora, može dohvatiti i podatke o knjigama koje je taj autor napisao. To postižemo dodavanjem novog polja unutar *AuthorType* tipa kao što je prikazano u kodu 5. U ovom slučaju novo polje *books* će biti tipa *GraphQLList*. Drugim riječima polje *books* će sadržavati niz svih knjiga autora, a svaka knjiga će biti instanca *BookType* tipa. To postižemo pretraživanjem baze podataka knjiga kako bi pronašli one koje imaju vrijednost polja *authorId* jednaku *id-u* autora.

Na ovaj način postignuta je veza između pojedinih tipova, što klijentu omogućuje slanje ugniježđenih upita. RESTful API ne omogućava ovakvo slanje upita. Za dohvaćanje dva tipa podataka korištenjem RESTful API-ja potrebno je poslati dva upita, dok je na ovaj način potrebno poslati samo jedan. Također ovim je završeno definiranje tipova podataka i njihovih polja što omogućuje pisanje tipova operacija koje omogućava GraphQL poslužitelj.

4.4.3 Tipovi GraphQL operacija

Za izradu GraphQL sheme potrebno je definirati tipove podataka i načine na koje ih korisnik može dohvatiti ili mijenjati. Načini dohvaćanja i izmjene podataka predstavljaju tipove GraphQL operacija. Kod izrade GraphQL sheme u okviru ovog projekta definirane su tri vrste operacija vezanih za dohvaćanje podataka, izmjene podataka i dohvaćanje podataka u stvarnom vremenu.

Vrsta GraphQL operacije koja se najčešće koristi je upit za dohvaćanje podataka sa poslužitelja. U kontekstu aplikacije korištenjem Express GraphQL-a za izradu GraphQL poslužitelja potrebno je definirati sve upite unutar *RootQueryType-a*. Sve upiti su vezani za tipove podataka koji su prethodno definirani u kodu GraphQL poslužitelja.

U kodu 6 prikazan je način definiranja *RootQueryType-a* koji sadrži sve upite vezane za dohvaćanje podataka. Iz razloga što se radi o tipu, jednako kao i tipovi podataka *RootQueryType* je instanca modula *GraphQLObjectType-a*. Osim što je potrebno definirati svojstva kao što su naziv i opis tipa, potrebno je navesti i polja kao što je prikazano u kodu 6. Polja u ovom slučaju ne označavaju varijable koje sadrže vrijednosti pojedinog resursa kao kod tipa podataka, nego sadrže sve upite vezane za dohvaćanje podataka sa poslužitelja koje klijentska aplikacija može poslati [20].

```
const RootQueryType = new GraphQLObjectType({
  name: 'Query',
  description: 'Root Query',
  fields: () => ({
    books: {
      type: new GraphQLList(BookType),
      description: 'List Knjiga',
      resolve: () => books
    }
})
```

Kod 6 - RootQueryType sa osnovnim upitom

U kodu 6 prikazan je primjer GraphQL upita vezanog za dohvaćanje knjiga sa poslužitelja. Na ovaj način definirano je što klijentska aplikacija može očekivati kao odgovor pošalje li upit vezan za listu knjiga. U upitu klijentska aplikacija mora koristiti izraz *books* kako bi dobila tražene informacije. U slučaju takvog upita poslužitelj kao odgovor vrača niz koji se sastoji od elemenata tipa *BookType*. Kakav tip podataka GraphQL poslužitelj vrača navedeno je u svojstvu *type* pojedinog GraphQL upita. U primjeru koji je prikazan u kodu 6, tip podataka je instanca modula *GraphQLList* sa elementima *BookType*. Drugim riječima klijentska aplikacija će kao odgovor dobiti niz koji sadrži sve knjige i može pristupiti poljima knjiga koja je naveo u upitu.

4.4.4 GraphQL Resolver

GraphQL poslužitelj mora znati kako popuniti podatke za svako polje unutar sheme kako bi mogao odgovoriti na upite koji zahtijevaju te podatke. U tu svrhu koriste se *resolveri*. Oni predstavljaju funkciju koja je odgovorna za popunjavanje podataka polja tipova. *Resolver* može

popuniti podatke na više načina, a u primjeru aplikacije u ovom radu podatke popunjava iz baze podataka koja je prikazana u tablicama 1 i 2. Na primjeru upita u kodu 6 funkcija *resolve()* popunjava polja *BookType* tipa svih knjiga podacima iz tablice 2.

Slika 13 - GraphQL resolver

Kao što je prikazano na slici 13 postupak dohvaćanja podataka sa GraphQL poslužitelja se sastoji od nekoliko koraka. Prvo klijentska aplikacija šalje upit vezano za podatke koje želi dohvatiti. U tom upitu mora navesti koja polja tipa podatka želi dohvatiti. Zatim *resolver* popunjava polja, za koja je klijent poslao upit, iz baze podataka. Na kraju korisniku se šalje odgovor koji sadrži tražene podatke. Definiranjem kako se popunjavaju podaci završena je implementacija upita na strani poslužitelja te je omogućeno slanje upita klijentskoj aplikaciji.

4.4.5 Pokretanje Express GraphQL poslužitelja

Jedino što preostaje kako bi klijentskoj aplikaciji bilo omogućeno korištenje podataka sa GraphQL poslužitelja je izraditi krajnju točku. Preko nje klijentu omogućavamo pristup informacijama koristeći URL. U ovom primjeru klijentska aplikacija pristupa informacija preko URL-a: „*http://localhost:5000/graphql*“.

```
app.use('/graphql', expressGraphQL({
  schema: schema,
  graphiql: true
}))
app.listen(5000, () => console.log('Sve ispravno'))
```

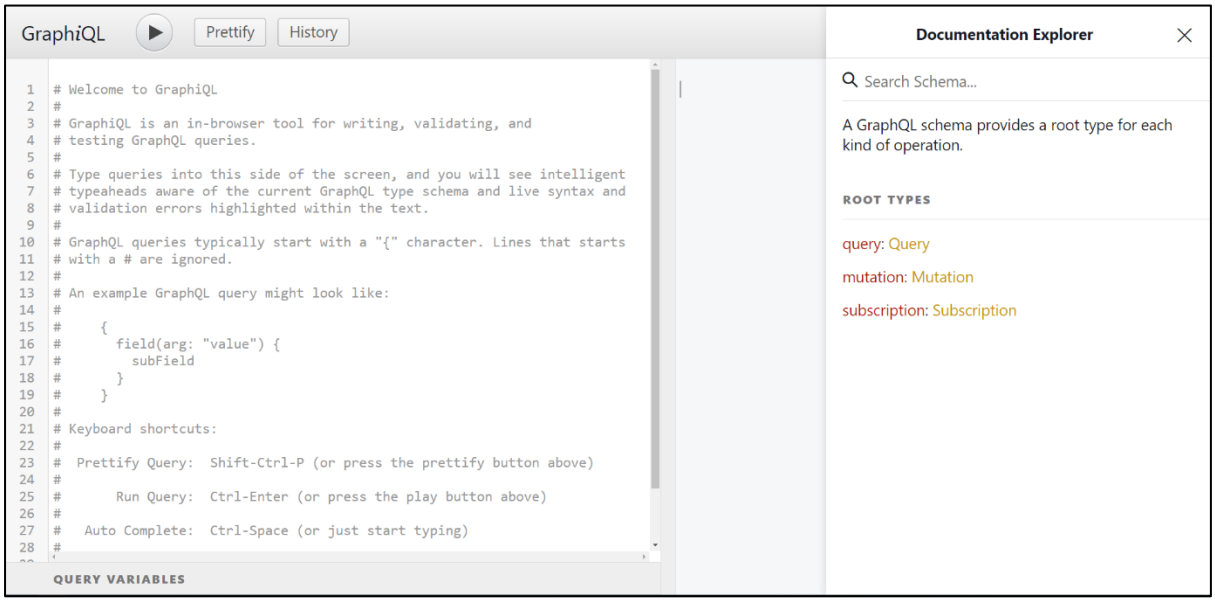
Kod 7 - app.use()

Kao što je prikazano na u kodu 7 pozivanjem metode *app.use()* kojoj šaljemo više vrijednosti koristimo krajnju točku kojoj pristupamo preko „*./graphql*“. Osim načina preko kojeg pristupamo krajnjoj točki, metodi *app.use()* šaljemo i instancu *expressGraphQL* modula. Prvo svojstvo te instance je shema koju smo prethodno definirali koja sadrži sve tipove i *resolvere*. Svojstvo *graphiql* u slučaju da je postavljeno na vrijednost *true* omogućava korištenje GraphQL IDE alata u web pregledniku. Korištenjem ovog alata možemo izravno pisati upite i na taj način uvjeriti se u ispravnost krajnje točke.

Važno je još spomenuti da GraphQL poslužitelj ima jednu krajnju točku, za razliku od poslužitelja temeljenog na REST arhitekturi. Kasnije pri dohvaćanju podataka vezanih za autora knjige nije potrebno izraditi drugu krajnju točku, nego i tim podacima pristupamo preko „*./graphql*“.

Na kraju pozivamo metodu *app.listen()* koja pokreče poslužitelj na portu *5000*. U slučaju da je kod ispravan, pri pristupu GraphQL poslužitelju preko preglednika u konzoli se ispisuje poruka „Sve ispravno“. U okviru ovog projekta za pokretanje poslužitelja, unutar direktorija projekta izvršavamo naredbu „*npm run devStart*“.

Nakon pokretanja procesa poslužitelja u terminalu bi trebali dobiti poruku „Express GraphQL poslužitelj sada radi na „*localhost:5000/graphql*“. Otvaranjem preglednika i unosom URL-a „*localhost:5000/graphql*“ trebao bi se prikazati GraphQL IDE kao što je prikazano na slici 15.



Slika 14 - GraphiQL

GraphQL IDE omogućava programeru testiranje funkcionalnosti GraphQL poslužitelja prije implementacije. Kao što je prikazano na slici 16, na postavljen upit dobijemo odgovor jedno pored drugog. Ovakav prikaz programeru može uvelike olakšati razumijevanje GraphQL upita i odgovora. Osim što podržava pisanje svih vrsta operacija kao što su upiti, mutacije i pretplate postoji mogućnost pregleda svih operacija koje poslužitelj omogućava kao što je prikazano u desnom dijelu slike 15.

4.4.6 Slanje upita

Kako bi testirali ispravnost upita vezanih za listu svih knjiga u GraphQL IDE-u postavljamo upit kao što je prikazano na slici 16. Upit za dohvaćanje podataka na početku bi trebao

sadržavati ključnu riječ „*query*“. Kako bi dohvatili listu knjiga potrebno je koristi naziv upita kao što je definirano unutar GraphQL poslužitelja. U ovom slučaju taj naziv je „*books*“.

Osim naziva, potrebno je navesti polja koja želimo dohvatiti. Upravo ovo predstavlja najznačajniju prednost GraphQL-a, u odnosu na REST. Iako se upit i dobiven odgovor u ovom slučaju ne razlikuju potpuno od REST GET zahtjeva GraphQL upit klijentu omogućava biranje polja koja želi dohvatiti. Korištenjem GET zahtjeva dohvaćamo sve podatke vezane za određeni resurs što klijentskoj aplikaciji nije uvijek potrebno. U primjeru na slici 16 dohvaćeni su svi podaci vezani za pojedinu knjigu, kako bi se uvjerili da su sva polja ispravno popunjena, iako unutar aplikacije neću koristiti ovaj upit.

Slika 15 - Lista knjiga

Korištenjem dokumentacije sa desne strane GraphQL IDE-a imamo uvid u polja pojedinog tipa. U slučaju *BookType* tipa to su polja *id*, *name*, *authorId*, *year*, *genre* i *author*. Usporedbom upita, odgovora i dokumentacije vidljivo je da odgovor odgovara upitu. Odgovor je prikazan u JSON formatu kao niz knjiga, gdje svaki element pojedine knjige sadrži definirana polja. Također odgovor strukturno odgovara upitu, što uvelike olakšava pisanje upita i omogućuje klijentskoj aplikaciji da predvidi kako će odgovor izgledati.

Osim podataka o knjigama, klijentskoj aplikaciji je potrebno omogućiti dohvaćanje podataka o autorima koji se nalaze unutar baze podataka kao što je prikazano u tablici 1. Implementacija upita za dohvaćanje podataka o autoru je slična kao implementacija upita o knjigama.

```
authors: {
      type: new GraphQLList(AuthorType),
      description: 'List Autora',
      resolve: () => authors
    },
```

Kod 8 - svi autori

Kako bi omogućili korisniku slanje upita za dohvaćanje podataka o autorima potrebno je u *RootQueryType* tipu dodati polje *authors* kao što je prikazano u kodu 8. Kao odgovor na *authors* upit klijentske aplikacije poslužitelj šalje niz čiji su elementi instance *AuthorType* tipa što je navedeno u svojstvu *type* upita *authors*. Pri slanju odgovora na upit, metoda *resolve()* popunjava polja svake instance *AuthorType* tipa iz baze podataka. Implementacijom ovog upita sa strane GraphQL poslužitelja korisniku je omogućeno slanje upita za dohvaćanje podataka o autorima.

Slanjem upita *authors* kao što je prikazano na slici 17, klijentska aplikacija može dohvati podatke vezane za sve autore sa poslužitelja. Potrebno je navesti koja polja želimo dohvatiti. U ovom slučaju to su polja *id* i *name* kao što je prikazano na desnom djelu slike 17. Možemo također vidjeti da je poslužitelj poslao ispravan odgovor sa podacima svih autora.

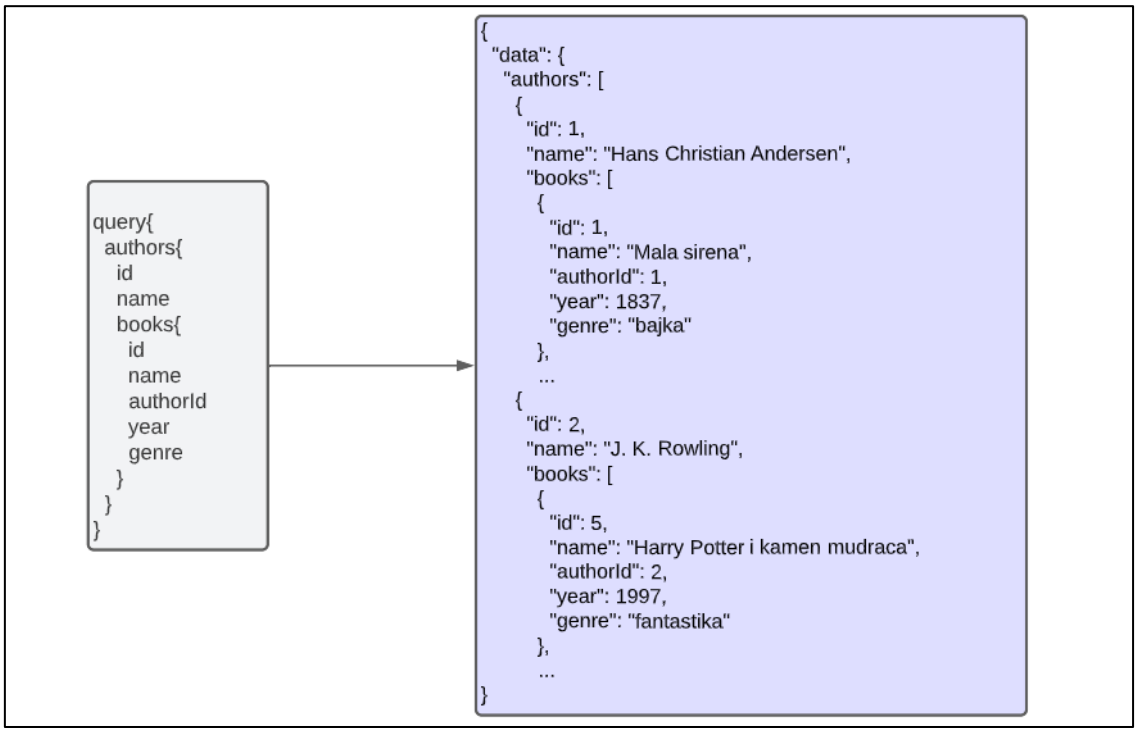
Slika 16 - Lista autora

4.4.7 Ugniježđeni upiti

Ono što GraphQL omogućava korisnicima, za razliku od drugih arhitektura i protokola je slanje ugniježđenih upita. Na primjeru koji je prikazan na ovom radu ta funkcionalnost se može iskoristiti za slanje upita za dohvaćanje podataka vezanih za autore i knjige istovremeno. Ova funkcionalnost osim što umanjuje broj upita koje je potrebno poslati, organizira podatke na način da međusobno povezuje autora i pripadajuće knjige.

Implementacija povezivanja različitih tipova prikazana je u kodu 4 i 5. Unutar pojedinog tipa definirano je polje koje predstavlja povezani tip. Polje *author BookType* tipa podataka predstavlja autora gdje polje *id* odgovara polju *authorId* pojedine knjige. Prema istom principu

polje *books AuthorType* tipa predstavlja knjige pojedinog autora. Takva implementacija klijentskoj aplikaciji omogućava slanje upita o autoru unutar upita o knjigama i obratno. Slanje ugniježđenog upita prikazano je na slici 18. Pri slanju ovakvog upita potrebno je navesti polja koja želimo dohvatiti za autore i za knjige. Odgovor koji je zaprimljen od GraphQL poslužitelja sadrži sve tražene podatke.

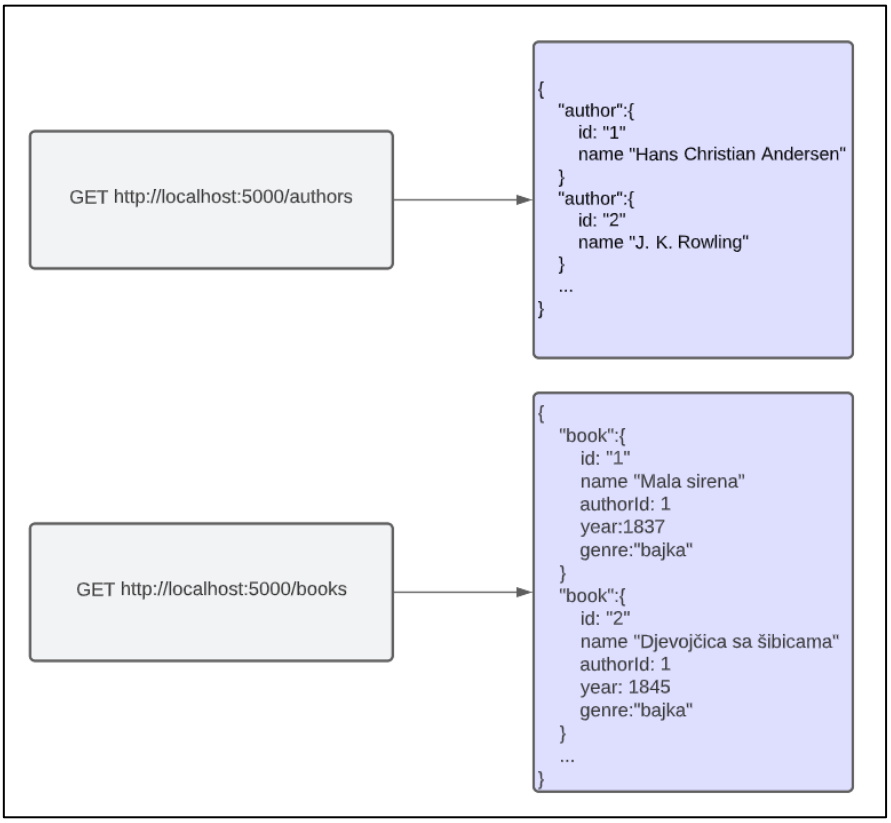


Slika 17 - ugniježđen GraphQL upit

Ugnježđivanje upita predstavlja jedan od razloga zašto programeri odabiru GraphQL u odnosu na slične arhitekture poput REST-a. Dok je u kontekstu ovog primjera potrebno poslati samo jedan GraphQL upit, korištenjem RESTful API-ja za dohvaćanje istih podataka trebalo bi poslati dva zahtjeva. To također znači da će GraphQL poslužitelj poslati jedan odgovor koji sadrži sve podatke, dok će Restful API poslati dva zasebna odgovora kao što je prikazano na slici 19 [12].

Korištenjem dva GET zahtjeva podaci o autorima i knjigama ne bi bili grupirani kao odgovor GraphQL poslužitelja, što znači da bi logika povezivanja autora i pripadajućih knjiga trebala biti unutar klijentske aplikacije, što klijenta dodatno opterećuje. Odabir polja unutar GraphQL upita klijentu omogućava dohvaćanje samo potrebnih podataka, odnosno klijent neće imati problema sa prevelikom količinom podataka koja ga dodatno opterećuje i zauzima podatkovni promet. U ovom primjeru korištenjem RESTful API-ja imali bi problem sa dohvaćanjem prevelike ili premale količine podataka, što bi dovelo do poteškoća koje GraphQL uspješno otklanja [29].

Korištenje GraphQL ugniježđenih upita donosi mnoge prednosti, ali treba biti oprezan. Takvi zahtjevi predstavljaju opterećenje za poslužitelja i kod prevelikih zahtjeva može doći do pada poslužitelja. Na primjeru ugniježđenog upita koji je prikazan na slici 18 moguće je *books* upitu dodati još jedan *authors* upit za dohvaćanje podataka o autoru. Broj upita koji se može dodati



Slika 18 - GET zahtjevi i odgovori

je neograničen, što predstavlja najčešći način na koji zlonamjerni softveri pokušavaju srušiti GraphQL poslužitelj. Kao zaštitu od takvih napada potrebno je koristiti jedan od načina obrane koji sam naveo u poglavlju GraphQL nedostaci.

4.4.8 Slanje argumenata unutar upita

Kako bi najbolje iskoristili prednosti koje GraphQL poslužitelj nudi klijentskoj aplikaciji, programeri pri izradi aplikacije moraju voditi računa o korištenju podataka. Temeljna ideja na kojoj je nastao GraphQL je smanjiti dohvaćanje prevelike količine podataka sa poslužitelja, odnosno ne opterećivati klijenta nepotrebnim informacijama.

U primjeru GraphQL upita koje sam dosad prikazao u ovom radu klijentska aplikacija dohvaća informacije o svim autorima i knjigama. To nije optimalno rješenje ako aplikacija zahtjeva samo neke od autora ili neke od knjiga. U tu svrhu potrebno je definirati upite tako da klijentu omogućimo upravo to. GraphQL upiti omogućuju prosljeđivanje argumenata u polja upita i ugniježđenih upita. Argumente je moguće proslijediti svakom polju unutar upita kako bi

dodatno specificirali koje podatke treba dohvatiti. Argumenti služe istoj svrsi kao parametri upita ili URL segmenti RESTful API-ja [21].

U primjeru koji je prikazan u ovom radu klijentskoj aplikaciji je potrebno omogućiti slanje upita koji sadrže argumente za dohvaćanje podataka o pojedinom autoru ili knjizi. Kao što sam već naveo argumente je moguće slati za sva polja pojedinog upita, ali u okviru projekta klijentskoj aplikaciji je potrebno omogućiti slanje upita koji prosljeđuje argument polju *id*. *Id* predstavlja jedinstvenu oznaku pojedinog autora ili knjige što klijentu omogućuje dohvaćanje podataka o tom resursu.

```
book: {
      type: BookType,
      description: 'Knjiga',
      args: {
        id: { type: GraphQLInt }
      },
      resolve: (parent, args) => books.find(book => book.id === args.id)
    },
author: {
      type: AuthorType,
      description: 'Autor',
      args: {
        id: { type: GraphQLInt }
      },
      resolve:
      (parent, args) => authors.find(author => author.id === args.id)
    }
```

Kod 9 - slanje argumenata

Na GraphQL poslužitelju unutar *RootQueryType-a* potrebno je definirati dva dodatna upita kao što je prikazano u kodu 9. Za razliku od prijašnjih upita, ovaj upit ima dodatno svojstvo *args* koje označava argumente koje poslužitelj može očekivati u upitu. Upit *book* klijentskoj aplikaciji omogućava dohvaćanje podataka o jednoj knjizi, a kao ulazni argument potrebno je poslati cjelobrojnu vrijednost. Isto vrijedi i za *author* upit.

Korištenjem funkcije *resolvera* GraphQL poslužitelj pronalazi odgovarajućeg autora ili knjigu koji ima vrijednost polja *id* jednaku cjelobrojnoj vrijednosti te vraća tražene podatke, odnosno polja koja su tražena za taj resurs. Implementacijom ova dva upita klijentu je omogućeno slanje upita sa argumentima, a primjer takvog upita je prikazan na slici 20.

Primjer upita koji je prikazan na slici 20 je sličan upitima prikazanim u prijašnjim primjerima. U ovom slučaju želimo dohvatiti podatke vezano samo za knjigu kojoj je vrijednost polja *id*

jednaka 2. Osim što je potrebno navesti polja čiju vrijednost želimo dohvatiti potrebno je navesti i argument pored ključne riječi upita. Odgovor koji je poslan sadrži samo jednu knjigu što znači da je implementacija upita valjana. To je bitno jer će upiti koji šalju argumente biti ključni za izradu klijentske aplikacije koju ću prikazati u ovom radu.

Slika 19 - slanje upita sa argumentima

4.5 Izrada klijentske aplikacije

GraphQL poslužitelj koji sam prikazao u radu, klijentskoj aplikaciji može omogućiti dohvaćanje podataka iz baze podataka koristeći slične upite kao što je prikazano na slici 20. Kako bi prikazao korištenje GraphQL upita i odgovora na stvarnom primjeru, izradit ću jednostavnu aplikaciju koja koristi podatke iz baze podataka. Aplikacija će imati 3 prikaza, a omogućiti će korisniku pregled podataka i izmjene. Prvi prikaz će prikazati sve autore koji se nalaze u bazi podataka, drugi knjige odabranog autora, a treći informacije o odabranoj knjizi.

Kod za izradu te aplikacije pisan je u alatu Visual Studio Code, a biblioteke koje su korištene za izradu su *React i Apollo Client. React* je besplatna front-end JavaScript biblioteka otvorenog koda za izgradnju korisničkih sučelja temeljenih na UI komponentama. Biblioteka koja je više zanimljiva u kontekstu ovog rada je Apollo Client. Ona predstavlja biblioteku za JavaScript koja omogućuje upravljanje lokalnim i udaljenim podacima pomoću GraphQL-a. Dok se ponajviše koristi za dohvaćanje i izmjenu podataka, podržava i neke naprednije funkcije poput predmemoriranja podataka. Jedan od razloga zašto sam odabrao Apollo Client je mogućnost dobre integracije sa React UI komponentama, što ću i prikazati u radu.

Za početak je potrebno izraditi novi React projekt i unutar tog projekta izvršiti naredbu „*npm install @apollo/client graphql*“. Instalacija ovog modula omogućava izradu klijentske aplikacije koja može komunicirati sa GraphQL poslužiteljem. Prije slanja upita potrebno je ostvariti vezu između poslužitelja i klijentske aplikacije. Kako bi to ostvarili potrebno je

definirati novu instancu ApolloClient modula kao što je prikazano u kodu 10. Sve što je potrebno navesti za inicijalizaciju ApolloClient-a su polja *uri* i *cache*. U polju uri navodimo URL GrahphQL krajnje točke sa koje želimo dohvatiti podatke. U slučaju ovog projekta to je „*http://localhost:5000/graphql*“. *Cache* je instanca modula *inMemoryCache* koju Apollo Client koristi za predmemoriranje rezultata upita nakon što ih dohvati.

```
import { ApolloClient,
         InMemoryCache,
         ApolloProvider } from '@apollo/client';
const client = new ApolloClient({
  uri: 'http://localhost:5000/graphql',
  cache: new InMemoryCache(),
});
```

Kod 10 - Apollo Client

Kako bi koristili Apollo Client zajedno sa Reactom potrebno je omotati React aplikaciju sa *ApolloProvider-om* kao što je prikazano u kodu 11. Na ovaj način omogućeno je slanje upita iz svih komponenata React aplikacije [22].

```
<ApolloProvider client={client}>
     <App />
 </ApolloProvider>
```

Kod 11 - ApolloProvider

4.5.1 Slanje GraphQL upita iz klijentske aplikacije

Za slanje upita vezanih za dohvaćanje podataka koristimo *useQuery()* metodu. U kodu 12 prikazana je funkcija koja prikazuje dohvaćene podatke. Dohvaćeni podaci su spremljeni u varijabli data. Za prikaz pojedinog autora koristimo React Link UI komponentu koja prikazuje ime autora. Ako je potrebno više vremena da se podaci dohvate ili u slučaju greške ova funkcija će korisniku ispisati poruku o tome.

```
function DISPLAY_AUTHORS() {
  const { loading, error, data } = useQuery(GET_AUTHORS);
  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error :</p>;
  return data.authors.map(({ id, name}) => (
    <div key={id}>
      <Link to="/books" state={{AUTHOR: name, ID: id}}>{name}</Link>
    </div>
  ));
}
```

Kod 12 - DISPLAY_AUTHORS funkcija

Kako bi metoda *useQuery()* mogla dohvatiti podatke potrebno je u kodu aplikacije definirati GraphQL upit. Ta metoda koristi *authors* upit koji je prikazan u kodu 13. Taj upit predstavlja jednostavan GraphQL upit koji dohvaća podatke o svim autorima, odnosno vrijednosti polja *name* i *id*.

```
const GET_AUTHORS = gql`
  query{
    authors{
      id
      name
    }
  }
`;
```

Kod 13 – korištenje authors upita

Kako bi klijentska aplikacija mogla dohvatiti podatke potrebno je prije pokrenuti server, a to možemo napraviti izvršavanjem naredbe „*npm run devStart*“ unutar direktorija GraphQL poslužitelja. Zatim možemo pokrenuti React aplikaciju izvršavanjem naredbe „*npm start*“ unutar direktorija klijentske aplikacije. Ako je sve ispravno napisano u internet pregledniku bi se trebao otvoriti novi prozor na URL-u „localhost:3000“ kao što je prikazano na slici 21.

Slika 20 - prvi prikaz klijentske aplikacije

Prvi prikaz aplikacije se sastoji od nekoliko Link UI elemenata gdje svaki element odgovara jednom autoru. Funkcija *DISPLAY_AUTHORS()* koristiti vrijednost varijable data u kojoj se

nalazi odgovor na GraphQL upit na način da se korisniku prikaže vrijednost *name* polja pojedinog autora. Vrijednost polja *id* se prosljeđuje sljedećem prikazu aplikacije pri odabiru autora. Odabirom pojedinog autora trebao bi se otvoriti novi prikaz sa svim knjigama autora koje se nalaze u bazi [23].

4.5.2 Slanje GraphQL upita sa argumentima iz klijentske aplikacije

Kako bi korisniku omogućili pregled knjiga pojedinog autora unutar aplikacije potrebno je izraditi još jedan prikaz sličan prikazu autora. Također kako bi dohvatili samo knjige odabranog autora potrebno je koristiti vrijednost polja *id* koja je poslana iz prethodnog prikaza. U kodu 14 je prikazan dio metode *DISPLAY_BOOKS()* koja šalje vrijednost polja *id* autora GraphQL upitu *author*.

```
const { loading, error, data } =
  useQuery(GET_BOOKS,{variables: {oznaka: state.ID}});
```

Kod 14 – DISPLAY_BOOKS()

Implementacija upita *author* prikazana je u kodu 15. Postoji više načina na koji se mogu dohvatiti podaci o knjigama pojedinog autora, ali u okviru ovog rada sam se odlučio na ovakav upit iz razloga što prikazuje korištenje ugniježđenih upita unutar klijentske aplikacije. Ovaj upit pronalazi odabranog autora koristeći vrijednost polja *id* koja je proslijeđena iz prijašnjeg prikaza i dohvaća polja *id* i *name* svih knjiga odabranog autora.

```
const GET_BOOKS = gql`
  query author($oznaka: Int){
    author(id: $oznaka){
      books{
        id
        name}
  }
}
`;
```

Kod 15 – korištenje author upita

Za razliku od RESTful API-ja nije bilo potrebno dohvatiti sve knjige, pa pretraživati one koje pripadaju odabranom autoru. Također ovim GraphQL upitom dohvaćene su samo vrijednosti polja *id* i *name* koje će se koristiti unutar komponente, odnosno nije bilo potrebno dohvatiti vrijednosti ostalih polja kao što *year* i *genre*. Korištenjem GraphQL-a u ovom primjeru ostvarena je relativno velika ušteda podataka.

Metoda *DISPLAY_BOOKS()* koristi dohvaćene podatke jednako kao i metoda *DISPLAY_AUTHORS()*. Vrijednost polja *name* pojedine knjige prikazana je korisniku, a vrijednost polja *id* se šalje sljedećem prikazu pri odabiru knjige. Prikaz podataka dohvaćenih *author* upitom prikazan je na slici 22.

Slika 21 - Prikaz knjiga pojedinog autora

4.5.3 Prikaz detalja pojedine knjige

Posljednji prikaz projekta koji sam prikazao u ovom radu sadrži podatke vezane za pojedinu knjigu. Korištenjem vrijednosti polja *id* odabrane knjige na prošlom prikazu koristimo metodu *DISPLAY_INFO()* kako bi prikazali informacije vezane za knjigu. Ta metoda slično kao i metoda *DISPLAY_BOOKS()* pohranjuje odgovor na GraphQL upit na način da ga sprema kao vrijednost varijable data.

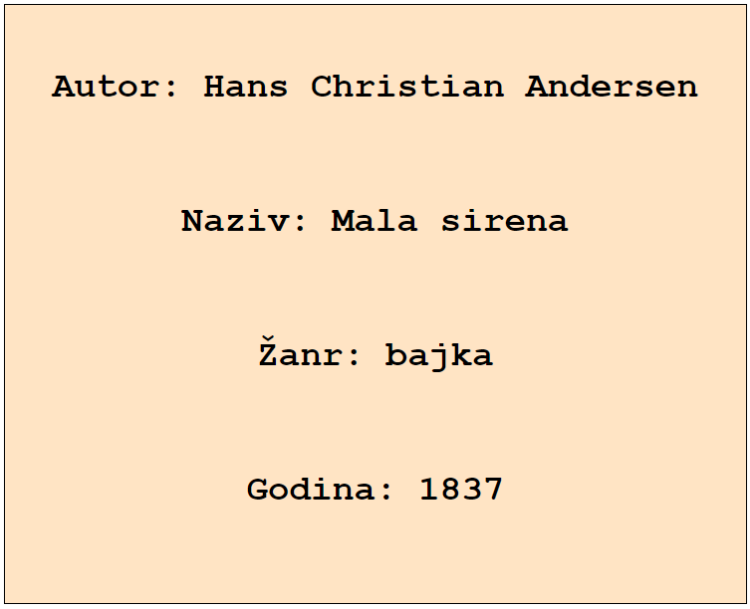
Za dohvaćanje podataka metoda koristi GraphQL upit *book* prikazan u kodu 16. Taj upit dohvaća vrijednosti polja knjige kojoj je vrijednost polja *id* jednaka *id-u* proslijeđenom iz

```
const GET_INFO = gql`
  query book($oznaka: Int){
    book(id: $oznaka){
      author{name}
      name
      year
      genre
  }
}`;
```

Kod 16 – korištenje book upita

prošlog prikaza. Osim što dohvaća vrijednosti polja *name*, *year* i *genre* knjige, dohvaća i vrijednost polja *name* autora knjige.

Za prikaz podataka dohvaćenim book GraphQL upitom koristimo jednostavnu formu koja se sastoji od osnovnih HTML elemenata. Podaci se korisniku prikazuju kao što je prikazano na slici 23. Unutar funkcije *DISPLAY_INFO()* pristupamo podacima koristeći vrijednost varijable data. Za korištenje na primjer vrijednosti polja *name* autora odabrane knjige, unutar HTML elementa H2, podacima pristupamo koristeći data.book.author.name.



Slika 22 - Prikaz detalja odabrane knjige

Implementacijom ovog prikaza prikazani su svi upiti za dohvaćanje podataka koje podržava GraphQL poslužitelj u ovom primjeru. Upiti su najčešća vrsta operacija koje klijentska aplikacija koristi. Principi koji smo koristili za izradu klijentske aplikacije mogu se koristi za slanje upita različitim GraphQL poslužiteljima, bitno je samo znati krajnju točku i strukturu podataka unutar poslužitelja.

4.6 Izmjene podataka korištenjem GraphQL poslužitelja

Osim što GraphQL poslužitelj klijentskoj aplikaciji omogućava dohvaćanje podataka, poželjno bi bilo da joj omogući i izmjene nad podacima. U REST arhitekturi za izmjene podataka koristimo HTTP metode PUT, POST i DELETE. Za razliku od toga GraphQL za izmjene podataka koristi samo jednu vrstu operacija odnosno mutacije. Mutacije se koriste za dodavanje novih podataka, izmjene postojećih ili brisanje podataka. GraphQL usporedno sa REST-om koristi samo jednu vrstu operacija za izmjene podataka što znači da je potrebno na GraphQL poslužitelju definirati što pojedina mutacija točno radi [20].

U okviru projekta koji je prikazan u ovom radu implementirano je šest mutacija. Mutacije se odnose na dodavanje, izmjenu vrijednosti polja *name*, i uklanjanje autora ili knjige. Kako bi omogućili klijentskoj aplikaciji pisanje operacija za izmjene podataka, mutacije je prvo potrebno definirati na poslužitelju jednako kao i ostale GraphQL operacije.

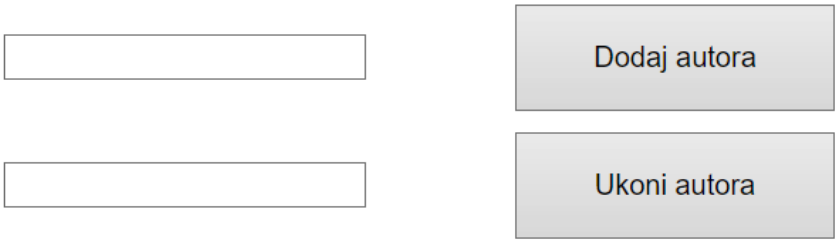
4.6.1 Korištenje mutacija za dodavanje novih podataka

Unutar projekta koji je prikazan u ovom radu potrebno je omogućiti korisniku dodavanje nove knjige ili autora. U tu svrhu potrebno je definirati mutacije na poslužitelju kako bi omogućili izmjene u bazi podataka. Slično kao i kod definiranja upita, mutacije za izmjenu podataka potrebno je definirati unutar sheme GraphQL poslužitelja kao što je prikazano u kodu 3. Osim što *RootMutationType* sadrži operacije za dodavanje novih objekata sadrži i ostale mutacije vezane za izmjene podataka.

```
const RootMutationType = new GraphQLObjectType({
  name: 'Mutation',
  description: 'Root Mutation',
  fields: () => ({
    addAuthor: {
      type: AuthorType,
      description: 'Dodaj autora',
      args: {
        name: { type: GraphQLNonNull(GraphQLString) }
      },
      resolve: (parent, args) => {
        const author = { id: authors.length + 1, name: args.name}
        authors.push(author)
        return author
      }
    }
})
```

Kod 17 – dodavanje autora

U kodu 17 je prikazan tip *RootMutationType* koji sadrži mutaciju za dodavanje novog autora. Mutacija *addAuthor* prima argument koji koristi za izradu novog autora. Taj argument predstavlja tekstualnu vrijednost koja će se spremiti unutar polja *name* novog autora. Nakon



Slika 23 – elementi za dodavanje i brisanje autora

što klijentska aplikacija pozove ovu mutaciju taj argument se šalje *resolveru* koji izrađuje novog autora sa poljima *id* i *name*, te ga dodaje ostalim autorima [20].

Kako bi korisniku omogućili unos novog imena autora na prvom prikazu su dodani elementi kao što je prikazano na slici 24. Odabirom „Dodaj autora“ klijentska aplikacija poziva mutaciju definiranu na poslužitelju, te joj šalje vrijednost koju je korisnik unio u tekstualni okvir.

```
const [addAuthor, { data, loading, error }] = useMutation(ADD_AUTHOR);
  if (loading) return 'Submitting...';
  if (error) return `Submission error! ${error.message}`;
```

Kod 18 – metoda NEW_AUTHOR

Metoda koja poziva mutaciju za dodavanje novog autora prikazana je u kodu 18. Osim toga unutar metode *NEW_AUTHOR()* definirani su elementi za dodavanje novog autora.

```
<form class="txt_autor"
        onSubmit={e => {
          e.preventDefault();
          addAuthor({ variables: { type: input.value}});
          input.value = '';
        }}
      >
```

Kod 19 – korištenje vrijednosti koja je unesena u formu

U kodu 19 prikazan je dio metode *NEW_AUTHOR()* koji je odgovoran za slanje vrijednosti unosa metodi *addAuthor()*. Ta metoda koristi *useMutation() hook* kako bi pozvala mutaciju *addAuthor*. Ovisno o odgovoru poslužitelja *NEW_AUTHOR* također prikazuje poruku korisniku kao što je prikazano u kodu 18 [24].

```
const ADD_AUTHOR = gql`
  mutation addAuthor($type: String!){
    addAuthor(name: $type)
    {id}
  }
`;
```

Kod 20 – korištenje addAuthor mutacije

ADD_AUTHOR, prikazan u kodu 20, poziva mutaciju *addAuthor* koja je prethodno definirana na poslužitelju i šalje joj tekstualnu vrijednost. Kao što je prikazano u kodu ta vrijednost koju je korisnik unio spremiti će se kao vrijednost polja *name* novog autora. Također kako svi upiti moraju vračati vrijednost nekog polja, bezobzira radili se o upitu ili mutaciji, u ovom slučaju vračamo vrijednost polja *id* čija vrijednost će biti jednaka broju autora u bazi podataka.

Nakon što korisnik unese naziv i klikne na „Dodaj autora“ novi autor bi trebao biti dodan. Osvježavanjem prikaza aplikacije naziv novog autora bi se trebao pojaviti na prikazu autora. Također postavljanjem upita „*query{authors{id name}}*“ u GraphQL IDE-u pojavit će se rezultat kao što je prikazano na slici 25.

Slika 24 - novi autor

Dodavanje nove knjige

Koristeći kod koji je implementiran za dodavanje novog autora, uz male izmjene jednostavno se može implementirati dodavanje nove knjige. Kako bi korisniku omogućili dodavanje nove knjige potrebno je na drugom prikazu aplikacije dodati UI elemente slično kao što je prikazano na slici 24. Korisnik unosi naziv knjige u tekstualni okvir, a klikom na tipku „Dodaj knjigu“ dodaje se nova knjiga autoru koje je trenutno odabran na tom prikazu.

```
addBook: {
      type: BookType,
      description: 'Dodaj knjigu',
      args: {
        name: { type: GraphQLNonNull(GraphQLString) },
        authorId: { type: GraphQLNonNull(GraphQLInt) }
      },
      resolve: (parent, args) => {
        const book = { id: books.length + 1, name: args.name,
          authorId: args.authorId, year:0,genre:"Žanr" }
        books.push(book)
        return book
      }
    }
```

Kod 21- mutacija addBook

Kako bi korisniku omogućili tu funkcionalnost potrebno je prvo definirati mutaciju na GraphQL poslužitelju. U kodu 21 je prikazana mutacija *addBook* slična mutaciji *addAuthor* prikazanoj u kodu 17.

Za razliku ova mutacija prima dva argumenta. Prvi argument je naziv knjige koji korisnik unosi, a drugi je *id authora* koji je potreban kako bi znali kojem autoru pridodati novu knjigu. Argumente mutacija prosljeđuje *resolveru* koji stvara novi objekt *book*. Vrijednosti polja *year* i *genre* su u ovom primjeru predefinirane, a vrijednosti polja *name* i *authorId* poprimaju vrijednosti iz argumenata. Na kraju nova knjiga je dodana u bazu podataka.

```
function NEW_BOOK() {
  const location = useLocation();
  const state = location.state;
  let input;
  const [updateAuthor, { data, loading, error }] = useMutation(ADD_BOOK);
  if (loading) return 'Submitting...';
  if (error) return `Submission error! ${error.message}`;
  return (
      <div>
      <form class="txt_book"
        onSubmit={e => {
          e.preventDefault();
          updateAuthor({ variables: { type: input.value, Aid:state.ID}});
          input.value = '';
        }}>
```

Kod 22 - metoda NEW_BOOK

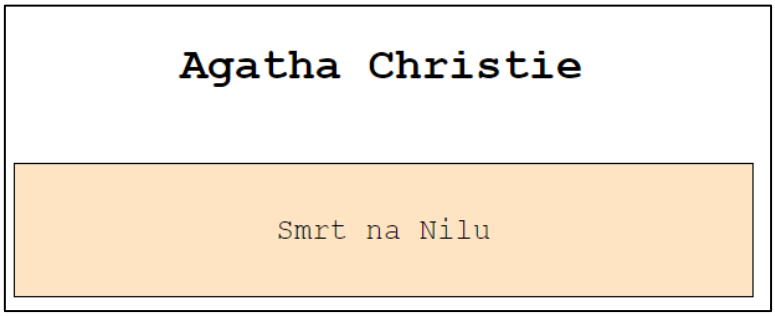
U kodu 23 prikazana je metoda *NEW_BOOK()* koja služi za prikupljanje informacija potrebnih za mutaciju i pozivanje mutacije. Nakon što korisnik unese ime u UI element *form* pritiskom na tipku „Dodaj knjigu“, metodi *updateAuthor()* šaljemo vrijednost koju je korisnik unio i vrijednost unutarnjeg stanja aplikacije. To stanje sadrži vrijednost *authorId* polja odabranog autora, te tu vrijednost na ovaj način prosljeđujemo mutaciji. Na kraju kao i u prošlom primjeru metoda *NEW_BOOK()* koristi *useMutation() hook* kako bi pozvala mutaciju *addBook* prikazanu u kodu 22.

```
const ADD_BOOK = gql`
  mutation addBook($type: String! $Aid: Int!){
    addBook(name: $type authorId:$Aid)
    {id}
  }`;
```

Kod 23 - korištenje addBook mutacije

Ovaj poziv koristi *addBook* mutaciju definiranu na GraphQL poslužitelju kako bi kreirala novu knjigu koristeći proslijeđene vrijednosti. S ovim je završena implementacija dodavanja nove knjige i autora.

Kako bi testirao funkcionalnost ove mutacije dodat ću knjigu autoru koji je prikazan na slici 25. Na slici 26 je prikazan novi autor i knjiga što potvrđuje funkcionalnost ovih mutacija. Slijedeće što je potrebno omogućiti korisniku je uklanjanje autora i knjiga, za što se također koriste mutacije.



Slika 25 - dodana knjiga

4.6.2 Korištenje GraphQL mutacija za uklanjanje podataka

Sljedeća funkcionalnost koju ću implementirati uz pomoć GraphQL mutacija je uklanjanje resursa sa poslužitelja. U primjeru aplikacije prikazane u ovom radu potrebno je korisniku omogućiti uklanjanje pojedinog autora ili knjige iz baze podataka. Za razliku od RESTful API-ja gdje bi koristili različite HTTP metode za dodavanje i uklanjanje autora ili knjige, korištenjem GraphQL sve izmjene podataka izvršavaju se jednom vrstom operacija, razlika je samo u implementaciji tih operacija. Za dodavanje autora i knjige na poslužitelju je bilo potrebno definirati dvije mutacije odnosno za svaki resurs po jednu. Isto je potrebno napraviti i za uklanjanje resursa [20].

Brisanje odabrane knjige

U kodu 24 prikazana je implementacija *removeBook* mutacije koju korisnik može pozvati kako bi uklonio pojedinu knjigu iz baze podataka. Ovu mutaciju je kao i prijašnje mutacije potrebno definirati unutar *RootMutationType-a*. Za korištenje mutacije potrebno je poslati argument, odnosno cjelobrojnu vrijednost koja će uspoređivati sa vrijednosti polja *id* pojedinih knjiga. Ta usporedba je zadatak *resolvera* koji pronalazi knjigu čija je vrijednost jednaka argumentu i uklanja je iz liste knjiga.

```
removeBook: {
      type: BookType,
      description: 'Ukloni knjigu pojedinog autora',
      args: {
        id: { type: GraphQLNonNull(GraphQLInt) }
      },
      resolve: (parent, args) => {
        books.forEach( (book, index) => {
          if (book.id === args.id) {
            books.splice(index,1)
          }
          return books
      })
    }},
```

Kod 24 – removeBook mutacija

Unutar klijentske aplikacije korištenje *removeBook* mutacije implementirano je korištenjem UI elemenata *form* i *button* slično kao što je prikazano na slici 24. Korisnik u tekstualni okvir unosi cjelobrojnu vrijednost i klikom na tipku „Ukloni knjigu“ poziva metodu *removeBook()* koja koristi *useMutation hook* kako bi pozvala mutaciju.

```
const REMOVE_BOOK = gql`
  mutation removeBook($type: Int!){
    removeBook(id: $type)
    {id}
  }
`;
```

Kod 25 – korištenje removeBook mutacije

RemoveBook mutacija koju klijentska aplikacija koristi prikazana je u kodu 25. Metoda *removeBook()* prosljeđuje mutaciji vrijednost koju je korisnik unio, koja se zatim koristi unutar mutacije za uklanjanje knjige. Nakon korištenja ove mutacije i osvježavanja prikaza aplikacije, knjiga sa vrijednosti polja *id* jednakim unosu korisnika uklonjena je sa GraphQL poslužitelja i prikaza aplikacije.

Brisanje odabranog autora

Uklanjanje autora znači i uklanjanje svih knjiga koje pripadaju tom autoru. U tom slučaju potrebno je definirati operaciju koja može ukloniti više resursa odjednom, a korištenjem samo jedne mutacije. *RemoveAuthor* mutacija prikazana u kodu 26 prima vrijednost koju je korisnik unio unutar tekstualnog okvira u prvom prikazu klijentske aplikacije kao što je prikazano na slici 24.

```
removeAuthor: {
      type: AuthorType,
      args: {
        id: { type: GraphQLNonNull(GraphQLInt) }
      },
      resolve: (parent, args) => {
        for(var i=0;i<=books.length-1;i++){
          if (books[i].authorId == args.id) {
            books.splice(i,1)
            i-=1
          }}
        authors.forEach( (author, index) => {
          if (author.id === args.id) {
            authors.splice(index,1)
          }
      })}},
```

Kod 26 – removeAuthor mutacija

Koristeći taj podatak *resolver* ove mutacije prvo pronalazi sve knjige čija je vrijednost polja *authorId* jednaka vrijednosti argumenta, te ih zatim uklanja iz liste knjiga. Nakon što su uklonjene sve knjige odabranog autora, *resolver* pronalazi i autora koristeći unesenu vrijednost i uklanja ga iz liste autora.

```
const REMOVE_AUTHOR = gql`
  mutation removeAuthor($type: Int!){
    removeAuthor(id: $type)
    {id}
  }
`;
```

Kod 27 – korištenje removeAuthor mutacije

Unutar klijentske aplikacije ista metoda koja korisniku prikazuje UI elemente za unos cjelobrojne vrijednosti, na klik korisnika poziva *removeAuthor* mutaciju koja je prikazana u kodu 27. Ovom mutacijom uklanjamo odabranog autora i njemu pripadajuće knjige. Kao i kod implementacije prijašnjih mutacija u slučaju pogreške ili pogrešnog unosa korisniku se ispisuje poruka o tome.

Slika 26 - uklanjanje autora i knjiga

Rezultat korištenja ove mutacije slanjem cjelobrojne vrijednosti 1 prikazan je na slici 27. Kako bi se uvjerili da se promjena nije dogodila samo u klijentskoj aplikaciji moguće je poslati upit unutar GraphQL IDE-a. Na slici je vidljivo da slanjem GraphQL upita ne dobivamo rezultate

vezane za autora čija je vrijednost polja *id* jednaka jedan što znači da su se promjene dogodile ne samo unutar klijentske aplikacije nego i na poslužitelju.

4.6.3 Korištenje GraphQL mutacija za izmjene podataka

Osim što se GraphQL mutacije koriste za dodavanje novih i uklanjanje postojećih resursa, koriste se za mijenjanje vrijednosti polja pojedinih resursa. Mutacija koja vrši izmjene na GraphQL poslužitelju funkcijski odgovara POST metodi. Kako bi korisniku omogućili izmjenu vrijednosti polja pojedinog resursa potrebno je na GraphQL poslužitelju definirati mutacije koje ovisno o vrsti resursa pronalaze određeno polje i mijenjaju njegovu vrijednost.

U primjeru koji prikazujem u ovom radu klijentskoj aplikaciji će biti omogućeno pozivati mutacije koje mijenjaju vrijednost polja *name* knjige ili autora. Bitno je naglasiti kako želimo postići da mutacija ne mijenja ostale vrijednosti polja, odnosno da svi ostali podaci vezani za pojedini resurs ostanu nepromijenjeni [25].

```
updateBook: {
      type: BookType,
      description: 'Azuriraj knjigu',
      args: {
        id: { type: GraphQLNonNull(GraphQLInt) },
        newName: { type: GraphQLNonNull(GraphQLString) }
      },
      resolve: (parent, args) => {
        books.forEach( (book, index) => {
          if (book.id === args.id) {
            books[index].name=args.newName
          }
          return books
      })
    }},
```

Kod 28 – updateBook mutacija

U kodu 28 prikazana je mutacija *updateBook* koja implementira izmjenu vrijednosti polja *name* odabrane knjige. Za identifikaciju resursa kojem želimo promijeniti ime potrebno je prilikom pozivanja mutacije navesti cjelobrojnu vrijednost. Ta vrijednost se u *resolveru* uspoređuje sa vrijednostima polja *id* svih knjiga kako bi locirali podatak. Sve što je ostalo napraviti je izmijeniti vrijednost polja *name* resursa sa tekstualnom vrijednosti koja je mutaciji proslijeđena kao argument zajedno sa *id-jem*.

Mutacija *updateAuthor* koja se koristi za izmjenu vrijednosti polja *name* autora prima iste argumente i implementirana je na isti način, jedina razlika je što radi promjene na *author* resursu umjesto na *book* resursu.

```
const [updateBook] = useMutation(UPDATE_BOOK);
return (
      <div>
      <form
        onSubmit={e => {
          e.preventDefault();
          updateBook({ variables: { nname: input.value, type:state.ID}});
          input.value = '';
        }}>
```

Kod 30 – korištenje updateBook mutacije

Kako bi korisniku omogućili izmjenu imena knjige ili autora, unutar klijentske aplikacije dodani su UI elementi *button* i *form*, kao i prošlim primjerima za pozivanje mutacija za dodavanje i uklanjanje autora. Korisnik u tekstualni okvir unosi novi naziv knjige, a klikom na tipku poziva se *updateBook()* metoda prikazana u kodu 30. Ta metoda koristi *updateBook* mutaciju kojoj prosljeđuje vrijednost koju je korisnik unio kao novi naziv i *id* knjige koji dohvaća iz stanja prikaza aplikacije.

```
const UPDATE_BOOK = gql`
  mutation updateBook($nname: String! $type: Int!){
    updateBook(newName: $nname id:$type)
    {id}
  }`;
```

Kod 29 – metoda updateBook()

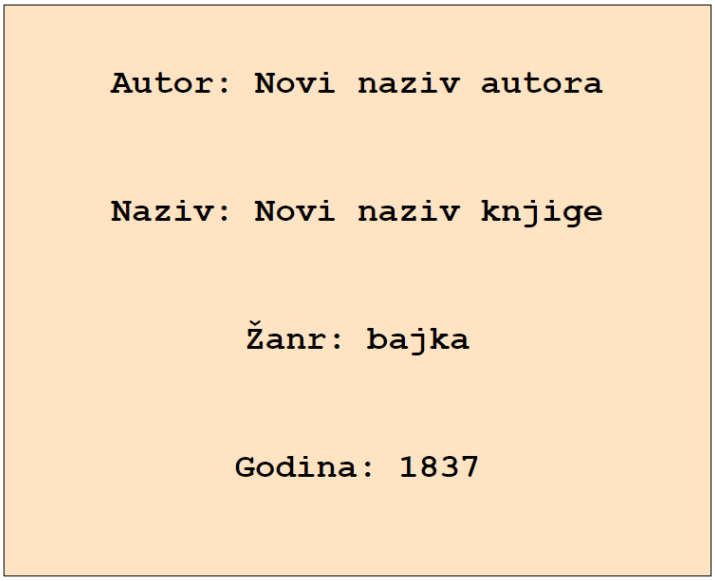
Kako bi izmijenili naziv knjige pozivamo mutaciju *updateBook* koja je prikazana u kodu 29. Korištenjem ove mutacije korisnik mijenja naziv odabrane knjige na način kako je definirano na GraphQL poslužitelju.

```
const UPDATE_AUTHOR = gql`
  mutation updateAuthor($nname: String! $type: Int!){
    updateAuthor(newName: $nname id:$type)
    {id}
  }`;
```

Kod 31 – korištenje updateAuthor mutacije

Promjena imena postojećeg autora implementirana je na isti način unutar klijentske aplikacije. Korisnik unosi novi naziv autora u tekstualni okvir, a *id* autora se dohvaća iz stanja prikaza aplikacije. Jedina razlika je što u slučaju autora aplikacija poziva *updateAuthor* mutaciju prikazanu u kodu 31. korištenjem te mutacije izvršavaju se promjene na GraphQL poslužitelju vezane za naziv odabranog autora.

Promjene naziva autora i knjige prikazane su na slici 28. Na ovom prikazu aplikacije može se vidjeti da su promijenjeni samo podaci koje želimo promijeniti. Ostatak resursa ostao je nepromijenjen.



Slika 27 – promjena vrijednosti polja name autora i knjige

S ovim je završena implementacija svih mutacija koje GraphQL poslužitelj u ovom primjeru omogućava. Korištenjem mutacija možemo na jednostavan način manipulirati podacima na poslužitelju bez potrebe za promjenama nad cijelim resursom. Također za razliku od RESTful API-ja GraphQL koristi samo jednu vrstu operacija za sve manipulacije podacima, što dodatno olakšava izmjene [25]

4.7 Korištenje GraphQL pretplata za dohvaćanje promjena

Klijentska aplikacija koja je prikazana u ovom radu korisniku omogućava uvid u podatke koji se nalaze u bazi podataka, te izmjene nad njima. Pri otvaranju novog prikaza aplikacija šalje GraphQL upit kako bi dohvatila podatke. Potencijalni problem u ovom slučaju nastaje kada korisnik napravi izmjene podataka koji se trenutno prikazuju. Iz razloga što upit dohvaća podatke samo pri pokretanju prikaza aplikacije, korisniku promjene koje je on ili neki drugi korisnik napravio neće biti vidljive sve dok ne osvježi prikaz. Kako bi GraphQL poslužitelj mogao omogućiti klijentskoj aplikaciji dohvaćanje promjena u stvarnom vremenu koristi treću vrstu operacija koji nazivamo pretplate [26].

GraphQL pretplate su na neki način slične upitima ili metodi GET RESTful API-ja na način da klijentskoj aplikaciji omogućava dohvaćanje podataka. Za razliku od tih operacija, pretplate su dugotrajne operacije koje mijenjaju rezultat tijekom vremena. Drugim riječima korištenjem pretplata klijentska aplikacija može primati nove podatke u stvarnom vremenu, bez potrebe za

osvježavanjem prikaza, odnosno slanjem novog upita svaki put kad očekujemo promjenu u bazi podataka. Pretplate su dakle korisne za obavještavanje klijenta u stvarnom vremenu o promjenama podatka, a u primjeru koji je prikazan u ovom radu između ostalog se mogu iskoristiti za slanje ažuriranja svaki put kada je dodan novi autor ili knjiga.

4.7.1 Implementacija pretplata na GraphQL poslužitelju

Kako bi ostvarili tu funkcionalnost na GraphQL poslužitelju je potrebno definirati pretplate te ih povezati sa promjenama o kojima želimo obavijestiti klijentsku aplikaciju. Za početak potrebno je u projekt dodati module koji omogućavaju implementaciju GraphQL pretplata kao što je prikazano u kodu 32. Modul *PubSub* GraphlQL poslužitelju omogućuje korištenje modela objave-pretplate (*Pub/Sub*) za praćenje događaja koji mijenjaju podatke. Instancu *SubscriptionServer* modula koristimo kako bi GraphQL poslužitelju omogućili da održava aktivnu vezu korištenjem tehnologija poput *WebSocketa*. *WebSocket* drugim riječima omogućuje otvaranje dvosmjerne interaktivne komunikacijske sesije između klijentske aplikacije i GraphQL poslužitelja [27].

```
const {subscribe} = require('graphql')
const {PubSub} = require('graphql-subscriptions')
const {SubscriptionServer} = require("subscriptions-transport-ws")
```

Kod 32 – dodatni moduli za implementaciju GraphQL pretplata

4.7.2 Dohvaćanje podataka o novim autorima i knjigama

Kako bi implementirali pretplate na GraphQL poslužitelju potrebno je definirati operacije koje je moguće koristiti, te ih povezati sa mutacijama o kojima želimo primati obavijesti. U kodu 33 je prikazana pretplata definirana unutar *RootSubscriptionType* tipa koji predstavlja sve pretplate GraphQL poslužitelja. Pretplata *AuthorAdded* omogućuje slanje podataka o novom resursu svaki put kad kada korisnik doda novog autora.

```
const RootSubscriptionType = new GraphQLObjectType({
  name: 'Subscription',
  description: 'Root Subscription',
  fields: () => ({
    AuthorAdded: {
      type: AuthorType,
      description: 'update autori',
      subscribe: () => pubsub.asyncIterator(["NEW_AUTHOR"])
    }})
```

Kod 33 – RootSubscritpionType

Sve što je ostalo je povezati mutaciju koja dodaje novog autora u bazu podataka i pretplatu koja šalje povratne informacije o tome. Kako bi to implementirali potrebno je unutar *resolvera* mutacije *addAuthor* dodati liniju koda kao što je prikazano u kodu 34. Ta naredba koristi metodu *publish()* modula *PubSub* kako bi proslijedila podatke pretplati *AuthorAdded* o autoru koji je upravo dodan. *AuthorAdded* pretplata koristi *asyncIterator* kako bi osluškivala događaje i dodala ih u red čekanja za obradu. Kod definiranja pretplata polje *subscribe* uvijek mora vratiti objekt *asyncIteratior* koji se sastoji od niza oznaka događaja od kojih treba očekivati promjene. U ovom slučaju to je mutacija *addAuthor* sa oznakom događaja *NEW_AUTHOR*.

```
pubsub.publish("NEW_AUTHOR", { AuthorAdded: author })
```

Kod 34 – objava novog autora

Kako bi omogućili istu funkcionalnost u slučaju dodavanja nove knjige potrebno je definirati pretplatu *BookAdded* unutar *RootSubscriptionType*-a kao što je prikazano u kodu 35. Ta pretplata prati događaj *NEW_BOOK*, odnosno očekuje pozivanje mutacije *addBook* koju koristimo za dodavanje nove knjige.

```
BookAdded: {
      type: BookType,
      description: 'update knjige',
      subscribe: () => pubsub.asyncIterator(["NEW_BOOK"])
    }
```

Kod 35 – BookAdded pretplata

Također potrebno je dodati liniju koda koja je prikazana u kodu 36 u *addBook* mutaciju koja objavljuje podatke nove knjige koristeći *publish()* metodu modula *PubSub*.

```
pubsub.publish("NEW_BOOK", { BookAdded: book })
```

Kod 36 – objava nove knjige

Ovim je završeno definiranje pretplata unutar *RootSubscriptionType* tipa. Taj tip operacija potrebno je dodati u shemu GraphQL poslužitelja kao što je prikazano u kodu 3. Zadnji korak implementacije pretplata na poslužitelju je implementacija *WebSocketa* korištenjem metode *create()* modula *SubscriptionServer* kao što je prikazano u kodu 37 [26].

```
SubscriptionServer.create(
  { schema, execute, subscribe },
  { server: httpServer, path: server.graphqlPath }
);
```

Kod 37 – implementacija WebSocketa

Ovim korakom na jednostavan način kreiramo jednostavnu konstantnu vezu između klijenta i poslužitelja koja se može koristi za dohvaćanje podataka u stvarnom vremenu. Za testiranje

pretplata prije implementacije na GraphQL klijentu možemo koristiti GraphQL IDE kao i za ostale vrste GraphQL operacija.

```
mutation AddAuthor($name: String!) {
  addAuthor(name: $name) {
    id
    name
  }
}
subscription AuthorAdded {
  AuthorAdded {
    id
    name
  }
}
```

Slika 28 - GraphQL IDE - testiranje pretplata

Pozivanjem GraphQL pretplate *AuthorAdded* u GraphQL IDE-u kao što je prikazano na slici 29 trenutno se neće prikazati nikakav odgovor sve dok se ne dogodi promjena u bazi podataka. Pozivanjem ove pretplate jedina promjena koju očekujemo je dodavanje novog autora, a to možemo napraviti korištenjem mutacije *addAuthor*. Tek nakon pozivanja te mutacije pretplata *AuthorAdded* će poslati odgovor kao što je prikazano na slici 30.

Slika 29 – odgovor pretplate AuthorAdded

Iz razloga što GraphQL pretplate koriste konstantnu vezu između poslužitelja i klijenta, pretplate *AuthorAdded* i *BookAdded* će slati odgovor svaki puta kada se dogodi očekivana promjena. Dakle za razliku od GraphQL upita pretplate je moguće koristi za konstantan pristup resursu u stvarnom vremenu kako bi korisniku prikazali najnoviju verziju podataka bezobzira na promjene koje su se dogodile [26].

U primjeru GraphQL poslužitelja koji je prikazan u ovom radu pretplate koristimo kako bi klijentsku aplikaciju obavijestili da je dodan novi autor ili knjiga, ali to im nije jednina primjena. GraphQL pretplate se mogu koristiti i za praćenje ostalih događaja, odnosno mutacija

kao što su uklanjanje resursa ili izmjena vrijednosti polja resursa koje su implementirane na poslužitelju.

4.7.3 Implementacija GraphQL pretplata unutar klijentske aplikacije

Iako GraphQL pretplate možemo koristiti umjesto upita to nije dobra praksa. Dohvaćanje velike količine podataka, te opterećivanje poslužitelja na ovakav način nije najbolja primjena GraphQL pretplata. Umjesto toga podatke je najbolje dohvatiti upitima jednom pri pokretanju prikaza aplikacije, a promjene nad podacima dohvatiti kada korisnik napravi neku promjenu. U primjeru koji je prikazan u ovom radu podaci se dohvaćaju koristeći GraphQL pretplate samo kada korisnik doda novog autora ili knjigu, što znači da se svi ostali autori i knjige ne dohvaćaju ponovno [26].

```
import { SubscriptionClient } from 'subscriptions-transport-ws'
import { WebSocketLink } from '@apollo/client/link/ws'
import { HttpLink } from '@apollo/client';
const httpLink = new HttpLink({
  uri: 'http://localhost:5000/graphql'
});
const wsLink = new WebSocketLink(
  new SubscriptionClient("ws://localhost:5000/graphql")
);
```

Kod 38 – Implementacija veze korištenjem WebSocketa

Za razliku od upita koji povremeno uspostavljaju vezu sa poslužiteljem, pretplate koriste konstantu vezu preko već spomenutog *WebSocketa*. Kako bi omogućili klijentskoj aplikaciji komunikaciju sa GraphQL poslužiteljem koristeći različite vrste veze za upite i pretplate implementiramo *httpLink* i *wsLink* kao što je prikazano u kodu 38. Upiti i mutacije ne zahtijevaju stalnu vezu sa poslužiteljem što HTTP čini učinkovitijim protokolom za korištenje takvih operacija. *SubscriptionClient* predstavlja modul *subscription-transport-ws* biblioteke koja nam omogućava implementacije veze između poslužitelja i klijenta korištenjem *WebSocketa* [28].

4.7.4 Korištenje GraphQL pretplata za ažuriranje popisa autora

Implementacija konstantne veze omogućava korištenje GraphQL pretplata unutar klijentske aplikacije. U aplikaciji pretplate koristimo kako bi osvježili podatke prikazane korisniku svaki put kada se doda novi autor ili knjiga. Pretplatu *AuthorAdded* prikazanu u kodu 39 koristimo kako bi dohvatili polja *id* i *name* autora dodanog mutacijom, iz razloga što promjene nisu odmah vidljive korisniku. Korištenjem samo GraphQL upita korisnik ne bi vidio promjene sve

dok ne osvježi prikaz aplikacije što nije poželjno, pogotovo ako na primjer izrađujemo aplikaciju za izravno slanje poruka [26].

```
const AUTHOR_SUBSCRIPTION = gql`
subscription Subscription {
  AuthorAdded {
    id
    name
  }
}`;
```

Kod 39 – korištenje AuthorAdded pretplate

Pretplate je moguće koristiti na dva načina unutar klijentske aplikacije. Prvi način je korištenjem *useSubscription() hook-a* kako bi dohvatili podatke slično korištenjem *useQuery() hook-a*. Već sam naveo da ovaj pristup nije idealan, te ako želimo pretplate koristiti kako bi samo ažurirali postojeće podatke koristimo *subscribeToMore()* funkciju.

U kodu 40 prikazana je upotpunjena verzija metode *DISPLAY_AUTHORS()* kojoj je dodana mogućnost ažuriranja podataka. U ovom slučaju ta metoda ima dva skupa podataka. Prvom možemo pristupiti koristeći data varijablu koja predstavlja podatke dohvaćene *authors* upitom. Drugi skup podataka se nalazi u varijabli *subscriptionData* i predstavlja podatke dohvaćene GraphQL pretplatom *AuthorAdded* unutar metode *SubscribeToMore()*.

```
const { loading, error, data, subscribeToMore } = useQuery(GET_AUTHORS);
  subscribeToMore({
    document: AUTHOR_SUBSCRIPTION,
    updateQuery: (prev, { subscriptionData }) => {
      if (!subscriptionData.data) return prev;
      const newauthor = {count: prev.authors.length+1,
                         __typename: "Author",
                         name: subscriptionData.data.AuthorAdded.name,
                         id: subscriptionData.data.AuthorAdded.id
                         };
      const exists = prev.authors.find(
         ({ id }) => id === newauthor.id
       );
       if (exists) return prev;
      return Object.assign({}, prev, {
         authors: [...prev.authors, newauthor],
      });
    }});
```

Kod 40 – proširenje DISPLAY_AUTHORS() metode

Svaki puta kada korisnik doda novog autora, odnosno pozove se mutacija *addAuthor*, *subscribeToMore()* dohvaća podatke dodanog autora i sprema ih u *subscriptionData*. Unutar

metode definiramo novi objekt koji predstavlja novog autora i provjeravamo postoji li već u podacima dohvaćenim GraphQL upitom. Ako ne postoji postojećim podacima dodajemo novog autora, koji se zatim prikazuju korisniku. Na ovaj način nema ponovnog dohvaćanja podataka, odnosno svaki autor je dohvaćen jednom. Iz razloga što pretplate koriste konstantnu vezu, metoda *subscribeToMore()* će se koristiti svaki put kad korisnik doda novog autora.

4.7.5 Korištenje GraphQL pretplata za ažuriranje popisa knjiga

Na sličan način je moguće implementirati ažuriranje popisa knjiga koji se korisniku prikazuje odabirom pojedinog autora. Svaki puta kada korisnik doda novog autora, odnosno koristi mutaciju *addBook* koristimo pretplatu *BookAdded* prikazanu u kodu 41. Ova pretplata je jako slična *AuthorAdded* pretplati, jedina razlika je što dohvaća vrijednosti polja *name* i *id* dodane knjige.

```
const BOOK_SUBSCRIPTION = gql`
subscription BookAdded{
  BookAdded {
    id
    name
  }
}`;
```

Kod 41 – Korištenje BookAdded pretplate

Jedino što je još potrebno napraviti je proširiti *DISPLAY_BOOKS()* metodu kako bi mogla dohvatiti ažurirane podatke i osvježiti prikaz aplikacije.

```
subscribeToMore({
    document: BOOK_SUBSCRIPTION,
    updateQuery: (prev, { subscriptionData }) => {
      if (!subscriptionData.data) return prev;
      const newbook = {
                        count: prev.author.books.length+1;,
                        __typename: "Books",
                        name: subscriptionData.data.BookAdded.name,
                        id: subscriptionData.data.BookAdded.id
                        };
      const exists = prev.author.books.find(({ id }) => id === newbook.id);
      if (exists) return prev;
      return Object.assign({}, prev, {
        author: { books: [...prev.author.books, newbook]}
      });
}});
```

Kod 42 – proširivanje DISPLAY_BOOKS() metode

Korištenjem metode *SubscribeToMore()* kako bi ažurirali podatke dohvaćene *author* upitom korisniku se trenutno prikazuju promjene nad podacima. Kao što je prikazano u kodu 42

SubscribeToMore() koristi *BOOK_SUBSCRIPTION* pretplatu kako bi dohvatila podatke dodane knjige, te ako je unos jedinstven dodaje novu knjigu u listu knjiga koja je povezana sa odabranim autorom.

Ovim je završena implementacija GraphQL pretplata unutar klijentske aplikacije. Kao što sam prikazao u ovom radu, korištenje pretplata kao i ostalih vrsta operacija koje podržava GraphQL, nudi mnoge prednosti pri izradi aplikacije koja koristi podatke sa poslužitelja. Pretplatama na jedinstven način dobivamo obavijesti o promjenama nad podacima kojima pristupamo, te na taj način u svakom trenutku imamo pristup najnovijim informacijama [28].

Korištenje GraphQL-a, odnosno operacija koje podržava omogućuje nam izradu moderne mrežne aplikacije bez potrebe za razumijevanjem tehnologija koje su mu prethodile ili protokola na kojima se temelji.

5 Zaključak

Korištenje GraphQL-a za izradu mrežne aplikacije ima mnoge prednosti u odnosu na slične protokole i arhitekture. Iz primjera koji je prikazan u ovom radu vidljiva je jednostavnost izrade GraphQL poslužitelja i klijenta, te svrhovitost korištenja upravo te tehnologije. Najveća prednost korištenja GraphQL-a je dohvaćanje upravo onih resursa koji su nam potrebni. Drugim riječima uklonjen je problem *over-fetchinga* i *under-fetchinga* koji je čest pri korištenju RESTful API-ja [12].

Dodatna prednost GraphQL-a je jednostavnost slanja upita. Sve što je potrebno kako bi dohvatili podatke je navesti koja polja resursa želimo dohvatiti. Također korištenje mutacija i pretplata je učinkovito i u usporedbi sa REST-om predstavljaju dodatnu funkcionalnost. Za razliku od PUT, POST i DELETE metoda REST-a, GraphQL mutacije sve te metode objedinjuju u jednu vrstu operacija što dodatno olakšava implementaciju.

Pretplate predstavljaju dodatnu funkcionalnost koja odgovara zahtjevima izrade modernih mrežnih aplikacija. Dohvaćanje podataka u stvarnom vremenu otvara nove mogućnosti, te nalazi primjenu u različitim vrstama mrežnih aplikacija poput društvenih mreža ili aplikacija za slanje izravnih poruka.

Bezobzira na prednosti koje GraphQL nudi u odnosu na REST, on ne predstavlja zamjenu toj arhitekturi, te odabir ovisi o zahtjevima aplikacije. REST je i dalje bolji odabir za jednostavne aplikacije koje ne koriste veliku količinu podataka. Također on je bolji odabir za početnike jer je jednostavniji za savladati, te je bolje podržan od zajednice, odnosno nudi više alata za razvoj.

GraphQL je relativno noviji što znači da nije jednako podržan kao REST, ali to također znači da podrška svakodnevno raste. Prednosti korištenja GraphQL-a prepoznale su i vodeće svjetske tvrtke poput Facebooka, Netflixa, Githuba i Paypala što čini razuvjeravanje GraphQL-a poželjnim od stane programera.

Smatram da u budućnosti možemo očekivati još veći interes za GraphQL od strane različitih tvrtki koje upravljaju velikom količinom podataka, što znači da možemo očekivati novije razvojne alate i više različitih primjena ove tehnologije.

6 Literatura

[1] What is Client Server Architecture, 2022, Pristup ostvaren 1.8.2022, https://intellipaat.com/blog/what-is-client-server-architecture/

[2] What is a Network?, 2022, Pristup ostvaren 1.8.2022, https://fcit.usf.edu/network/chap1/chap1.htm

[3] Types of Network, 2022, Pristup ostvaren 1.8.2022, https://www.educba.com/types-of-network/

[4] Service-Oriented Architecture, 2019, Pristup ostvaren 1.8.2022, https://www.ibm.com/cloud/learn/soa

[5] Application Program Interface, 2022, Pristup ostvaren 1.8.2022, https://www.shorturl.at/iRUY8

[6] Types of APIs and their differences?, 2022, Pristup ostvaren 1.8.2022, https://www.shorturl.at/iOP25

[7] XML Soap, 2022, Pristup ostvaren 5.8.2022, https://www.w3schools.com/xml/xml_soap.asp

[8] REST API Definition: Understanding the Basics of REST APIs, 2022, Pristup ostvaren 8.8.2022, https://www.astera.com/type/blog/rest-api-definition/

[9] Rest API Response Codes And Types Of Rest Requests, 2022, Pristup ostvaren 8.8.2022, https://www.softwaretestinghelp.com/rest-api-response-codes/

[10] A query language for your API, 2022, Pristup ostvaren 10.8.2022, https://graphql.org

[11] GraphQL vs. REST: What You Didn't Know, 2022, Pristup ostvaren 10.8.2022, https://www.shorturl.at/FKTW5

[12] To GraphQL or not to GraphQL? Pros and Cons, 2021, Pristup ostvaren 10.8.2022, https://slicknode.com/blog/graphql-or-not-graphql-pros-and-cons/

[13] Explore GraphQL, 2022, Pristup ostvaren 10.8.2022, https://www.graphql.com

[14] What is GraphQL?, 2019, Pristup ostvaren 10.8.2022, https://www.redhat.com/en/topics/api/what-is-graphql

[15] Most Popular APIs for Books, 2020, Pristup ostvaren 10.8.2022, https://www.programmableweb.com/news/10-most-popular-apis-books-2022/brief/2020/01/26

[16] Integrated Development Environment, 2021, Pristup ostvaren 11.8.2022, https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-integrated-development-environment

[17] GraphQL Projects and Tools You Have To Try, 2021, Pristup ostvaren 11.8.2022, https://hasura.io/blog/10-popular-open-source-graphql-projects-and-tools-you-have-to-try/

[18] Running an Express GraphQL Server, 2022, Pristup ostvaren 12.8.2022, https://graphql.org/graphql-js/running-an-express-graphql-server/

[19] Nenad Crnko, Uvod u GraphQL, 2018, Pristup ostvaren 13.8.2022, https://sistemac.srce.hr/sites/sistemac.srce.hr/files/docs/seminari/2018/srce_seminar_za_it-specijaliste_graphql_20180907.pdf

[20] Queries and Mutations, 2022, Pristup ostvaren 15.8.2022, https://graphql.org/learn/queries/

[21] Passing Arguments, 2022, Pristup ostvaren 15.8.2022, https://graphql.org/graphql-js/passing-arguments/

[22] Introduction to Apollo Client, 2022, Pristup ostvaren 17.8.2022, https://www.apollographql.com/docs/react/

[23] Apollo Queries, 2022, Pristup ostvaren 17.8.2022, https://www.apollographql.com/docs/react/data/queries

[24] Mutations in Apollo Client, 2022, Pristup ostvaren 17.8.2022, https://www.apollographql.com/docs/react/data/mutations

[25] GraphQL Mutations - Writing data, 2022, Pristup ostvaren 19.8.2022, https://hasura.io/learn/graphql/intro-graphql/graphql-mutations/

[26] Apollo Subscriptions, 2022, Pristup ostvaren 19.8.2022, https://www.apollographql.com/docs/react/data/subscriptions/

[27] Web socket and how it is different from the HTTP?, 2022, Pristup ostvaren 19.8.2022, https://www.geeksforgeeks.org/what-is-web-socket-and-how-it-is-different-from-the-http/

[28] Realtime Updates with GraphQL Subscriptions, 2022, Pristup ostvaren 19.8.2022, https://www.howtographql.com/react-apollo/8-subscriptions/

[29] REST APIs Exhaustion Signs, 2022, Pristup ostvaren 20.8.2022, https://www.programmersinc.com/over-fetching-and-under-fetching-rest-apis-exhaustion-signs/

[30] SOAP vs REST What's the Difference?, 2020, Pristup ostvaren 20.8.2022, https://smartbear.com/blog/soap-vs-rest-whats-the-difference/

[31] What is the Presentation Layer in Programming?, 2022, Pristup ostvaren 20.8.2022, https://www.coderscampus.com/what-is-the-presentation-layer-in-programming/

[32] Two-Tier Client/Server, 2013, Pristup ostvaren 20.8.2022, https://www.techopedia.com/definition/23846/two-tier-clientserver

7 Tablice

8 Slike

9 Kod