Strategije predmemoriranja u GraphQL-u

Novaković, Marin

SVEUČILIŠTE U SPLITU
PRIRODOSLOVNO MATEMATIČKI FAKULTET

ZAVRŠNI RAD

STRATEGIJE PREDMEMORIRANJA U GRAPHQL-U

Marin Novaković

Split, rujan 2023.

Temeljna dokumentacijska kartica

Sveučilište u Splitu Završni rad

Prirodoslovno-matematički fakultet

Odjel za informatiku

Ruđera Boškovića 33, 21000 Split, Hrvatska

STRATEGIJE PREDMEMORIRANJA U GRAPHQL-U

Marin Novaković

SAŽETAK

Rad se bavi strategijama predmemoriranja podataka u GraphQL programskim infrastrukturama. Teorijski dio rada obuhvaća pregled osnova HTTP predmemoriranja, koncepte rada GraphQL-a, opis izazova kod predmemoriranja odgovora u GraphQL-u te objašnjenja tehnika koje ga omogućuju, skupa s drugim tehnikama predmemoriranja podataka. U praktičnom dijelu rada se kroz razvijenu web aplikaciju demonstriraju spomenute tehnike predmemoriranja, korištenjem programskih biblioteka poput *Apollo Server*-a, *Apollo Client*-a i *DataLoader*-a. Također se bilježi učinak tih tehnika na poslužiteljski i korisnički dio aplikacije. U zaključku rada se ističu implikacije njegovih ishoda na buduća istraživanja te se daju preporuke za njihovo proširenje.

Ključne riječi: GraphQL, predmemoriranje, HTTP, Apollo Server, DataLoader

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

Rad sadrži: 49 stranica, 29 grafičkih prikaza, 0 tablica i 14 literaturnih 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*

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

Mirna Marić, *asistent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Rad prihvaćen: 18. rujna 2023.

Basic documentation card

University of Split

Undergraduate Thesis

Faculty of Science

Department of Informatics

Ruđera Boškovića 33, 21000 Split, Croatia

GRAPHQL CACHING STRATEGIES

Marin Novaković

ABSTRACT

The topic of this thesis deals with data caching strategies within GraphQL software infrastructures. The theoretical part of the thesis includes an overview of HTTP caching basics, working concepts behind GraphQL, a description of challenges with response caching in GraphQL as well as explanations for techniques that enable it, including other data caching techniques. In the practical part of the thesis, these techniques are demonstrated through a developed web application, using software libraries such as *Apollo Server*, *Apollo Client* and *DataLoader*. The impact of the mentioned techniques on the server and user sides of the application is also noted. The conclusion of the thesis highlights possible implications of its outcomes on future research topics and provides recommendations for their expansion.

Keywords: GraphQL, caching, HTTP, Apollo Server, DataLoader

Thesis deposited in the library of Faculty of Science, University of Split

Thesis consists of: 49 pages, 29 figures, 0 tables and 14 references, original in: Croatian

Mentor: Goran Zaharija, doc. dr. sc. *Assistant Professor of Faculty of Science, University of Split*

Reviewers: Goran Zaharija, doc. dr. sc. *Assistant Professor of Faculty of Science, University of Split*

Monika Mladenović, doc. dr. sc. *Assistant Professor of Faculty of Science, University of Split*

Mirna Marić, *Teaching Assistant of Faculty of Science, University of Split*

Thesis accepted: September 18th, 2023

IZJAVA

kojom izjavljujem s punom materijalnom i moralnom odgovornošću da sam završni rad s naslovom *Strategije predmemoriranja u GraphQL-u* izradio samostalno pod voditeljstvom doc.dr.sc. Gorana Zaharije. U radu sam primijenio metodologiju znanstvenoistraživačkog rada i koristio literaturu koja je navedena na kraju završnog rada. Tuđe spoznaje, stavove, zaključke, teorije i zakonitosti koje sam izravno ili parafrazirajući naveo u završnom radu na uobičajen, standardan način citirao sam i povezao s fusnotama s korištenim bibliografskim jedinicama. Rad je pisan u duhu hrvatskog jezika.

Student
Marin Novaković

Sadržaj

UVOD

U kontekstu računarstva, predmemoriranje se odnosi na postupak privremene pohrane podataka u priručnoj memoriji kako bi se mogli učinkovitije isporučiti u budućnosti. Priručna memorija je općeniti naziv za bilo koju vrstu privremenog spremnika podataka koji pruža brži pristup podacima u odnosu na uobičajeni slijed operacija potrebnih za njihov pronalazak i obradu. Kvalitetno implementirani mehanizmi predmemoriranja mogu značajno poboljšati performanse računalnih sustava i rasteretiti računalne resurse.

U dobu konstantno rastućih zahtjeva za što efikasnijom i efektivnijom razmjenom podataka putem Interneta, korištenje web arhitektura baziranih na konceptu krajnjih točaka sve više pokazuje svoje nedostatke. Postojeći web API-ji ne uspijevaju udovoljiti širok spektar različitih načina uporabe podataka koje poslužitelji nude, što dovodi do kompromisa u njihovom dizajnu.

GraphQL tehnologija pruža alternativan pristup razmjeni informacija između klijenta i poslužitelja, omogućujući klijentskim aplikacijama da preciziraju točno one podatke koje trebaju, izbjegavajući pri tome suvišan mrežni promet. Međutim, inicijative protiv korištenja GraphQL-a u razvoju web aplikacija često se argumentiraju pogrešnim pretpostavkama o njegovim sposobnostima predmemoriranja podatka, za koje se vjeruje da su ograničene u odnosu na arhitekture poput REST-a.

Teorijski dio rada pruža pregled osnovnih značajki HTTP predmemoriranja i GraphQL-a te izazova koji se mogu pojaviti njihovom uzajamnom primjenom u izradi mrežnih aplikacija. Navođenjem potencijalnih strategija za prevladavanje tih izazova nastoje se opovrgnuti negativne konotacije vezane za predmemoriranje u GraphQL-u.

Aplikacija razvijena u sklopu projekta demonstrira kako implementacija tih strategija utječe na cjelokupan tijek izvršavanja mrežnih zahtjeva, a pri tome se bilježe razne optimizacije na poslužiteljskoj i korisničkoj strani interakcije.

1. OSNOVE HTTP PREDMEMORIRANJA

Tehnike predmemoriranja prostiru se kroz gotovo sve aspekte računarstva, od podatkovnih i instrukcijskih predmemorija procesora, priručnih međuspremničkih memorija podatkovnih sustava OS-a, do ARP i DNS predmemorija usmjernika [1]. Web, kao Internet usluga za pregledavanje hipertekstulanih dokumenata, također nudi standardiziranu podršku predmemoriranja odgovora poslanih s poslužitelja, a pri tome se oslanja na postojeću specifikaciju HTTP-a [2].

HTTP predmemorija automatski pohranjuje odgovore na poslane zahtjeve te ih isporučuje pri budućim zahtjevima, izbjegavajući komunikaciju s izvornim poslužiteljem. Na taj način smanjuje se suvišan prijenos podataka i zagušenje mreže, kako na korisničkoj, tako na poslužiteljskoj strani.

1.1. Pogodci, promašaji i svježina odgovora

Kada klijent pošalje zahtjev za pregled odabranog dokumenta, vrši se provjera je li odgovor na taj zahtjev pohranjen u priručnoj memoriji. Ukoliko jest i ukoliko je ispravan, on se izravno prosljeđuje klijentu, bez da se vrši komunikacija s izvornim poslužiteljem. Ta pojava naziva se pogotkom (engl. *cache hit*) [1], a omjer odgovora koji je isporučen klijentu na taj način zove se stopom pogodaka (engl. *hit rate*). U protivnom se radi o promašaju (engl. *cache miss*). Promašaj se javlja kada je zahtjev za danim resursom poslan po prvi put, kada odgovor nije moguće predmemorirati ili kada je odgovor postao nevažeći ili se uklonio da se oslobodi prostor.

U idealnim okolnostima, zahtjev za nekim resursom samo bi se jednom morao slati poslužitelju, dok bi svi naredni zahtjevi bili udovoljeni pogotkom u priručnoj memoriji kao posljedica predmemoriranja prvotnog odgovora. To naravno nije izvedivo zato što se podaci na poslužitelju s vremenom mogu mijenjati, što znači da podaci u priručnoj memoriju postaju beskorisni. Da se to izbjegne, predmemoriranim odgovorima dodjeljuje se svojevrstan vijek trajanja koji određuje koliko dugo se pohranjeni odgovori smiju isporučivati klijentu nakon što se inicijalno pohrane u priručnoj memoriji. Temeljem toga, uvodi se pojam svježine odgovora [1].

Odgovor je svjež ako pripadajući vijek trajanja u trenutku zahtjeva još nije istekao, a ako jest, odgovor se smatra ustajalim i zahtjev se prosljeđuje poslužitelju. Naime, svježina odgovora ne mora nužno odražavati aktualno stanje podataka na poslužitelju jer se radi isključivo o pretpostavci koliko učestalo se oni mijenjaju. Poslužitelji u sklopu poslanog odgovora ponekad mogu izostaviti eksplicitnu naznaku svježine, zbog čega određivanje iste može predstavljati izazov. U takvim se slučajevima potrebno oslanjati na heurističko računanje svježine pomoću ostalih informacija u odgovoru. Doduše, specifikacija HTTP-a ne pruža standardne algoritme za njihovu primjenu, ali daje primjere dobre prakse [2].

1.2. Privatne i dijeljene predmemorije

Specifikacija HTTP-a opisuje dvije osnovne vrste priručne memorije: privatne i dijeljene [3]. Privatne predmemorije namijenjene su za korištenje od strane pojedinačnih korisnika te su najčešće implementirane na razini web preglednika. Glavni im je zadatak pohrana odgovora koji su prilagođeni odabranom korisniku i kojima ni jedan drugi korisnik ne smije imati pristup. S obzirom na to, za njihov rad nije potrebna dodatna sklopovska podrška, a kapacitetom su zanemarive veličine. S druge strane, dijeljene predmemorije pohranjuju odgovore kojima više korisnika ima pravo pristupa, a obično su realizirane u obliku odvojenog sklopovlja koje posreduje između poslužitelja i klijenata. Tako se pospješuje ponovna iskoristivost odgovora i stopa pogodaka jer je vjerojatnost ponovnog pristupa predmemoriranom dokumentu veća ako je i broj aktivnih korisnika veći.

1.3. *Cache-Control* zaglavlje

Protokol HTTP nudi nekoliko načina za upravljanje i prilagodbu ponašanja predmemorije na razini odgovora. Najkorišteniji takav mehanizam svakako je postavljanje *Cache-Control* zaglavlja unutar poslanog odgovora [4]. *Cache-Control* zaglavlje sastoji se od jedne ili više direktiva ili instrukcija (engl. *cache directives*) za kontrolu priručne memorije preglednika i dijeljenih predmemorija. Navode se jedna za drugom, odvojene zarezom, a neke mogu primati i opcionalne argumente.

max-age / s-maxage

Uvođenje direktive *max-age* jedan je od preporučenih postupaka eksplicitne deklaracije svježine odgovora. Predstavlja maksimalno proteklo vrijeme od trenutka slanja odgovora s poslužitelja, tijekom kojeg se on smatra svježim i poslužuje iz priručne memorije.

Postavljanje njezine vrijednosti na 0 označava da se odgovor ne smije dohvaćati izravno iz predmemorije. Ako se odgovor želi posluživati iz dijeljene predmemorije, potrebno je umjesto *max-age* koristiti *s-maxage* direktivu.

private / public

Direktiva *private* označava da je isporučeni odgovor namijenjen isključivo jednom korisniku i da ga je dozvoljeno pohraniti samo u privatnoj predmemoriji. U slučajevima kada se želi omogućiti pohrana odgovora u dijeljenim priručnim memorijama, ona se zamjenjuje s direktivom *public*. Važno je napomenuti da direktiva *public* također dozvoljava predmemoriranje odgovora kojima to inače ne bi bilo moguće, poput odgovora čiji zahtjevi imaju autorizacijsko zaglavlje.

must-revalidate

Odgovore koji više nisu svježi nije uvijek poželjno odmah ukloniti iz priručne memorije. Direktiva *must-revalidate* nalaže da se odgovor smije dohvaćati iz priručne memorije dok god je svjež, a kada postane ustajao potrebno je potvrditi njegovu točnost s poslužiteljem prije ponovne upotrebe. Koristi se u kombinaciji s direktivom *max-age*.

no-cache

Unatoč imenu, direktiva *no-cache* ne zabranjuje predmemoriranje odgovora. Odgovorima koji sadrže ovu direktivu je omogućena pohrana u priručnu memoriju, ali pod uvjetom da se za svaki poslani zahtjev izvrši provjera ispravnosti podataka s poslužiteljem. To se može poistovjetiti s korištenjem kombinacije direktiva *must-revalidate* i *max-age* s vrijednošću 0.

no-store

Za situacije u kojima uopće nema smisla pohranjivati odgovore u priručnu memoriju, bilo zbog prevelike učestalosti izmjene povezanih podataka na poslužitelju ili strogih sigurnosnih zahtjeva, rabi se direktiva *no-store*. Njezina prisutnost u *Cache-Control* zaglavlju izričito zabranjuje bilo kakav oblik predmemoriranja odgovora, neovisno o drugim direktivama koje su možda navedene u zaglavlju.

1.4. Validacija

Kao što je već spomenuto, direktiva *must-revalidate* dozvoljava ponovnu upotrebu ustajalih odgovora iz predmemorije nakon provjere njihove ispravnosti s poslužiteljem. Slijed operacija potrebnih da se provjeri aktualnost predmemoriranih podataka zove se validacija,

odnosno revalidacija, a najčešće se postiže slanjem uvjetnih zahtjeva (engl. *conditional requests*) [3]. Princip korištenja uvjetnih zahtjeva je relativno intuitivan:

1) Kada priručna memorija zaprimi zahtjev za dokumentom koji je u njoj pohranjen, ali nije svjež, poslužitelju se šalje zahtjev za istim dokumentom, s dodatnim zaglavljima koji služe za validaciju.

2) Pri primitku zahtjeva, poslužitelj izlučuje podatke iz validacijskih zaglavlja i vrši usporedbu s povezanim zaglavljima iz posljednjeg poslanog odgovora koji je evidentiran za taj resurs.

3) Ako se zaglavlja podudaraju, priručnoj memoriji se šalje odgovor sa statusnim kodom 304, koji označava da se podaci na poslužitelju nisu promijenili u odnosu na one iz priručne memorije. Tijelo odgovora je prazno jer nema potrebe za slanjem novih podataka, a informacije iz zaglavlja se koriste za obnovu svježine predmemoriranog odgovora. Ako nema poklapanja, poslužitelj na uobičajen način dohvaća novi dokument i šalje ga u tijelu odgovora sa statusnim kodom 200, što znači da odgovor sadrži nove podatke. Pri dospijeću odgovora, priručna memorija zamjenjuje ustajali dokument s novim i koristi nova zaglavlja za računanje njegove svježine.

Specifikacija HTTP-a podržava dvije tehnike validacije putem uvjetnih zahtjeva: vremenske oznake dobivene iz *Last-Modified* zaglavlja i entitetske oznake iz *ETag* zaglavlja [1].

1.4.1. *Last-Modified* i *If-Modified-Since* zaglavlja

Slanjem odgovora, poslužitelj može definirati *Last-Modified* zaglavlje s vremenskom oznakom za koju pretpostavlja da je odabrani dokument posljednji put izmijenjen. Za validaciju ustajalog odgovora preko vremenske oznake, potrebno ju je poslati unutar *If-Modified-Since* zaglavlja u uvjetnom zahtjevu. Ako je ta vremenska oznaka jednaka onoj koja je posljednja zabilježena za traženi resurs na poslužitelju, odgovor iz priručne memorije se smatra važećim. Korištenje vremenskih oznaka za validaciju može u određenim situacijama uzrokovati neočekivane ishode [1]. Na primjer, ovisno o postavkama poslužitelja, vremenske oznake se pri premještanju dokumenata s jedne lokacije na drugu mogu promijeniti, bez utjecaja na sami sadržaj. Neusklađenost vremena na povezanim poslužiteljima također može poremetiti validaciju.

1.4.2. *ETag* i *If-None-Match* zaglalvlja

Entitetske oznake drugi su tip validacijskog mehanizma, a zapisuju se unutar *ETag* zaglavlja. Riječ je o *stringovima* koji služe za jedinstvenu identifikaciju specifične instance dokumenta. Poslužitelj u pravilu za svaku modifikaciju nekog dokumenta stvara novu entitetsku oznaku i isporučuje ju u sklopu odgovora. Ukoliko se pomoću nje želi provesti validacija, šalje se unutar *If-None-Match* zaglavlja, slično kao vremenska oznaka. Ako predmemorija posjeduje više ustajalih odgovora za isti dokument, sve pripadajuće entitetske oznake se koriste pri sljedećoj validaciji, tako što se zajedno navedu u *If-None-Match* zaglavlju. Specifikacija, doduše, ne sprječava zajedničko korištenje vremenskih i entitetskih oznaka za validaciju istog odgovora, no važno je znati da *If-None-Match* zaglavlje ima prioritet nad *If-Modified-Since* zaglavljem, koje će se u tada koristi jedino ako nema podudaranja ni s jednom navedenom entitetskom oznakom.

2. UVOD U GRAPHQL

Osnovni cilj svakog web poslužitelja, neovisno o korištenoj arhitekturi, je jednak: pravovremeno isporučiti tražene podatke svim korisnicima s pravom pristupa. Očekivano, klijenti imaju različite potrebe što se tiče informacija koje žele konzumirati, a zadatak poslužitelja je da što bolje zadovolji sve moguće slučajeve korištenja. To predstavlja poseban izazov poslužiteljima koji primjenjuju API-je bazirane na krajnjim točkama, gdje je svaki resurs jednoznačno opisan jedinstvenim identifikatorom, tj. URI-jem.

Kako svakom klijentu nisu uvijek potrebni svi podaci dobiveni iz zaprimljenog resursa, nemaju izbor nego preuzeti cjeloviti odgovor, a suvišne podatke odbaciti. Premda je moguće definirati veći broj krajnjih točaka za potencijalne varijacije u korisničkim zahtjevima, dobiveni API-ji su u pravilu manje održivi i slabije optimizirani.

S namjerom da nadiđu te poteškoće, Facebook je 2012. razvio GraphQL specifikaciju, a 2015. ju učinio dostupnom za javnost [5]. GraphQL kombinira upitni jezik po ugledu na SQL i okruženje za izvršavanje tih upita na poslužitelju. Oslanja se na sustav tipova za reprezentaciju podataka, što klijentima pruža slobodu da zahtijevaju isključivo one podatke koji njima trebaju. GraphQL nije vezan za specifičan programski jezik, kao ni način skladištenja i prijenosa podataka, što ga čini veoma fleksibilnim i lako upotrebljivim u sklopu brojnih vrsta web aplikacija.

2.1. GraphQL shema

U središtu svakog GraphQL API-ja nalazi se shema koja objedinjuje definicije svih tipova [5]. Prije izvršavanja korisničkog zahtjeva, on se prvo mora interpretirati preko sheme da bi se provjerila njegova ispravnost. Stoga se za upitni jezik GraphQL-a kaže da snažno tipiziran (engl. *strongly-typed*) [6, 7], što podrazumijeva manju učestalost sintaktičkih pogrešaka u upitima, a klijentima daje mogućnost integracije alata koji automatski popunjavaju zahtjeve i vrše njihovu validaciju prije slanja.

2.1.1. Objektni tipovi

Tipovi se u kontekstu GraphQL sheme mogu opisati kao entiteti koji određuju strukturu i značajke podataka unutar API-ja. Najosnovniji takvi tipovi su objektni tipovi (engl. *object*

types) [5, 8], a predstavljaju vrste objekata koje klijenti mogu dohvaćati i polja koja oni sadržavaju. Samim poljima također je pridružen vlastiti tip za vrstu podatka, a mogu biti jedan od ugrađenih skalarnih tipova poput cijelog broja i *stringa* ili neki drugi objektni tip. Za polja koja ne smiju biti prazna, tj. vraćati vrijednost *null*, koristi se oznaka uskličnika (!) uz dodijeljeni tip.

```
type Student{
    JMBAG: ID!
    ime: String!
    prezime: String!
    dob: Int
    kolegiji: [Kolegij]!
}
```

Kod 2.1: Primjer objektnog tipa u GraphQL shemi

Na isječku koda (Kod 2.1) prikazan je objektni tip *Student* s poljima *JMBAG* (skalarni tip ID neprazne vrijednosti), *ime* i *prezime* (neprazni *stringovi*), *dob* (neprazni cijeli broj) i *kolegiji* (obavezni niz s elementima objektnog tipa *Kolegij* koji mogu imati vrijednost *null*).

2.1.2. Operacijski tipovi

GraphQL razlikuje tri vrste operacija za rad s podacima: upite, mutacije i pretplate [6]. Upiti služe isključivo za dohvaćanje podataka s poslužitelja, u obliku objekata definiranih objektnim tipovima. Mutacije se koriste za mijenjanje podataka, bilo stvaranjem novih ili ažuriranjem i brisanjem postojećih podataka, s opcijom slanja povratnog objekta, slično upitu. Pretplate stvaraju postojanu vezu s poslužiteljem, u namjeri da detektiraju promjene nad podacima u stvarnom vremenu i isporuče tražene informacije klijentu. Za korištenje operacija potrebno je u GraphQL shemi definirati operacijske tipove (engl. *root operation types*) [5, 8]. Oni su ulazna točka u GraphQL API jer određuju skup svih dozvoljenih upita, mutacija i pretplata. U shemi se deklariraju slično objektnim tipovima, pri čemu svako polje opisuje jednu operaciju, argumente koje ona prihvaća te povratni tip.

```
type Query{
    student(JMBAG: ID!): Student!
}

type Mutation{
    novi_student(podaci: StudentPodaci!): Student
}
```

Kod 2.2: Primjeri operacijskih tipova u GraphQL shemi

U primjeru (Kod 2.2) deklarirani su operacijski tipovi za upite i mutacije. Tip za upite *Query* sadrži upit naziva *student* koji prima obavezni argument *JMBAG* skalarnog tipa ID te vraća neprazni objekt tipa *Student*. Mutacijski tip *Mutation* sadrži mutaciju *novi_student* koja prima podatke ulaznog tipa *StudentPodaci*, a opcionalno vraća objekt tipa *Student*.

2.2. Izvršavanje zahtjeva

Unatoč neovisnosti GraphQL specifikacije o načinu razmjene informacija između klijenta i poslužitelja, GraphQL API-ji najčešće se poslužuju preko HTTP-a [9]. Specifičnost GraphQL poslužitelja leži u tome što se svi zahtjevi obrađuju putem jedne krajnje točke. Preporučeni način slanja zahtjeva je u obliku *stringova*, kodiranih u JSON formatu unutar tijela HTTP POST metode. Neovisno o operaciji i uspješnosti njezina izvršavanja, podaci se šalju u tijelu odgovora, također u JSON formatu. *String* GraphQL operacije se sastoji od deklaracije njezinog tipa, naziva specifične operacije, argumenata i skupa željenih polja povratnog objekta, tj. selekcijskog seta (engl. *selection set*). Ovisno o potrebama korisnika, selekcijski set može sadržavati sva ili dio polja definiranih za povezani objektni tip.

```
query {
  student(JMBAG: "ltm6210ll2fsg62") {
    ime
    prezime
    kolegiji {
      naziv
      }
    }
  }
```

Kod 2.3: Primjer upita u GraphQL-u

Prikazani zapis (Kod 2.3) poziva upit pod nazivom *student* te mu prosljeđuje vrijednost argumenta *JMBAG*. U odgovoru se očekuje ime, prezime i popis kolegija studenta, dok su njegova dob i JMBAG izostavljeni. S obzirom na to da polje kolegij vraća objekt tipa

Kolegij, unutar njega se također može precizirati skup polja, u ovome slučaju samo naziv kolegija. Ako nije došlo do greške prilikom izvršavanja zahtjeva, oblik odgovora se poklapa s poljima iz operacijskog *stringa*, gdje su elementi sačinjeni od ključ-vrijednost parova (Kod 2.4).

```
{
  "data": {
    "student": {
      "ime": "Ivan",
      "prezime": "Horvat ",
      "kolegiji": [
        {
          "naziv": "Uvod u umjetnu inteligenciju"
        },
        {
          "naziv": "Baze podataka"
        }
      ]
    }
  }
}
```

Kod 2.4: Primjer strukture odgovora u GraphQL-u

2.2.1. *Resolver* funkcije

GraphQL shema sama po sebi ne opisuje način kako se pristupa i upravlja s podacima na poslužitelju. Za to su zadužene *resolver* funkcije (engl. *resolver functions*) [5]. Glavni im je zadatak implementirati mehanizme za dohvaćanje i ažuriranje podataka, neovisno radi li se to putem baze podataka, odvojenog API-ja ili neke druge usluge, te ih zatim prilagoditi da budu u skladu s tipovima iz GraphQL sheme. Sustav tipova automatski povezuje polja povratnog objekta s vrijednostima istoimenih svojstava iz dohvaćenih podataka. Ako se nazivi polja ne podudaraju ili se dobiveni podaci moraju dodatno prilagoditi, potrebno je koristiti posebne *resolver* funkcije zvane trivijalni *resolveri*. Svako polje objekta može imati vlastiti trivijalni *resolver* za razrješenje pripadajuće vrijednosti, što ponekad zahtijeva naknadni pristup usluzi koja isporučuje podatke.

2.3. GraphQL u odnosu na REST

REST je vodeća arhitektura za dizajniranje web API-ja koji rade po principu krajnjih točaka. Za rad s resursima REST se najčešće oslanja na skup metoda definiranih u HTTP

specifikaciji, a to su metode HTTP GET, POST, PUT i DELETE. Usprkos širokoj adaptaciji, REST API-ji pokazuju bitne nedostatke u situacijama kada moraju posluživati korisničke aplikacije različitih podatkovnih zahtjeva [6, 7]. GraphQL svojim značajkama nastoji ciljano eliminirati te nedostatke.

2.3.1. Deklarativno dohvaćanje podataka

Svaki resurs unutar REST API-ja pruža pristup fiksnoj kolekciji podataka, definiranom od strane poslužitelja. To može biti nepraktično korisnicima kojima nisu potrebne sve informacije u sklopu resursa jer su slanjem zahtjeva prisiljeni preuzeti suvišne podatke [6]. Jedan od načina na koji se može dozvoliti filtracija podataka unutar resursa jest parametrizacijom URI-jeva, no mogućnosti toga pristupa su veoma ograničene [8].

GraphQL pruža potpunu slobodu u odabiru podataka putem upitnog jezika, što ima pozitivne implikacije ne samo za veličinu dobivenog odgovora, već i za brzinu njegovog primitka na klijentskoj strani. Pojedini resursi u REST-u ponekad ne pružaju sve potrebne informacije korisniku, zbog čega je neophodno slati uzastopne zahtjeve za različitim resursima [6]. Dobro dizajnirana GraphQL shema uvelike smanjuje broj takvih zahtjeva korištenjem ugniježđenih upita nad poljima koja vraćaju objekt s podacima vezanim za roditeljski objekt.

2.3.2. Upravljanje krajnjim točkama

Kako se zahtjevi web aplikacija konstantno mijenjaju, API-ji koji ih podržavaju moraju skupa s njima evoluirati kako bi im nastavili pružati relevantne informacije. U slučaju REST arhitektura, to često podrazumijeva uspostavljanje novih krajnjih točaka prilagođenih za specifične potrebe pojedinačnih aplikacija. Porast u broju takvih krajnjih točaka znatno smanjuje preglednost API-ja i dugoročno ga čini jako teškim za održavanje [6]. U GraphQL-u, arhitektura se temelji na jednoj krajnjoj točki kao polazištu za cijeli podatkovni model poslužitelja. Zahvaljujući tome, klijenti ne trebaju voditi računa o tome na kojoj lokaciji se pojedini resursi nalaze, već se u mogu oslanjati na značajke upitnog jezika i definicije tipova unutar sheme.

2.3.3. Introspekcija

Dokumentiranje funkcionalnosti je neophodan korak u dizajniranju bilo kojeg web API-ja jer korisnicima pruža uvid u moguće načine upravljanja podacima na poslužitelju. REST nudi široku podršku za alatima koji pospješuju izradu dokumentacije, ali njihova primjena zahtijeva opsežno razumijevanje i ručno testiranje API-ja.

Jedno od najmoćnijih svojstava sustava tipova u GraphQL-u je automatska introspekcija sheme [5, 6, 8]. Introspekcija omogućuje definiranje upita s informacijama o GraphQL shemi poput detalja o definiranim operacijskim i objektnim tipovima. Svaka shema dolazi sa skupom ugrađenih introspekcijskih polja koja automatski dohvaćaju podatke vezane za sustav tipova. To bitno olakšava razumijevanje osnovne funkcionalnosti API-ja i služi kao temelj za integraciju brojnih alata za generiranje dokumentacije i testiranje. Na primjer, ugrađeno razvojno okruženje GraphiQL koristi introspekciju za validaciju operacija prije izvršavanja, isticanje sintakse, automatsko popunjavanje koda i prikazivanje referenci API-ja u web pregledniku.

3. PREDMEMORIRANJE U GRAPHQL-u

Iako su prednosti korištenja GraphQL-a neosporive, infrastrukture koje ga koriste također mogu pogodovati od mehanizama predmemoriranja za dodatno smanjenje suvišnog podataka prometa između poslužitelja i klijenata. Međutim, manjak standardizacije u implementaciji tih mehanizama stvorio je pogrešnu predodžbu o tome da je predmemoriranje u GraphQL-u neizvedivo, posebice u kontekstu HTTP predmemoriranja [8].

Dodatnu raspravu oko optimizacije i isplativosti izaziva granularnost GraphQL upita, pri čemu se u pitanje stavlja ponovna iskoristivost predmemoriranih odgovora i kompleksnost njihove validacije. Ipak, rastući interes doveo je do razvoja brojnih programskih solucija koje na različite načine facilitiraju predmemoriranje u GraphQL-u, ne samo na razini odgovora, nego i u fazama obrade upita i pristupa podacima.

3.1. Izazovi predmemoriranja odgovora

Većina nedoumica oko predmemoriranja odgovora u GraphQL-u može se pripisati načinu na koji GraphQL iskorištava HTTP za prijenos podataka. Pri tome su usporedbe s REST-om dosta česte budući da je REST dizajniran da usko prati standarde i preporuke HTTP specifikacije. Počevši od toga kako klijenti upravljaju s podacima na poslužitelju, jasno je da GraphQL ne može predmemorirati odgovore na razini HTTP-a, barem ne bez uvođenja dodatnih modifikacija u njegovo razvojno okruženje.

Jedinstvena krajnja točka za pristup GraphQL API-ju sama po sebi ne pruža dovoljno informacija za jednoznačnu identifikaciju resursa, kao što bi to bio slučaj s URI-jevima u REST-u. Osim toga, HTTP POST, kao preporučena metoda za slanje GraphQL operacija poslužitelju, nije zamišljena da podržava predmememoriranje jer bi se po specifikaciji trebala koristiti prvenstveno za stvaranje novih resursa [2]. HTTP GET je jedina sigurna metoda s podrškom za predmemorianjem odgovora, ali njezina primjena u GraphQL-u nije optimalna jer zanemaruje tijelo zahtjeva u kojemu se inače navodi *string* GraphQL operacije. Moguće rješenje toga bilo bi proslijediti operaciju kao parametar u sklopu URI-ja, što bi također riješilo problem identifikacije resursa. Naravno, to bi jedino bilo prihvatljivo za upite jer služe isključivo za dohvat podataka, dok bi se mutacije i pretplate nastavile prenositi u tijelu HTTP POST metode.

Ključan izazov u svemu tome ponovno leži u identifikaciji podataka u priručnoj memoriji jer će se upiti s različitim redoslijedom navođenja polja tretirati kao različiti resursi, te neće dijeliti predmemorirane odgovore. Isto će vrijediti i za upite u kojima nisu navedena sva polja u odnosu na odgovor iz predmemorije.

Drugu poteškoću izaziva reprezentacija podataka u GraphQL-u, gdje kompleksne relacije između entiteta, kroz ugniježđena objektna polja, značajno otežavaju određivanje svježine predmemoriranog odgovora. U idealnim okolnostima, svakom polju bi se dodijelila vlastita oznaka svježine, što bi potencijalno povećalo ponovnu iskoristivost odgovora ukoliko bi se u narednim upitima tražila polja čija svježina još nije istekla. Nažalost, HTTP ne razumije semantiku GraphQL upita pa je svježinu potrebno deklarirati na razini cijelog odgovora. Ipak, da bi vjerojatnost korištenja zastarjelih podataka bila što manja, svježina odgovora bi se onda trebala određivati prema polju čija se vrijednost najčešće mijenja. Zbog istih ograničenja, djelomična validacija upita nije provediva, već se odgovor mora promatrati kao cjelina pri provjeri njegove valjanosti.

Trenutna specifikacija za posluživanje i konzumaciju GraphQL API-ja putem HTTP-a ne daje izričite smjernice za upravljanje HTTP predmemorijom [10], zbog čega se implementacija tih mehanizama razlikuje između programskih rješenja. Pored toga, većina GraphQL klijenata i razvojnih okvira nudi vlastite priručne memorije, odvojene od predmemorije preglednika, koje normaliziraju dohvaćene podatke radi lakšeg referenciranja i ažuriranja. Ažuriranje predmemoriranih odgovora u skladu s drugih provedenim operacijama može smanjiti potrebu za njihovom validacijom te osigurati konzistentnost, bez dodatne komunikacije s poslužiteljem.

3.2. Postojani upiti

Složenost GraphQL upita iziskuje potrebu za provjerom njihove ispravnosti nakon što pristignu na stranu izvornog poslužitelja. Kod normalnog tijeka izvršavanja upita, pripadni *string* se prvo leksički analizira, tijekom čega se razdvaja na posebne leksičke jedinice, odnosno tokene, koji moraju pratiti GraphQL sintaksu [5]. Temeljem tih tokena, upit se parsira, što rezultira generiranjem apstraktnog sintaksnog stabla (engl. *abstract syntax tree*) u kojem svaki čvor odgovara jednom elementu upita, primjerice polju, dok veze odražavaju njihovu međusobnu kompoziciju. Apstraktno sintaksno stablo se zatim koristi za validaciju upita u odnosu na GraphQL shemu, a to osigurava da se samo upiti koji prate sustav tipova

sheme u konačnici i izvrše. Očito je da ovaj slijed operacija pridonosi sveukupnom vremenu izvršavanja upita, pa tako i vremenu potrebnom za isporuku odgovora, ali je kao korak nužan jer ništa ne jamči da će poslani zahtjev biti u skladu s opisanom GraphQL shemom.

U produkcijskom okruženju, većina klijentskih aplikacija definira upite koji se po strukturi ne mijenjaju između zahtjeva, što znači da poslužitelj bespotrebno ulaže vrijeme i računalne resurse kako bi iznova provjerio njihovu ispravnost. Veliki upiti su posebno problematični jer njihova validacija može dovesti do primjetnih kašnjenja kod primitka odgovora i do degradacije u performansama klijentskih aplikacija. Da se u takvim situacijama poveća učinkovitost mrežnog prometa, neki GraphQL poslužitelji nude podršku za postojanim upitima (engl. *persisted queries*), gdje se prethodno verificirani upiti predmemoriraju na poslužitelju skupa s jedinstvenim identifikatorom za indeksiranje.

Kod postojanih upita, korisnici umjesto operacijskog *stringa* prosljeđuju njegov identifikator preko kojeg poslužitelj određuje koji upit se treba izvršiti. Time se smanjuje veličina poslanog zahtjeva te se optimizira izvršavanje upita jer više nije nužno provjeravati je li strukturno i sintaktički korektan. Identifikatore postojanih upita preporučeno je generirati deterministički, najčešće šifriranjem samog operacijskog *stringa*, jer to omogućuje klijentima da iz njega sami odrede pod kojim bi se identifikatorom upit indeksirao. Dakako, ovo je provedivo samo ako je poslužitelj prije toga pohranio postojani upit u priručnoj memoriji. Općenita komunikacije između klijenta i poslužitelja kod korištenje postojanih upita je sljedeća [8]:

1) Klijent započinje slanjem zahtjeva koji sadrži identifikator željenog upita, dobivenog šifriranjem njegovog operacijskog *stringa*.
2) Kada poslužitelj primi zahtjev, provjerava postoji li u priručnoj memoriji upit s navedenim identifikatorom. Ako ne postoji, korisnika se u odgovoru o tome obavijesti.
3) Korisnik zatim šalje naknadni zahtjev poslužitelju, u kojem uz identifikator upita navodi i njegov operacijski *string*.
4) Ovaj put, poslužitelj na uobičajen način izvršava upit, vršeći validaciju operacijskog *stringa*, i šalje dobivene podatke klijentu. Nakon toga, registrira postojani upit u priručnoj memoriji pod dobivenim identifikatorom.
5) Za naredne zahtjeve, korisniku je omogućeno slanje identifikatora postojanog upita kojeg će poslužitelj sada prepoznati i izvršiti bez ponovne provjere ispravnosti.

Vidljivi nedostatak ove tehnike je potreba za slanjem dodatnog zahtjeva ako poslužitelj prethodno nije imao priliku registrirati postojani upit s poslanim identifikator. Promatrajući širu sliku, to je donekle zanemariv trošak za klijenta jer je time osigurao da on i potencijalno drugi klijenti u budućnosti mogu izbjeći slanje cjelovitog upita. Uz to, postojani upiti su posebno efektivni kada se prenose putem HTTP GET metode jer pružaju konzistentnu tehniku identifikacije resursa za svrhe HTTP predmemoriranja.

3.3. N+1 problem

Dinamičko generiranje podatkovnih reprezentacija, pospješeno korištenjem *resolver* funkcija, je jedno od najbitnijih obilježja svakog GraphQL sustava. *Resolveri* su obično dizajnirani da se izvršavaju asinkrono [5] jer razrješavanje vrijednosti pripadnih polja često zahtijeva dohvaćanje dodatnih podataka iz vanjskih izvora, što može uzrokovati kašnjenja u njihovoj isporuci. Asinkronost također dozvoljava paralelno izvršavanje *resolvera* za različita polja unutar objekta, čime je obrada samih upita primjetno brža. Uz pogodnosti koje koncept *resolver* funkcija sa sobom donosi, jedan nedostatak kod načina pribavljanja podataka za potrebe razrješavanja vrijednosti polja se posebno ističe.

```
query {
    kolegiji_po_studentu(JMBAG: "ltm6210ll2fsg62") {
      naziv
      nositelj_kolegija {
        ime
        prezime
      }
    }
  }
```

Kod 3.1: Primjer selekcijskog seta s ugniježđenim objektnim tipom

U sljedećem upitu (Kod 3.1). Selekcijski set sastoji se od naziva kolegija i objektnog polja *nositelj_kolegija* s poljima za ime i prezime nositelja kolegija. Pod pretpostavkom da se podaci za razrješenje polja *nositelj_kolegija* nisu unaprijed dohvatili prilikom izvršavanja *resolver* funkcije upita *kolegiji_po_studentu*, potrebno ih je naknadno dohvatiti temeljem podataka iz roditeljskog objekta, primjerice identifikatora kolegija unutar baze podataka. *Resolver* za polje *nositelj_kolegija* tada će trebati stvoriti novu vezu s bazom podataka i izvršiti SQL upit za pronalazak podataka o nositelju kolegija.

To je problematično iz više razloga. Kao prvo, definirana *resolver* funkcija je ograničena na kontekst svojeg roditeljskog objekta, u ovom slučaju na individualni kolegij iz popisa, što bi

značilo da će se za svaki takav kolegij izvršiti po jedna *resolver* funkcija polja *nositelj_kolegija* te će se za svaku od njih stvoriti po jedna dodatna konekcija s bazom podataka. Drugo, GraphQL nema ugrađeni podršku za pamćenje vrijednosti dosada razriješenih polja, zbog čega će se za kolegije s istim nositeljem *resolveri* neovisno jedno o drugom izvršiti te će preko odvojenih veza s bazom podataka dohvatiti identične podatke.

To se kolektivno naziva N+1 problemom [8] jer se uz jednu konekciju za dohvat podataka o popisu kolegija stvara još N konekcija za dohvat podataka o nositelju kolegija, po jedna za svaki kolegij iz popisa. N+1 problem je naročito izražen kod sustava koji podatke dobivaju preko DBaaS-ova ili drugih komercijalnih API-ja, gdje se troškovi njihova korištenja često temelje na broju pristupa usluzi. Problem je još ozbiljniji ako se upiti nastave gnijezditi, što kod ovako definiranih *resolvera* može dovesti do umnažanja broja potrebnih poziva ka vanjskoj usluzi.

3.3.1. Grupiranje ugniježđenih upita

Kod uobičajenog tijeka izvršavanja GraphQL upita, *resolveri* polja se momentalno pozivaju prema redoslijedu definiranom od strane GraphQL izvršitelja. Za polja s asinkronim *resolverima* to znači da će izvršitelj čekati sve dok njihove vrijednosti ne postanu dostupne te će tek onda nastaviti s izvršavanjem *resolvera* ugniježđenih polja [8, 11]. Naravno, *resolveri* nepovezanih polja na istoj razini, uključujući asinkrone *resolvere*, će se dalje nesmetano izvršavati, paralelno jedno s drugim.

Da se izbjegne N+1 problem, trebalo bi biti moguće zakazati uzajamno razrješenje vrijednosti odabranog polja u svim objektima na istoj razini, tako da se svi potrebni podaci dohvate u sklopu jedne grupacije te adekvatno raspodijele između pripadajućih objekata. Taj se postupak naziva grupnim izvršavanjem ugniježđenih upita (engl. *query batching*) [8], a zamišljen je kao svojevrstan oblik lijenog učitavanja podataka (engl. *lazy loading*) jer se dohvat podataka odgađa sve do trenutka kada *resolver* za neko polje nije pozvan za sve objekte unutar kolekcije. Grupno izvršavanje ugniježđenih upita se provodi u dvije faze:

1) U prvoj fazi, za svaki objekt iz kolekcije poziva se *loader* funkcija, čija je uloga učitati skup akumuliranih ključeva, odnosno argumenata koji bi se inače proslijedili *resolveru* za razrješenje vrijednosti polja, u jedan niz. Pri učitavanju tih ključeva, u mjestu vrijednosti polja vraća se objekt koji će nakon isporuke podataka poprimiti pripadajuću vrijednosti za svako polje. *Loader* funkcija preuzima ulogu tipične

resolver funkcije, a u slučaju upita *kolegiji_po_studentu*, za polje *nositelj_kolegija* u niz ključeva se učitavaju jedinstveni identifikator svakog od kolegija.

2) U drugoj fazi, nakon što je *loader* funkcija učitala sve ključeve u niz, oni se prosljeđuju *batch* funkciji, koja će u sklopu jedne optimizirane procedure pristupiti vanjskoj usluzi i dohvatiti potrebne podatke u skladu s navedenim ključevima. Za upit *kolegiji_po_studentu*, to može biti poseban SQL upit koji će u bazi podataka pronaći sve profesore koji su nositelji kolegija s identifikatorom među učitanim ključevima. Niz dohvaćenih podataka će se temeljem ključeva mapirati s odgovarajućim instancama polja te im tako dodijeliti vrijednosti.

Iz opisanog je jasno da efektivnost grupnog izvršavanja ugniježđenih upita u svrhu izbjegavanja N+1 problema ovisi ponajviše o dozvoljenim metodama za pristup podacima preko usluge koja ih pruža. Na primjeru relacijskih baza podataka, grupno izvršavanje je itekako isplativo jer SQL dozvoljava pisanje podupita koji bi prema nizu učitanih ključeva mogli unutar jednog upita isporučiti više redaka iz baze podataka odjednom. Obratno, kod API-ja bez podrške za grupiranjem zahtjeva učinci grupnog izvršavanja bili bi zanemarivi jer bi se podaci svakako dohvaćali putem odvojenih poziva API-ju.

3.3.2. Predmemoriranje po zahtjevu

Uz spomenutu mogućnost istovremenog razrješenja vrijednosti ugniježđenog polja za sve objekte na istoj razini, većina programskih solucija za grupno izvršavanje ugniježđenih upita koristi neki oblik memoizacijske predmemorije [12] za privremenu pohranu parova učitanih ključeva i objekata koji će poprimiti neku vrijednost.

U situacijama kada se u kontekstu jednog korisničkog zahtjeva ponavlja polje sa specifičnim ključem, umjesto da se ključ ponovno učita u niz, asocirano polje će se razriješiti pomoću vrijednosti koja se nakon grupnog izvršavanja dodijelila objektu u priručnoj memoriju. Ta tehnika deduplikacije ključeva osigurava da se tijekom pristupa podacima identični ključevi obrađuju samo jednom unutar iste *batch* funkcije. Predmemoriani parovi se također mogu koristiti u drugim dijelovima zahtjeva, na primjer za dublje ugniježđena polja koja koriste istu *loader* funkciju za svoje *resolvere*.

Ovo funkcionira kao vrsta predmemorije na razini zahtjeva jer se sprječava dohvaćanje istih podataka neovisno o tome gdje su unutar zahtjeva potrebni. Preporuka je da se za svaki naredni zahtjev stvori nova instanca mehanizma za grupno izvršavanje zato što postoji

vjerojatnost da će se podaci između zahtjeva promijenit pa vrijednosti pohranjene u priručnoj memoriji više neće biti aktualne. Opcija dijeljenja jedne takve instance između zahtjeva je svakako dostupna, no tada je izrazito važno voditi računa o pravilnom ažuriranju vrijednosti u predmemoriji s obzirom na provedene promjene nad podacima. Nadalje, nužno je ograničiti kapacitet predmemorije da između zahtjeva ne bi mogla neograničeno rasti te zauzeti veliku količinu radne memorije poslužitelja.

Memoizacijska predmemorija se nikako ne bi smjela koristiti kao zamjena za priručne memorije na razini aplikacije, ali može biti koristan dodatak poslužitelju za optimizaciju faze dohvaćanja podataka i formacije odgovora na korisničke zahtjeve.

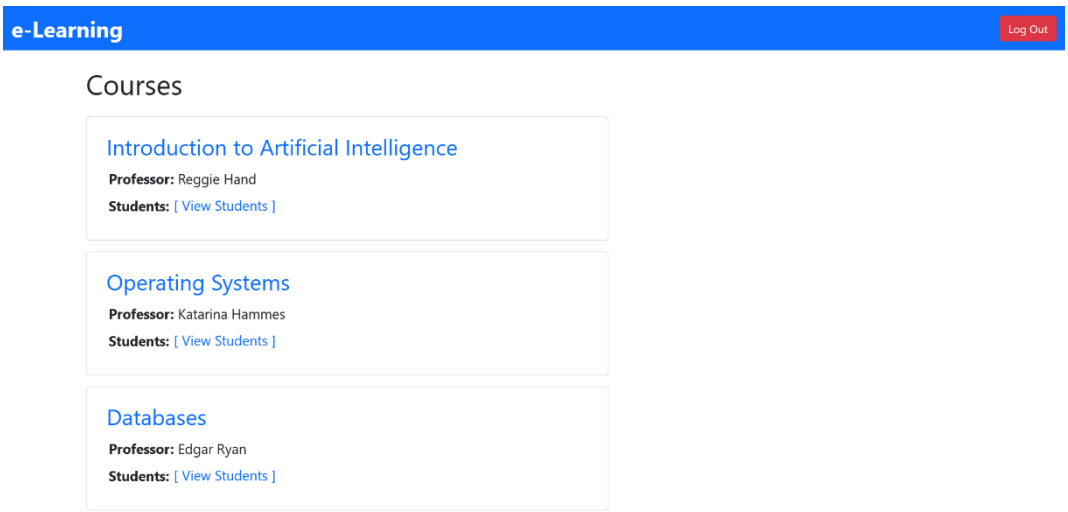
4. PROJEKTNI ZADATAK

Cilj završnog projekta je demonstrirati spomenute strategije predmemoriranja u GraphQL-u, primjenom podržanih tehnologija unutar konkretnog i ostvarivog slučaja korištenja. S tom namjerom, razvijena je web aplikacija u kojoj korisnički i poslužiteljski dio integriraju GraphQL u svoje radno okruženje. Komunikacija između korisnika i poslužitelja odvija se putem GraphQL API-ja, a uvođenjem tehnika predmemoriranja javljaju se optimizacije u međusobnoj razmjeni podataka.

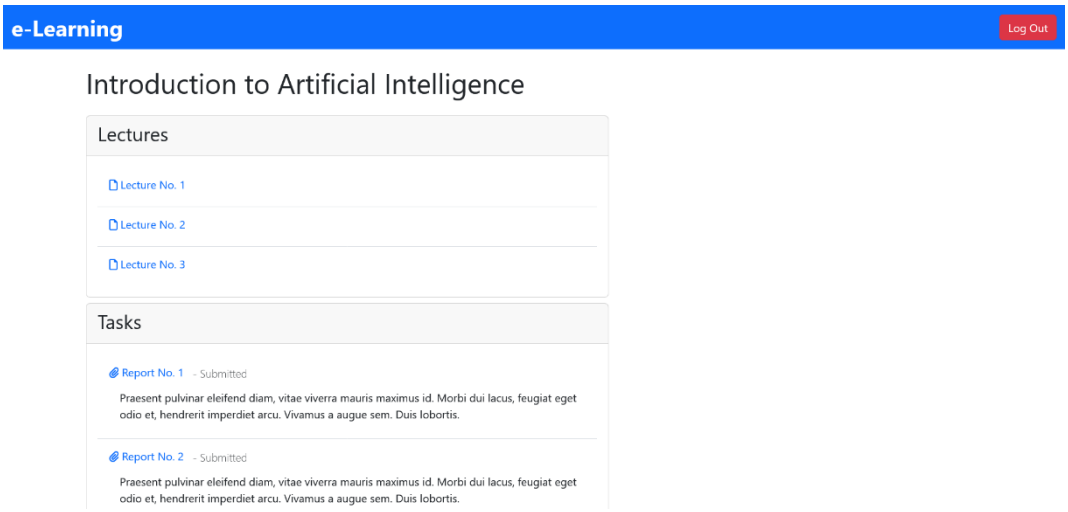
4.1. Pregled funkcionalnosti i korištenih tehnologija

Razvijena web aplikacija ima ulogu pojednostavljene platforme za e-učenje, pri čemu je članovima odabrane obrazovne ustanove pristup aplikaciji omogućen sa sustavom za prijavu. Sustav provodi autorizaciju korisnika, prema kojoj razlikuje uloge studenata i profesora.

Unutar aplikacije, studentima je ponuđen pregled svih upisanih kolegija s informacijama o nositelju kolegija i uvidom u popis drugih upisanih studenata (Slika 4.1). Na stranici odabranog kolegija nalazi se lista poveznica za preuzimanje materijala s predavanja te lista zadataka koje trebaju predati (Slika 4.2). Predajom riješenog zadatka preko ponuđenog sučelja, studenti mogu motriti njegov status ocjenjivanja od strane profesora.

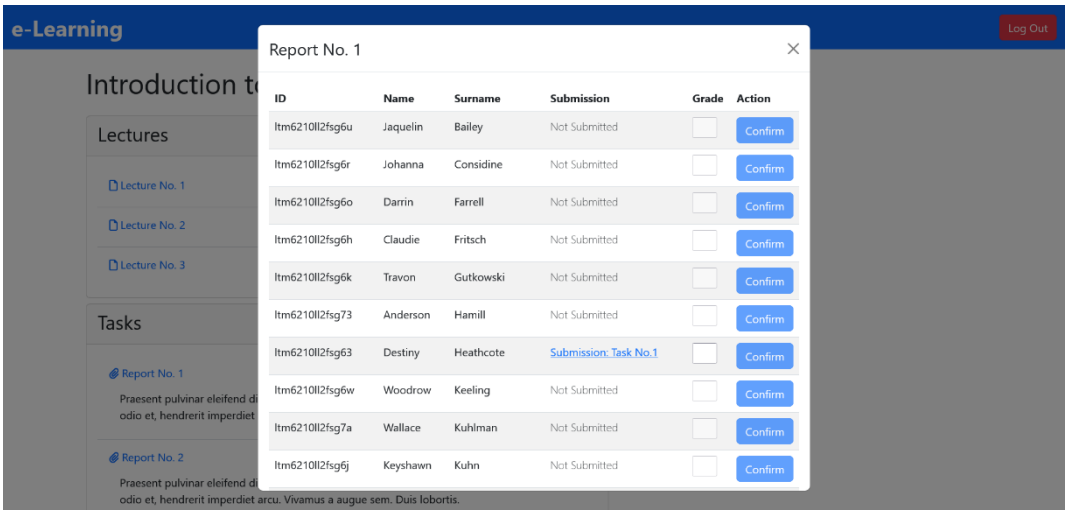


Slika 4.1: Pregled upisanih kolegija unutar aplikacije



Slika 4.2: Pregled popisa materijala i zadataka s kolegija

Slično, korisnici s ulogom profesora imaju početni prikaz svih kolegija za koje su nositelji. Svakom od tih kolegija imaju pravo dodavati nove ili uklanjati postojeće poveznice s materijalima i zadatke. Odabir jednog od zadataka iz popisa izaziva prikaz statusa njegove predaje za svakog od upisanih studenata (Slika 4.3). Ako je neki student predao zadatak, profesor tu predaju može preuzeti i ocijeniti.



Slika 4.3: Sučelje za pregled i ocjenjivanje predanih zadataka

Iz opisane funkcionalnosti je evidentno da korisnička interakcija nije ograničena samo na pregled sadržaja unutar aplikacije, već također nudi više metoda za njegovu modifikaciju. Ta spoznaja je ključna jer ističe potrebu za kontinuiranim upravljanjem podacima u priručnoj memoriji, neovisno radi li se to na strani klijenta ili poslužitelja.

Poslužiteljski dio se sastoji od *NodeJS* aplikacije na kojoj je pokrenut *Express* server. *Express* razvojni okvir se inače koristi kao samostalni alat za postavljanje HTTP poslužitelja,

no u aplikaciji služi kao „omotač“ za GraphQL API zbog podrške za *middleware* funkcijama, poput CORS-a za prihvaćanje korisničkih zahtjeva s vanjskih izvora. Sami GraphQL API je konfiguriran unutar *Apollo Server* poslužitelja, što čini realizaciju priručne memorije poslužitelja i HTTP predmemoriranje pomoću ugrađenih dodataka veoma jednostavnim. Svi podaci s kojima se raspolaže unutar aplikacije su pohranjeni u lokalnoj *SQLite* bazi podataka, a pristup im je posredovan *Knex* alatom za povezivanje na bazu podataka i pisanje SQL upita. Za potrebe testiranja, u terminalu razvojnog okruženja se nakon svake konekcije s bazom podataka ispisuje izvršena SQL naredba. *Dataloader* paket je odabran za potrebe grupnog izvršavanja ugniježđenih upita, a poslužitelj je postavljen tako da se pri pokretanju može birati između *resolvera* koji ga koriste i onih koji individualno dohvaćaju podatke iz baze podataka za svaki pozvani *resolver*.

Korisnički dio je zamišljen kao jednostranična aplikacija, gdje se ovisno o korisnikovim akcijama mijenja njezin sadržaj stranice.U tu svrhu, korištena je *ReactJS* biblioteka jer dozvoljava podjelu korisničkog sučelja na individualne komponente, čijim se ponašanjem i sadržajem upravlja preko *hook* funkcija. *React Router* biblioteka olakšava istovremeno mijenjanje prikaza više takvih komponenti odjednom, pružajući vlastite komponente za definiranje ruta između kojih korisnici mogu prelaziti. Za slanje korisničkih zahtjeva u obliku GraphQL operacija kombiniraju se *GraphQL.js* biblioteka i *Apollo Client* klijentsko okruženje za *React*. Poseban naglasak je stavljen na *Apollo Client* biblioteku jer uključuje normaliziranu predmemoriju za odgovore te opciju slanja postojanih upita.

4.2. Apollo Server

Apollo Server je vodeća poslužiteljska biblioteka otvorenog koda namijenjena za rad s GraphQL aplikacijama [13]. Pruža unificirano razvojno okruženje za uspostavljanje i testiranje GraphQL API-ja te alate za jednostavno povezivanje sheme s deklariranim *resolver* funkcijama. Od posebnog je interesa za opisani projekt jer dolazi s vlastitom priručnom memorijom općenite namjene koja automatski pohranjuje i isporučuje različite podatke pri primitku korisničkih zahtjeva.

Apollo Server podržava predmemoriranje odgovora na udaljenim CDN-ovima, u spremnicima kao što su *Redis* i *Memcached,* ali je za potrebe projekta korištena dijeljena predmemorija na samom poslužitelju. Pokretanjem poslužitelja stvara se instanca *InMemoryLRUCache* objekta čija je uloga predmemorirati odgovore i postojane upite

prateći LRU strategiju za upravljanje alociranom memorijom. Naime, kada kapacitet priručne memorije dosegne maksimalnu popunjenost, dodavanjem novih vrijednosti uklanjaju se zapisi nad kojima se pogodak nije zabilježio najduži period vremena.

Za predmemoriranje odgovora u *InMemoryLRUCache* memoriji, na raspolaganju je *apollo/server-plugin-response-cache* biblioteka, čija se *responseCachePlugin()* funkcija može navesti u listi dodataka. Da bi se dozvolilo HTTP predmemoriranje u pregledniku i dijeljenim predmemorijama, *Apollo Server* dolazi s pojednostavljenom značajkom računanja *Cache-Control* zaglavlja s ugrađenim *CacheControl* dodatkom, koji je prema zadanim postavkama aktivan pri pokretanju poslužitelja.

```
const apolloServer = new ApolloServer({
    typeDefs,
    resolvers: useDataLoader ? resolversDataLoader : resolversRegular,
    cache: new InMemoryLRUCache(),
    plugins: [
      ApolloServerPluginCacheControl(
        {
          calculateHttpHeaders: true
        }
      ),
      responseCachePlugin({
        sessionId: ({ request }) => request.http.headers.get('authorization') || null
      })
    ]
  })
```

Kod 4.1: Konfiguracija GraphQL poslužitelja

U sljedećem isječku koda (Kod 4.1) prikazan je konstruktor *Apollo Server* poslužitelja. Kao što je vidljivo, konstruktor prima jedan parametar u obliku objekta sa svim potrebnim postavkama za njegovu inicijalizaciju. Prvo navedeno svojstvo jest *typeDefs* svojstvo, zaslužno za prosljeđivanje GraphQL sheme u *string* formatu. Ono se povezuje s *resolvers* svojstvom, koje sadrži mapu svih deklariranih *resolver* funkcija.

Kada se aplikacija pokrene, moguć je izbor između *resolvera* s *DataLoader* integracijom i onih bez, temeljem čega se u *resolvers* svojstvo učitavaju različite mape funkcija. *cache* svojstvo opisuje tip priručne memorije na poslužitelju, a radi demonstracije je stvorena instanca zadane *InMemoryLRUCache* klase. U *plugins* svojstvu obuhvaćeni su svi dodaci predviđeni za instalaciju na poslužitelju, a to su već spomenuti *CacheControl* i *responseCachePlugin*. Osobitost *responseCachePlugin* dodatka je sposobnost podjele priručne memorije na korisničke sesije, tako da pristup određenim predmemoriranim

odgovorima bude dozvoljen samo pojedinim korisnicima. Zbog jednostavnosti, korisničke sesije se u slučaju razvijene aplikacije međusobno raspoznaju prema tokenima iz autorizacijskog zaglavlja zahtjeva jer su specifični za svakog prijavljenog korisnika.

4.2.1. *@cacheControl* direktiva

U *Apollo Server* okruženju ponašanje predmemoriranih odgovora se može regulirati prilagodbom dviju oznaka [13]. Prva je maksimalni vijek trajanja odgovora koja, po analogiji *max-age* direktive iz *Cache-Control* zaglavlja, naznačuje koliko dugo se odgovor smatra svježim te se smije posluživati direktno iz priručne memorije. Druga oznaka je doseg odgovora, na osnovu čije se vrijednosti obuhvaćeni odgovori u predmemoriji dijele na javne i privatne, slično principu rada *public* i *private* direktiva. Dodjela ovih oznaka može se provesti na dva načina. Prvi je dinamički, a uključuje postavljanje *cacheControl* objekta u *info* parametru *resolver* funkcije (Kod 4.2).

```
Query: {
        student: (_, { JMBAG }, _, info) => {
          info.cacheControl.setCacheHint({ maxAge: 60, scope: 'PRIVATE' });
          return find(student, { JMBAG });
        }
      }
```

Kod 4.2: Primjer dinamičke prilagodbe ponašanja predmemoriranog odgovora u *resolveru*

Drugi pristup, koji je zbog praktičnih razloga izabran za potrebe projekta, je statički i ostvaruje se u samoj GraphQL shemi, definiranjem *@cacheCoctrol* direktive. Direktive se mogu opisati kao proširenja GraphQL sheme jer utječu na normalan tijek izvršavanja GraphQL operacija. *Apollo Server* automatski prepoznaje *@cacheControl* direktivu, a u shemi se definira kako slijedi (Kod 4.3):

```
//schema.graphql
enum CacheControlScope { PUBLIC, PRIVATE }
directive @cacheControl(
  maxAge: Int
  scope: CacheControlScope
  inheritMaxAge: Boolean
) on FIELD_DEFINITION | OBJECT | INTERFACE | UNION
```

Kod 4.3: Definicija *@cacheControl* direktive

Uz oznake *maxAge* i *scope* za postavljanje maksimalnog vijeka trajanja i dosega odgovora, oznaka *inheritMaxAge* je naročito važna za pravilno računanje svježine odgovora ovisno o poljima koja su navedena u selekcijskom setu upita. Oznake je dopušteno postavljati na razini svakog ne-skalarnog polja objekta, ali i na razini definicije cijelog objektnog tipa. U slučajevima kada se želi osigurati da objektno polje preuzme vrijednost *maxAge* oznake roditeljskog objekta, uz njega se navodi *inheritMaxAge* oznaka s vrijednošću true. To je ključno jer zadana vrijednost *maxAge* oznake svakog objektnog polja iznosi 0, a kako se maksimalni vijek svježine odgovora računa prema najmanjoj takvoj vrijednosti iz selekcijskog seta, odgovor se tada uopće neće pohraniti u priručnoj memoriji.

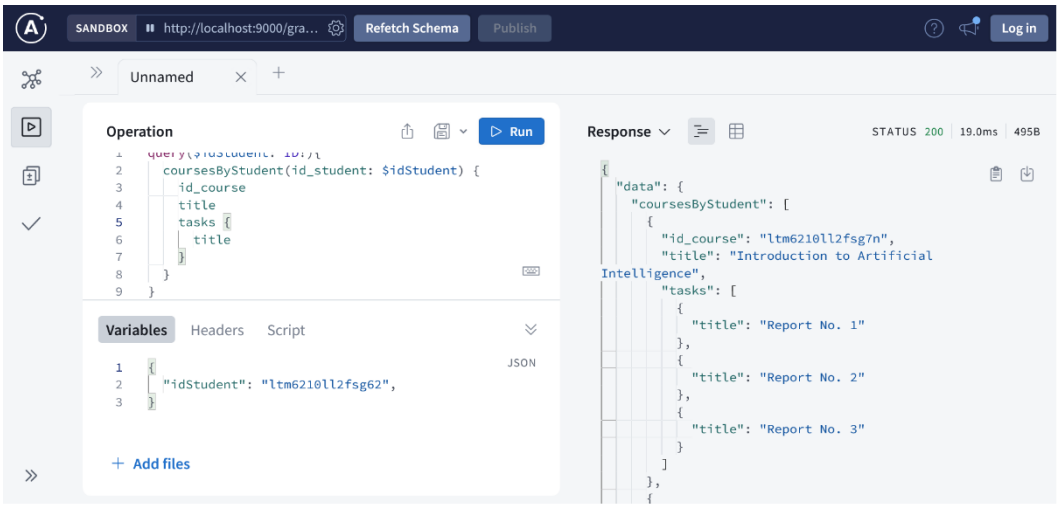
```
//schema.graphql
type Query{
    //…
    coursesByStudent(id_student: ID!): [Course]! @cacheControl(maxAge: 360, scope:
PRIVATE)
}

type Course @cacheControl(inheritMaxAge: true){
    id_course: ID!
    title: String!
    professor: Professor!
    students: [Student]! @cacheControl(inheritMaxAge: true)
    tasks: [Task]! @cacheControl(maxAge: 60)
    lectures: [Lecture]! @cacheControl(maxAge: 60)
}
```

Kod 4.4: Korištenje *@cacheControl* direktive unutar sheme aplikacije

Na primjeru upita (Kod 4.4) *coursesByStudent* vidi se kako različita objektna polja utječu na vrijednost *maxAge* oznake ako se navedu u selekcijskom setu upita. Samo polje upita iz operacijskog tipa *Query* ima vijek trajanja od 360 sekundi i privatnog je dosega. Taj vijek trajanja će naslijediti objektni tip *Course*, a isto će vrijediti i za njegova ugniježđena polja *professor* i *students*. Međutim ako selekcijski set upita sadrži polje *tasks* ili polje *lectures*, vijek trajanja pripadnog odgovora će iznositi 60 sekundi. To ponašanje je poželjno jer se očekuje da će nositelj kolegija htjeti mijenjati popise materijala i zadataka.

4.2.2. Testiranje u *Apollo Sandbox*-u

Utjecaj uvođenja predmemoriranje odgovora na izvršavanje poslanih zahtjeva testiran je uz pomoć ugrađenog *Apollo Sanbox* okruženja za izvršavanje GraphQL operacija u pregledniku (Slika 4.4).



Slika 4.4: Izvršavanje upita u *Apollo Sandbox* okruženju

Operacija u pitanju je ranije opisani *coursesByStudent* upit, a u selekcijskom setu navedena su polja za identifikator kolegija *id_course*, naslov kolegija *title*, i popis zadataka *tasks*, gdje je za svaki zadatak tražen njegov naslov u ugniježđenom polju *title*.

```
------------------NEW REQUEST--------------------
[db] select `tb_Courses`.`id_course`, `tb_Courses`.`title` from `tb_Courses` inner join `tb_Enrollments
` on `tb_Courses`.`id_course` = `tb_Enrollments`.`id_course` where `tb_Enrollments`.`id_student` = 'ltm
6210ll2fsg62'
[db] select * from `tb_Tasks` where `tb_Tasks`.`id_course` = 'ltm6210ll2fsg7n'
[db] select * from `tb_Tasks` where `tb_Tasks`.`id_course` = 'ltm6210ll2fsg7p'
[db] select * from `tb_Tasks` where `tb_Tasks`.`id_course` = 'ltm6210ll2fsg7r'
------------------NEW REQUEST--------------------
------------------NEW REQUEST--------------------
------------------NEW REQUEST--------------------
------------------NEW REQUEST--------------------
```

Slika 4.5: Isječak terminala prilikom uzastopnog izvršavanja upita *coursesByStudent*

Iz zabilježenog ispisa u terminalu (Slika 4.5) može se opaziti da će primitak početnog zahtjeva s upitom *coursesByStudent* uzrokovati povezivanje poslužitelja s bazom podataka i izvršavanje nekoliko SQL upita za dohvat potrebnih podataka. Svaki ponovljeni primitak zahtjeva u rasponu 60 sekundi od slanja prvotnog odgovora će izazvati pogodak u priručnoj memoriji, čime će se izbjeći ponovno dohvaćanje podataka iz baze.

4.3. DataLoader

DataLoader je generička inačica mehanizma za grupno izvršavanje ugniježđenih upita u *NodeJS* aplikacijama [12]. Uključena *DataLoader* klasa stvara sučelje između poslužitelja i usluge za pristup i skladištenje podataka, a svaka njezina instanca pruža jedinstvenu memoizacijsku predmemoriju za pamćenje učitanih parova ključeva i JavaScript *Promise* objekata. Imajući na umu implikacije dijeljenja jedne takve instance *DataLoader* klase

između više zahtjeva, u razvijenom projektu se nastojala osigurati konzistentnost predmemoriranih podataka, primjenom metoda za ažuriranje učitanih ključeva nakon provedbe mutacijskih operacija.

Kao što je prethodno opisano, pokretanjem poslužitelja nudi se izbor između učitavanja *resolver* funkcija koje razrješuju vrijednosti objektnih polja stvaranjem individualnih konekcija sa *SQLite* bazom podataka i *reslovera* koji to čine grupnim izvršavanjem ugniježđenih upita. U primjerima (Kod 4.5, Kod 4.6) se vidi razlika u izvedbi tih *resolver* funkcija za polje *tasks* u *Course* objektnom tipu.

```
//tasks.js
const getTasksByCourse = async (id_course) => {
    return await connection
        .select()
        .from('tb_Tasks')
        .where('tb_Tasks.id_course', id_course)
}
//rootResolver.js
Course: {
    //…
    tasks: (course) => getTasksByCourse(course.id_course),
}
```

Kod 4.5: Primjer *resolvera* bez *DataLoader* implementacije

Uobičajeni *resolver* (Kod 4.5) poziva funkciju *getTasksByCourse()*, kojom se spaja na bazu podataka i izvršava upit za dohvat svih zadataka kolegija s identifikatorom *id_course*. To se radi za svaki od kolegija iz popisa dobivenog izvršavanjem *resolvera* roditeljskog objekta.

```
//taskLoaders.js
const taskByIdLoader = new DataLoader(async (ids_task) => {
    const tasks = await connection
        .select()
        .from('tb_Tasks')
        .whereIn('tb_Tasks.id_task', ids_task)

    return ids_task.map((id_task) => tasks.find((task) => task.id_task === id_task))
}, {
    cache: new LRUMap(15)
})
//rootResolverDataLoader.js
Course: {
    //…
    tasks: (course, _args) => tasksByCourseLoader.load(course.id_course),
}
```

Kod 4.6: Primjer *resolvera* s *DataLoader* implementacijom

S druge strane, *resolver* s *DataLoader* podrškom (Kod 4.6) poziva *loader* funkciju *tasksByCourseLoader* instance s kojom će za svaki kolegij u memoizacijsku predmemoriju učitati par njegovog identifikatora i *Promise* objekta. Kapacitet ove predmemorije je ograničena na 15 takvih parova. Na kraju JavaScript petlje događaja poziva se *callback* funkcija *tasksByCourseLoader* objekta, a ona će, spojivši se na bazu podataka, pronaći liste zadataka svih kolegija s učitanim identifikatorima. Nakon toga, dohvaćene vrijednosti će se mapirati s odgovarajućim *Promise* objektima, prema *id_course* ključevima.

Budući da je riječ o globalnoj instanci *DataLoader* klase, za svaku mutacijsku operaciju koja utječe na povezane podatke u bazi podataka potrebno je osigurati da učitani ključevi u predmemoriji aktivno odražavaju provedene promjene. U projektu je to postignuto modifikacijom *resolvera* polja u *Mutation* operacijskom tipu.

Resolveri mutacija *addTask* i *removeTask*, kreiranih za stvaranje novih i brisanje postojećih zadataka, nakon uspješnog pristupa bazi podataka uklanjaju identifikator kolegija pripadnog zadatka iz učitanog niza ključeva, prosljeđujući ga *clear()* funkciji *tasksByCourseLoader* instance (Kod 4.7). Ako se to ne napravi, polje *tasks* će se za taj kolegij nastaviti razrješavati s vrijednošću predmemoriranog para koja više nije aktualna.. Sljedeći put kada se pozove *resolver* polja *tasks*, u niz ključeva će se učitati uklonjeni identifikator kolegija, a *batch* funkcijom će se dohvatiti njegova lista zadataka iz baze i mapirati s njegovim *Promise* objektom u predmemoriji.

```
//taskMutationResolver.js
addTask: async (_root, args, context) => {
    //…
    const taskData = {
       //…
    }
    const response = await createTask(taskData)
    tasksByCourseLoader.clear(response[0].id_course)
    //…
    return await response[0]
},
removeTask: async (_root, args, context) => {
    //…
    const response = await deleteTask(args.id_task)
    tasksByCourseLoader.clear(args.id_course)
    //…
    return await response
}
```

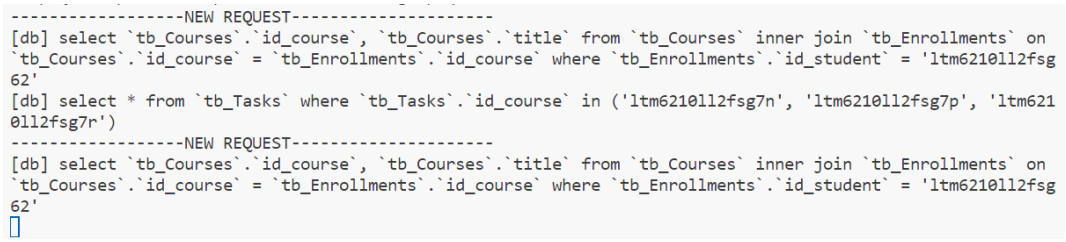
Kod 4.7: Primjer ažuriranja ključeva u memoizacijskoj predmemoriji *DataLoader* instance

Prethodni primjer upita *coursesByStudent* (Slika 4.4) ponovno je služio za testiranje izvršavanja zahtjeva, ovaj put sa svrhom usporedbe broja stvorenih konekcija s bazom podataka, između implementacija sa i bez *tasksByCourseLoader*-a.

```
------------------NEW REQUEST---------------------
[db] select `tb_Courses`.`id_course`, `tb_Courses`.`title` from `tb_Courses` inner join `tb_Enrollments` on
`tb_Courses`.`id_course` = `tb_Enrollments`.`id_course` where `tb_Enrollments`.`id_student` = 'ltm6210ll2fsg
62'
[db] select * from `tb_Tasks` where `tb_Tasks`.`id_course` = 'ltm6210ll2fsg7n'
[db] select * from `tb_Tasks` where `tb_Tasks`.`id_course` = 'ltm6210ll2fsg7p'
[db] select * from `tb_Tasks` where `tb_Tasks`.`id_course` = 'ltm6210ll2fsg7r'
------------------NEW REQUEST---------------------
[db] select `tb_Courses`.`id_course`, `tb_Courses`.`title` from `tb_Courses` inner join `tb_Enrollments` on
`tb_Courses`.`id_course` = `tb_Enrollments`.`id_course` where `tb_Enrollments`.`id_student` = 'ltm6210ll2fsg
62'
[db] select * from `tb_Tasks` where `tb_Tasks`.`id_course` = 'ltm6210ll2fsg7n'
[db] select * from `tb_Tasks` where `tb_Tasks`.`id_course` = 'ltm6210ll2fsg7p'
[db] select * from `tb_Tasks` where `tb_Tasks`.`id_course` = 'ltm6210ll2fsg7r'
```

Slika 4.6: Izvršavanje *coursesByStudent* upita bez *DataLoader* implementacije

Kod izvršavanja upita bez podrške za *DataLoader*-om (Slika 4.6), poslužitelj se sveukupno 4 puta povezao s bazom podataka; jednom pri pozivanju *resolvera coursesByStudent* polja te 3 puta pri pozivanju *resolvera* polja *tasks*, jednom za svaki od 3 kolegija. Broj ovih veza nije se promijenio nakon ponovljenog izvršavanja upita.



Slika 4.7: Izvršavanje *coursesByStudent* upita sa *DataLoader* implementacijom

Za upit koji pak uključuje *tasksByCourseLoader* u *resolveru* polja *tasks* (Slika4.7), stvorene su samo dvije konekcije s bazom podataka; jedna za dohvat popisa kolegija te jedna za dohvat svih lista zadataka za sva 3 kolegija. Kod drugog izvršavanja upita stvorila se samo jedna konekcija s bazom za *resolver* polja *coursesByStudent*, dok su se liste zadataka isporučile iz memoizacijske predmemorije.

4.4. Apollo Client

Apollo Client je komplementarna klijentska solucija za *Apollo Server* poslužitelje [14]. Njezina inačica za *React* podržava slanje zahtjeva preko posebnih *hook* funkcija koje se vežu uz odgovarajuću *React* komponentu, što osigurava da se rezultati izvršenih GraphQL operacija momentalno odražavaju u korisničkom sučelju aplikacije. Jedna od ključnih značajki ovih *hook* funkcija je mogućnost udovoljavanja GraphQL upita korištenjem lokalno dostupnih vrijednosti, dobivenih kao rezultat prijašnje komunikacije s izvornim poslužiteljem. To je postignuto normaliziranom predmemorijom u radnoj memoriji preglednika, dostupne s ugrađenom *InMemoryCache* klasom (Kod 4.8).

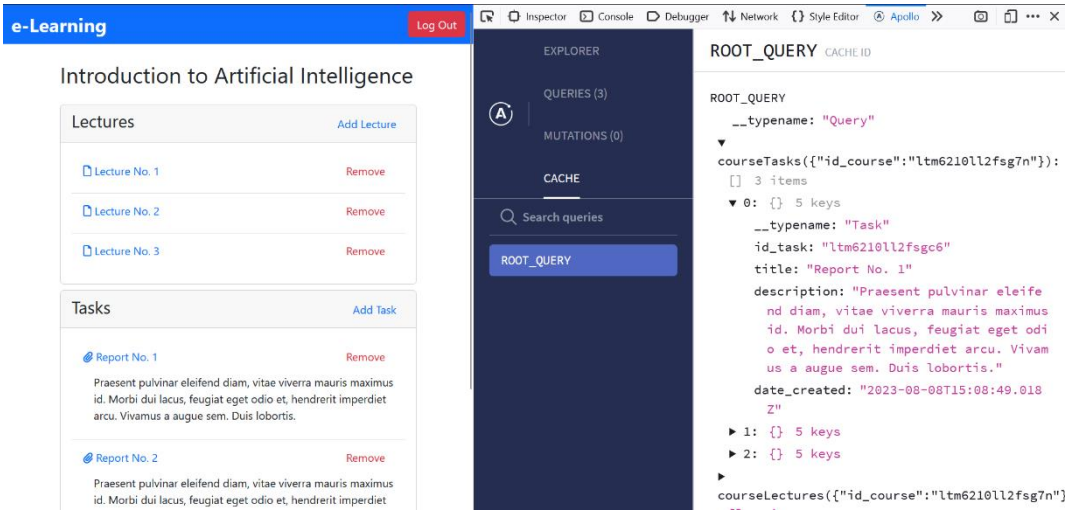
```
const client = new ApolloClient({
    //…
    cache: new InMemoryCache(),
    //…
});
```

Kod 4.8: Primjer postavljanja InMemoryCache instance u konfiguraciji klijentske aplikacije

Ako postavke *hook* funkcije to dozvoljavaju, podaci iz odgovora se automatski pohranjuju u *InMemoryCache* objektu. Slično HTTP predmemoriji preglednika, ponovno slanje zahtjeva izazvat će pogodak u *InMemoryCache*-u te će se iz predmemoriranih zapisa formirati odgovor bez pozivanja GraphQL API-ja i preuzimanja podataka s poslužitelja.

Zahvaljujući normalizaciji predmemoriranih zapisa, objektna polja iz odgovora se mogu individualno referencirati te tako prenamijeniti za potrebe drugih upita. To također sprječava predmemoriranje objektnih polja s istim vrijednostima. *InMemoryCache* funkcionira kao *lookup* tablica objekata indeksiranih po dodijeljenim referencama. Reference su u pravilu generirane povezivanjem polja *id* i *__typename* iz pripadajućeg objekta, a ako jedno od tih polja nije dostupno, odgovor će se referencirati kao cjelina po *stringu* upita iz kojeg je proizišao. Nažalost, to će onemogućiti referenciranje objektnih polja odgovora, no predmemorirani odgovor će dalje biti koristan u slučaju da se upit iz reference pokuša ponovno izvrši.

Za svaki objektni tip je dozvoljeno definirati vlastitu tehniku generiranja referenci, ali jednostavnost razvijene aplikacije to nije zahtijevala pa se referenciranje provodilo na razini odgovora. Pregled predmemoriranih vrijednosti tijekom rada aplikacije potpomognut je *Apollo Client Devtools* ekstenzijom u pregledniku (Slika 4.8).



Slika 4.8: Pregled *InMemoryCache* predmemorije uz *Apollo Client Devtools* ekstenziju

4.4.1. *fetch-policy* parametar

Apollo Client pruža dvije osnovne *hook* funkcije za izvršavanje GraphQL upita [14]. *useQuery() hook* se poziva kod svakog prikazivanja pripadne *React* komponente, dok se *useLazyQuery() hook* poziva samo ako je i njegova *callback* funkcija pozvana, obično kao rezultat neko sporednog događaja u aplikaciji. U postavkama ovih *hook* funkcija dostupan je *fetch-policy* parametar, preko kojeg se određuje kako će povezani upit stupati u interakciju s predmemorijom dok se izvršava.

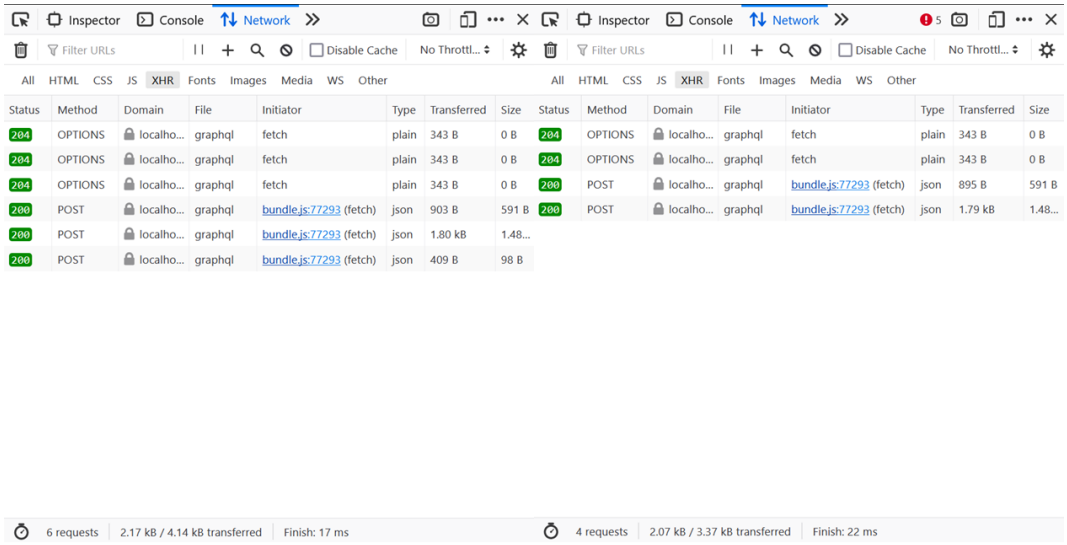
```
//hooks.js
export function useCourseTasksByStudent(id_course, id_student) {
    const [executeCourseTasksByStudent, { loading, error, data }] =
useLazyQuery(queries.courseTasksByStudentQuery, {
        variables: {
            input: {
                id_course,
                id_student
            }
        },
        fetchPolicy: 'network-only',
        skip: !(id_student && id_course)
    })
    return { executeCourseTasksByStudent, courseTasksByStudent:
data?.courseTasksByStudent, loading, error }
}
```

Kod 4.9: Postavljanje fetch-policy parametra

U *useCourseTaskByStudent() hook* funkciji (Kod 4.9), namijenjenoj za dohvat liste zadataka kolegija i statusa predaje za odabranog studenta, *fetch-policy* je postavljen na vrijednost *'network-only'*, što znači da će se pri svakom njegovom pozivu upit izvršavati dohvaćanjem podataka s poslužitelja. Ovo je slično *no-cache* direktivi jer će se isporučeni odgovor svejedno pohraniti u priručnoj memoriji. Inače, zadana vrijednost *fetch-policy* parametra je postavljena na *'cache-first'*, što nalaže da će se pozivanjem *hook* funkcije prvo provjeriti je li odgovor već spremljen u predmemoriji prije nego li se zahtjev proslijedi poslužitelju.

Razlika tih dviju vrijednosti *fetch-policy* parametra prikazana je u sljedećem primjeru (Slika 4.9), gdje se motrio broj poslanih zahtjeva kod prvog učitavanja stranice kolegija i drugog učitavanja nakon povratka s prethodne stranice popisa kolegija. Za prvo učitavanje stranice, izuzevši HTTP OPTIONS metode, poslano je sveukupno tri zahtjeva, po jedan za svaku pozvanu *hook* funkciju. Pri drugom učitavanju stranice, u istim okolnostima su poslana samo

2 zahtjeva, a razlog tome je taj što su dvije od tri pozvane *hook* funkcije imale postavljen *fetch-policy* na vrijednost *'network-only'* dok je kod treće bio postavljen na *'cache-first'*.



Slika 4.9: Usporedba broja poslanih zahtjeva između prvog i drugog prelaska na stranicu kolegija

4.4.2. Ažuriranje predmemoriranih odgovora

Glavni nedostatak *InMemoryCache* klase leži u činjenici da nema izvornu podršku za računanjem svježine predmemoriranih odgovora. Stoga je upravljanje njezinim sadržajem, u kombinaciji s prikladnih *fetch-policy* postavkama, neophodno da bi se osigurala konzistentnost podataka u aplikaciji.

Upravo iz tog razloga, *hook* funkcija za izvršavanje mutacijskih operacija *useMutation()* dolazi sa vrlo praktičnom značajkom automatskog ažuriranja predmemoriranih objekata, ali samo ako su određeni uvjeti ispunjeni. Naravno, neophodno je da odgovor poslužitelja za izvršenu mutaciju uključuje sve objektne koji su njome zahvaćeni, no također se očekuje da se ti objekti mogu normalizirati po istom principu kao objekti istog tipa iz predmemorije. Premda nijedan od ta dva uvjeta nije u potpunosti ispunjen u okviru projekta, ažuriranje predmemorije je srećom još uvijek provedivo.

useMutation() hook, uz prethodno opisanu značajku, nudi pristup metodi *update()* za ručno mijenjanje predmemoriranih podataka. Metodi se kao prvi parametar prosljeđuje ciljana *InMemoryCache* instanca, dok u mjestu drugog parametra prima *callback* funkciju zaduženu za ažuriranje podataka.

```
//hooks.js
export function useAddTask(id_course, title, description) {
    const [addTask, { loading, error, data }] = useMutation(queries.addTaskMutation,
{
        variables: {
            //…        },
        update(cache, { data }) {
            const { courseTasks } = cache.readQuery({
                query: queries.courseTasksQuery,
                variables: {
                    idCourse: id_course
                }
            })
            cache.writeQuery({
                query: queries.courseTasksQuery,
                variables: {
                    idCourse: id_course
                },
                data: {
                    courseTasks: courseTasks.concat(data?.addTask)
                }
            })
        }
    })
    return { addTask, loading, error }
}
```

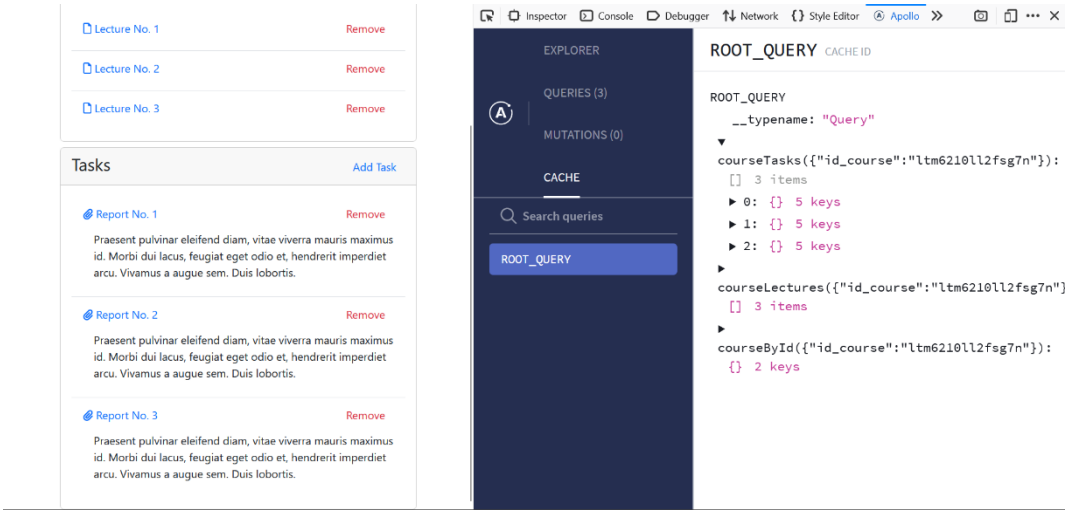
Kod 4.10: Ažuriranje predmemoriranih podataka korištenjem *update()* metode

U priloženom isječku koda (Kod 4.10) izdvojena je definicija *hook* funkcije *useAddTask()*, koja se poziva kada nositelj kolegija pokušava dodati novi zadatak u trenutni popis. Uz uobičajeno izvršavanje mutacijske operacije, uočljivo je da se podaci iz dobivenog odgovora prosljeđuju *callback* funkciji *update()* metode.

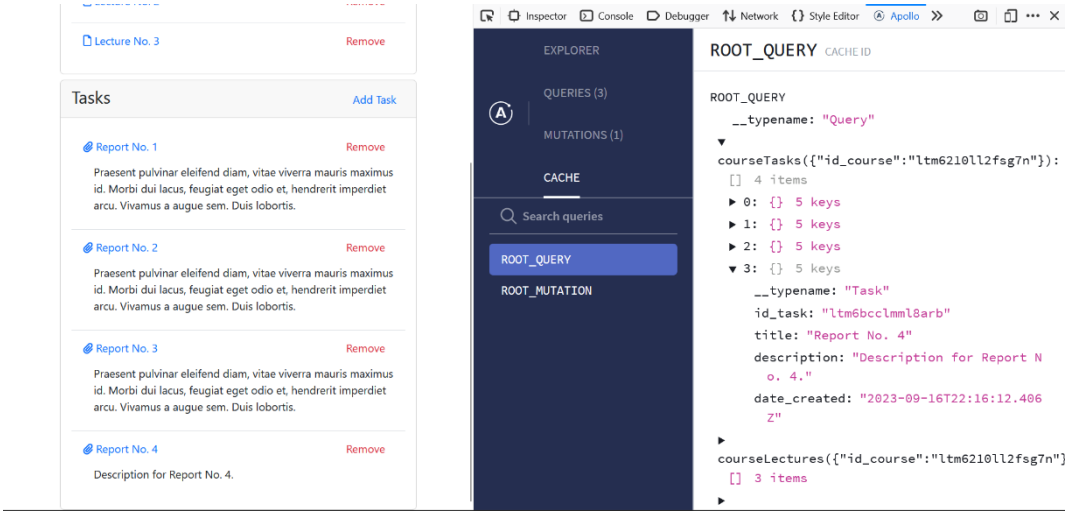
Prvo što *callback* funkcija radi jest dohvat ranije predmemoriranog odgovora s dosadašnjim popisom zadataka, pozivom readQuery() metode specificiranog *InMemoryCache* objekta. Odgovor se referencira po *stringu* upita *courseTasksQuery* te po varijablama koje je primio kada se izvršavao. U sljedećem koraku, nad instancom predmemorije se poziva metoda *writeQuery()*, s istom referencom predmemoriranog odgovora koji se želi izmijenti. U ovom

slučaju, to se postiže dodavanjem povratnog objekta iz odgovora izvršene mutacije na kraj trenutnog popisa zadataka.

Ova promjena će se trenutačno odraziti u korisničkom sučelju aplikacije zato što *hook* funckije upita odgovore reprezentiraju kao *React* stanja, a isto će se moći vidjeti u prikazu predmemorije u Apollo Client Sandbox-u (Slika 4.10, Slika 4.11).



Slika 4.10: Prikaz sučelja aplikacije prije dodavanja novog zadatka u popis



Slika 4.11: Prikaz sučelja aplikacije nakon dodavanja novog zadatka u popis

4.4.3. Primjena postojanih upita i HTTP predmemoriranja

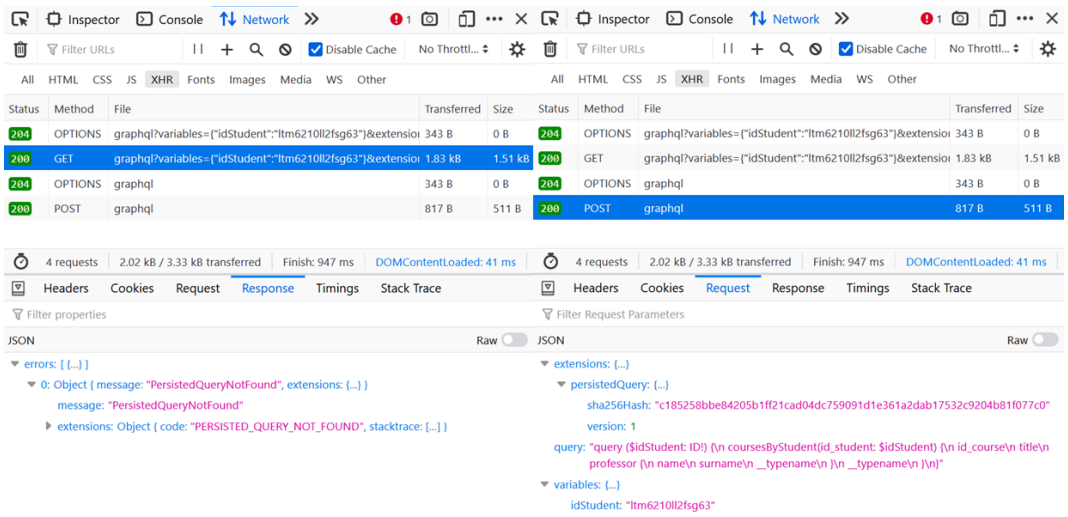
Klijentska aplikacija automatski šalje postojane upite poslužitelju pri pozivu *useQuery()* i *useLazyQuery() hook* funkcija. Za samu implementaciju postojanih upita, *Apollo Client* biblioteka ima na raspolaganju *createPersistedQueryLink()* funkciju, koja se poziva u sklopu *link* svojstva iz *options* parametra *Apollo Client* instance (Kod 4.11).

```
//apolloClient.js
const client = new ApolloClient({
    link: createPersistedQueryLink({ sha256, useGETForHashedQueries: true
}).concat(authLink).concat(httpLink),
    cache: cache,
});
```

Kod 4.11: Omogućavanje postojanih upita u klijentskoj aplikaciji

Funkcija *createPersistedQueryLink()* definira proceduru provođenja postojanih upita, djelujući kao *middleware* između upita i veze s poslužiteljem iz *httpLink* objekta. Identifikator za indeksiranje postojanog upita se generira šifriranjem njegovog *string* zapisa sa *sha256* algoritmom, a svojstvo *useGETForHashedQueries* nalaže da se identifikator treba slati poslužitelju kao parametar u HTTP GET metodi.

Koraci izvršavanja postojanog upita koji nije predmemoriran na poslužiteju su konzistentni s tehnikom opisanom u prethodnom poglavlju. To je evidentno i iz analize mrežnog prometa stranice s popisom kolegija (Slika 4.12), na kojoj se izvršava upit za dohvat tog popisa. Prvi pokušaj izvršavanja postojanog upita slanjem zahtjeva u HTTP GET metodi rezultira greškom u tijelu odgovora, s porukom da postojani upit nije pronađen. Zatim se HTTP POST metodom šalje zahtjev u čijem se tijelu nalaze identifikator postojanog upita, njegov *string* zapis i potrebne varijable. Taj se zahtjev potom normalno izvršava i u odgovoru se isporučuju željeni podaci, a postojani upit se pohranjuje u predmemoriju poslužitelja za potrebe izvršavanja budućih upita, poslanih HTTP GET metodom.



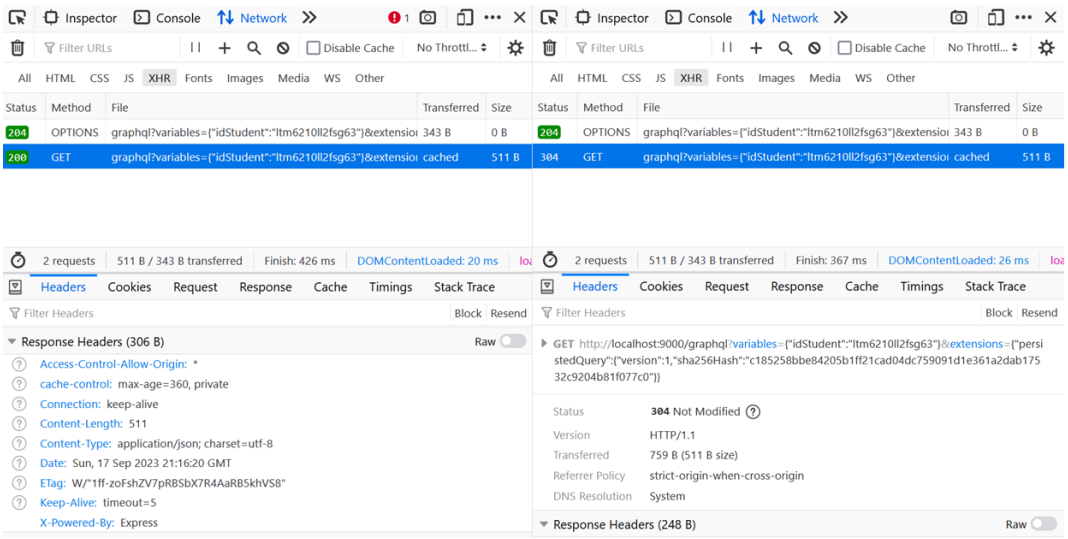
Slika 4.12: Primjer poslanih zahtjeva tijekom prvog izvršavanja postojanog upita

Ova konfiguracija će, u koordinaciji s opcijom računanja *Cache-Control* zaglavlja na poslužitelju, osigurati da se dobiveni odgovori mogu spremati u HTTP predmemoriji preglednika i iz nje posluživati ovisno o dodijeljenim direktivama. Pogodnosti toga pristupa su još izraženije ako se predmemoriranje odgovora simultano izvršava i u normaliziranoj predmemoriji *Apollo Client*-a. Primjerice, kod *hook* funkcija kojima je *fetch-policy* parametar postavljen na *'network-only'* vrijednost, zahtjevom će se prvo pristupiti priručnoj memoriji preglednika i pokušati pronaći predmemorirani odgovor preko URI-ja. Istovremeno, upiti *hook* funkcija s *fetch-policy* parametrom vrijednosti *'cache-first'* će prvo provjeriti prisutnost odgovora u normaliziranoj predmemoriji.

Kada se na stranici s popisom upisanih kolegija osvježi kartica preglednika u kojem je aplikacija otvorena, ponovljeno izvršavanje postojanog upita uzrokovat će da se pošalje samo putem HTTP GET metode.

Ponovnim osvježavanjem kartice, neposredno nakon toga, poslani zahtjev će dobiti odgovor iz predmemorije preglednika, što se vidi po *cached* oznaci unutar *Transferred* stupca zahtjeva (Slika 4.13). Može se uočiti i da su u *Cahce-Control* zaglavlju odgovora postavljene *max-age* i *private* direktive i da vrijednost max-age direktive iznosi 360 sekundi. To je u sladu s postavljenim *maxAge* i *scope* oznakama *@cacheControl* direktive za polje izvršenog upita u GraphQL shemi.

Nakon proteklih 360 sekundi od primitka originalnog odgovora, osvježavanje kartice će izazvati njegovu validaciju u predmemoriji. Poslužitelju će se tada poslati uvjetni zahtjev, no validacija se neće provest usporedbom entitetskih ili vremenskih oznaka, već će se podaci ponovno dohvatiti iz baze podataka i usporediti s predmemoriranim odgovorom unutar *InMemorLRUCache*-a. Budući da se ovom slučaju ti podaci poklapaju, klijentu se šalje odgovor sa statusnim kodom 304, a odgovoru iz predmemorije preglednika se ažuriranjem zaglavlja obnavlja svježina i omogućuje ponovno korištenje.



Slika 4.13: Dohvaćanje i validacija odgovora iz predmemorije preglednika

Zaključak

Cilj ovog rada je bio pobliže opisati problematiku predmemoriranja podataka u GraphQL-u te ponuditi teorijsko objašnjenje i praktičnu demonstraciju trenutno dostupnih strategija za njihovo nadilaženje. Solucije implementirane u sklopu razvijenog projekta su prikazale kako ove strategije mogu pronaći primjenu u stvarnim okolnostima razvoja Web aplikacija i koji utjecaj one imaju na provođenje komunikacije između poslužitelja i klijenata.

Zabilježene su brojne optimizacije koje se ne tiču samo smanjenog broja poslanih korisničkih zahtjeva, već i smanjene složenosti njihove obrade na poslužitelju te efikasnije formacije odgovora preko sloja za pristup podacima. Upoznavši se sa pogodnostima deklarativnog dohvaćanja podataka koje pruža upitni jezik GraphQL-a, saznanja proizišla iz ovoga rada bi mogla dodatno motivirati zainteresirane razvojne programere da razmotre korištenje GraphQL-a kao moguće alternative za REST.

Budući da se u sklopu projekta nastojalo demonstrirati što više tehnika predmemorianja podataka u GraphQL-u, fokus nije bio na prikupljanju preciznih mjerenja performansi aplikacije kao rezultat njihova korištenja. Preporuka je da budući istraživački radovi na sličnu temu bolje prouče utjecaj uvođenja opisanih tehnika na izvedbu cjelokupne aplikacije iz aspekata vremena obrade zahtjeva i isporuke odgovora te količine razmijenjenih podataka. U tu svrhu se mogu definirati standardizirani jedinični i E2E testovi za testiranje pojedinačnih, ali i kombinacija više strategija predmemoriranja.

Literatura

[1] Wessels, Duane. *Web caching*. "O'Reilly Media, Inc.", 2001.

[2] Fielding, Roy T., et al. *RFC 7234 - Hypertext Transfer Protocol (HTTP/1.1): Caching*. "HTTP Working Group, IETF", svibanj 2014. Web. (Pristup: 24.8.2023.). https://httpwg.org/specs/rfc7234.html

[3] Gourley, Davd, and Brian Totty. *HTTP: the definitive guide*. "O'Reilly Media, Inc.", 2002.

[4] Mozilla Contributors. *Cache-Control - HTTP*. MDN Web Docs. "Mozilla Developer Network", srpanj 2023. Web. (Pristup: 24.8.2023.) https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching

[5] GraphQL Contributors. *GraphQL Working Draft*. GraphQL. "Joint Developer Foundation", srpanj 2023. Web. (Pristup: 24.8.2023.) http://spec.graphql.org/draft/

[6] Porcello, Eve, and Alex Banks. *Learning GraphQL: declarative data fetching for modern web apps*. "O'Reilly Media, Inc.", 2018.

[7] Wieruch, Robin. *The Road to GraphQL: Your journey to master pragmatic GraphQL in JavaScript with React. js and Node. js*. Robin Wieruch, 2018.

[8] Giroux, Marc-Andre. *Production Ready GraphQL*. Marc-Andre Giroux, 2020.

[9] *Serving over HTTP*. GraphQL. "The GraphQL Foudation", (n.d.) (Pristup: 25.8.2023.) https://graphql.org/learn/serving-over-http/

[10] benjie et. al. *GraphQL Over HTTP*. Github. “Joint Development Foundation", travanj 2023. (Pristup: 3.9.2023.) https://github.com/graphql/graphql-over-http/blob/main/spec/GraphQLOverHTTP.md

[11] *Execution*. "The GraphQL Foudation", (n.d.) (Pristup: 10.9.2023.) https://graphql.org/learn/execution/

[12] Byron, Lee et. al. *Dataloader README*. Github “Joint Development Foundation" siječanj 2023 (Pristup: 12.9.2023.) https://github.com/graphql/dataloader/blob/main/README.md

[13] *Apollo Server*. Apollo GraphQL Docs. "Apollo Graph Inc." (n.d.) (Pristup: 14.9.2023.) https://www.apollographql.com/docs/apollo-server

[14] *Apollo Client (React)*. Apollo GraphQL Docs. "Apollo Graph Inc." (n.d.) (Pristup: 14.9.2023.) https://www.apollographql.com/docs/react

Popis slika

Popis isječaka koda

Skraćenice

API	*Application Programming Interface*	sučelje za programiranje aplikacija
REST	*Representational State Transfer*	reprezentacijski prijenos podataka
HTTP	*Hyper Text Transfer Protocol*	protokol za prijenos hiperteksta
OS	*Operating System*	operacijski sustav
ARP	*Adress Resolution Protocol*	protokol za razrješavanje adresa
DNS	*Domain Name System*	sustav domenskih imena
URI	*Uniform Resource Identifier*	uniformni identifikator resursa
SQL	*Structured Query Language*	strukturirani upitni jezik
JSON	*JavaScript Object Notation*	notacija JavaScript objekata
DBaaS	*Database as s Service*	baza podataka kao usluga
CORS	*Cross-origin Resource Sharng*	među-izvorno dijeljenje resursa
CDN	*Content Delivery Network*	mreža za isporuku podataka
LRU	*Least Recently Used*	algoritam najmanje nedavno korišteni
E2E	End-to-End	testovi od kraja do kraja