Razvoj web servisa temeljenih na upitnom jeziku GraphQL i arhitekturi bez poslužitelja

Bel, Filip

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Bel

Razvoj web servisa temeljenih na upitnom jeziku GraphQL i arhitekturi bez poslužitelja

DIPLOMSKI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Bel

Matični broj: 45896/17-R

Studij: Informacijsko i programsko inženjerstvo

Razvoj web servisa temeljenih na upitnom jeziku GraphQL i arhitekturi bez poslužitelja

DIPLOMSKI RAD

Mentor/Mentorica:

Prof. dr. sc. Danijel Radošević

Varaždin, lipanj 2022.

Filip Bel

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Tema ovog rada je razvoj web servisa temeljenih na upitnom jeziku GraphQL i arhitekturi bez poslužitelja. U uvodu biti će pojašnjena uloga web aplikacija u svakodnevnim životu ljudi te REST standard razmjene podataka između poslužitelja i klijenta. Potom će biti objašnjen HTTP protokol koji je temelj rada REST arhitekture nakon čega će fokus biti na novom načinu razmjene podataka - GraphQL upitnom jeziku. Biti će primjerima objašnjeno što je to GraphQL, kako se razlikuje od REST-a te koji pristupi u njegovoj implementaciji postoje. Potom ćemo se dotaknuti pojma računarstva u oblaku koji danas s razlogom privlači mnogo pažnje. Biti će pojašnjeno što je to arhitektura bez poslužitelja, navedeni neki od AWS servisa koji će se koristiti u praktičnom dijelu rada te obrazloženo koje su prednosti u korištenju takve arhitekture. Kao praktičan dio rada biti će razvijena web aplikacija koja će restoranima omogućiti izradu i održavanje svog digitalnog jelovnika, dok će gostima restorana aplikacija omogućiti pregled jelovnika te ocjenjivanje jela s jelovnika. Na kraju rada slijedi zaključak te komentari na obrađenu temu.

Ključne riječi: web aplikacija, HTTP, REST, GraphQL, serverless, AWS

Sadržaj

1. Uvod

Web aplikacije danas imaju važnu ulogu u našim svakodnevnim životima. Nekad je korištenje weba značilo pristupanje statičnim web stranicama u svrhu pronalaska potrebnih informacija dok danas to uključuje visoku razinu interaktivnosti korisnika s web aplikacijom. Web i mobilne aplikacije postaju zrelije i složenije te zahtijevaju brzu prilagodbu i efikasnost u pogledu potreba korisnika pa tako softverski inženjeri dolaze do novih načina poboljšanja interakcije između strane klijenta i poslužitelja aplikacije. [1]

Jedna od najvećih promjena tijekom posljednjih nekoliko godina u tom pogledu jest GraphQL - jezik za upite i manipulaciju podacima otvorenog koda koji se primjenjuje kod komunikacije između klijenta i poslužitelja. REST (*Representation State Transfer*) je predstavljen 2000. godine te je bio brzo prihvaćen standard. Međutim, zbog rasta kompleksnosti aplikacija te promjene njihovih potreba, ova arhitektura otkrila je neke slabosti vezane uz izvedbu i fleksibilnost aplikacija. Navedeni problemi mogu se riješiti pomoću GraphQL-a te iako je to još dosta mlada tehnologija, široko je prihvaćena te korištena od strane velikih tvrtaka kao što su Facebook, Netflix i GitHub upravo zbog brojnih prednosti. [2]

Kako raste kompleksnost web aplikacija, tako i proces njihovog pokretanja (eng. *deployment)* postaje sve složeniji. Posljednjih je godina tako sve zastupljeniji pojam računarstva u oblaku (eng. *cloud computing*) te time i arhitektura bez poslužitelja (eng. *serverless architecture*) koja programerima omogućuje razvoj i pokretanje servisa, odnosno aplikacija, bez potrebe za upravljanjem temeljnom infrastrukturom na kojoj će se ta aplikacija pokretati.

U ovom ću radu prvo krenuti od HTTP protokola koji je temelj današnje komunikacije na webu. Potom ću se osvrnuti na REST arhitekturu koja je već posljednjih 20 godina standard za razmjenu podataka između klijenta i poslužitelja te je usporediti sa GraphQL upitnim jezikom na kojem će biti fokus ovog rada. Nakon pojašnjenja i razrade GraphQL upitnog jezika dotaknut ću računarstvo u oblaku te pojasniti arhitekturu bez poslužitelja u kontekstu ovog rada. Kao praktičan dio rada implementirat ću web aplikaciju koja restoranima omogućuje izradu i održavanje svog digitalnog jelovnika, a korisnicima omogućuje pregled i pretraživanje restorana i njihovih jelovnika te ocjenjivanje jela s jelovnika.

2. Korištene tehnologije

Pri razvoju aplikacije u praktičnom dijelu rada osim HTML-a i CSS-a korištene su tehnologije koje će u nastavku biti pojašnjene. Obje strane aplikacije (klijent i poslužitelj) implementirane su koristeći TypeScript programski jezik koji je zapravo nad skup JavaScript jezika. Na strani poslužitelja bitno je istaknuti Nest.js razvojni okvir koji potiče modularnu strukturu koda te u pozadini koristi Apollo biblioteku za implementaciju GraphQL web servisa. Na strani klijenta korišten je fleksibilni razvojni okvir Next.js koji se temelji na popularnoj React biblioteci. Arhitektura back end dijela aplikacije biti će strukturirana primjenom arhitekture bez poslužitelja (eng. *serverless architecture*) te će za tu svrhu biti korištene usluge AWS-a.

2.1. TypeScript

TypeScript je programski jezik razvijen od strane Microsofta koji se temelji na JavaScript jeziku. To je strogi sintaktički nad skup JavaScripta koji mu daje statičku provjeru tipova, što je upravo jedan od nedostataka JavaScripta koji rezultira težim i nepravovremenim uočavanjem grešaka u kodu. Osim programskog jezika, TypeScript se može smatrati i alatom koji svojim mogućnostima olakšava pisanje i održavanje čistog koda. Pri kompiliranju (eng. *compile*) TypeScript koda vrši se statička provjera sintakse i tipova te se on potom prevodi u čisti JavaScript kod koji se tada pokreće u web pregledniku ili na poslužitelju (Node.js). [3]

2.2. Nest.js

Nest.js je razvojni okvir koji služi za izgradnju učinkovitih i skalabilnih Node.js aplikacija na strani poslužitelja. Okvir je razvijen koristeći TypeScript i kombinira elemente objektno orijentiranog programiranja (OOP), funkcionalnog programiranja (FP) i funkcionalno reaktivnog programiranja (FRP). U pozadini Nest.js koristi robusne okvire za pokretanje HTTP poslužitelja kao što je Express ili Fastify. Posljednjih je godina zahvaljujući Node.js okruženju JavaScript postao moglo bi se reći standard za razvoj kako front end strane web aplikacija tako i back end strane. To je dovelo do izrade mnogo odličnih razvojnih okvira i biblioteka za razvoj web aplikacija, no ni jedna od njih učinkovito ne rješava glavni problem - arhitekturu. Sama filozofija Nest.js okvira jest da nudi gotovu arhitekturu aplikacije koja programerima i timovima omogućuje stvaranje aplikacija koje se mogu lako testirati, skalabilne su, slabo povezane te omogućuju vrlo lagano održavanje. Ova arhitektura uvelike je inspirirana Angular razvojnim okvirom. [4]

2.3. Next.js

Next.js je razvojni okvir koji se temelji na React biblioteci. React je deklarativna, učinkovita i fleksibilna JavaScript biblioteka otvorenog koda koja omogućuje izradu kompleksnog korisničkog sučelja od malih i izoliranih dijelova koda koje se nazivaju komponente. Jedan od nedostataka React biblioteke jest taj što je teško provesti SEO optimizaciju stranice zato što se aplikacija renderira potpuno na strani klijenta, odnosno u web pregledniku pomoću JavaScripta. [5] Next.js rješava taj problem na način što omogućuje renderiranje na strani poslužitelja tako da klijent dobije gotov HTML kod. Neke od ostalih značajki Next.js okvira su optimizirano učitavanje slika, pojednostavljeno upravljanje rutama te općenito bolje iskustvo prilikom razvoja. [6]

2.4. AWS

Amazon Web Services (AWS) podružnica je tvrtke Amazon te je jedan od najpopularnijih pružatelja usluga računarstva u oblaku. Trenutno AWS pruža više od 200 usluga raspoređenih u kategorije računalne snage (eng. *computing*), pohrane (eng. *storage*), baze podataka (eng. *databases and caching*), mrežna infrastruktura (eng. *network infrastructure*) itd. [7]

Većina AWS usluga nije izložena izravno krajnjim korisnicima, već umjesto toga funkcionalnost nude putem API-ja koje programeri mogu integrirati u svoje aplikacije. Tim servisima se tako pristupa putem HTTP-a koristeći REST arhitekturu. Te usluge organizacijama pomažu u bržem razvoju, nižim troškovima i lakšem skaliranju njihovih proizvoda. U praktičnom dijelu rada koristit će se tri usluge AWS-a - S3 za pohranu datoteka, odnosno slika, Aurora za pokretanje baze podataka te AWS Lambda - servis koji omogućuje pokretanje koda bez potrebe za upravljanje infrastrukturom.

2.5. Serverless Framework

Serverless Framework je razvojni okvir koji olakšava razvoj i pokretanje AWS Lambda funkcija te ostale AWS infrastrukture potrebne za rad aplikacije. To je CLI koji nudi strukturu, automatizaciju i najbolje prakse omogućujući da se usredotočimo na razvoj sofisticiranih arhitektura bez poslužitelja vođene događajima koje se sastoje od funkcija i događaja. [8]

3. HTTP protokol

Od svog razvoja ranih 1990-ih, web je postao središnji dio života mnogih ljudi, stvarajući prilike za rad, učenje, trgovinu, socijalnu povezanost i još mnogo toga. Otvaranjem web preglednika moguće je pristupiti gotovo beskrajnom svemiru informacija. Da bi to bilo moguće, jedno od načela iza dizajna weba jest otvorenost - otkrivanje i standardizacija pravila prema kojima bi web aplikacije morale raditi i međusobno komunicirati. [9] Ova otvorenost omogućuje interoperabilnost što različitim aplikacijama koje se pridržavaju dogovorenih pravila omogućuje kako bi one međusobno ispravno radile. Iako se moderni web dosta razvio te danas omogućava razmjenu ne samo tekstualnih podataka, temeljni mehanizam za dohvaćanje informacija i dalje je isti kao što je bio izvorno zamišljen. Ključ ove komunikacije naziva se HTTP - protokol koji služi za razmjenu podataka na webu. [10]

HTTP *(Hypertext Transfer Protocol*) je skup pravila između klijenta (mrežni resurs koji zahtijeva podatke) i poslužitelja (mrežni resurs koji prima i odgovara na zahtjev klijenta). Drugim riječima, HTTP je "jezik" koji klijent i poslužitelj koriste za razmjenu podataka. Svaki HTTP zahtjev koji se šalje poslužitelju sadrži sljedeće:

- HTTP metodu
- URL, odnosno putanju na koju se zahtjev šalje
- verziju HTTP protokola
- zaglavlje zahtjeva
- opcionalno tijelo (eng. *body*) zahtjeva

HTTP metoda označava radnju koju HTTP zahtjev očekuje od poslužitelja. Na primjer, dvije najčešće HTTP metode su GET i POST. Zahtjev GET zauzvrat očekuje informacije (npr. "dohvati jelovnik restorana"), dok zahtjev POST obično označava da klijent šalje informacije web poslužitelju (npr. "dodaj novo jelo u jelovnik"). Osim dvije spomenute metode još su često korištene metode PUT i PATCH za ažuriranje podataka na poslužitelju (npr. "ažuriraj cijenu određenog jela na jelovniku) te DELETE metoda za brisanje podataka (npr. "obriši određeno jelo s jelovnika"). [10]

Zaglavlje zahtjeva sadrži tekstualne informacije pohranjene u parovima ključ/vrijednost i uključene su u svaki HTTP zahtjev. Ova zaglavlja sadrže neke osnovne informacije o zahtjevu kao npr. tip podataka koje klijent prihvaća kao odgovor, podaci o autentifikaciji itd. Tijelo zahtjeva može sadržavati same podatke koje želimo poslati poslužitelju - npr. ako šaljemo zahtjev za kreiranje nove stavke u jelovnika, onda ćemo u tijelu tog zahtjeva poslati npr. naziv stavke, cijenu, opis itd.

HTTP odgovor je ono što klijent dobije od strane poslužitelja kao odgovor na poslani zahtjev koji sadrži sljedeće:

- HTTP statusni kod
- zaglavlja odgovora
- opcionalno tijelo (eng. *body*) odgovora

HTTP statusni kod sastoji se od 3 znamenke te obično označuje je li HTTP zahtjev bio uspješan. Statusni kodovi podijeljeni su u 5 kategorija, gdje "xx" predstavlja različite brojeve od 00 do 99: [11]

- 1xx - informativni
- 2xx - uspjeh
- 3xx - redirekcija
- 4xx - pogreška na strani klijenta
- 5xx - pogreška na strani poslužitelja

Tako npr. statusni kod koji započinje s "2" označava uspjeh. Npr. ako klijent od poslužitelja zatraži web stranicu, uspješan odgovor će najčešće nositi statusni kod 200. Sami dohvaćeni podaci nalazit će se u tijelu odgovora, dok će se u zaglavlju isto kao kod HTTP zahtjeva nalaziti neki dodatni podaci.

3.1. Aplikacijsko programsko sučelje (API)

Aplikacijsko programsko sučelje (API) je skup pravila koja nekom programu omogućuju prijenos podataka drugom programu. API je sučelje što znači da definira način interakcije. API omogućuje programerima da izbjegnu suvišan rad - umjesto ponovne izgradnje aplikacijskih funkcija koje već postoje, programeri mogu integrirati već postojeće (eksterne) funkcije u svoju aplikaciju. [12] Zamislimo da izrađujemo aplikaciju za ocjenjivanje jela koja se nalaze na jelovniku restorana te da se jelovnici često mijenjaju. Mogli bismo potrošiti mnogo vremena i novaca na izradi vlastitog sustava za dohvaćanje jelovnika od velikog broja restorana, ili pak sustava kojim ćemo omogućiti svakom restoranu da u našu aplikaciju unese svoj jelovnik. No, ukoliko postoji neki vanjski servis za dohvaćanje ažurnih jelovnika svih restorana isplativije bi bilo integrirati taj postojeći servis u našu aplikaciju što bi nam dalo mogućnost usredotočenja na druge aspekte naše aplikacije.

3.2. REST servisi

REST je akronim za ***RE****presentational* ***S****tate* ***T****ransfer* koji predstavlja skup arhitektonskih ograničenja za distribuirane hipermedijske sustave. Prvi puta ga spominje Roy Fielding 2000. godine u svojoj disertaciji gdje predstavlja principe i ograničenja koji moraju biti zadovoljeni da bi se neki servis mogao nazvati REST servisom. Kao glavni principi REST servisa izdvajaju se ne postojanje stanja (eng. *statelessness*), mogućnost keširanja (eng. *cacheability*), jedinstveno sučelje (eng. *uniform interface*), slojevita sistemska arhitektura (eng. *layered system architecture*) i kod na zahtjev (eng. *code on demand*). [13]

REST servisi nemaju stanje što znači da svaki zahtjev mora uključiti sve potrebne informacije za obradu, odnosno poslužitelj ne sprema nikakve podatke vezane za sam zahtjev klijenta. REST servisi također imaju slojevitu sistemsku arhitekturu što znači da je moguće da zahtjevi prije obrade prolaze kroz više slojeva sustava. Npr. ako klijent pošalje zahtjev nekom REST servisu, poslužitelj može taj zahtjev proslijediti nekom drugom servisu za obradu, a da to klijent zapravo ni ne zna. [14]

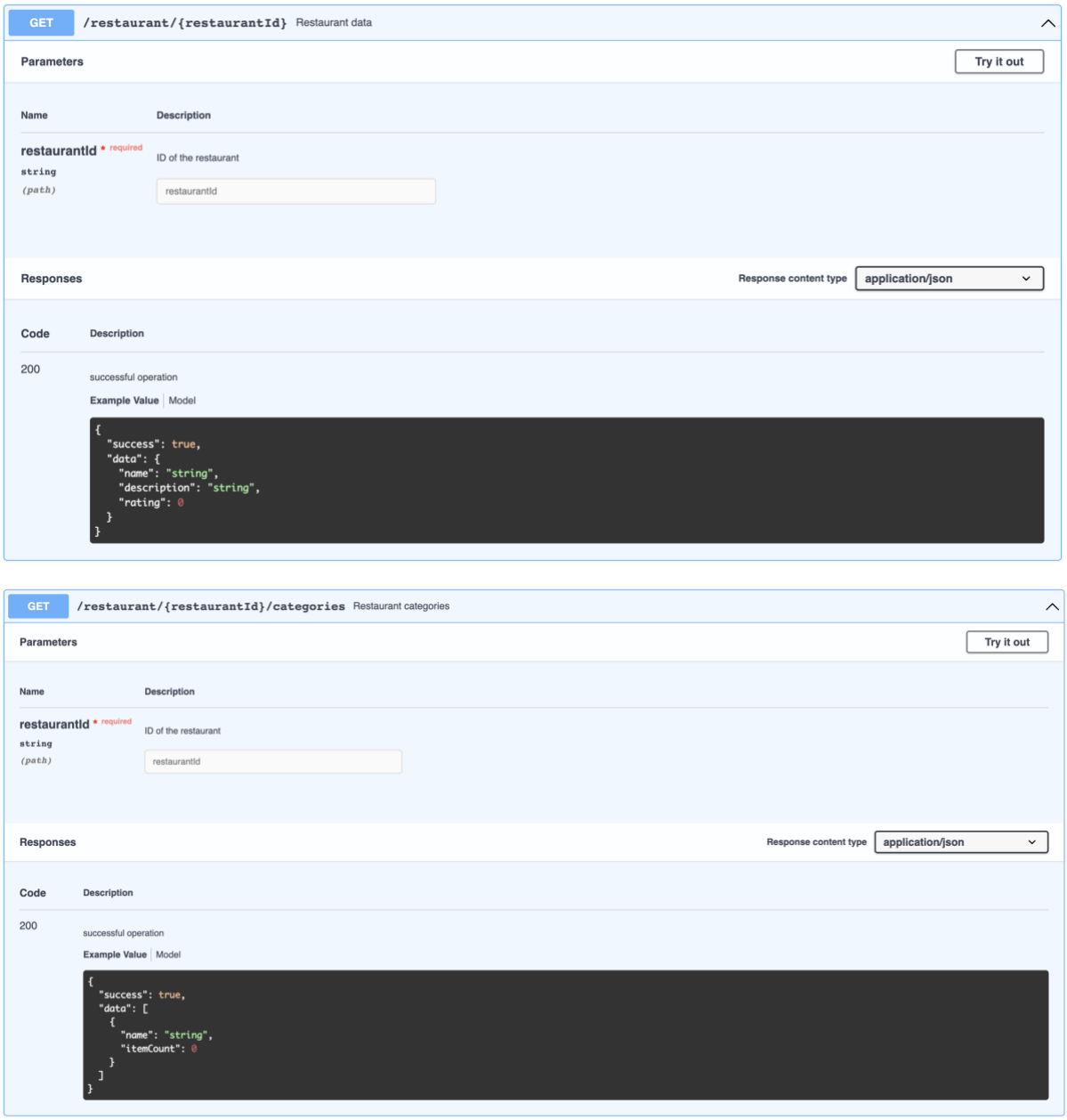
Uzmimo za primjer da razvijamo aplikaciju koja će imati tri stranice. Na prvoj stranici prikazat će se podaci o restoranu (ime, opis i ocjena). Na drugoj će se prikazati sve kategorije sa jelovnika tog restorana (naziv kategorije i broj jela u kategoriji za svaku kategoriju) gdje će biti moguće obrisati željenu kategoriju. Na trećoj stranici će se prikazati jela iz odabrane kategorije (naziv, opis, cijena i slika jela). Ako taj scenarij sagledamo sa strane REST servisa, biti će nam potrebne četiri krajnje točke: jedna za dohvaćanje podataka o restoranu, jedna za dohvaćanje svih kategorija restorana, jedna za dohvaćanje svih jela u odabranoj kategoriji te jedna za brisanje odabrane kategorije.

3.2.1. OpenAPI

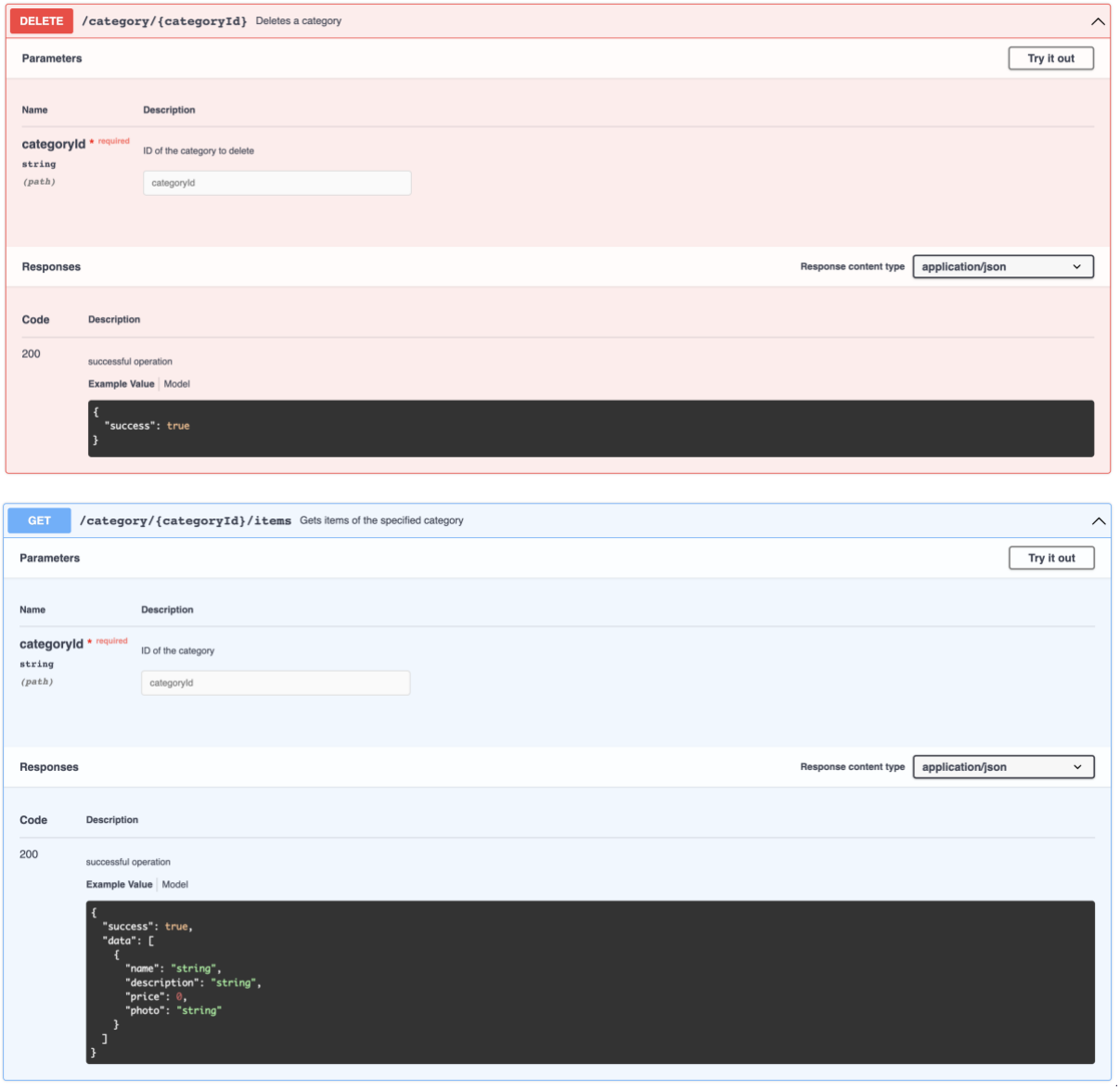
OpenAPI specifikacija, ranije poznata kao Swagger specifikacija, je specifikacija za opisivanje, korištenje i vizualizaciju REST servisa. Drugim riječima, OpenAPI je specifikacija koja služi za dokumentiranje REST servisa. [15]

Vratimo se na ranije spomenut primjer REST servisa te zamislimo da smo servis implementirali i da će neki drugi programer front end dio aplikacije integrirati sa servisom koji smo razvili. Ono što taj drugi programer mora znati su pravila REST servisa kojeg smo razvili - mora znati na koje krajnje točke aplikacija mora slati zahtjeve, koristeći koje metode (npr. GET, POST) te ostala pravila kako bi mogao slati ispravan zahtjev. Za tu ćemo svrhu napraviti dokumentaciju REST servisa prema OpenAPI specifikaciji sa koje će biti moguće sve to iščitati. Dokumentacija će izgledati kao na slikama 1 i 2 te će za svaku krajnju točku REST servisa biti

navedeni mogući parametri, potrebno tijelo (eng. *body*) zahtjeva te će biti prikazan primjer/model odgovora na taj zahtjev:



Slika 1: OpenAPI specifikacija krajnjih točaka za dohvaćanje podataka o restoranu te za dohvaćanje svih kategorija restorana



Slika 2: OpenAPI specifikacija krajnjih točaka za brisanje određene kategorije te za dohvaćanje svih stavaka iz određene kategorije

OpenAPI specifikaciju moguće je napraviti ručno koristeći OpenAPI sintaksu ili je moguće koristiti biblioteke koje automatski generiraju specifikaciju. Nest.js razvojni okvir tako ima poseban modul koji prepoznaje krajnje točke implementiranog REST servisa te automatski generira OpenAPI specifikaciju za iste. Sada kada imamo specifikaciju napravljenog servisa, drugi programeri će lako znati kako koristiti naš servis.

4. GraphQL

GraphQL je skraćenica od **Graph Q**uery **L**anguage, ali za razliku od drugih upitnih jezika kao što je SQL, to nije jezik za izravnu komunikaciju s bazom podataka, već jezik koji definira pravila i oblik komunikacije klijenta s API poslužiteljem. GraphQL specifikacija je otvoreni standard koji opisuje pravila i karakteristike jezika te pruža upute za izvršavanje GraphQL upita. S obzirom da je GraphQL definiran otvorenim standardom, ne postoji službena implementacija GraphQL-a već se ona može napisati bilo kojim programskim jezikom, integrirati s bilo kojom bazom podataka te podržati bilo koji klijent (kao što su mobilne i web aplikacije) sve dok slijedi pravila navedena u specifikaciji. [1]

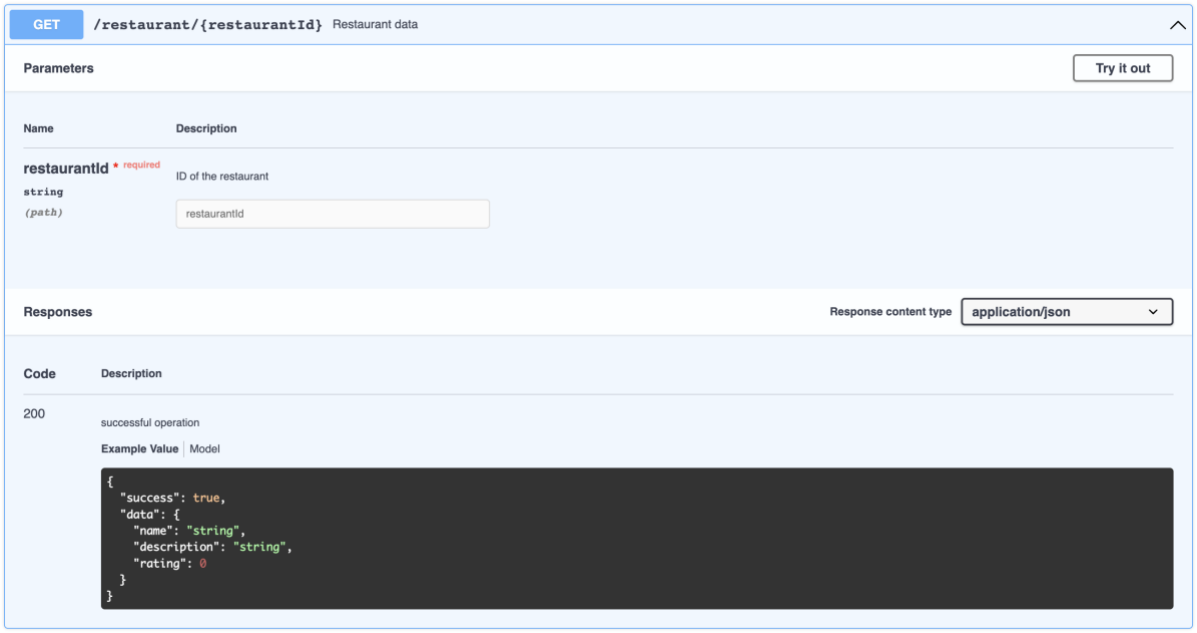
Priča o nastanku ideje GraphQL upitnog jezika započinje 2012. godine kada se Facebook suočio sa problemima oko efikasnosti rada njihove aplikacije na mobilnim uređajima. Naime, implementacija Facebookovog tzv. zida s objavama (eng. *news feed*) bila je dosta kompleksna jer su objave bile međusobno povezane, isprepletene i rekurzivne. Postojeća implementacija s perspektive potrošnje mrežnih resursa nije bila dovoljno efikasna, pa samim time i brzina nije bila najbolja. Na aplikaciji za mobilne uređaje nije bilo potrebno prikazivati količinu podataka neke objave jednako kao na verziji za stolna računala, no ona je svejedno dohvaćala sve podatke što je bilo nepotrebno trošenje mrežnih resursa. Glavni problem tome je bio što su programeri mobilne aplikacije bili limitirani mogućnostima REST servisa, odnosno nisu bez velikih promjena servisa mogli dohvaćati samo one podatke koji su aplikaciji potrebni. [16]

U kolovozu iste godine objavljena je prva verzija Facebook aplikacije koja je koristila novu GraphQL tehnologiju koja je programerima omogućila da smanje potrošnju mrežnih resursa korištenjem njezine mogućnosti dohvaćanja podataka. Tijekom sljedećih godinu i pol GraphQL tehnologija se nastavila razvijati kako bi 2015. prvi puta njena specifikacija bila javno objavljena zajedno s referentnom implementacijom u JavaScriptu. Postoji nekoliko ključnih karakteristika GraphQL dizajna. GraphQL upiti su deklarativni i hijerarhijski, a shema je strogo tipizirana i introspektivna, o čemu će biti više riječi u nastavku. [16]

4.1. Upiti i mutacije

U prošlom smo poglavlju definirali jedan jednostavan REST servis koji se sastoji od 4 krajnjih točaka (eng. *endpoint*) i spomenuli kako je korištenje REST servisa zapravo slanje zahtjeva na njegove krajnje točke putem HTTP protokola. GraphQL također koristi HTTP protokol za razmjenu podataka te predstavlja strukturiran način slanja zahtjeva koji ima dvije osnovne vrste operacija - upite i mutacije.

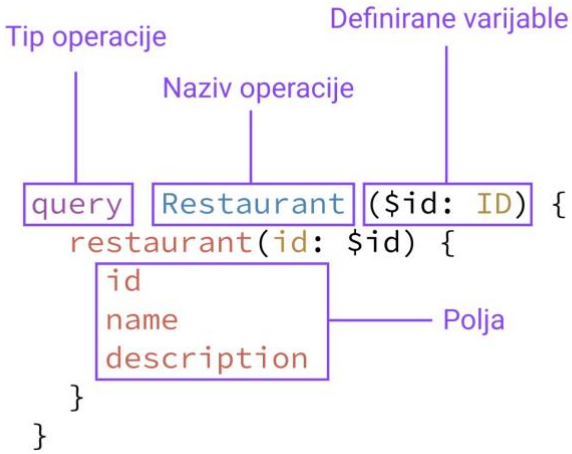
Kod korištenja REST servisa možemo istaknuti dva ključna pojma - akciju koju želimo provesti (koja je opisana HTTP metodom zahtjeva koji šaljemo) te entitet kojem želimo pristupiti (što je određeno putanjom, odnosno krajnjom točkom na koju šaljemo zahtjev). Tako npr. slanjem zahtjeva na krajnju točku `/restaurant/{restaurantId}` (slika 3) pristupamo entitetu restorana. S druge strane, slanjem zahtjeva na krajnju točku `/category/{categoryId}/items` pristupamo entitetima proizvodima određene kategorije.



Slika 3: OpenAPI specifikacija krajnje točke `/restaurant/{restaurantId}`

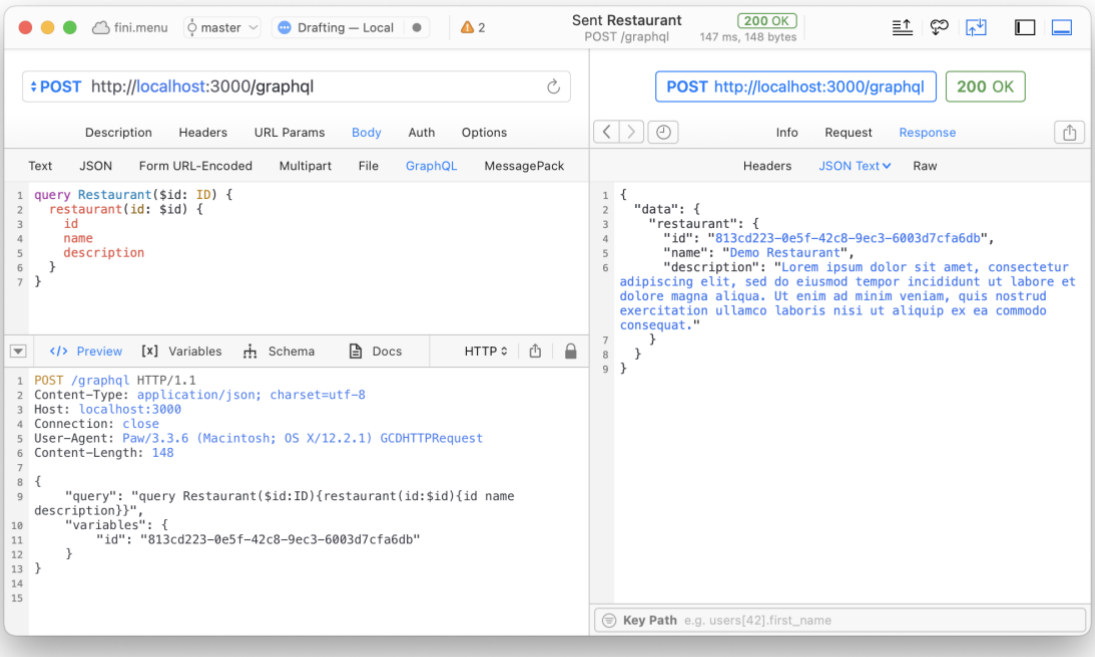
GraphQL upitni jezik funkcionira na način da postoji samo jedna krajnja točka na koju se preko POST metode u tijelu HTTP zahtjeva šalje GraphQL operacija koju želimo izvršiti. Operacija koja služi za dohvaćanje podataka naziva se upit (eng. *query*) te je sinonim GET zahtjevima kod REST servisa. Mutacije (eng. *mutation*) su operacije koje signaliziraju da želimo provesti neku promjenu u sustavu, slično kao POST ili DELETE metode kod REST servisa. [17]

Uzmimo za primjer da želimo napraviti ekvivalentan upit zahtjevu iz REST servisa iz poglavlja 3 za dohvaćanje podataka o restoranu (slika 3). Upit će izgledati kao na slici 4 koji se sastoji od nekoliko stavaka. Tip operacije može biti `query` ili `mutation`. U ovom slučaju to je `query`, odnosno upit, jer želimo samo dohvatiti podatke (ne želimo ih zapisati ili ažurirati). Naziv operacije je proizvoljan te se obično kod upita definira isti kao i entitet koji dohvaćamo. Potom se opcionalno unutar zagrada definiraju varijable upita koje se mogu proslijediti što je slično kao parametri kod REST servisa. [18] U idućem redu stavljamo naziv upita kojeg želimo izvršiti koji mora biti implementiran na API-ju te potom definiramo polja (eng. *fields*) koja želimo da nam API vrati u odgovoru (slika 4).



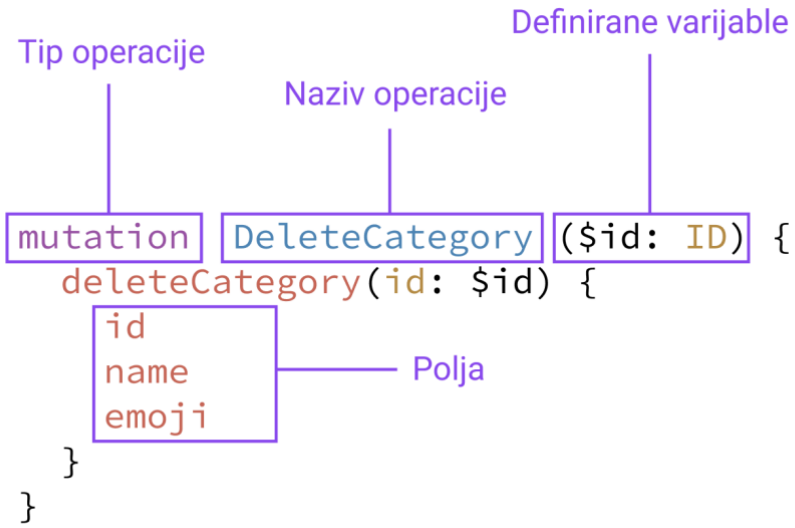
Slika 4: Primjer strukture GraphQL upita za dohvaćanje podataka o restoranu

Za slanje zahtjeva na GraphQL poslužitelj koristit ću aplikaciju Paw (slika 5). Na slici možemo vidjeti da se šalje HTTP POST zahtjev čije tijelo sadrži gore prikazan GraphQL upit. Paw alat ima mogućnost prikaza sintakse GraphQL jezika, no ako pogledamo donji dio zaslona primjećujemo da se zapravo radi o običnom POST zahtjevu koji kao tijelo sadrži upit koji smo sastavili. S desne strane zaslona nalazi se odgovor poslužitelja. Možemo primijetiti da je poslužitelj vratio ona polja koja smo stavili u upit, dakle to su `id`, `name` i `description`. Upravo to je jedna od najvećih prednosti GraphQL upitnog jezika - na klijentu možemo definirati strukturu odgovora koji želimo očekivati od poslužitelja.



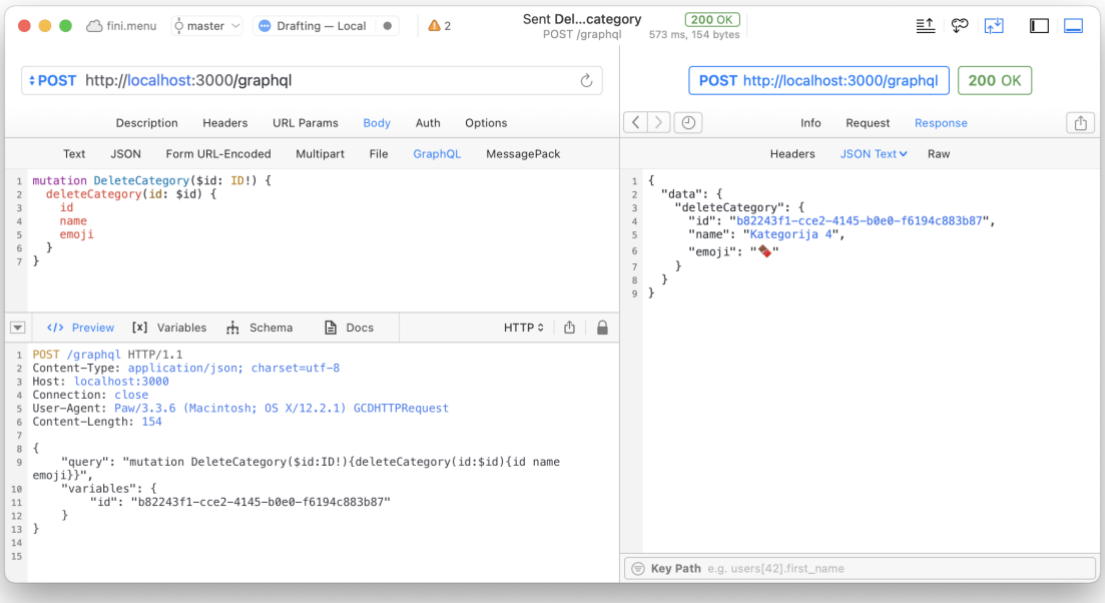
Slika 5: Izvršen upit sa slike 4 na GraphQL poslužitelju

Mutacije slijede istu sintaksu kao i upiti, samo se kao tip operacije koristi ključna riječ `mutation`. Naziv operacije je također proizvoljan, no obično se koristi glagol kojim se opisuje akcija koja će se provesti. [19] Na slici 6 prikazana je mutacija koja je ekvivalentna krajnjoj točki iz poglavlja 3 za brisanje kategorije (slika 2).



Slika 6: Primjer strukture GraphQL mutacije za brisanje određene kategorije

Ako izvršimo ovu mutaciju u Paw alatu, opet vidimo da se na poslužitelj šalje mutacija kao tijelo POST HTTP zahtjeva te kao odgovor od poslužitelja (desni dio zaslona) dobivamo točno ona polja koja smo naveli u mutaciji (slika 7).



Slika 7: Izvršena mutacija sa slike 6 na GraphQL poslužitelju

Sada smo upoznati s time kako se šalju upiti i mutacije na GraphQL poslužitelj te kako izgleda njihova sintaksa, no jedno ključno pitanje još nije odgovoreno - na koji su način definirana polja entiteta koja u upitu možemo zatražiti. Odgovor na to pitanje pojasnit ćemo u idućem poglavlju.

4.2. Tipovi i shema

GraphQL poslužitelj koristi shemu za opisivanje oblika dostupnih podataka koje može vraćati klijentu. Shema definira hijerarhiju tipova s poljima koja se popunjavaju na strani poslužitelja - npr. iz baze podataka. Osim što shema definira tipove podataka, ona također definira i koji su upiti i mutacije dostupni klijentima za izvršavanje.

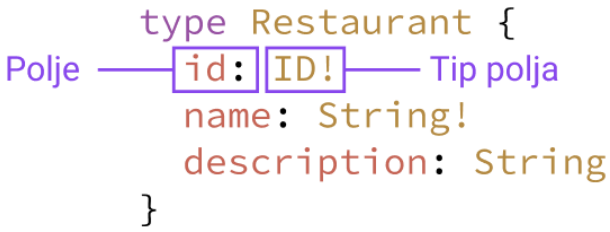
Kao što smo vidjeli iz prijašnjih upita, glavna značajka GraphQL upitnog jezika je mogućnost odabira polja nad objektima koja želimo da nam poslužitelj vrati. Pogledajmo sljedeći primjer upita na slici 8. Na lijevoj strani nalazi se dio upita, dok se na desnoj strani nalazi odgovor poslužitelja na postavljen upit.

Slika 8: Primjer upita i pripadajućeg odgovora s GraphQL poslužitelja

Pravilo pri oblikovanju upita je da uvijek počinjemo sa "korijenskim" objektom. Potom odabiremo polje koje želimo dohvatiti (u ovom slučaju `restaurant`) te za odabrani restaurant objekt odabiremo koja polja tog objekta želimo dohvatiti - `id`, `name` i `description`. Budući da se oblik upita podudara s rezultatom, odnosno odgovorom poslužitelja, možemo predvidjeti što će upit vratiti bez da znamo mnogo o samoj implementaciji poslužitelja. No, korisno je imati točan opis podataka koje možemo dohvatiti kako bismo znali koja polja možemo odabrati, koje vrste objekata možemo dobiti kao odgovor itd. [20] Ovdje u igru dolazi shema koju ćemo detaljnije pojasniti malo kasnije. Svaki GraphQL servis definira skup tipova koji u potpunosti opisuju skup mogućih podataka koji se mogu dohvatiti na tom GraphQL poslužitelju. Zatim, kada upit pristigne na poslužitelj, tipovi se provjeravaju te izvršavaju prema toj shemi.

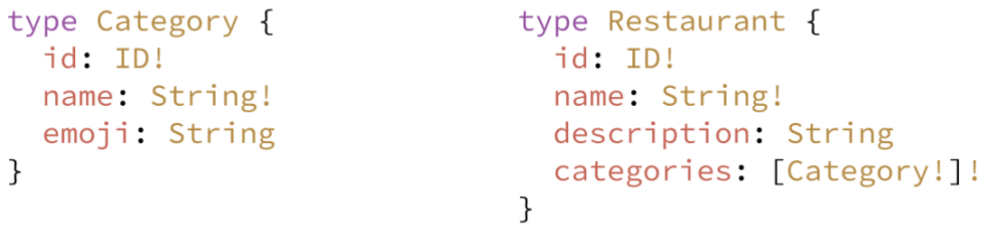
4.2.1. Tipovi objekata i polja

Najosnovnije komponente GraphQL sheme su tipovi objekata koji predstavljaju vrstu objekta koji se može dohvatiti i polja koja taj objekt sadrži [21]. U GraphQL jeziku opis sheme za tipove objekta (u ovom slučaju tip `Restaurant`) možemo predstaviti kao na slici 9:



Slika 9: Definiran tip Restaurant u shemi GraphQL poslužitelja

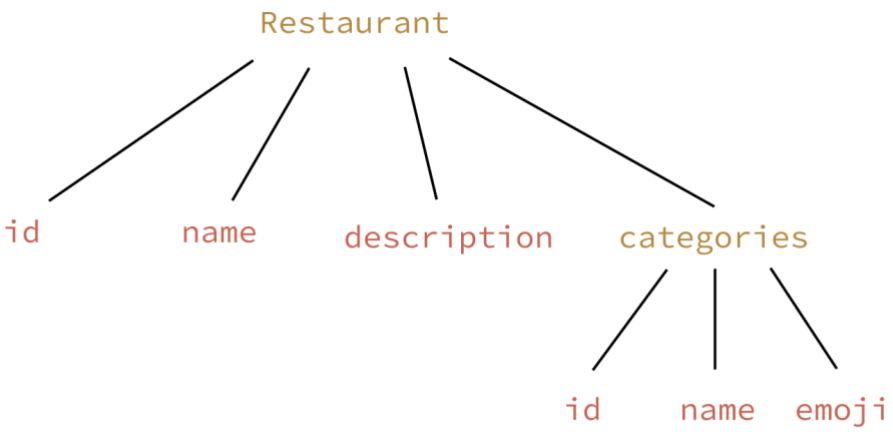
Svaki objekt sastoji se od polja te je svakom polju pridružen tip. `ID` je sličan tipu `String` no samo označava da je to jedinstven identifikator tog objekta. Oznaka "`!`" označava da to polje mora poprimiti vrijednost te da nikada neće biti `null`. U ovom primjeru tipovi `ID` i `String` nazivaju se skalarni tipovi, odnosno to su tipovi koji su ugrađeni u GraphQL jezik. Osim skalarnih tipova, možemo definiranom tipu dodijeliti polje koje će biti nekog drugog tipa koji smo definirali. Na slici 10 definiran je tip `Category` (lijevo) te je ažuriran tip `Restaurant` (desno) tako da sada sadrži polje `categories` čiji tip je niz (eng. *array*) tipova `Category`.



Slika 10: Definirani tipovi `Category` i `Restaurant`

Kao što smo prije imali primjer, `Category!` tip označava da objekt `Category` ne može biti `null`. `[Category!]` znači da će svaki `Category` objekt u tom nizu (eng. *array*) poprimiti neku vrijednost koja nikada neće biti `null`. `[Category!]!` pak znači da samo polje `categories` nikada neće biti `null` već će uvijek biti niz kategorija koji može biti prazan, no nikada neće sadržavati `null` vrijednosti.

Nakon toliko priče o GraphQL jeziku, ovdje se konačno prvi puta dotičemo sa - grafovima. Naime, ako pogledamo zadnju verziju tipa `Restaurant` koji smo definirali (slika 10), mogli bismo vizualno prikazati njegov reprezentirajući graf (slika 11). Polja u shemi koja vraćaju skalarne tipove su listovi grafa (crveno obojani). [20]



Slika 11: Vizualna reprezentacija tipa `Restaurant` sa slike 10

4.2.2. Skalarni tipovi

Vidjeli smo da se GraphQL objektni tip sastoji od naziva i polja koja se mogu granati, no u nekom trenutku ta polja moraju vratiti neke konkretne podatke. Tu dolaze skalarni tipovi koji predstavljaju listove upita, odnosno grafa. Već smo se susreli s nekim tipovima u dosadašnjim primjerima, a ovo su svi skalarni tipovi definirani GraphQL jezikom [21]:

- `Int` - 32 bitni integer
- `Float` - decimalna vrijednost dvostruke preciznosti
- `String` - niz UTF-8 znakova
- `Boolean` - `true` ili `false`
- `ID` - predstavlja jedinstven identifikator objekta koji je zapravo isto kao i `String` - niz UTF-8 znakova. Obično se koristi na front endu kao ključ kod keširanja podataka

4.2.3. Enumeracije

Enumeracije se u GraphQL jeziku nazivaju `Enums` te su to posebna vrsta skalara koji su ograničeni na točno određeni skup vrijednosti. [21] Slijedi primjer enumeracije:

```
enum RestaurantType {
  FAST_FOOD
  BUFFET
  FINE_DINING
  CAFE
}
```

Slika 12: Primjer definirane enumeracije

Sada nekom polju kao tip možemo pridružiti ovu enumeraciju te možemo očekivati da će to polje vraćati jedno od sljedećih vrijednosti `FAST_FOOD`, `BUFFET`, `FINE_DINING`, `CAFE`.

4.2.4. Input tipovi

Do sada smo govorili o prosljeđivanju skalarnih vrijednosti kao što su `String` ili `Int` kao argumente upita i mutacija. No, isto tako možemo proslijediti i kompleksnije objekte što je posebno korisno kod mutacija gdje bismo npr. kod mutacije za kreiranje nekog entiteta željeli proslijediti cijeli objekt koji želimo kreirati. U GraphQL jeziku se takvi tipovi nazivaju input tipovi te se oni definiraju identično kao i obični tipovi, samo s ključnom riječi `input` umjesto `type`. [21] Uzmimo za primjer sljedeću mutaciju:

```
mutation CreateItem($name: String!, $description: String, $price: Float!) {
  createItem(name: $name, description: $description, price: $price) {
    id
    name
    description
    price
  }
}
```

Slika 13: Primjer mutacije za kreiranje novog proizvoda

Ova mutacija za kreiranje novog proizvoda ima više argumenata koji bi se zajedno mogli pretvoriti u `input` tip kako bi mutacija izgledala preglednije - slika 14.

```
input ItemInput {
  name: String!
  description: String
  price: Float!
}
```

Slika 14: Definirani `input` tip za argumente mutacije `createItem`

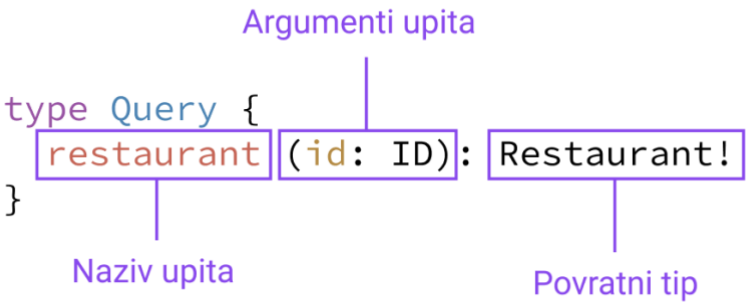
Ako sada izmijenimo mutaciju da koristi `input` tip koji smo kreirali, ona će izgledati dosta preglednije (slika 15). Korištenje input tipova dobra je praksa te je posebno korisno kod mutacija s većim brojem argumenata.

```
mutation CreateItem($item: ItemInput!) {
  createItem(item: $item) {
    id
    name
    description
    price
  }
}
```

Slika 15: Mutacija sa slike 13 izmijenjena tako da sada prima input tip `ItemInput`

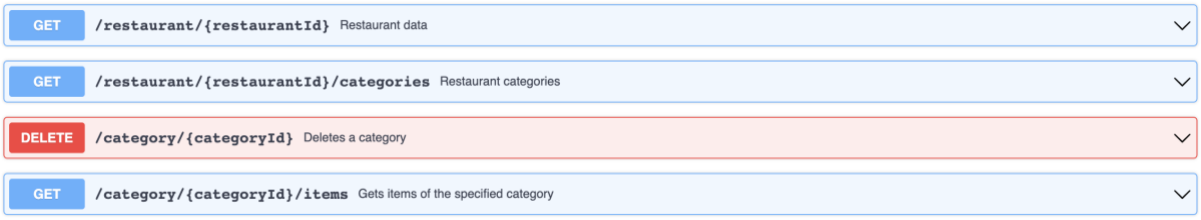
4.2.5. Shema

GraphQL specifikacija definira tzv. jezik za definiranje sheme (eng. *schema definition language*) ili SDL kojim se definira shema GraphQL poslužitelja koja je zapravo datoteka koja sadrži sve tipove i definirane upite i mutacije GraphQL poslužitelja. [21] Da bi GraphQL poslužitelj prepoznao upit `restaurant` koji dohvaća podatke o restoranu, koji smo ranije izvršili (slika 5), potrebno je definirati tip tog upita u shema datoteci. U shemi se može definirati jedan tip `Query` u kojem se definiraju svi dostupni upiti koje će klijent moći izvršiti:



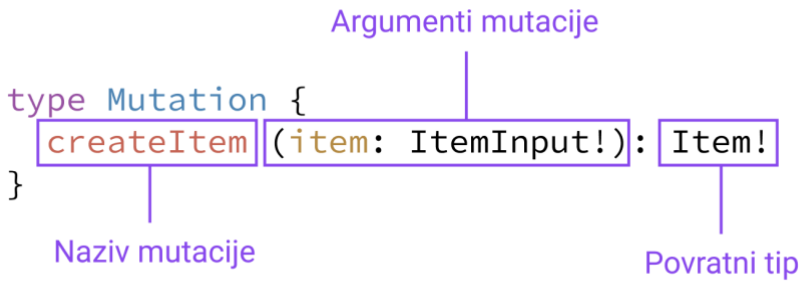
Slika 16: Definiran upit za dohvaćanje podataka o restoranu u shema datoteci

Prilikom definiranja tipa upita moramo definirati naziv upita, opcionalno njegove argumente te obavezno povratni tip. Argumente upita možemo poistovjetiti sa parametrima kod REST servisa - `restaurantId` i `categoryId` na primjeru sa slike 17. Svaki argument upita mora imati definiran svoj tip te sam upit mora imati definiran povratni tip koji označava tip podataka koji će vratiti klijentu.



Slika 17: OpenAPI specifikacija svih ranije definiranih krajnjih točaka

Slično kao i kod upita, shema može imati jedan tip `Mutation` u kojem se definiraju sve raspoložive mutacije. One se definiraju na isti način kao i upiti:

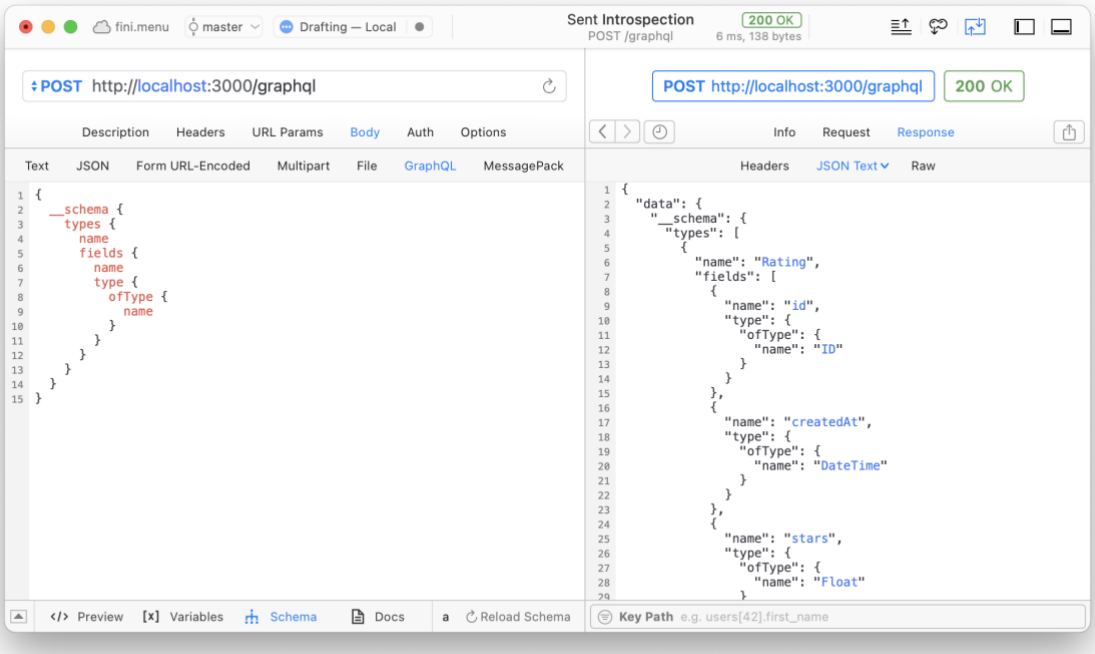


Slika 18: Definirana mutacija za kreiranje nove stavke

4.3. Dokumentiranje GraphQL servisa

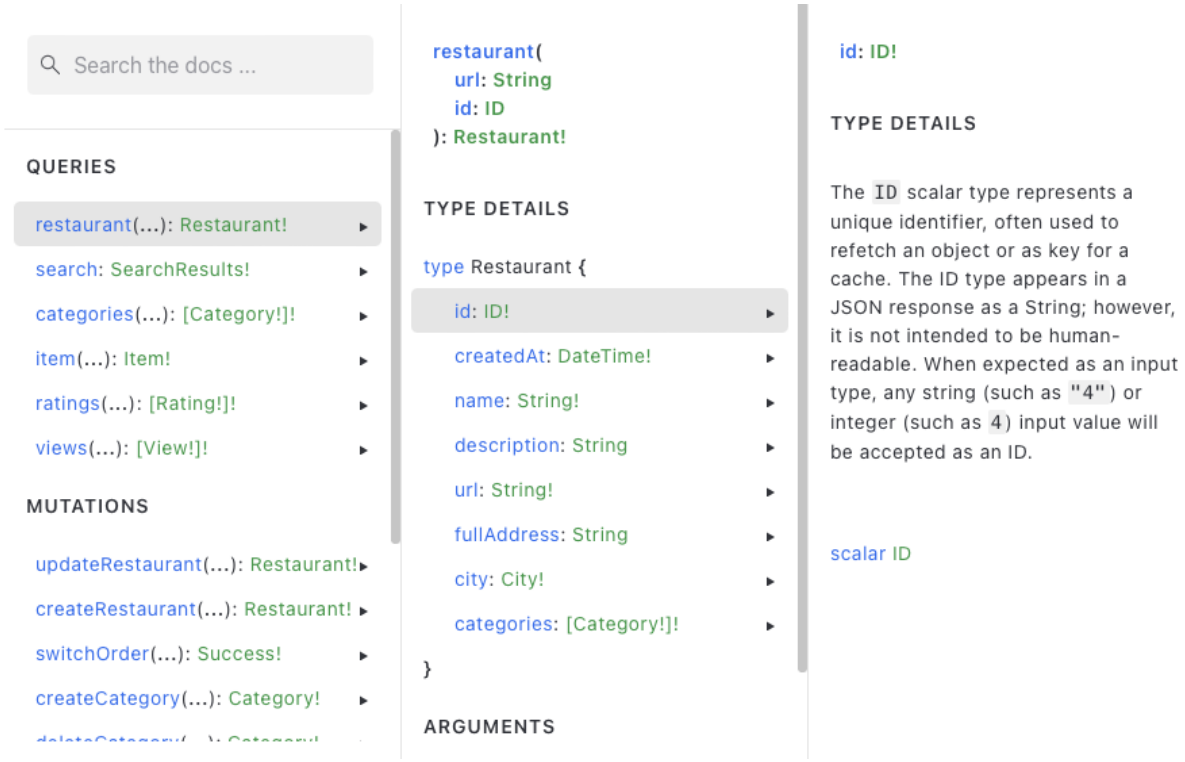
Spomenuli smo da se za dokumentaciju REST servisa koristi OpenAPI specifikacija. Postoje alati i razvojni okviri koji automatski vode dokumentaciju REST servisa ažurnom, no to nije uvijek slučaj pa je ponekad potrebno ručno izrađivati dokumentaciju pisanjem raznih notacija u kodu.

Sama SDL shema može se smatrati dokumentacijom GraphQL servisa pošto su tamo definirani upiti i mutacije koje se mogu provesti te tipovi podataka koje možemo očekivati kao odgovor. Dakle posao dokumentiranja servisa već je odrađen kod same implementacije te jedino što preostaje je kako prikazati shemu u nekom GraphQL klijentu (npr. Paw). Ovdje u igru dolazi introspekcija GraphQL servisa što predstavlja mogućnost postavljanja upita poslužitelju koji su resursi dostupni u trenutnoj shemi (upiti, mutacije, tipovi). [22] Na slici 19 prikazan je upit introspekcije GraphQL poslužitelju za sve definirane tipove.



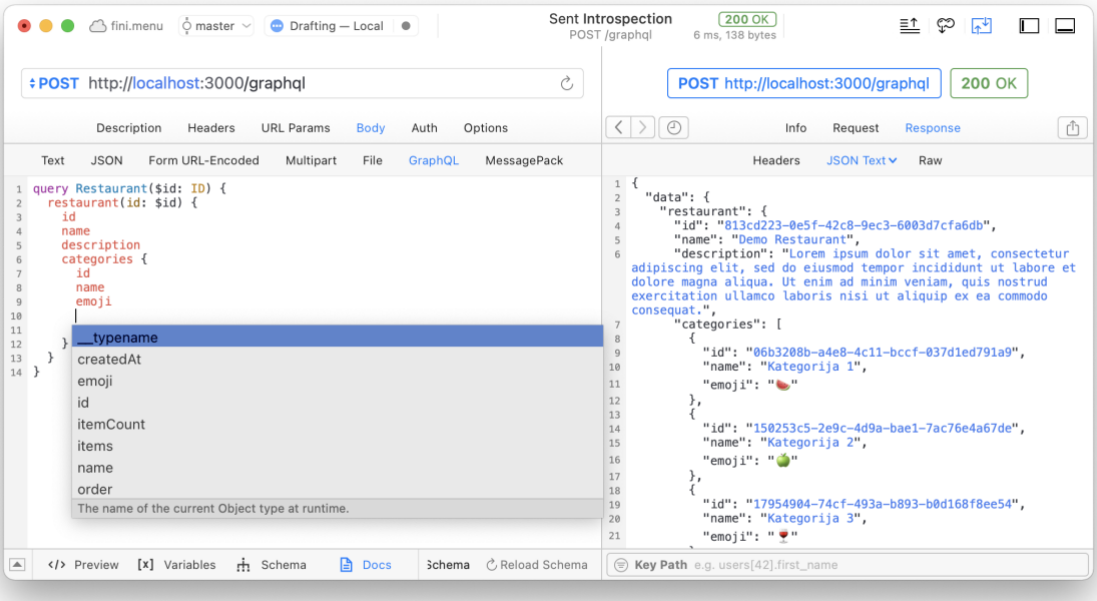
Slika 19: Prikaz upita introspekcije za dohvaćanje svih dostupnih tipova na poslužitelju

Kao odgovor (desna strana) poslužitelj je vratio sve definirane tipove iz sheme. Na isti način možemo dohvatiti sve definirane upite i mutacije. Paw (a i drugi klijenti za testiranje GraphQL servisa) ima opciju prikaza dokumentacije GraphQL servisa (slika 20) koju je izgradio na temelju introspekcije poslužitelja.



Slika 20: Dokumentacija GraphQL poslužitelja u alatu Paw

Osim pregleda dokumentacije, alati kao Paw obično prilikom pisanja upita imaju mogućnost nadopunjavanja upita s mogućim poljima prema dokumentaciji dobivenoj introspekcijom (slika 21) što testiranje i korištenje servisa čini još jednostavnijim.



Slika 21: Mogućnost nadopunjavanja upita u alatu Paw

Tako su programeru na pregledan način vidljivi svi mogući upiti, mutacije i tipovi na GraphQL servisu. Također pozitivna stvar je što je ova dokumentacija uvijek ažurna jer ako promijenimo shemu na poslužitelju, ova dokumentacija se preuzima direktno s poslužitelja. U SDL shemi je također moguće dodavanje komentara za pojedini upit, mutaciju, tip ili pak polje s kojima se još pobliže servis može dokumentirati. [23]

4.4. Načini implementacije

Nakon pojašnjenja tipova i sheme u GraphQL jeziku dolazimo do samog načina implementacije koji se, nakon definirane sheme, zapravo može svesti na pitanje kako odrediti koji podaci će se vraćati za koji tip podataka te za koji upit ili mutaciju koju smo definirali. Implementacija se tako može podijeliti na dva dijela: definiranje sheme poslužitelja te definiranje tzv. *resolver* funkcija koje govore poslužitelju kako dohvatiti podatke povezane s određenim tipom definiranim u shemi. U posljednje dvije godine povećao se broj alata za razvoj GraphQL API-ja te bi se mogla izdvojiti dva pristupa u načinu implementacije istog - *schema first* i *code first*. [24]

4.4.1. Schema first

Kada je GraphQL 2015. pušten u javnost, ekosustav alata za razvoj bio je oskudan. Postojala je samo službena specifikacija i njena referentna implementacija u JavaScriptu - `graphql-js`[1]. Sve do danas, najpopularnije biblioteke za implementaciju GraphQL poslužitelja koriste istu `graphql-js` biblioteku. Koristeći tu biblioteku, GraphQL shema se definira kao JavaScript objekt gdje se definicija tipa te implementacija resolver funkcije objedinjuje:

[1] https://github.com/graphql/graphql-js

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

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      hello: {
        type: GraphQLString,
        args: {
          name: { type: GraphQLString },
        },
        resolve: (_, args) => `Hello ${args.name || 'World!'}`,
      },
    },
  }),
})
```

Slika 22: Primjer definiranja upita `hello` koristeći `graphql-js` biblioteku [25]

Kao što se može vidjeti iz primjera, API za kreiranje GraphQL shema vrlo je opsežan. SDL prikaz sheme puno je sažetiji i lakši za shvatiti:

```
type Query {
  hello(name: String): String
}
```

Slika 23: Upit sa slike 22 definiran koristeći SDL [25]

Kako bi olakšao razvoj GraphQL poslužitelja, Apollo je krenuo sa razvojem `graphql-tools` biblioteke u ožujku 2016. godine. Cilj je bio odvojiti definiciju sheme od stvarne implementacije logike, odnosno *resolver* funkcija. Upravo to je dovelo do trenutno popularnog schema first pristupa razvoja koji se provodi u dva koraka:

1. Ručno pisanje GraphQL shema definicije koristeći SDL
2. Implementacija pripadajućih resolver funkcija

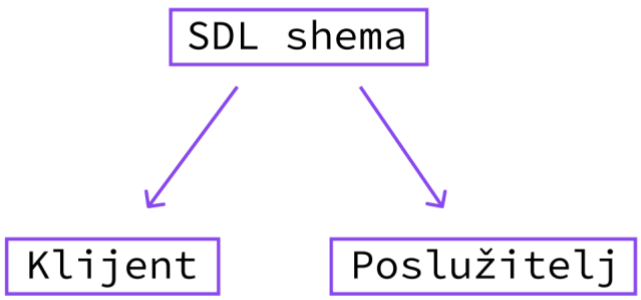
Schema first pristup označava pristup kojim prvo definiramo shema datoteku za GraphQL servis, a potom implementiramo kod koji će povezivati definicije iz te sheme. Za definiranje sheme koristi se Schema Definition Language (SDL) pa se ovaj pristup može nazvati i SDL

first. [25] Na slici 24 prikazan je primjer implementacije GraphQL poslužitelja tim pristupom. Ovaj kod ekvivalentan je prijašnjem primjeru (slika 22).

```
const { makeExecutableSchema } = require('graphql-tools')

const typeDefs = `
type Query {
  hello(name: String): String
}
`

const resolvers = {
  Query: {
    hello: (_, args) => `Hello ${args.name || 'World!'}`,
  },
}

const schema = makeExecutableSchema({
  typeDefs,
  resolvers,
})
```

Slika 24: Implementacija upita `hello` SDL first pristupom [25]

Ovim pristupom prvo se definira shema (`typeDefs`) koja se zatim učita funkcijom `makeExecutableSchema`. Shema se isto tako može definirati u datoteci koja se potom na isti način učita. Istu shema datoteku koju učita poslužitelj šalje se također klijentu (npr. Paw) koji potom može vidjeti dokumentaciju poslužitelja (slika 25).



Slika 25: Prikaz korištenja sheme u shema first pristupu

Osim preglednosti i bolje čitljivosti koda, ovakav pristup ima niz drugih prednosti. Naime, ovakav pristup je jednostavan za razumijevanje te je prikladan za brzu implementaciju GraphQL poslužitelja. Sama definicija sheme tako može poslužiti kao API dokumentacija (slično kao OpenAPI specifikacija kod REST servisa) te samim time kao komunikacijski alat između front end i back end timova. [25]

4.4.2. Code first

Iako SDL first pristup ima mnoge prednosti, posljednje dvije godine pokazale su da ga je izazovno primijeniti na većim projektima. Postoji niz problema koji se javljaju u kompleksnijim projektima o kojima će biti više riječi kasnije. Svi problemi su sami po sebi uglavnom rješivi, no stvarni problem je što njihovo rješavanje zahtijeva korištenje, a i učenje, mnogih dodatnih alata. Posljednjih dviju godina je tako objavljeno mnogo alata koji pokušavaju poboljšati tijek razvoja GraphQL poslužitelja SDL first pristupom no izuzetni troškovi u učenju, upravljanju i integraciji svih tih alata usporavaju programere u razvoju. [24]

Jedan od glavnih problema SDL first pristupa jest nekonzistentnost između shema definicije i implementacije resolver funkcija. Sa SDL first pristupom definicija sheme mora odgovarati točnoj strukturi implementacije resolver funkcija što znači da programer mora osigurati njihovu međusobnu konzistentnost. Iako je to već izazov i za male sheme, to postaje praktički nemoguće kako sheme rastu na stotine ili tisuće redaka (za referencu, shema GitHub GraphQL servisa ima više od 10 000 redaka). [25]

Drugi problem jest modularnost sheme. Kada pišemo velike sheme najmanje što želimo je da se definicije svih tipova nalaze u istoj datoteci. Umjesto toga, želimo ih podijeliti na manje dijelove što bi osiguralo bolju preglednost i lakše održavanje. Treći problem se odnosi na IDE podršku i općenito iskustvo u razvoju. Kako se GraphQL shema temelji na sustavu tipova koji može biti ogromna prednost tijekom razvoja jer omogućuje statičku analizu koda (kao npr. TypeScript), to nažalost nije slučaj jer je SDL definiran kao običan tekst, odnosno string, pa IDE obično neće biti u mogućnosti raditi statičku provjeru tipova. [25] Pitanje se postavlja kako iskoristiti GraphQL tipove u IDE-ovima kako bismo imali koristi od značajki kao što su *autocomplete* i provjera pogrešaka prilikom kompiliranja SDL koda.

Za svaki od navedenih problema postoje alati koji nastoje zaobići te probleme i učiniti iskustvo u razvoju GraphQL poslužitelja što bolje. Kako se radi ne samo o jednom problemu nego više njih, to zahtijeva korištenje i učenje više novih alata. Većina SDL first problema proizlazi iz činjenice da ručno moramo mapirati SDL shemu u programski jezik. Upravo to mapiranje razlog je zbog kojeg su nam potrebni razni alati. Umjesto povećanja složenosti razvoja GraphQL poslužitelja s ogromnom količinom alata, trebali bismo težiti jednostavnijem

pristupu gdje bismo u idealnom slučaju GraphQL shemu definirali pomoću programskog jezika kojeg koristimo (npr. JavaScript) - upravo to je ideja code first pristupa.

Ranije smo u prošlom poglavlju prikazali definiranje sheme pomoću `graphql-js` biblioteke. To je suština onoga što znači code first pristup. Kod ovog pristupa ne postoji ručno održavanje sheme nego se umjesto toga SDL generira iz koda koji implementira shemu. Slijedi primjer code first pristupa pri definiranju sheme koristeći `nexus-js` biblioteku, a u praktičnom dijelu rada vidjet ćemo isti princip koristeći Nest.js razvojni okvir.

```
const Query = objectType('Query', t => {
  t.string('hello', {
    args: { name: stringArg() },
    resolve: (_, { name }) => `Hello ${name || `World`}!`,
  })
})

const schema = makeSchema({
  types: [Query],
  outputs: {
    schema: './schema.graphql'),
    typegen: './typegen.ts',
  },
})
```

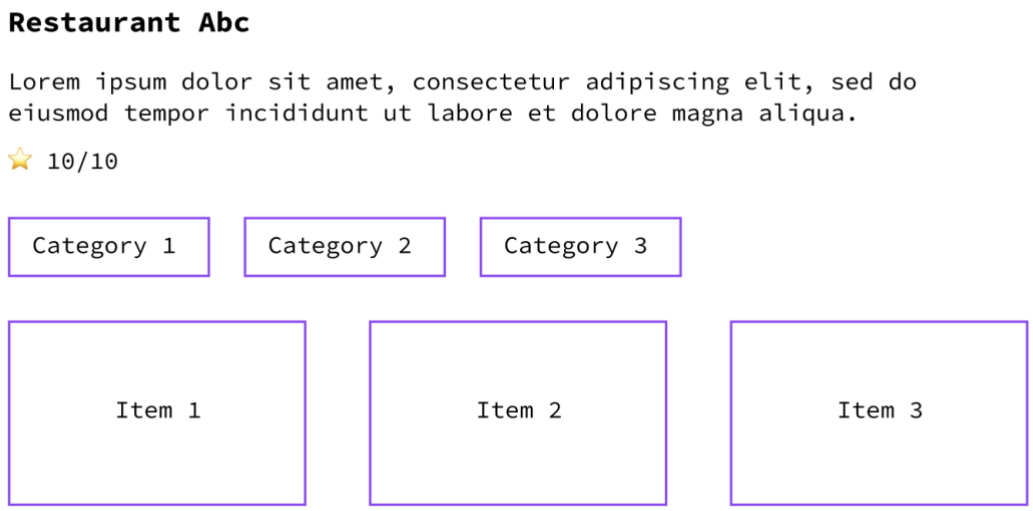
Slika 26: Implementacija upita `hello` code first pristupom [25]

S ovim pristupom GraphQL tipove definiramo direktno koristeći JavaScript/TypeScript te ćemo tako imati sve prednosti IDE-a kojeg koristimo kao što su *autocomplete* i statička provjera tipova. Tipovi koji se na taj način definiraju se proslijede u funkciju (`makeSchema`) koja iz koda generira SDL shemu (slika 27).

Slika 27: Proces generiranja SDL sheme u code first pristupu

4.5. Usporedba s REST servisima

Kod REST servisa podatke dohvaćamo putem različitih krajnjih točaka. Npr. da bismo dohvatili podatke o restoranu koristit ćemo krajnju točku `/restaurant/{id}`, da bismo dohvatili kategorije nekog restorana koristit ćemo krajnju točku `/restaurant/{id}/categories`, a da bismo dohvatili stavke iz neke kategorije koristit ćemo `/restaurant/{id}/categories/{categoryId}`. Uzmimo za primjer stranicu aplikacije gdje se prikazuju stavke neke kategorije (slika 28).

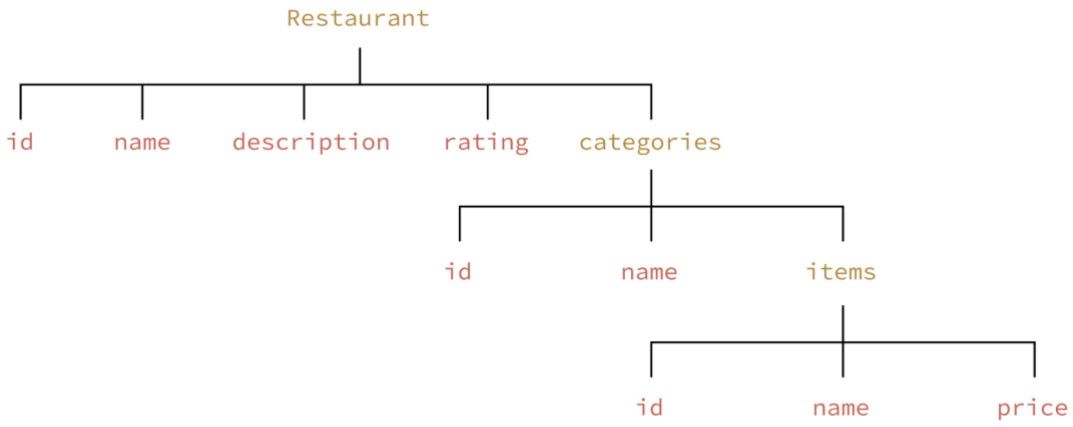


Slika 28: Primjer stranice aplikacije gdje se prikazuju stavke neke kategorije

Ovdje moramo prikazati podatke s 3 različite krajnje točke što bi značilo da klijent mora poslati 3 zasebna zahtjeva. Naravno, u mnogo slučaja se neće desiti da ćemo za ove podatke morati pristupati tri različite krajnje točke već će sve potrebne podatke vratiti jedna, npr. `/restaurant/{id}`. No, mnogo puta će upravo ovo biti slučaj. Ako dođe do toga, GraphQL nam osigurava da ćemo sve potrebne podatke moći dohvatiti jednim zahtjevom zato što GraphQL zahtijeva drukčiji pogled na razvoj API servisa - ne pogled u smislu krajnjih točaka (slika 17) već u obliku grafova (slika 11). Slijedi primjer tog GraphQL upita (slika 29) te njemu pripadni graf (slika 30).

```
query Restaurant($id: ID) {
  restaurant(id: $id) {
    id
    name
    description
    rating
    categories {
      id
      name
      items {
        id
        name
        price
      }
    }
  }
}
```

Slika 29: Primjer upita za dohvaćanje podataka o restoranu, njegovim kategorijama te stavaka unutar kategorija



Slika 30: Vizualna reprezentacija upita sa slike 29 u obliku grafa

Ovaj problem REST servisa naziva se *underfetching problem* što znači da od jedne krajnje točke (eng. *endpoint*) ne dobivamo sve potrebne podatke pa moramo napraviti dodatne zahtjeve na druge krajnje točke kako bismo dobili sve potrebne podatke. [26] U tom pogledu GraphQL ima prednost što rezultira boljim performansama. Idući problem kod REST servisa jest tzv. *overfetching* što znači da krajnja točka vrati više podatka nego što je klijentu potrebno. U tom slučaju nepotrebno se koristi više mrežnih resursa (eng. *bandwidth*) nego što je potrebno. Uzmimo za primjer stranicu prikazanu na slici 31. Recimo da nam za svaku stavku REST servis vrati `naziv`, `cijenu`, `opis`, `kategorije`, `sliku`, `varijante`, `ocjene` itd. Pošto

su nama za tu stranicu potrebni samo `naziv` i `cijena`, klijent će primiti puno više podataka nego što mu je potrebno. GraphQL taj problem rješava tako da u upitu možemo definirati polja koja želimo primiti u odgovoru.

Slika 31: Primjer stranice aplikacije gdje se prikazuju stavke

Uobičajeni uzorak kod REST servisa jest strukturiranje krajnjih točaka prema pogledima unutar aplikacije. Taj pristup je praktičan jer omogućuje klijentu da dobije sve potrebne podatke za određeni pogled pa tako ne dolazi do *underfetching* problema. Glavni nedostatak ovog pristupa je što ne dopušta brze iteracije na front endu. Sa svakom promjenom koja se napravi na korisničkom sučelju postoji velik rizik da je sada potrebno više ili manje podataka nego prije jer to zahtijeva promjene na back endu. Zahvaljujući fleksibilnošću GraphQL jezika, ovaj je problem riješen na način da je omogućeno klijentu uz minimalno dodatnog rada napraviti promjene u pogledu dohvaćanja količine podataka budući da klijenti mogu specificirati točan oblik podataka koji žele primiti od poslužitelja. [26] Također jedna od prednosti GraphQL-a je lagano dokumentiranje što smo obuhvatili u ranijem poglavlju.

Kako sve ima svoje nedostatke, tako i GraphQL. Obično GraphQL zahtijeva nešto više vremena za naučiti s obzirom da je potrebno biti upoznati sa jezikom za definiranje sheme. Isto tako, u slučaju prethodnog korištenja REST principa, možda će trebati programeru jedno vrijeme da se navikne na GraphQL pošto je potrebno promijenit način razmišljanja o krajnjim točkama u način razmišljanja u grafovima. Koliko god GraphQL ima prednosti, nije primjeren za svaki slučaj korištenja. Kada radimo s entitetima, GraphQL se čini dosta prirodan s obzirom da relacije možemo prikazati pomoću grafova. S druge strane, obično je primjerenije za npr. autentifikaciju koristiti REST arhitekturu s obzirom da ne dobivamo nikakvu prednost koristeći GraphQL u tom slučaju. [17] Isto tako, ako će neke druge aplikacije koristiti naš servis za autentifikaciju, tada je najsigurnije ići u smjeru s REST servisom jer te druge aplikacije možda uopće neće koristiti GraphQL koji zahtijeva dodatne biblioteke i konfiguracije.

5. Računarstvo u oblaku

Računarstvo u oblaku (eng. *cloud computing*) se definira kao dostupnost resursa računalnih sustava, posebno pohrane podataka (pohrana u oblaku) i računalne snage, bez izravnog upravljanja od strane korisnika. Umjesto da organizacije moraju uložiti velika ulaganja u kupnju opreme, obuku osoblja i trajno održavanje, te potrebe rješava pružatelj usluga u oblaku [27]. Time se stvara jedna razina apstrakcije za korisnike usluga čime im se omogućuje fokus na njihov posao, npr. razvoj aplikacije koja će biti postavljena na cloud, a ne zamaranje detaljima oko mreže, skaliranja itd.

Većina tvrtki preusmjerava svoje poslovanje u oblak zbog brojnih prednosti koje im omogućavaju lako pohranjivanje podataka bez potrebe za fizičkim prostorom. Pristup podacima također je postao lakši jer se to može učiniti s bilo kojeg mjesta u bilo koje vreme putem interneta. Tri su glavna modela računarstva u oblaku: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) i Software as a Service (SaaS). [28]

5.1. Infrastructure as a Service (IaaS)

IaaS pruža organizaciji iste tehnologije i mogućnosti kao i tradicionalni podatkovni centar, uključujući potpunu kontrolu nad instancama poslužitelja. Sistemski administratori unutar tvrtke odgovorni su za upravljanje aspektima kao što su baze podataka, aplikacije, sigurnost itd., dok davatelj usluga u oblaku upravlja samim poslužiteljima, mrežom, hardverom itd. [29]

5.2. Platform as a Service (PaaS)

PaaS uklanja potrebe organizacijama da upravljaju temeljnom infrastrukturom (hardverom i operacijskim sustavima) poslužitelja te im omogućuju da se usredotoče na implementaciju i razvoj aplikacija. Time se tvrtkama pomaže da budu učinkovitije jer ne trebaju brinuti o nabavi računalnih resursa, planiranju kapaciteta, održavanju softvera, popravljanju itd. [29]

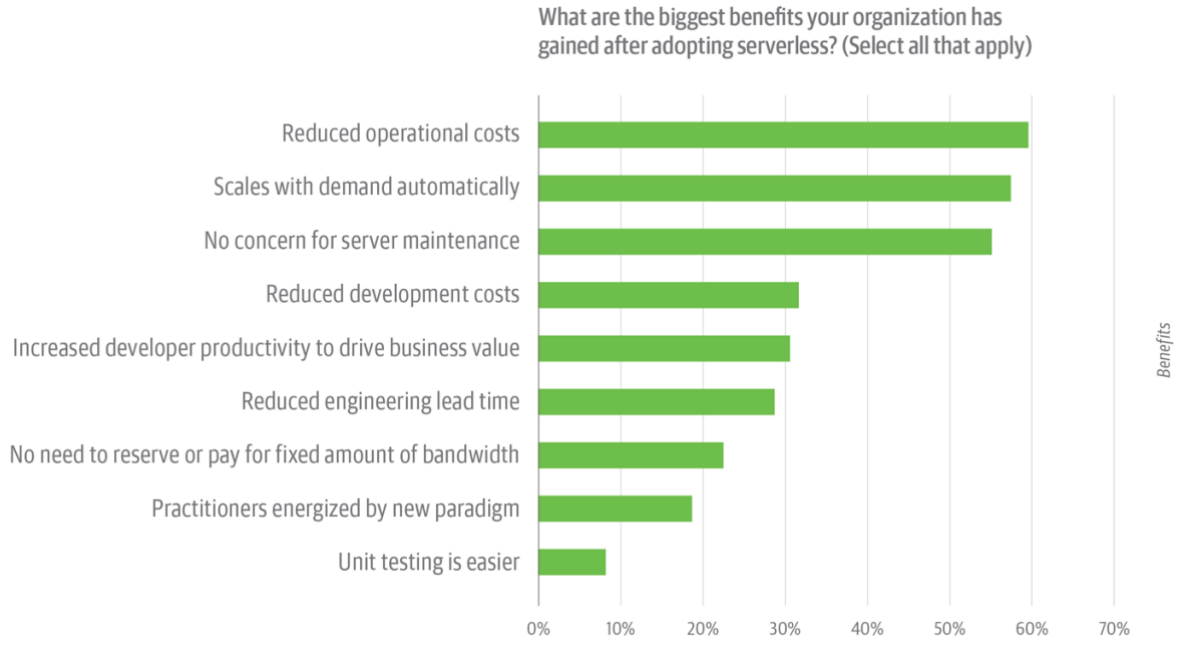
5.3. Software as a Service (SaaS)

SaaS pruža krajnjim korisnicima gotov proizvod koji pokreće i njime upravlja davatelj usluga. Ovim modelom krajnji korisnik ne mora razmišljati o instalaciji proizvoda, o tome kako se on održava i kako se upravlja temeljnom infrastrukturom na kojoj se pokreće - može bezbrižno koristiti taj određeni proizvod, odnosno softver. [29]

5.4. Arhitektura bez poslužitelja

Kako organizacije i njihovi tehnološki ekosustavi sazrijevaju, često smatraju da je upravljanje infrastrukturom velik izazov. Umjesto da postanu profesionalci u upravljanju infrastrukturom, softverski timovi radije će posvetiti svoje vrijeme i resurse razvoju aplikacija. Tako dolazi nova alternativa - arhitektura bez poslužitelja (eng. *serverless architecture*). [30]

Arhitektura bez poslužitelja je model računarstva u oblaku koji opisuje način na koji tvrtke mogu izgraditi i pokretati aplikacije, ali da pri tome ne moraju upravljati infrastrukturom. Naziv "serverless" predstavlja da nam zapravo za pokretanje aplikacije nije potrebna priprema i upravljanje poslužiteljem, već će se za to pobrinuti davatelj usluga. [30] Neke od prednosti arhitekture bez poslužitelja su brži deployment aplikacija, veća fleksibilnost, skalabilnost i niži troškovi (slika 32).



Slika 32: Prikaz rezultata ankete o arhitekturi bez poslužitelja provedenoj od strane tvrtke O'Reilly [30]

Na početku rada spomenuli smo da AWS nudi usluge raspoređene u nekoliko kategorija. U nastavku biti će pojašnjena usluga S3 koja pripada kategoriji pohrane (eng. *storage*) te usluga Lambda koja pripada kategoriji računalne snage (eng. *computing*). Obje usluge koristit će se u praktičnom dijelu rada.

5.4.1. S3

S3 jedna je od prvih usluga koje je AWS pokrenuo. To je usluga pohrane objekata (datoteka) koja nudi vrhunsku skalabilnost, dostupnost podataka, sigurnost i performanse. Može se koristiti za pohranu bilo kojeg tipa podataka bilo koje veličine, dok se u web aplikacijama često koristi za pohranu statičkih datoteka kao što su slike. Što se tiče kontrole pristupa, Amazon S3 nudi široke opcije konfiguriranja kontrole pristupa kako bi se zadovoljili bilo koji poslovni zahtjevi aplikacije. Pohrana je organizirana na način da možemo kreirati tzv. *buckete* u koje se mogu pohranjivati podaci. Tako npr. možemo za svaku aplikaciju imati poseban *bucket*. [31]

5.4.2. Lambda

AWS Lambda je računalna usluga koju pruža Amazon te pripada kategoriji računalne snage. Lambda funkcionira na način da korisnik kreira funkcije koje se definiraju kao samostalne aplikacije napisane u jednom od podržanih jezika, potom prenesu aplikaciju u AWS Lambda koji te funkcije tada izvršava. Koristeći Lambdu, možemo se fokusirati samo na svoj kod kojeg samo moramo prenijeti (eng. *upload*) na Lambda servis - AWS će se pobrinuti za pokretanje tog koda što uključuje postavljanje radne okoline, dodjeljivanje računalne snage, automatsko skaliranje itd. U odnosu na standardno pokretanje aplikacije na poslužitelju, tamo bismo prvo morali instalirati operacijski sustav, potom instalirati sve potrebne alate radnog okruženja nakon čega bi tek mogli pokrenuti aplikaciju. [32]

Ono što je najveća prednost Lambda funkcija jest skalabilnost. Kada dođe zahtjev koji će pokrenuti Lambda funkciju, kod funkcije se pokreće u containeru kako bi mogao izvršiti zahtjev. Nakon što se kod funkcije izvrši, funkcija može nastaviti izvršavati idući zahtjev. No, ako za vrijeme izvršavanja zahtjeva pristigne još jedan zahtjev, Lambda će alocirati dodatnu instancu containera koji sadrži kod funkcije što znači da se povećava konkurentnost (eng. *concurrency*) funkcije. Ovo svojstvo iznimno je korisno ako aplikacija ima veliki promet zato što to možemo zamisliti da se naš kod izvršava na više poslužitelja te da je opterećenje ravnomjerno raspodijeljeno između instanca naše Lambda funkcije. Nakon što se izvrši zahtjev funkcije, količina alociranog RAM-a funkcije pomnoži se s trajanjem obrade zahtjeva funkcije u milisekundama te se na taj način ova usluga naplaćuje - dakle prema broju obrađenih zahtjeva, trajanju zahtjeva i količini alocirane memorije. [32] U praktičnom dijelu rada vidjet

ćemo kako na jednostavan način pomoću Serverless Frameworka deployati Lambda funkciju. Velik nedostatak Lambda funkcija je tzv. *cold start* - vrijeme potrebno da pri dolasku zahtjeva za izvršavanje Lambda pokrene container koji će izvršavati kod funkcije. S obzirom da se Lambda funkcije izvršavaju na zahtjev, naš kod nije pokrenut 24/7 kao što je to tradicionalno na poslužitelju već se pokreće samo kada pristigne zahtjev - zato je i trošak manji. Cold start pokretanje obično varira između 100ms i 1s te ovisi o količini/veličini koda koji se pokreće. [33]

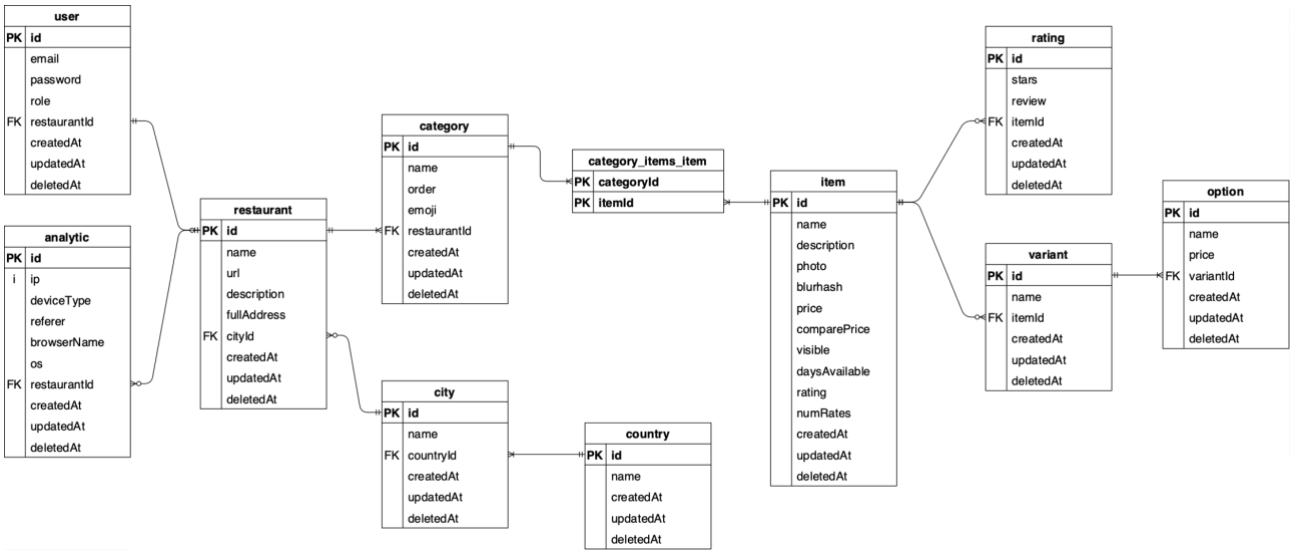
6. Implementacija aplikacije

6.1. Opis aplikacijske domene

U praktičnom dijelu ovog rada izradit ću web aplikaciju koja restoranima omogućuje izradu i održavanje svog digitalnog jelovnika, a korisnicima omogućuje pregled i pretraživanje restorana i njihovih jelovnika te ocjenjivanje jela s jelovnika. Restorani će tako imati mogućnosti dodavanja i uređivanja kategorija, dodavanja i uređivanja stavaka jelovnika te pregleda analitike svog digitalnog jelovnika što uključuje podatke kao što su broj ukupnih pregleda jelovnika po danu, broj jedinstvenih pregleda jelovnika po danu itd. S druge strane, gosti restorana imati će mogućnost pregleda digitalnog jelovnika restorana gdje će moći za svaku kategoriju vidjeti njoj pripadne stavke. Isto tako, kao jedna od korisnih mogućnosti aplikacije biti će sustav ocjenjivanja stavaka sa jelovnika - korisnici će moći ocjenjivati jela te pregledavati njihove ocjene.

Slučaj korištenja ove aplikacije biti će sljedeći: gost dođe u restoran te sjedne za stol na kojem se nalazi QR kod. Skeniranjem koda korisniku se na mobitelu otvara digitalni jelovnik tog restorana gdje on može vidjeti kategorije i jela prema kategorijama. Korisnik može pretraživati jela te za svako vidjeti pripadajuću sliku te ostale podatke (opis i cijena). Što se tiče sustava ocjena, za svako jelo korisnik može vidjeti ocjenu te svako može također ocijeniti ocjenom od 1 do 10. Restoran će prijavom u aplikaciju u bilo kojem trenutku moći uređivati svoj digitalni jelovnik - dodavati i uređivati kategorije i stavke unutar kategorija. Svakom restoranu dodijeljen je jedan QR kod, odnosno URL koji vodi do njihovog jelovnika. QR kod restoran može ispisati na papir te ga staviti na stolove. Restoran u aplikaciji može pregledavati statistiku posjeta njegovog digitalnog jelovnika - koliko je korisnika i jedinstvenih korisnika posjetilo digitalni jelovnik po danima. Osim pregleda digitalnog jelovnika određenog restorana, putem kojeg se može doći skeniranjem QR koda, korisnici na početnoj stranici mogu vidjeti neke korisne podatke kao najbolje ocjenjena jela, restorane s najboljim ocjenama te isto tako mogu pretraživati druge restorane.

6.2. ERA model



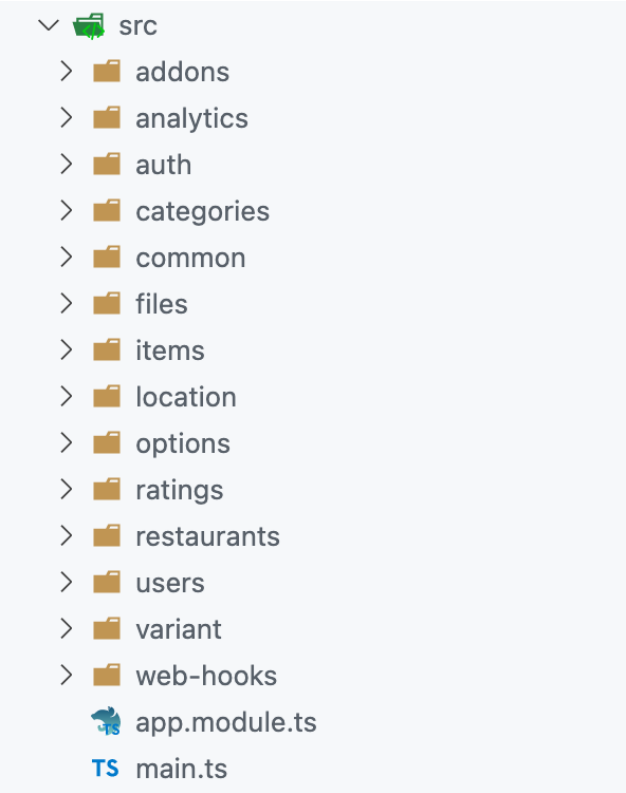
Slika 33: ERA model baze podataka aplikacije

Na slici 33 prikazan je ERA model baze podataka u koju se spremaju podaci aplikacije - restorani, kategorije, stavke jelovnika itd. Kao sustav za upravljanjem baze podataka odabran je PostgreSQL, a ORM biblioteka koja će se koristiti u sklopu Nest.js razvojnog okvira biti će popularan TypeORM.

Svaki restoran (`restaurant`) vezan je za pripadajućeg korisnika (`user`). Svaki restoran može imati kreiranih više kategorija u svojem jelovniku koje se spremaju u relaciji `category`. Stavke menija (npr. jela) pohranjuju se u relaciji `item` te jedna stavka može pripadati više kategorija, a jedna kategorija može sadržavati više stavaka. Za svakog restorana bilježi se analitika posjećenosti njegovog jelovnika što se zapisuje u relaciju `analytic`. Tamo se na temelju user agenta korisnikovog preglednika spremaju podaci o tipu uređaja (`deviceType`), stranice s koje je korisnik došao na stranicu jelovnika (`referer`), naziv web preglednika (`browserName`), i operacijskog sustava (`os`). Svakom restoranu pridružen je grad u kojem se nalazi (relacija `city`), a njemu država u kojem se nalazi (relacija `country`). Svaka stavka može imati varijante (npr. veličina) koje se pohranjuju u relaciji `variant` te svaka varijanta može imati više opcija koje se pohranjuju u relaciji `option` - npr. mala, srednja, velika.

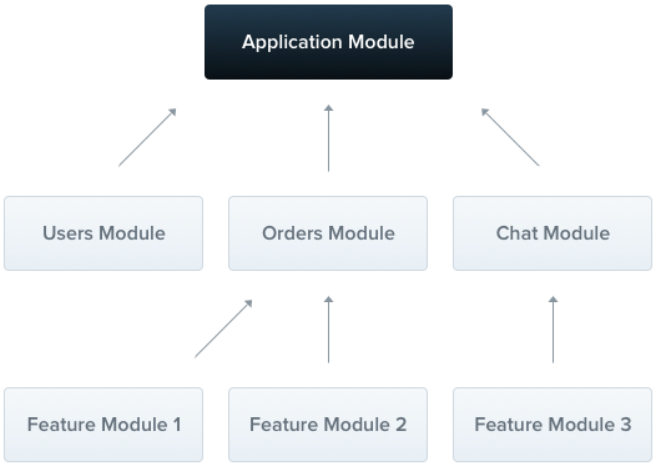
6.3. Razvoj na strani poslužitelja

Poslužitelj je razvijen u Node.js radnoj okolini koristeći Nest.js razvojni okvir koji se generalno sastoji od GraphQL servisa te nekih REST krajnjih točaka. Osim toga, back end dio aplikacije još se sastoji od nekih *serverless* servisa - S3 bucketa za pohranu slika stavaka s jelovnika te Lambda funkcije koja je zaslužna za kompresiju uploadanih slika. Spomenuli smo da Nest.js nastoji riješiti jedan od glavnih problema koji se javlja prilikom razvoja web aplikacija koristeći neku biblioteku ili razvojni okvir - arhitekturu. Na slici 34 prikazana je struktura Nest.js projekta.



Slika 34: Prikaz strukture Nest.js projekta

Nest.js strukturira projekt prema modulima. Svaka aplikacija ima barem jedan modul - korijenski modul (`app.module.ts`). Korijenski modul je početna točka koju Nest koristi za izgradnju grafa aplikacije - internu strukturu podataka koju Nest koristi za rješavanje odnosa i ovisnosti između modula (slika 35).



Slika 35: Primjer grafa Nest.js aplikacije [34]

Modul aplikacije definira se dekoratorom `@Module()` koji prima jedan objekt koji se sastoji od sljedećih svojstva [35]:

- `providers` - temeljni koncept koji predstavlja klase kao što su servisi, repozitoriji, pomoćne klase, resolver klase itd. Glavna ideja provider klase jest da se ona može injektirati u druge klase kao ovisnost.
- `controllers` - klase prema MVC arhitekturi koje sadrže poslovnu logiku
- `imports` - lista svih uvezenih (eng. *import*) modula koji izvoze (eng. *export*) provider klase koje su potrebne ovom modulu
- `exports` - podskup provider klasa koje želimo da budu dostupni za uključivanje u druge module.

Koristeći module, Nest.js čini strukturu projekta jako modularnom te lakom za održavanje. Slijedi primjer `Categories` modula aplikacije. Ovdje kao provider klase koristimo `CategoriesResolver` klasu koja je sadrži GraphQL resolver funkcije za upite i mutacije vezane za kategorije te `CategoriesService` - klasa koja sadrži logiku komunikacije s bazom. U imports arrayu nalazimo TypeORM model `Category` koji je zapravo ORM klasa entiteta iz baze te modul `RestaurantsModule`. Exports array sadrži klasu `CategoriesService` jer se ona koristi u drugim modulima.

```
1   @Module({
2    providers: [CategoriesResolver, CategoriesService],
3    imports: [TypeOrmModule.forFeature([Category]), forwardRef(() => RestaurantsModule)],
4    exports: [CategoriesService],
5   })
6   export class CategoriesModule {}
```

6.3.1. Autentifikacija

Ranije u radu spomenuli smo da GraphQL nije prikladan za svaki slučaj upotrebe - autentifikacija je jedan od tih slučajeva. GraphQL je prikladan kada podaci koji se razmjenjuju reprezentiraju entitete iz baze podataka - tada je struktura u obliku grafa prirodna s obzirom da se njime na intuitivan način mogu prikazati relacije. No s druge strane, kod autentifikacije nije u prvom planu toliko razmjena podataka u obliku entiteta zbog čega bi REST servis i dalje bio prikladniji. Idući razlog tome jest ako bismo npr. implementirali autentifikaciju na našem poslužitelju te bi se u budućnosti možda neka druga aplikacija služila našim servisom za autentifikaciju. Ako ta druga aplikacija ne bi koristila GraphQL, onda bi trebala samo zbog autentifikacije uključiti dodatne biblioteke i dodatne konfiguracije, dok to u slučaju REST-a nije potrebno. Isto tako, s obzirom da GraphQL nema više krajnjih točaka, autentifikacija nije toliko izravna za implementirati. Iz spomenutih razloga autentifikacija je jedan od modula aplikacije koji je implementiran koristeći REST arhitekturu. Slijedi kod modula za autentifikaciju:

```
1    @Module({
2      controllers: [AuthController],
3      imports: [
4        UsersModule,
5        PassportModule,
6        JwtModule.register({
7          secret: 'fifi',
8          signOptions: { expiresIn: '60s' },
9        }),
10     ],
11     providers: [AuthService, LocalStrategy, JwtStrategy],
12   })
13   export class AuthModule {}
```

Dosta modula koji se u Auth modulu koriste potječu iz Passport biblioteke koja autentifikaciju čini doista lakom za implementiranje. U liniji 2 uključuje se `AuthController` koji zapravo definira krajnje točke servisa prema REST arhitekturi na koje će se moći slati zahtjevi (slika 36).

Slika 36: OpenAPI specifikacija krajnjih točaka za autentifikaciju

Slijedi kod Auth controllera koji definira gore navedene tri krajnje točke koje su zaslužne za autentifikaciju.

```
1   @Controller('auth')
2   export class AuthController {
3     constructor(private authService: AuthService) {}
4
5     @UseGuards(LocalAuthGuard)
6     @Post('login')
7     async login(@Request() req, @Response({ passthrough: true }) res: FastifyReply) {
8       // Sign access token
9       const accessToken = await this.authService.login(req.user);
10
11      // Create httpOnly cookie
12      let cookieOptions: CookieSerializeOptions =
13         { httpOnly: true, domain: '.fini.menu', path: '/', secure: true };
14
15      // If its not production, do not set domain and secure flag
16      if (process.env.NODE_ENV !== 'production') cookieOptions = { httpOnly: true, path: '/' };
17
18      res.setCookie('Authorization', `Bearer ${accessToken}`, cookieOptions);
19
20      return { success: true };
21    }
22
23    @UseGuards(AuthGuard('jwt'))
24    @Post('verify')
25    async verify() {
26      return { success: true };
27    }
28
29    @UseGuards(AuthGuard('jwt'))
30    @Post('logout')
31    async logout(@Response({ passthrough: true }) res: FastifyReply) {
32      res.clearCookie('Authorization');
33
34      return { success: true };
35    }
36  }
```

Ovdje vidimo i način injektiranja servisa u controller - injektiranjem preko konstruktora na liniji 3. Taj servis se potom koristi u liniji 9 gdje se poziva login funkcija koja sadrži logiku za generiranje jwt tokena.

6.3.2. GraphQL poslužitelj

Velik dio aplikacije prikladan je za implementaciju koristeći GraphQL. Kao što smo vidjeli u poglavlju 6.2, model podataka sadrži mnogo relacija - restoran može imati više kategorija, kategorija može imati više stavaka dok jedna stavka (jelo) može imati više ocjena. To je upravo slučaj u kojem možemo iskoristiti prednosti koje pruža GraphQL upitni jezik. Spomenuli smo ranije dva pristupa u razvoju GraphQL servisa - shema first i code first pristup. S obzirom na niz prednosti koje dobivamo code first pristupom, to je bio moj odabir. Isto tako, rekli smo da se razvoj GraphQL servisa može podijeliti u dva dijela - definiranje sheme (ovim pristupom će to biti pomoću TypeScript koda) te implementacija resolver funkcija koje sadrže logiku prema kojoj se podaci dohvaćaju i vraćaju za pripadajući tip.

Nest.js omogućuje code first pristup u razvoju GraphQL servisa koristeći dekoratore i TypeScript klase kako bi se generirala korespondirajuća GraphQL shema. Prilikom pokretanja, Nest.js će na temelju koda generirati SDL shema datoteku koja će se potom koristiti pri pokretanju servisa. Slijedi primjer koda kojim definiramo GraphQL tip `Category`:

```
1   import { Field, Int, ObjectType } from '@nestjs/graphql';
2
3   @ObjectType()
4   export class Category {
5     @Field()
6     name: string;
7
8     @Field({ nullable: true })
9     emoji: string;
10
11    @Field((type) => Int)
12    order: number;
13
14    @Field((type) => [Item])
15    items: Item[];
16
17    @Field((type) => Int)
18    itemCount: number;
19  }
```

U liniji 1 uključujemo nekoliko dekoratora koji služe za definiranje tipa. U liniji 3 koristimo ObjectType dekorator koji je ekvivalentan ključnoj riječi type u SDL jeziku. Dekorator Field predstavlja jedno polje unutar tipa kojem se može dodijeliti tip koji je `String` (linije 5 i 8) ukoliko se to ne definira drukčije (linije 11, 14 i 17). Zadano je da je svako polje non-nullable (što se u SDL jeziku označava znakom `!`) ukoliko se to ne označi drukčije (linija 8). U liniji 14 kao tip polja odabrali smo `Item` što je zapravo druga klasa kojom smo definirali tip Item. Ovaj kod će se u SDL jezik prevesti kao što slijedi:

```
type Category {
    name: String!
    emoji: String
    order: Int
    items: [Item!]!
    itemCount: Int!
}
```

Tipove smo definirali, no prije nego što definiramo tip za upit kojim ćemo dohvatiti kategorije nekog restorana, još ćemo se malo zadržati na gore definiranoj klasi Category. Za komunikaciju sa PostgreSQL bazom podataka koristimo ORM biblioteku TypeORM koja također za definiranje ORM entiteta koristi dekoratore. S obzirom da nam je GraphQL tip koji smo definirali reprezentacija entiteta iz baze, definiranu klasu nadopunit ćemo dekoratorima kojima ćemo je pretvoriti u ORM entitet:

```
1   import { Column, Entity, JoinTable, ManyToMany, ManyToOne } from 'typeorm';
2   import { Field, Int, ObjectType } from '@nestjs/graphql';
3
4   import { Restaurant } from '../restaurants/restaurant.entity';
5   import { BaseEntity } from '../common/base.entity';
6   import { Item } from '../items/item.entity';
7
8   @Entity()
9   @ObjectType()
10  export class Category extends BaseEntity {
11    @Column()
12    @Field()
13    name: string;
14
15    @Column({ nullable: true })
16    @Field({ nullable: true })
17    emoji: string;
18
19    @Column()
20    @Field((type) => Int)
21    order: number;
22
23    @ManyToMany(() => Item, (item) => item.categories)
24    @JoinTable()
25    @Field((type) => [Item])
26    items: Item[];
27
28    @Field((type) => Int)
29    itemCount: number;
30
31    @ManyToOne(() => Restaurant)
32    restaurant: Restaurant;
33
34    @Column()
35    restaurantId: string;
36    constructor(name: string, order: number, restaurant: Restaurant) {
37      super();
38      this.name = name;
```

```
39      this.order = order;
40      this.restaurant = restaurant;
41    }
42  }
```

Iako se većina svojstava preklapaju između definiranja TypeORM entiteta i GraphQL tipa, primjećujemo da to nije slučaj za sva svojstva - npr. u liniji 28-29 definirali smo GraphQL polje `itemCount` koje ne postoji kao stupac u bazi podataka. Dakle, ovo svojstvo neće reflektirati podatak iz baze nego ćemo implementirati resolver funkciju koja će vraćati taj podatak. Isto tako imamo definirana neka svojstva u ORM entitetu koje nemamo u GraphQL tipu i to je sasvim uobičajeno jer nam nisu uvijek bitna sva svojstva entiteta iz baze podatka. Sada možemo krenuti na implementaciju resolver klase koju ćemo nazvati `CategoriesResolver`:

```
1   @Resolver((of) => Category)
2   export class CategoriesResolver {
3     constructor(private restaurantsService: RestaurantsService) {}
4
5     @Query((returns) => [Category])
6     async categories(@Args() args: GetCategoriesArgs) {
7       const { restaurantId, url } = args;
8
9       const restaurant = await this.restaurantsService.findOne({
10        where: { id: restaurantId, url },
11        relations: ['categories', 'categories.items', 'categories.items.categories'],
12        order: {
13          categories: {
14            order: 'ASC',
15          },
16        },
17      });
18
19      if (!restaurant) throw new Error('Restaurant not found');
20
21      return restaurant.categories;
22    }
23  }
```

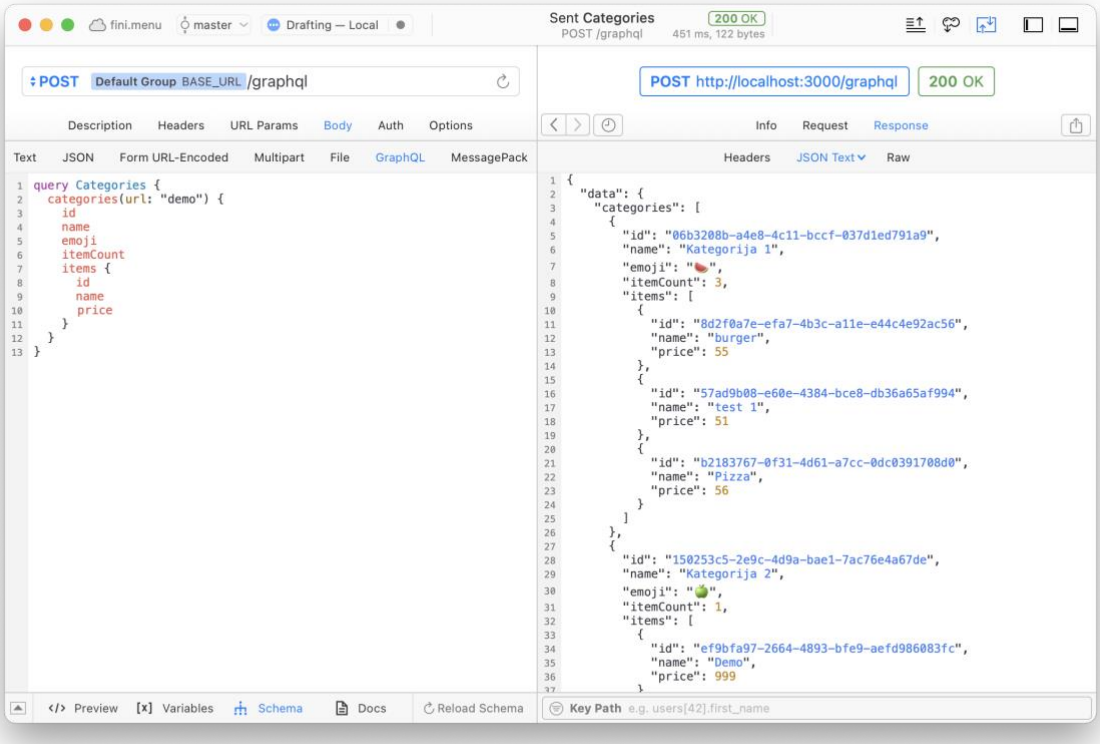
U liniji 1 koristili smo dekorator `Resolver` koji određuje da se radi o implementaciji resolver funkcije. U liniji 3 pomoću konstruktora klase injektirali smo ovisnost o klasi `RestaurantsService` koja predstavlja servis koji komunicira s bazom podataka preko TypeORM biblioteke. U liniji 5 dekorator `Query` definira da se radi o definiranju upita te je ekvivalentan ključnoj riječi `query` u SDL-u. Dekorator sadrži povratni tip `[Category]` koji definira da će ovaj upit vratiti niz tipova `Category`. U idućoj liniji definirana je sama funkcija čiji naziv predstavlja naziv upita koji ćemo definirati. U liniji 9 pozivamo funkciju iz klase `RestaurantsService` koja komunicira s bazom podataka te izgleda ovako:

```
1   @Injectable()
2   export class RestaurantsService {
3     constructor(
4       @InjectRepository(Restaurant)
5       private readonly restaurantsRepository: Repository<Restaurant>,
6     ) {}
7
8     async findOne(query: FindOneOptions<Restaurant>): Promise<Restaurant> {
9      return this.restaurantsRepository.findOne(query);
10    }
11
12    async create(name: string, url: string, user: User): Promise<Restaurant> {
13      const restaurant = this.restaurantsRepository.create({ name, url, user });
14      await this.restaurantsRepository.save(restaurant);
15
16      return restaurant;
17    }
18
19    async update(dto: Partial<Restaurant>): Promise<Restaurant> {
20      const restaurant = await this.restaurantsRepository.save(dto);
21      return restaurant;
22    }
23   }
```

U liniji 1 korišten je Nest.js dekorator `Injectable` koji označava da se ova klasa može injektirati kao ovisnost unutar drugih klasa. U liniji 4 injektiran je repozitorij restorana iz TypeORM biblioteke. Potom je definirano nekoliko funkcija koje komuniciraju s bazom podataka preko TypeORM biblioteke.

Da se vratimo na gornji kod definiranja `categories` upita, u liniji 7 se od argumenata upita uzimaju argumenti `restaurantId` i `url` kako bismo mogli prema tim argumentima pretražiti restoran. Nakon što se provede upit u bazi na liniji 9, rezultati se vrate u liniji 21. Ovaj kod će pomoću SDL-a generirati shemu kao što slijedi. Ako izvršimo ovaj upit, dobit ćemo odgovor kao na slici 37.

```
type Query {
  categories(restaurantId: ID, url: String): [Category!]!
}
```

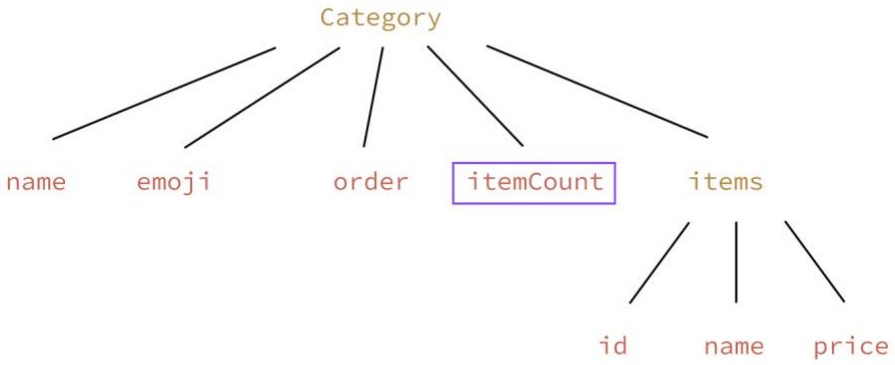


Slika 37: Primjer izvršenog categories upita

U klasi Category ranije smo definirali jedno polje koje se ne nalazi u bazi podataka - `itemCount`, no na slici 37 vidimo da to polje vraća podatak. Resolver funkcije također možemo definirati na razini polja pa ćemo sada unutar `CategoriesResolver` klase definirati resolver funkciju koja će vraćati podatke za polje `itemCount`:

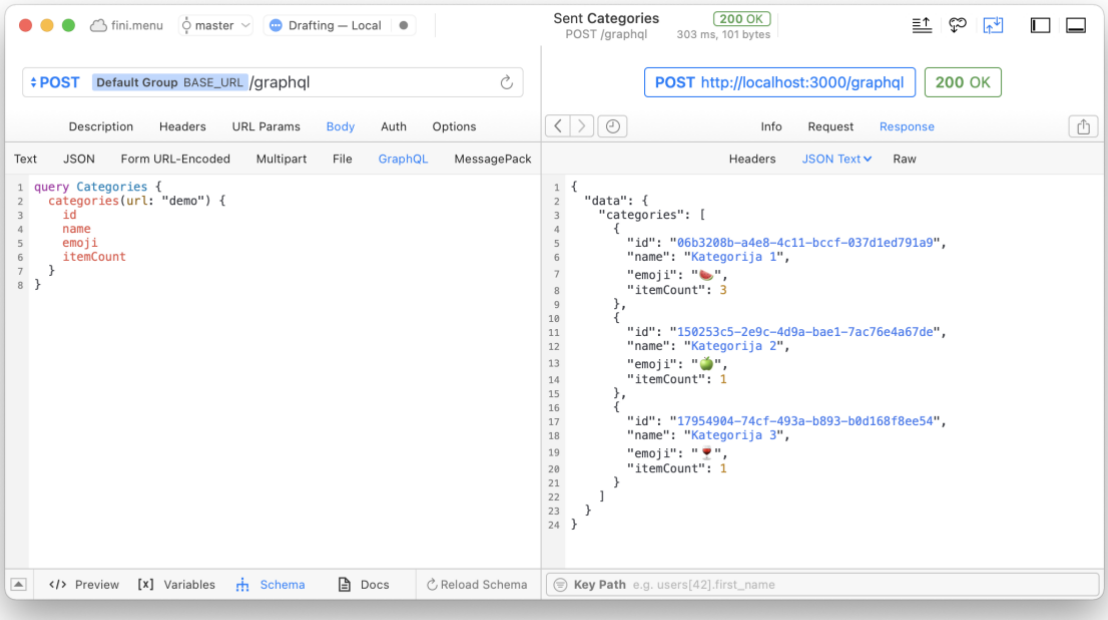
```
1   @ResolveField()
2   async itemCount(@Parent() category: Category) {
3     return category.items.length;
4   }
```

U liniji 1 koristili smo `ResolveField` Nest.js dekorator koji predstavlja da se radi o resolver funkciji na razini GraphQL polja. U liniji 2 nazivom funkcije označili smo za koje polje će ova funkcija vratiti rezultate dok smo kao argument iskoristili `Parent` dekorator. Za objašnjenje ovog dijela pogledajmo sljedeći graf koji reprezentira upit categories sa slike 37 :



Slika 38: Graf koji reprezentira upit categories sa slike 37

Kako upit vraća tip Category, on je na najvišoj razini u grafu pa sva njegova polja/svojstva možemo mu smatrati djecom. Ako je `itemCount` polje dijete od tipa Category, tada je Category roditelj (eng. *parent*) polju `itemCount`. Upravo to predstavlja dekorator `Parent` - njime možemo pristupiti roditelju u grafu, odnosno u ovom slučaju cijelom objektu tipa Category. Na temelju toga gore u kodu u liniji 3 vraćamo broj elemenata u nizu dohvaćenih kategorija. Ako ponovo pogledamo odgovor na naš upit kojeg smo dobili od poslužitelja na slici 36, možemo postaviti još jedno pitanje - kako je poslužitelj znao vratiti za određenu kategoriju njoj pripadajuće stavke (tip `Item`)? Ako se osvrnemo na kod klase `CategoriesResolver` gdje smo implementirali resolver našeg GraphQL upita, sve postaje jasnije jer smo u liniji 11 definirali `relations` niz na temelju kojeg će TypeORM provesti JOIN operaciju nad definiranim relacijama unutar tog niza. Što će se desiti ako izvršimo upit kao što je na slici 39?



Slika 39: Izvršen categories upit koji ne dohvaća podatke o stavkama

U tom upitu nismo zatražili od poslužitelja polje `items`, no, hoće li se svojstvo `items` (stavke kategorije) na poslužitelju zapravo dohvatiti iz baze podataka? Odgovor je da, i to je problem poznat kao overfetching (sjetimo se, ovaj problem spomenuli smo u kod REST servisa ali u malo drukčijem obliku). Overfetching kod GraphQL-a jest problem kada poslužitelj dohvaća iz izvora podataka (npr. baze podataka) više podataka nego što je klijent zatražio. Ovaj problem je sasvim normalna pojava u razvoju GraphQL poslužitelja te je često neizbježan - no veći problem predstavlja overfetching kod REST servisa koji zapravo utječe na klijenta jer on troši više bandwidtha, nego overfetching kod GraphQL servisa jer se tada overfetching odvija na poslužitelju - klijent nema znanja o tom problemu jer će on uvijek kao odgovor primiti podatke koje je zatražio.

Postoji način kako bismo mogli izbjeći ovaj problem. Možemo iz resolver funkcije za upit `categories` maknuti dio koda kojim se u bazi podataka izvršava JOIN funkcija, a za polje `items` implementirati resolver funkciju koja će dohvatiti stavke te kategorije. Kod će izgledati ovako:

```
1    @UseGuards(GqlAuthGuard, RolesGuard)
2    @Query((returns) => [Category])
3    async categories(@CurrentUser() user: User, @Args() args: GetCategoriesArgs) {
4      const { restaurantId, url } = args;
5      const id = restaurantId || user.restaurant.id;
6
7      const restaurant = await this.restaurantsService.findOne({
8        where: { id, url },
9        order: {
10         categories: {
11           order: 'ASC',
12         },
13       },
14     });
15
16     if (!restaurant) throw new Error('Restaurant not found');
17
18     return restaurant.categories;
19   }
20
21   @ResolveField()
22   async items(@Parent() category: Category) {
23     return this.itemsService.find({
24       where: { categoryId: category.id }
25     })
26   }
```

Ukoliko sada izvršimo upit sa slike 39 neće doći do overfetching problema jer se neće na poslužitelju dohvatiti stavke kategorije iz baze podataka, dakle izvršit će se jedan upit prema

bazi podataka. No, ako sada izvršimo upit sa slike 37, dobit ćemo isto odgovor ali za svaku kategoriju će se izvršiti po jedan upit prema bazi koji će dohvatiti njegove stavke (kod gornje resolver funkcije od linije 21 do 26). Sada će se prilikom ovog upita, ako postoje 3 kategorije u bazi podataka, izvršiti ukupno 4 upita prema bazi podataka što je jako neefikasno. Navedeni problem naziva se N+1 problem te je također čest u razvoju GraphQL servisa. [35] Sam naziv problema predstavlja da ukoliko ćemo dohvaćati neki entitet koji ima definirane resolver funkcije za svoja polja koja izvršavaju dodatan upit prema bazi podataka tada ćemo uvijek imati 1 upit prema bazi podataka za dohvat svih entiteta te za svaki dohvaćeni entitet dodatan upit za njegovo polje nad kojim smo definirali resolver funkciju (N zahtjeva). Od mogućih rješenja ovog problema jedno je implementirati upit tako da će pretrpjeti overfetching problem, koji je puno manji problem nego N+1 problem glede performansa.

Kako smo definirali potrebne tipove i pojasnili implementaciju jednog upita, preostalo nam je pogledati primjer implementacije jedne mutacije. Uzet ćemo za primjer mutaciju za ažuriranje neke kategorije - `updateCategory` mutacija. Implementacija ove mutacije nalazit će se također u CategoriesResolver klasi te će izgledati ovako:

```
1    @Mutation((returns) => Category)
2    async updateCategory(@Args() args: UpdateCategoryArgs): Promise<Category> {
3      return this.categoriesService.update(args);
4    }
```

U liniji 1 koristimo Nest.js dekorator koji je ekvivalentan ključnoj riječi `mutation` u SDL-u. U dekoratoru se također, jednako kao i kod Query dekoratora, definira povratni tip. U liniji 2 definiramo funkciju čiji naziv predstavlja naziv mutacije. Kod argumenta funkcije iskoristili smo dekorator `Args` kojim se definiraju argumenti te mutacije (isto kao i definiranje `input` tipa u SDL-u). Taj argument tipa je klase `UpdateCategoryArgs` čija implementacija izgleda kao što slijedi:

```
1    @ArgsType()
2    export class UpdateCategoryArgs {
3      @Field((type) => ID)
4      id: string;
5
6      @Field({ nullable: true })
7      name?: string;
8
9      @Field({ nullable: true })
10     emoji?: string;
11
12     @Field((type) => Int, { nullable: true })
13     order?: number;
14   }
```

Dekorator `ArgsType` ekvivalentan je ključnoj riječi `input` u SDL-u te se polja definiraju na isti način koristeći iste dekoratore kao i kod definiranja tipova. Prikazani kod će generirati sljedeću shemu pomoću SDL-a:

```
input UpdateCategoryArgs {
    id: ID!
    name: String
    emoji: String
    order: Int
}
```

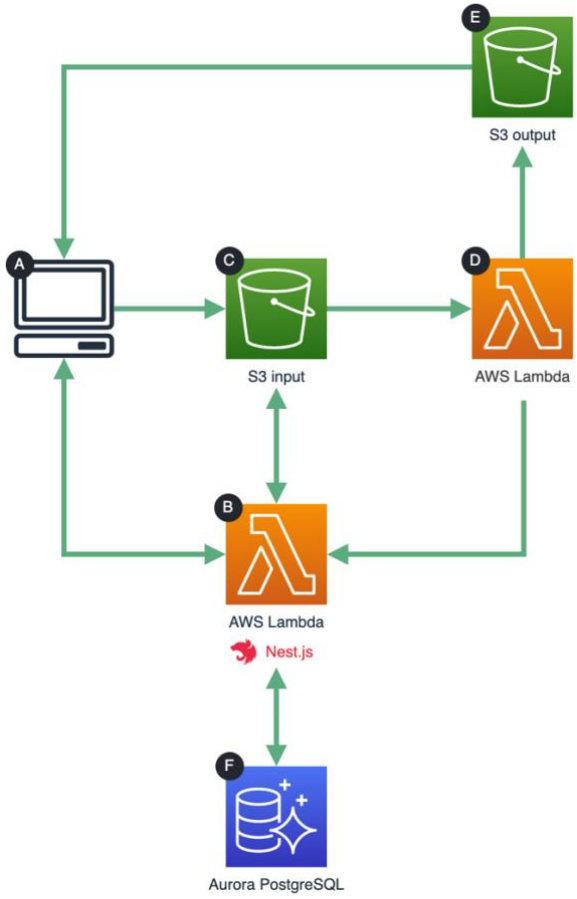
Da se vratimo na kod mutacije `updateCategory`, u liniji 3 poziva se funkcija iz klase `CategoriesService` koja pomoću TypeORM-a izvršava upit u bazi podataka za ažuriranje kategorije. Taj kod će generirati sljedeći tip kojim se definira `updateCategory` mutacija pomoću SDL-a:

```
type Mutation {
  updateCategory(data: UpdateCategoryArgs!): Category!
}
```

Ovo je bio osnovni princip kako su implementirani tipovi, upiti, mutacije te njima pripadne resolver funkcije na back end strani za modul kategorija. Na isti način implementirani su ostali moduli aplikacije.

6.3.3. Arhitektura bez poslužitelja

S obzirom se želimo fokusirati na razvoj aplikacije, a upravljanje i održavanje infrastrukture želimo prepustiti AWS-u, pomoću Serverless Frameworka ćemo sastaviti serverless arhitekturu aplikacije. Pri tome ćemo koristiti tri AWS servisa - S3, Aurora i Lambda. Na slici 40 prikazana je arhitektura back end strane aplikacije kakvu ćemo postaviti.



Slika 40: Prikaz arhitekture bez poslužitelja aplikacije

Korisnik (A) će komunicirati s Nest.js aplikacijom (B) koja će se pokretati pomoću Lambda funkcije. Nest.js aplikacija komunicirati će s bazom podataka koja će biti pokretana na AWS Aurora PostgreSQL servisu (F) što je servis koji pruža pokretanje *serverless* baze podataka - bazu podataka koju možemo jednim klikom pokrenuti bez da brinemo o raznim instalacijama, dodjeljivanju računalnih resursa i ono što je najbolje - AWS se brine o automatskom skaliranju. U aplikaciji će biti moguće uploadati slike jela koje će se spremati u S3 bucket. Ono što je potrebno dodatno napraviti jest procesuirati sliku - smanjiti rezoluciju i kvalitetu tako da se slike brzo učitavaju za korisnika. To će biti implementirano na način da ćemo imati dva S3 bucketa - input i output. Korisnici će slike uploadati direktno u input bucket

te nakon što slika bude tamo uploadana pokrenut će se Lambda funkcija koja će uzeti tu sliku, smanjit joj rezoluciju i kvalitetu te ju pohraniti u output bucket. Korisnici će u aplikaciji pristupati slikama iz output bucketa - dakle procesuirane slike čija rezolucija je smanjena.

Koristeći Serverless Framework inicijalizirat ćemo novi projekt te u serverless.yml datoteku dodati AWS resurse koji će se koristiti (input i output S3 buckete):

```
resources:
  Resources:
    InputBucket:
      Type: "AWS::S3::Bucket"
      Properties:
        BucketName: ${self:custom.inputBucket}
        CorsConfiguration:
          CorsRules:
            - AllowedOrigins:
                - "*"
              AllowedMethods:
                - PUT
              AllowedHeaders:
                - "*"
              MaxAge: 3600
    OutputBucket:
      Type: AWS::S3::Bucket
      Properties:
        BucketName: ${self:custom.outputBucket}
        AccessControl: PublicRead
        CorsConfiguration:
          CorsRules:
            - AllowedOrigins:
                - "*"
              AllowedMethods:
                - GET
              AllowedHeaders:
                - "*"
              MaxAge: 3600
```

S obzirom da se pokretanje Lambda funkcija temelji na događajima, definirat ćemo u istoj datoteci da se pokrene Lambda funkcija za obradu slika kada se desi događaj kreiranja objekta u input bucketu (kada se uploada slika u input bucket). Kasnije ćemo implementirati kod te Lambda funkcije.

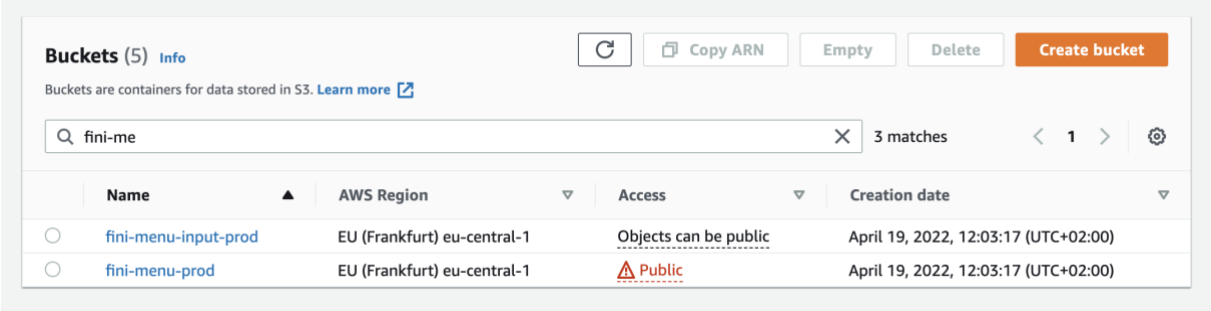
```
functions:
  itemPhotoProcess:
    description: Triggered when an item photo is uploaded
    handler: src/itemPhoto.handler
    events:
      - s3:
          bucket:
            Ref: InputBucket
          existing: true
          event: s3:ObjectCreated:*
          rules:
            - prefix: item/
    environment:
      INPUT_BUCKET: ${self:custom.inputBucket}
      OUTPUT_BUCKET: ${self:custom.outputBucket}
```

U liniji 4 definirali smo putanju gdje se nalazi kod Lambda funkcije koji ćemo implementirati, dok smo u liniji 5 definirali događaj koji će pokrenuti tu funkciju. Sam kod funkcije koji slijedi je dosta jednostavan.

```
1  export const handler = async (event: S3Event) => {
2    const s3 = new AWS.S3();
3
4    for (const record of event.Records || []) {
5      // Get uploaded photo
6      const uploadedPhoto = await s3.getObject({ Bucket: record.s3.bucket.name,
7                                                  Key: record.s3.object.key }).promise();
8
9      // Compress photo
10     const compressed = await sharp(uploadedPhoto.Body as Buffer)
11       .rotate()
12       .resize({ width: 500, height: 500, fit: "cover" })
13       .jpeg({ quality: 90 })
14       .toBuffer();
15
16     // Calculate blurhash of the photo
17     const blurhashPhoto = await sharp(uploadedPhoto.Body as Buffer)
18       .raw()
19       .ensureAlpha()
20       .resize(32, 32, { fit: "inside" })
21       .toBuffer({ resolveWithObject: true });
22
23     const blurhash = encode(new Uint8ClampedArray(blurhashPhoto.data),
24                             blurhashPhoto.info.width, blurhashPhoto.info.height, 4, 4);
25
26     // Upload photo to output bucket
27     await s3
28       .putObject({
29         Bucket: process.env.OUTPUT_BUCKET,
30         Key: record.s3.object.key,
31         Body: compressed,
32         ContentType: "image",
33         ACL: "public-read",
34       })
35       .promise();
36
37     // Delete original photo
38     await s3.deleteObject({ Bucket: record.s3.bucket.name,
39                             Key: record.s3.object.key }).promise();
40
41     // Call api webhook
42     await fetch("https://api.fini.menu/web-hooks/photo", {
43       method: "POST",
44       body: JSON.stringify({ photoId: uploadedPhoto.Metadata?.photoid, blurhash }),
45       headers: { "Content-Type": "application/json" },
46     });
47   }
48 };
```

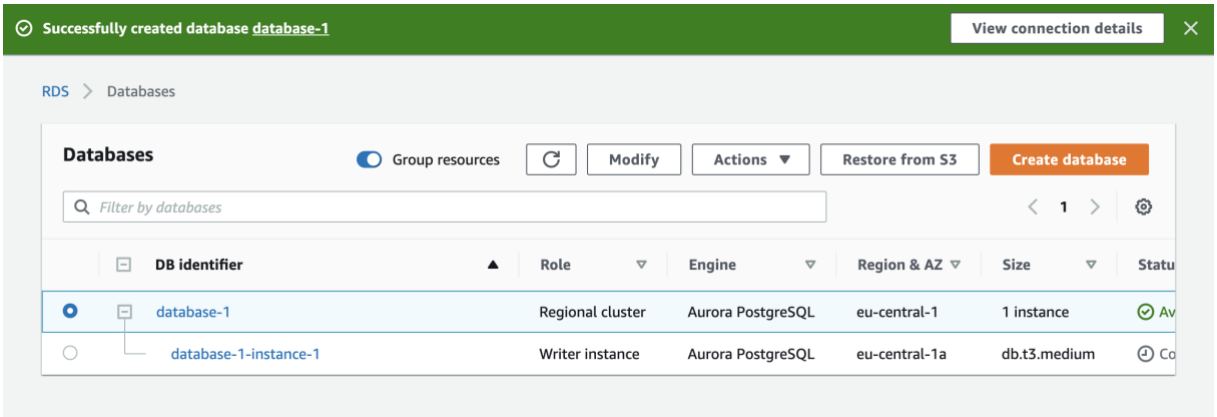
Za obradu slike koristi se biblioteka Sharp, dok se u liniji 23-24 izračunava tzv. BlurHash slike o čemu će biti nešto više riječi u idućem poglavlju.

U linijama 41-46 podaci o slici (izračunati BlurHash i ID slike) šalju se Nest.js aplikaciji koja te podatke tada zapisuje u bazu podataka. Naredbom `serverless deploy` možemo deployati Serverless Framework projekt te ako tada pogledamo u AWS upravljačkoj ploči, vidjet ćemo da su kreirana dva S3 bucketa:



Slika 41: Prikaz kreiranih S3 bucketa u AWS upravljačkoj ploči

Ako se osvrnemo na sliku 39 koja prikazuje arhitekturu back end dijela aplikacije, ono što nam još preostaje je kreirati bazu podataka pomoću Aurora PostgreSQL servisa te deployati Nest.js projekt pomoću Serverless Frameworka kao Lambda funkciju. Bazu podataka pomoću Aurora servisa možemo jednostavno napraviti preko AWS upravljačke ploče gdje potom dobijemo podatke za spajanje na bazu (slika 42).



Slika 42: Pregled kreirane baze podataka u AWS upravljačkoj ploči

Preostaje nam još samo deployati Nest.js projekt pomoću Serverless Frameworka što ćemo učiniti tako da u Nest.js projekt dodamo serverless.yml datoteku sa sljedećim sadržajem:

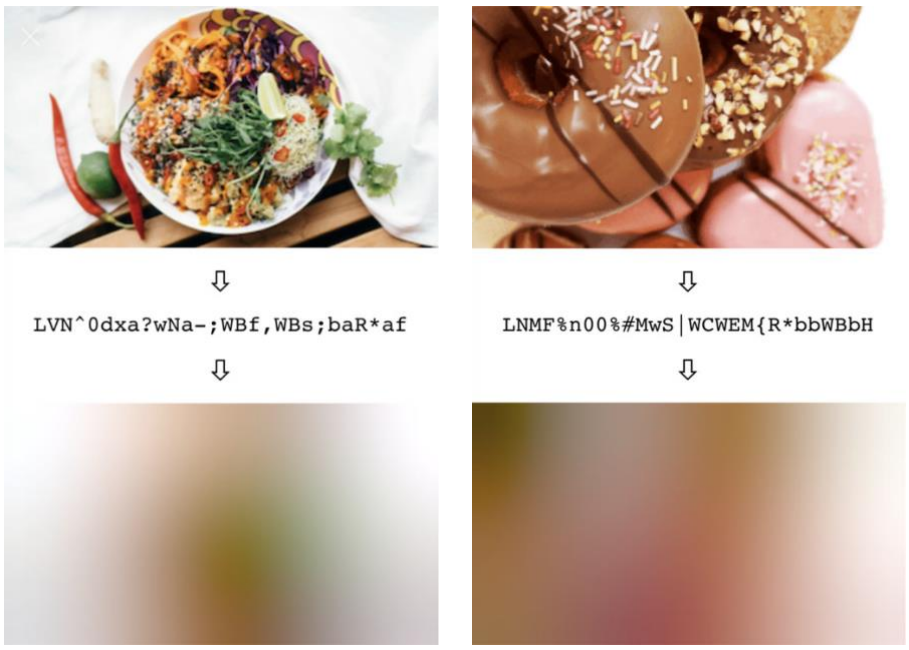
```
1   service: graphql
2
3   plugins:
4     - serverless-offline
5
6   provider:
```

```
7       name: aws
8       region: eu-central-1
9
10    functions:
11      main:
12        handler: dist/src/index.handler
13        events:
14          - http:
15              cors: true
16              method: ANY
17              path: '/'
18          - http:
19              method: ANY
20              path: '{proxy+}'
```

U liniji 12 definirana je putanja do koda koji će se pokrenuti u Lambda funkciji. Taj kod je zapravo funkcija koja pokreće Nest.js aplikaciju koja koristi `aws-lambda-fastify` biblioteku.

6.3.3.1. BlurHash algoritam

Kako slike danas predstavljaju jedan od ključnih sadržaja web stranica, one se često sporo učitavaju ukoliko stranica sadrži veći broj slika. Tako će sve dok se slika ne učita, na njezinom mjestu biti prikazana praznina što vizualno s perspektive UX-a ne izgleda najbolje. Jedno od rješenja ovog problema jest prikaz tzv. placeholdera slike za vrijeme njezinog učitavanja koji se može izračunati pomoću BlurHash algoritma koji je razvijen od strane tvrtke Wolt. Jednostavno rečeno, BlurHash placeholder predstavlja kompaktnu vizualnu reprezentaciju slike i koristi se na front endu za prikaz slike za vrijeme njezinog učitavanja kako bi se poboljšalo korisničko iskustvo (slika 43).



Slika 43: Blurhash algoritam za izračun kompaktnog prikaza slike [36]

Ovaj algoritam koji se danas široko koristi zapravo je jednostavan te je za njegovu inicijalnu implementaciju Wolt timu bilo potrebno svega dva dana. BlurHash algoritam temelji se na diskretnoj kosinusnoj transformaciji (DCT) koja se koristi također pri kompresiji slika (JPEG i HEIF formati). Potom se rezultat diskretne kosinusne transformacije enkodira u base 83 format te se kao rezultat dobiva niz koji izgleda kao što slijedi [37]:

`LlMF%n00%#MwS|WCWEM{R*bbWBbH`

BlurHash niz se tada sprema u bazu podataka te samo što je još preostalo jest na front endu aplikacije prikazati BlurHash reprezentaciju slike koja se učitava. To je ostvareno pomoću `react-blurhash` biblioteke koja u pozadini koristi `blurhash` biblioteku kako bi se proveo obrnut algoritam od gore navedenog, odnosno kako bi se dekodirao BlurHash niz i na temelju njega stvorila slika koja izgleda kao mutna verzija originalne slike (slika 43). Slijedi kod kojim se pomoću komponente `BlurhashCanvas` iz biblioteke `react-blurhash` renderira BlurHash placeholder slika ispod slike koja se učitava, te nakon što se slika učita ona se prikazuje iznad BlurHash slike:

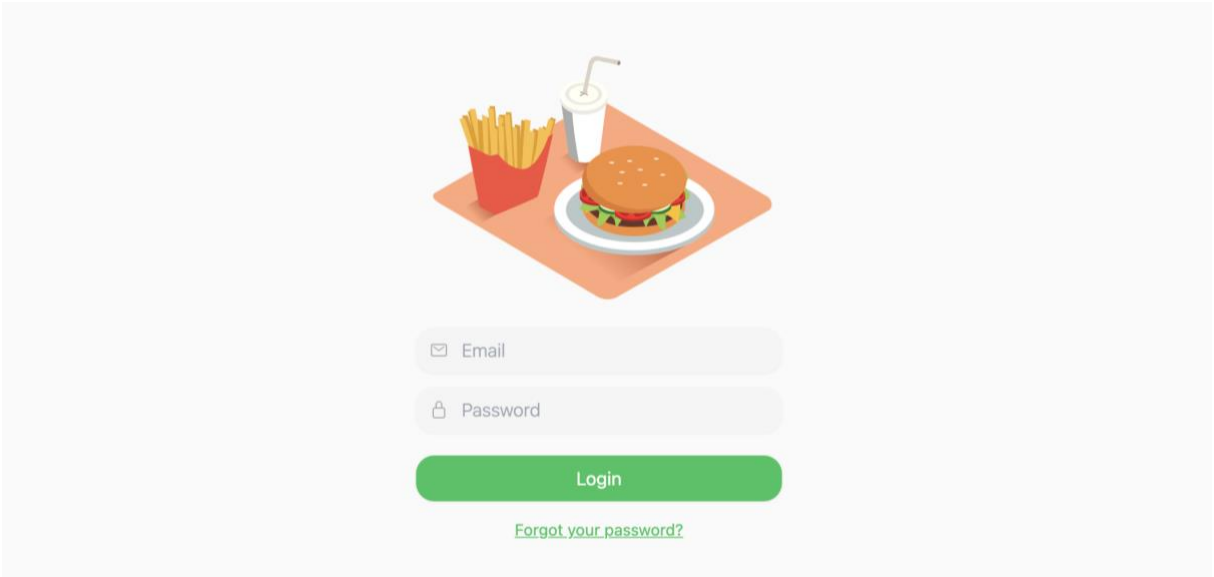
```
<div className="relative">
  {item.blurhash &&
    <BlurhashCanvas
      hash={item.blurhash}
      punch={1}
      className="absolute w-full h-full rounded-2xl rounded-b-none"
    />
  }
  <Image
    src={item.photo || require("@common/assets/placeholder.svg")}
    width="100%"
    height={70}
    layout="responsive"
    objectFit="cover"
    className="rounded-2xl rounded-b-none"
    alt=""
  />
</div>
```

6.4. Razvoj na strani korisnika

Aplikacija je na strani korisnika razvijena koristeći Next.js - razvojni okvir za React koji olakšava i ubrzava razvoj interaktivnih korisničkih sučelja. Od značajnijih biblioteka koje su korištene pri razvoju bitno je istaknuti Apollo biblioteku koja je GraphQL klijent pomoću kojeg možemo slati upite GraphQL poslužitelju. Osim toga, Apollo sadrži napredan cache mehanizam zahvaljujući kojem se aplikacija čini responzivna. Za upravljanje stanjem aplikacije (eng. *state management*) korištena je Redux biblioteka. U nastavku će biti prikazane funkcionalnosti aplikacije te njihova pripadajuća implementacija.

6.4.1. Prijava

Restoran ima mogućnost prijave u aplikaciju pomoću email adrese i lozinke (slika 44). Ovdje se poziva krajnja točka `/auth/login` sa slike 35 te se zahtjev šalje pomoću Axios biblioteke.



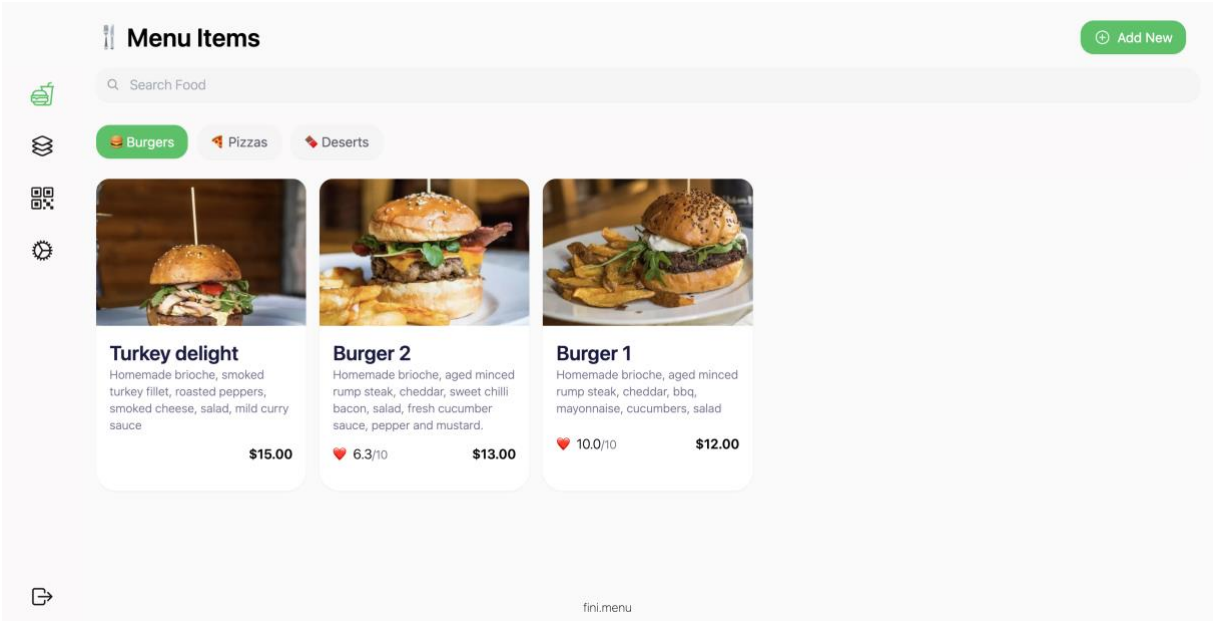
Slika 44: Prijava u aplikaciju

Slijedi dio koda kojim su definirali pozivi prema REST krajnjim točkama za autentifikaciju pomoću Axios biblioteke. Kod je prilično jednostavan, a kasnije ćemo vidjeti na koji način se šalju upiti GraphQL poslužitelju pomoću Apollo biblioteke.

```ts
import axios from "axios";

export const api = axios.create({
  baseURL: process.env.NEXT_PUBLIC_API_URL,
  withCredentials: true,
});

export const login = async (email: string, password: string) => {
  return await api.post("/auth/login", { email, password });
};

export const logout = async () => {
  return await api.post("/auth/logout");
};

export const verify = async () => {
 return await api.post("/auth/verify");
};
```

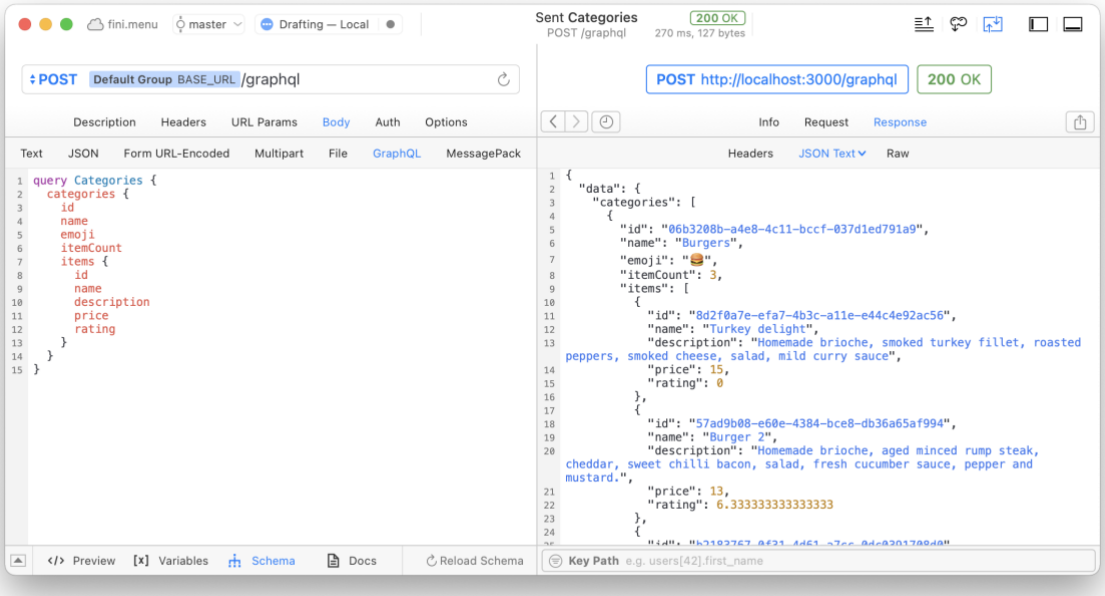
6.4.2. Pregled i uređivanje jela

Nakon prijave prikazuje se početni zaslon koji omogućuje pregled svih dostupnih kategorija te stavaka po kategorijama. Na ovom zaslonu mogu se pretraživati stavke jelovnika te se klikom na gumb "Add New" može dodati nova stavka.



Slika 45: Zaslon za prikaz i uređivanje jela

Ovdje imamo slučaj sličan kao sa slike 28 gdje moramo dohvatiti kategorije i pripadne stavke. Za taj prikaz koristit ćemo sljedeći GraphQL upit koji dohvaća sve kategorije i pripadajuće stavke za trenutno prijavljeni restoran:



Slika 46: Izvršeni upit za dohvaćanje svih kategorija i jela za trenutno prijavljen restoran

Za izvršavanje upita na GraphQL poslužitelj koristit ćemo Apollo biblioteku. Prvo moramo definirati upit koji želimo poslati:

```
export const GET_CATEGORIES_ITEMS = gql`
  query GetCategoriesAndItems {
    categories {
      id
      name
      emoji
      items {
        id
        name
        description
        price
        photo
        daysAvailable
        blurhash
        rating
        categories {
          id
          name
          emoji
        }
      }
    }
  }
`;
```

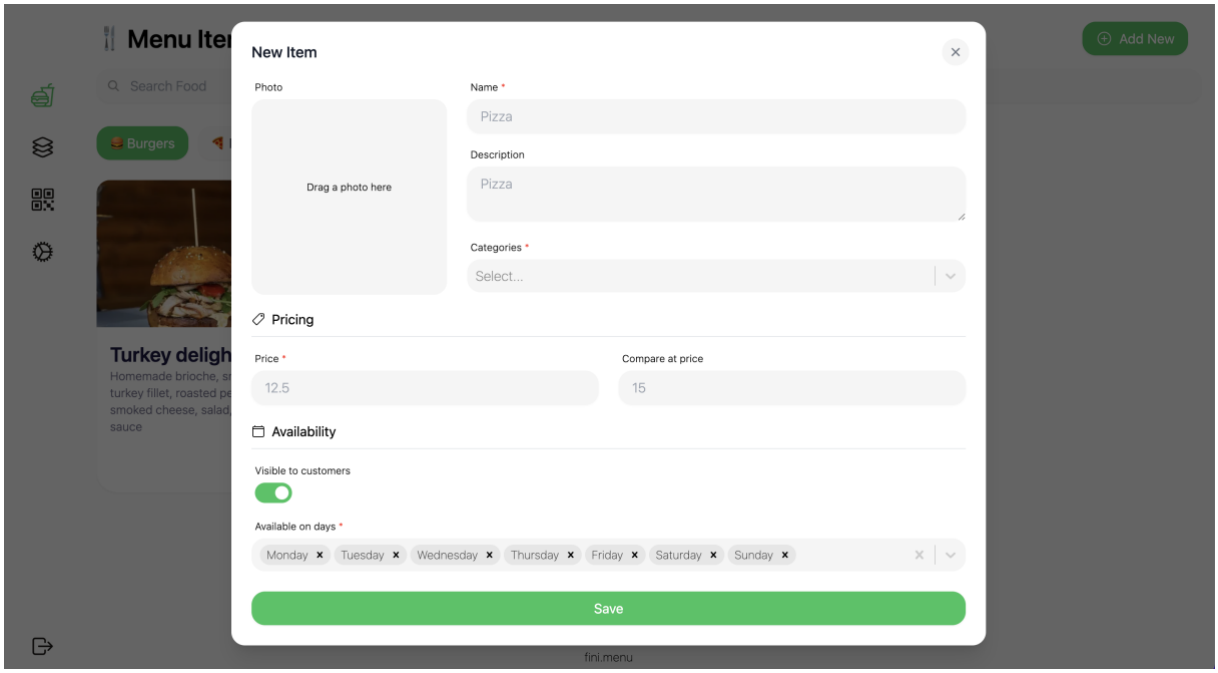
Zatim opcionalno možemo definirati TypeScript tip za argumente upita i za podatke koje vraća upit. Iako je taj dio opcionalan, jako se preporuča definirati tipove kako bi iskoristili prednosti statične provjere tipova koju nam TypeScript pruža. Definirat ćemo tako povratni tip `CategoriesData` koji se podudara s tipom koji nam vraća `categories` upit koji smo gore definirali:

```
export interface CategoriesData {
  categories: Category[];
}

interface Category {
  id: string;
  name: string;
  emoji?: string;
  items: Item[];
}

export interface Item {
  id: string;
  name: string;
  description: string;
  price: number;
  comparePrice: number;
  photo: string;
  blurhash: string;
  daysAvailable: string[];
  rating: number;
  categories: Category[];
}
```

Nakon toga koristeći useQuery hook iz Apollo biblioteke možemo poslati zahtjev GraphQL poslužitelju kao što slijedi. Varijabla `data` sadržat će podatke tipa `CategoriesData` koji smo definirali gore kada pristigne odgovor od poslužitelja, dok je varijabla `loading` boolean koji označava je li zahtjev još u tijeku.

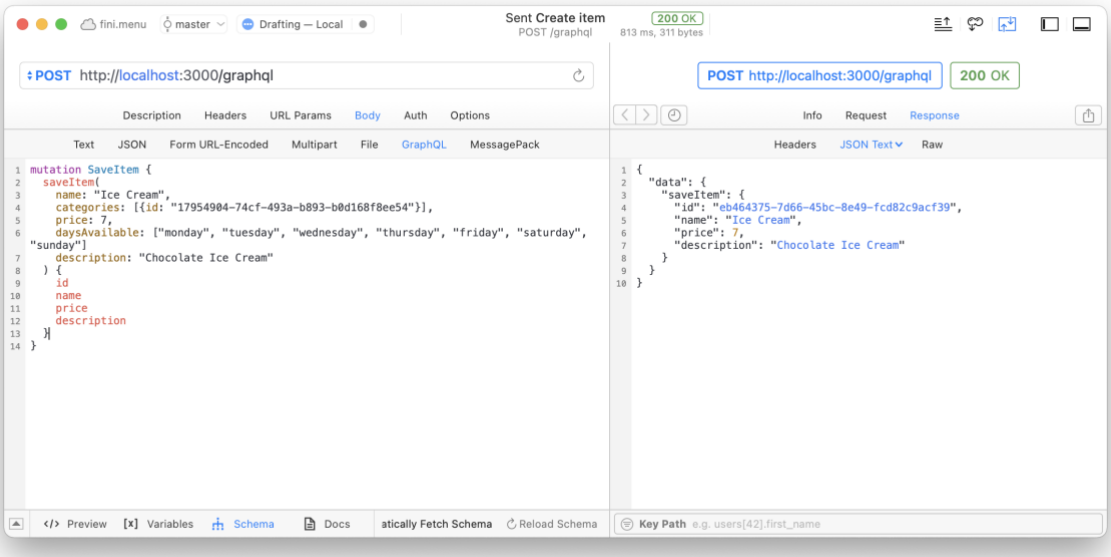
```
const { data, loading } = useQuery<CategoriesData>(GET_CATEGORIES_ITEMS);
```

Na istom zaslonu klikom na gumb Add New otvara se modal za dodavanje nove stavke kao na slici 47. U tom modalu mogu se ispuniti podaci o jelu kojeg želimo dodati, kao npr. naziv, opis, cijena, kategorije u koje želimo smjestiti to jelo itd.



Slika 47: Modal za kreiranje novog jela

Ovdje moramo pozvati mutaciju za spremanje novog jela koju smo nazvali `saveItem` (slika 48).



Slika 48: Izvršena mutacija za dodavanje novog jela

Ova mutacija funkcionira na način da ukoliko se proslijedi argument `id`, tada se jelo ažurira s podacima koji su poslani, a ukoliko nije proslijeđen argument `id` tada se u bazu podataka unosi novo jelo. Mutacija također ima argument `uploadFile` koji je tipa `Boolean`. Ukoliko se za ovaj

argument pošalje vrijednost `true`, tada će poslužitelj generirati S3 link kojeg će u odgovoru vratiti klijentu pomoću kojeg on potom može uploadati sliku direktno u S3 input bucket. Slijedi implementacija mutacije `saveItem` na poslužitelju:

```
@Mutation((returns) => Item)
async saveItem(@Args() args: SaveItemArgs): Promise<Item> {
  let uploadUrl = null;

  // If user is about to upload an image, generate S3 url
  if (args.uploadFile) {
    const { key, url } = await this.filesService.generateUrl('item');
    args.photo = key;
    uploadUrl = url;
    delete args.uploadFile;
  }

  const item = await this.itemsService.save(args);
  return { ...item, uploadUrl };
}
```

Mutacije se pomoću Apollo biblioteke izvršavaju slično kao i upiti - prvo moramo definirati mutaciju koju želimo poslati:

```
const SAVE_ITEM = gql`
  mutation SaveItem(
    $id: ID
    $uploadFile: Boolean
    $name: String!
    $categories: [CategoriesInput!]!
    $price: Float!
    $daysAvailable: [String!]!
    $description: String
  ) {
    saveItem(
      id: $id
      uploadFile: $uploadFile
      name: $name
      categories: $categories
      price: $price
      daysAvailable: $daysAvailable
      description: $description
    ) {
      id
      name
      description
      price
      uploadUrl
      photo
      rating
      daysAvailable
      categories {
        id
      }
    }
  }
`;
```

Za razliku od prijašnjeg upita `categories`, ovdje imamo argumente koje ćemo slati na poslužitelj pa ćemo tako definirati TypeScript tipove za njih i isto tako povratni tip podataka:

```
1   interface SaveItemData {
2     saveItem: Item;
3   }
4
5   interface SaveItemVars {
6     id?: string;
7     uploadFile: boolean;
8     name: string;
9     categories: { id: string }[];
10    price: number;
11    blurhash?: string;
12    daysAvailable: string[];
13    description?: string;
14  }
```

Tip `Item` na liniji 2 je već onaj koji smo ranije definirali za upit `categories`. Nakon toga koristimo useMutation hook iz Apollo biblioteke:

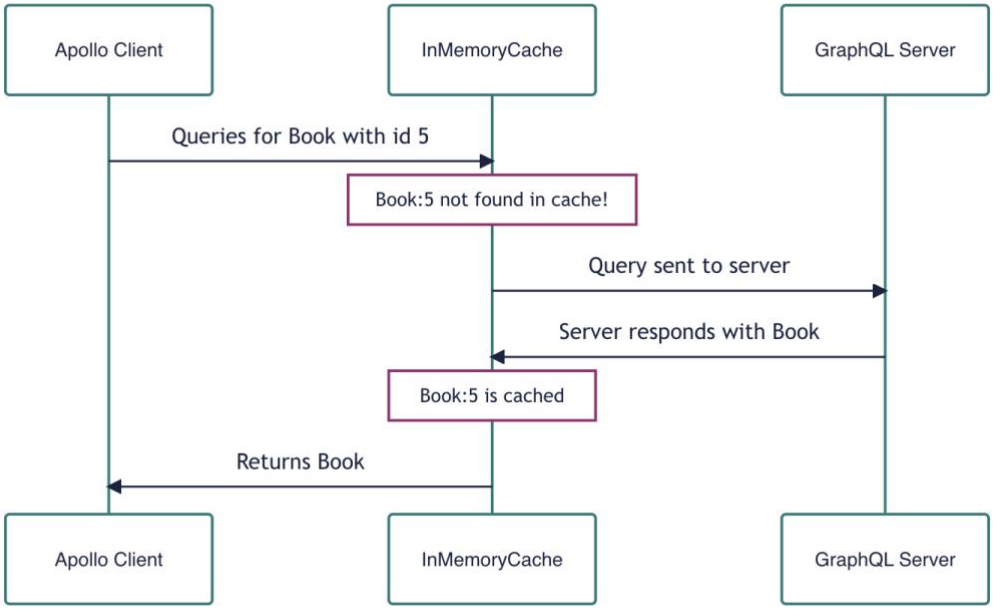
```
const mutation = useMutation<SaveItemData, SaveItemVars>(SAVE_ITEM);
```

Nakon što smo poslali zahtjev GraphQL poslužitelju za dodavanje novog jela, imamo dva načina kako dinamički prikazati to novo jelo u sučelju:

1. Ponovno slanje upita za dohvaćanje svih jela te će se tako prikazati novo dodano jelo
2. Dodavanje novo dodanog jela čije podatke je vratila `saveItem` mutacija u Apollo cache

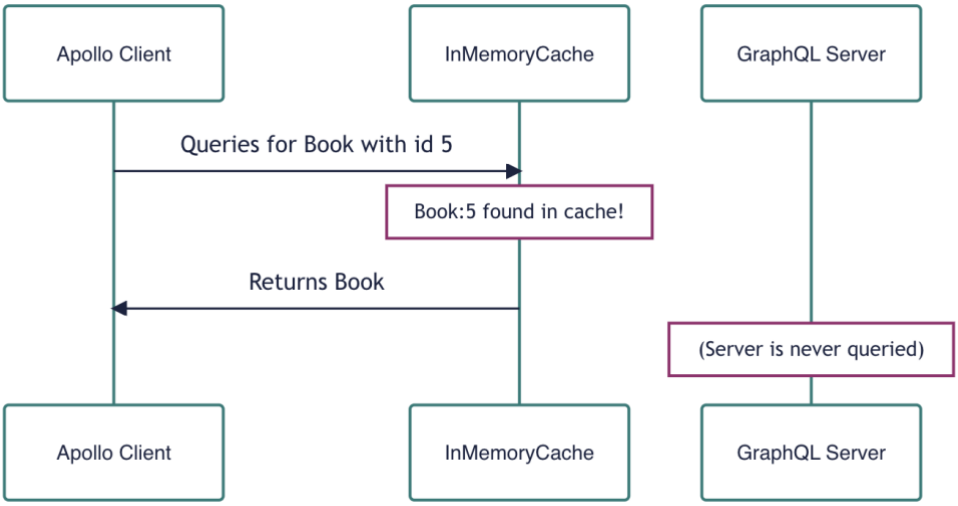
Prva opcija je manje efikasna i sporija pošto će biti potrebno poslati dodatan zahtjev poslužitelju nakon dodavanja novog jela. Druga opcija je optimalnija zato što ne moramo izvršiti dodatan zahtjev već je samo podatke koje smo dobili kao odgovor od `saveItem` mutacije potrebno dodati u Apollo cache. Apollo biblioteka pohranjuje GraphQL upite u lokalni normalizirani cache koji se pohranjuje u memoriji. To omogućuje gotovo instantno dohvaćanje podataka koji se već nalaze u cacheu bez potrebe za slanjem zahtjeva poslužitelju.

Uzmimo za primjer da dohvaćamo objekt Book s identifikatorom 5, tada će tijek izgledati kao na slici 49. Apollo će prvo provjeriti cache te ako se knjiga s identifikatorom 5 ne nalazi u njemu tada će poslati zahtjev poslužitelju. Nakon što odgovor sa poslužitelja pristigne, knjiga s identifikatorom 5 se sprema u memorijski cache. [38]



Slika 49: Prikaz rada Apollo cachea kada se zatraženi objekt ne nalazi u cache memoriji [38]

Idući puta kada će Apollo dohvaćati knjigu s identifikatorom 5 tijek će izgledati kao na slici 50. Apollo će provjeriti nalazi li se knjiga u memorijskom cacheu te će ju, pošto se nalazi, odmah vratiti bez slanja zahtjeva poslužitelju. Apollo cache ima mnogo konfigurabilnih opcija - možemo ga potpuno prilagoditi svojim potrebama. Dakle Apollo u bilo kojem trenutku u memoriji ima spremljene podatke koje smo već dohvatili s poslužitelja. Podaci koji se prikazuju u korisničkom sučelju se zapravo prikazuju iz memorijskog cachea.



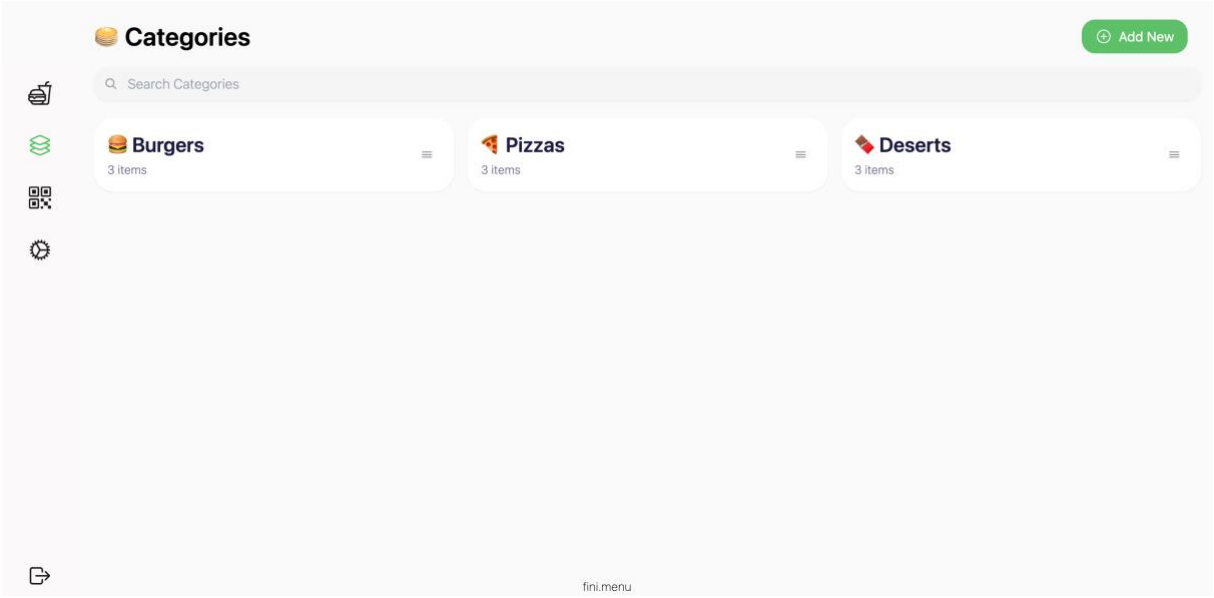
Slika 50: Prikaz rada Apollo cachea kada se zatraženi objekt nalazi u cache memoriji [38]

Pošto ćemo slijediti ranije spomenutu drugu opciju, morat ćemo ručno u Apollo cache dodati novo dodano jelo što je u nekim slučajevima izazovno no osigurava najbolji UX - jelo koje korisnik doda instantno će se prikazati u popisu jela. Ranije definirani `saveItem` upit ćemo tako izolirati u hook funkciju `useCreateItem` gdje ćemo dodati logiku za manipulaciju Apollo cacheom. Nakon primitka odgovora od poslužitelja izvršit će se funkcija u liniji 3. U liniji 4 dohvatit će se podaci o jelu koje je dodano te će se u linijama 8-23 to jelo dodati u Apollo cache. Potom se u linijama 27-42 prolazi svakom kategorijom u Apollo cacheu te se novo dodano jelo dodaje u tu kategoriju ako pripada u nju. Iako po količini koda izgleda kompleksno, logika za manipulaciju Apollo cachea u ovom slučaju dosta je jednostavna.

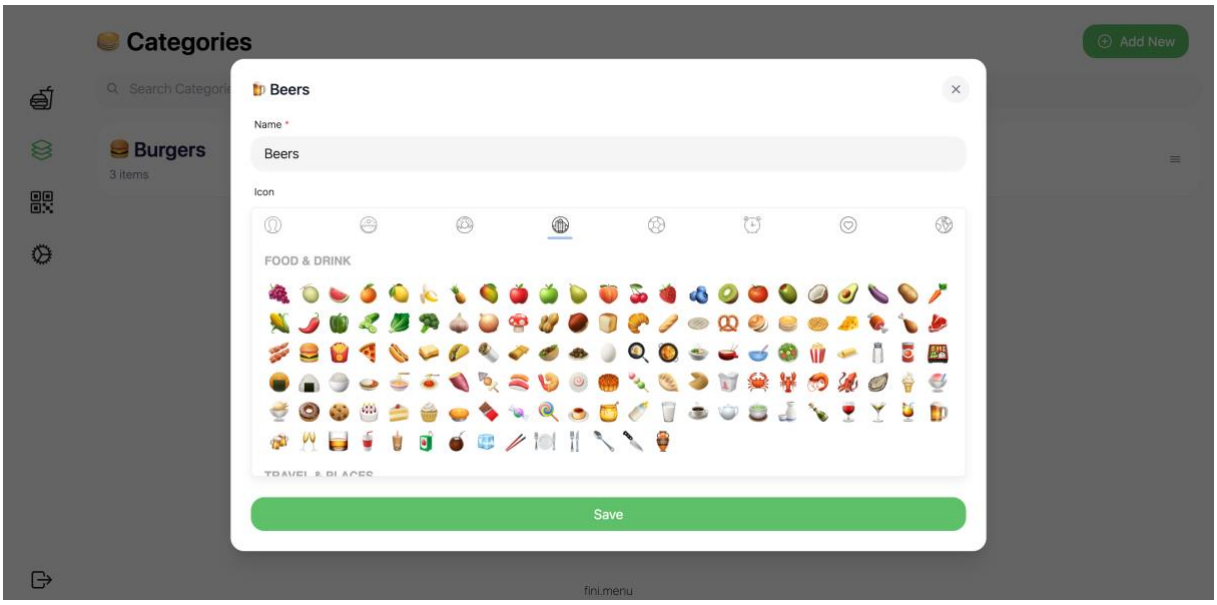
```
1   export const useCreateItem = () => {
2     const mutation = useMutation<SaveItemData, SaveItemVars>(SAVE_ITEM, {
3       update: (cache, { data }) => {
4         const newItem = data?.saveItem;
5
6         if (!newItem) return;
7
8         const newItemRef = cache.writeFragment({
9           data: newItem,
10          fragment: gql`
11            fragment NewItem on Item {
12              id
13              name
14              description
15              price
16              uploadUrl
17              photo
18              categories {
19                id
20              }
21            }
22          `,
23        });
24
25        if (!newItemRef) return;
26
27        for (const category of newItem.categories) {
28          cache.modify({
29            id: `Category:${category.id}`,
30            fields: {
31              items(existingItems: Reference[] = []) {
32                const newItems = [...existingItems];
33
34                // If we added a new item, add it to the array
35                if (!newItems.find((i) => i.__ref === newItemRef.__ref))
36                   newItems.push(newItemRef);
37
38                return newItems;
39              },
40            },
41         });
42        }
43      },
44    });
45    return mutation;
46  };
```

6.4.3. Pregled i uređivanje kategorija

Restorani također imaju opciju pregleda i uređivanja kategorija svog jelovnika (slika 51). Osim uređivanja i dodavanja novih kategorija (slika 52), također je moguće promijeniti redoslijed kategorija prema kojem se one prikazuju. Upiti za dohvaćanje kategorija i za spremanje kategorija izvršavaju se na jednak način kao i za stavke.



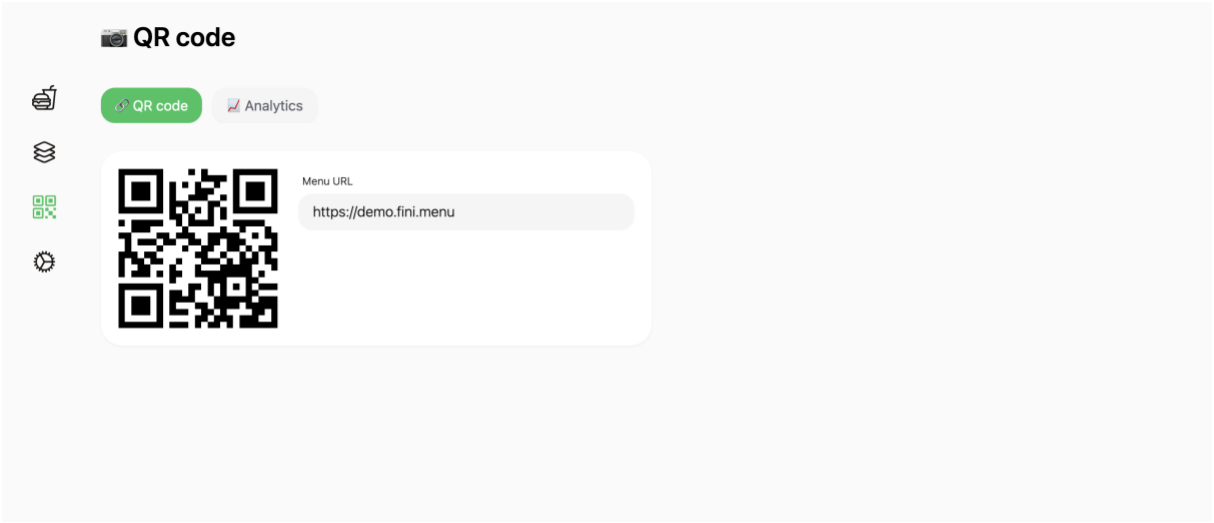
Slika 51: Zaslon s prikazom svih kategorija jelovnika



Slika 52: Modal za dodavanje nove kategorije

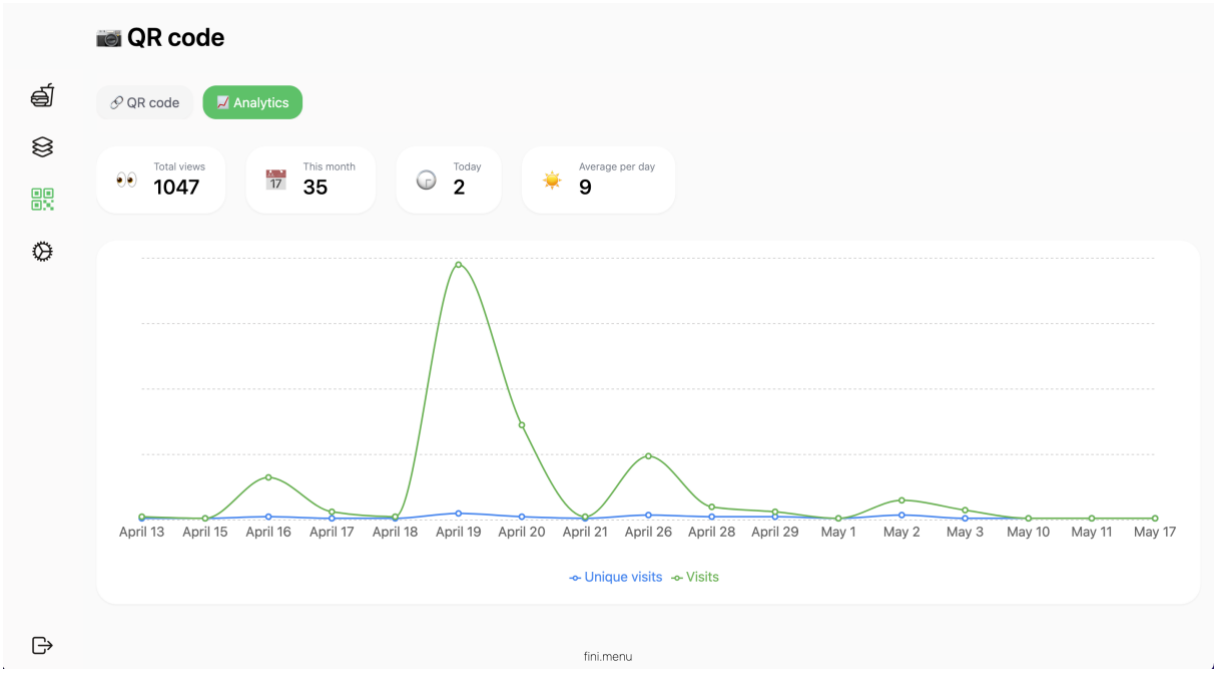
6.4.4. QR kod i analitika

Svaki restoran ima svoj QR kod koji vodi na njegov digitalni jelovnik do kojeg je također moguće doći preko URL-a koji se također vidi na zaslonu koji je prikazan na slici 53.



Slika 53: Prikaz QR koda restorana i URL-a koji vodi do jelovnika restorana

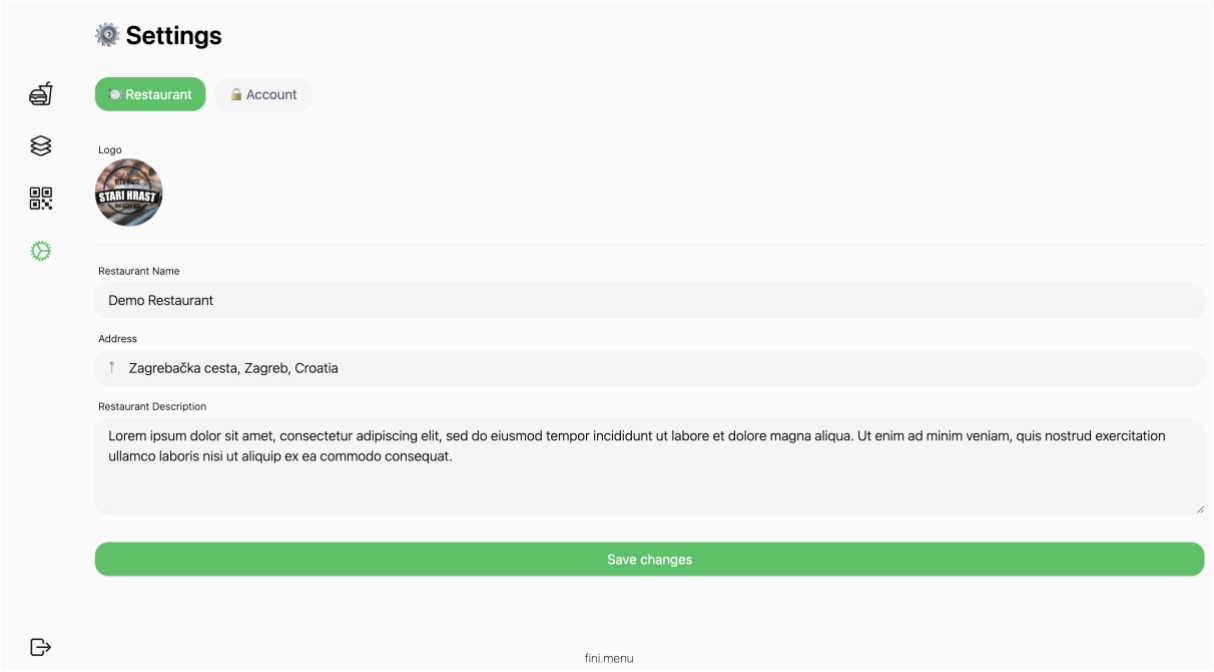
Osim QR koda, također je moguće vidjeti analitiku posjećenosti digitalnog jelovnika restorana (slika 54). Po danima je moguće vidjeti ukupan broj posjeta te broj jedinstvenih posjeta. Osim toga dostupni su i neki drugi podaci kao npr. ukupan broj pregleda jelovnika do sada, broj pregleda jelovnika za tekući mjesec, za trenutni dan te prosječan broj posjeta dnevno.



Slika 54: Prikaz analitike posjećenosti digitalnog jelovnika restorana

6.4.5. Postavke restorana

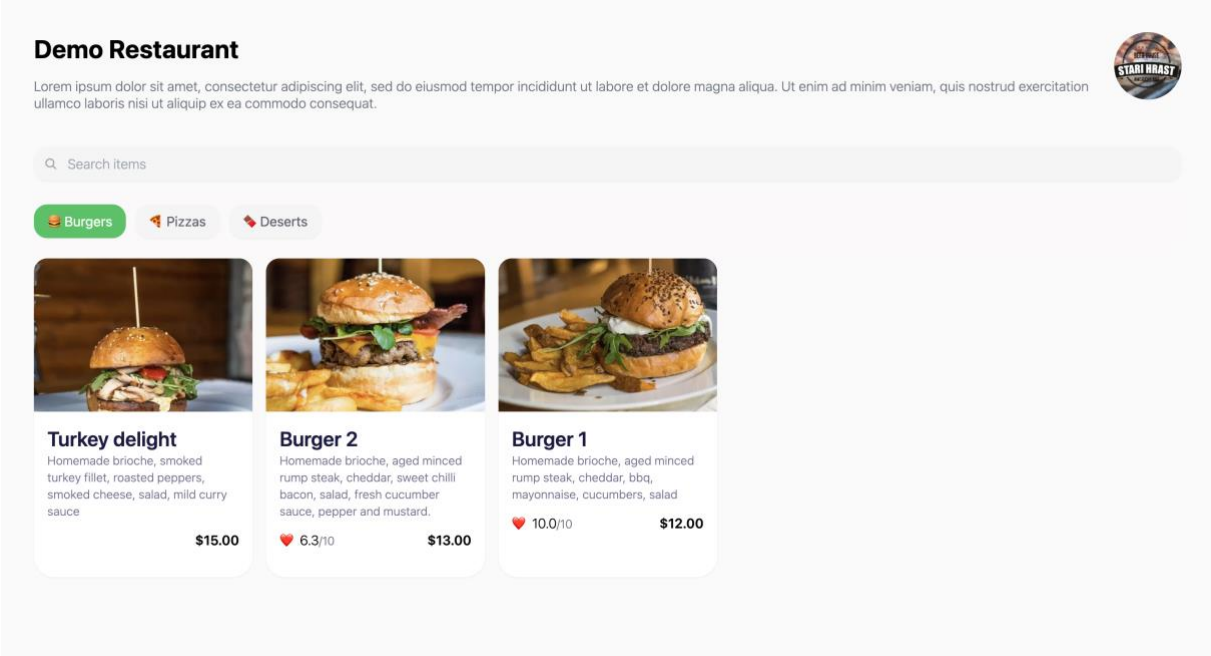
Restoran, osim postavki svog računa (promjena lozinke i emaila), ima mogućnost izmjene podataka o restoranu kao što je opis koji se prikazuje kada gost pomoću QR koda otvori digitalni jelovnik (slika 55).



Slika 55: Prikaz zaslona s postavkama restorana

6.4.6. Pregled jelovnika sa strane gosta

Skeniranjem QR koda koji se nalazi na stolu gostima se na mobitelu otvara pregled jelovnika restorana gdje mogu vidjeti sve kategorije i stavke kategorija te je isto tako moguće pretraživati jela (slika 56).



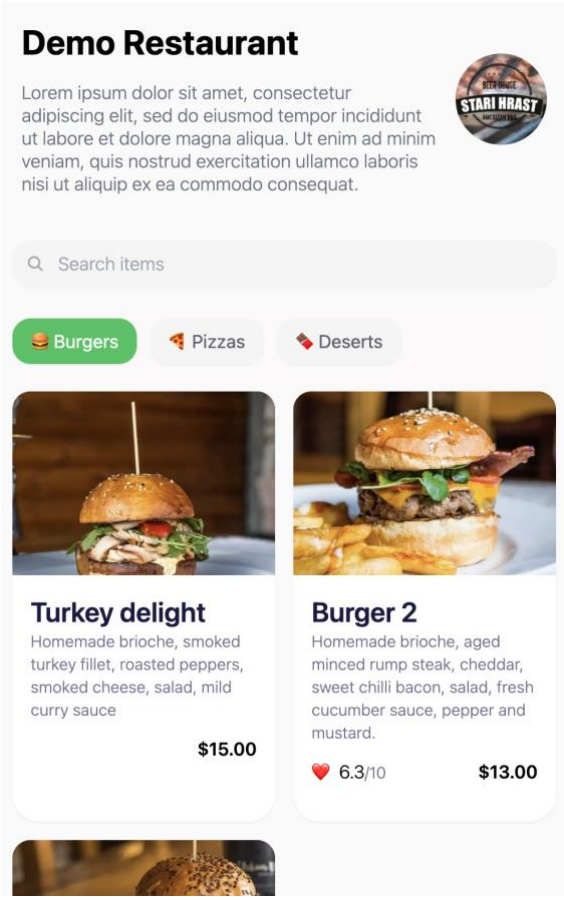
Slika 56: Prikaz jelovnika sa strane gosta

Kod svakog posjeta stranice s jelovnikom bilježi se analitika na način da se na REST servis šalje `POST` zahtjev na krajnju točku `/analytics` čija implementacija s back end strane slijedi u nastavku:

```
1   @Post()
2   async analytic(@Request() req: FastifyRequest, @Body() body: MarkAnalyticDto) {
3     const referer = req.headers.referer;
4     const userAgent = req.headers['user-agent'];
5     const ip = req.headers['remote-address'].toString();
6
7     const ua = UAParser(userAgent);
8
9     await this.analyticsService.create({
10      restaurantId: body.restaurantId,
11      deviceType: ua.device.type || 'desktop',
12      browserName: ua.browser.name,
13      os: ua.os.name,
14      referer,
15      ip,
16    });
17
18    return { success: true };
19  }
```

Ono što se ovdje događa jest dohvaća se URL stranice s koje je korisnik došao na stranicu jelovnika restorana (npr. ako je kliknuo na link jelovnika na nekoj društvenoj mreži), user agent zahtjeva te IP adresa zahtjeva (linije 3-5). Potom se u liniji 7 pomoću biblioteke `ua-parser-`

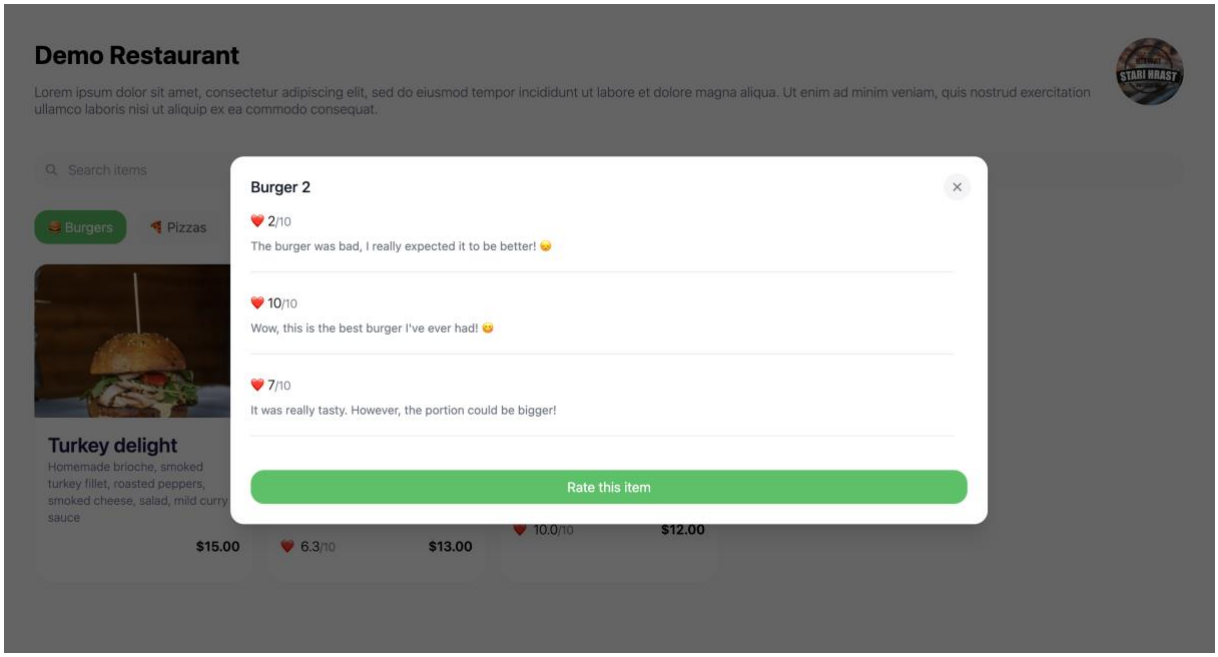
`js` na temelju user agenta izvlače neki podaci kao što su tip uređaja, naziv preglednika, operacijski sustav uređaja itd. Prikupljeni podaci potom se u linijama 9-16 spremaju u bazu podataka. Stranica je također prilagođena jednostavnijem pregledu na mobilnom uređaju (slika 57) s obzirom da će to biti najčešće kako će gosti pristupati jelovniku.



Slika 57: Prikaz jelovnika sa strane gosta na mobilnom uređaju

6.4.7. Sustav ocjenjivanja

Važan dio aplikacije je upravo sustav ocjenjivanja gdje gosti mogu ostavljati ocjene jelima sa jelovnika. Na stranici za pregled jelovnika moguće je za jela koja imaju ocjene vidjeti njihovu prosječnu ocjenu, a klikom na neko od jela moguće je vidjeti svaku ocjenu te njoj pripadajuću recenziju tog jela (slika 58).



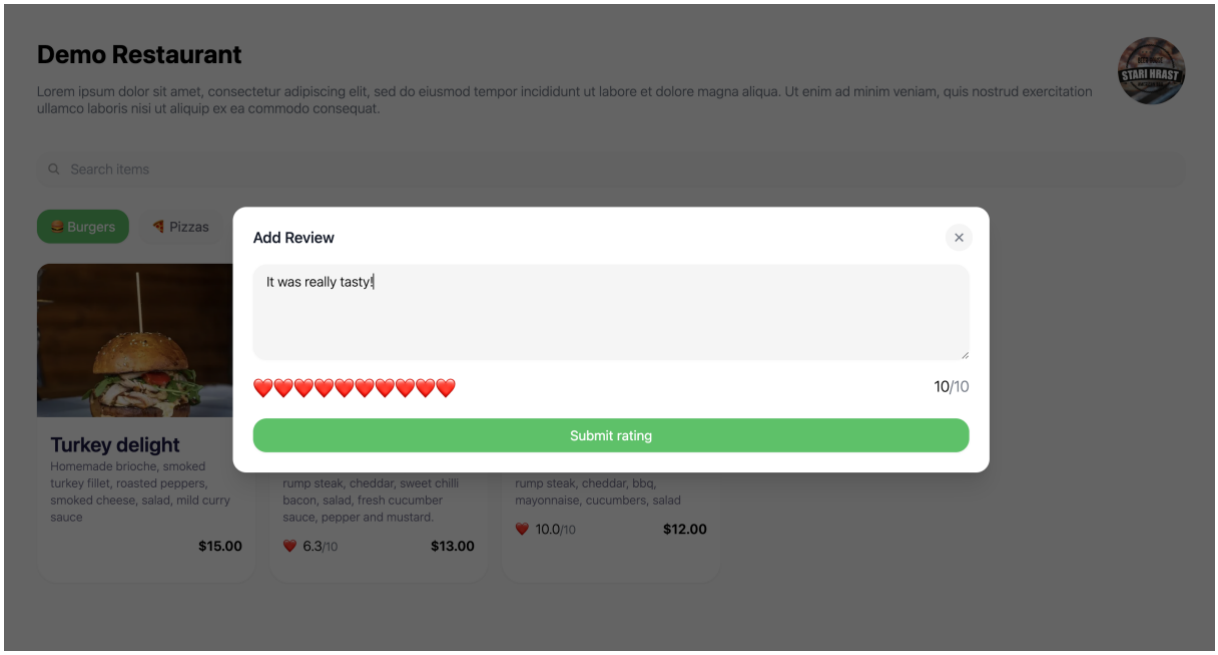
Slika 58: Prikaz ocjena odabranog jela

Za dohvaćanje ocjena i recenzija jela koristi se upit `item` za što je napravljena hook funkcija `useItemReviews`:

```
1    import { gql, useQuery } from "@apollo/client";
2    import { Rating } from "@features/ratings/types";
3
4    const GET_ITEM_REVIEWS = gql`
5      query GetItemReviews($itemId: ID!) {
6        item(itemId: $itemId) {
7          id
8          ratings {
9            id
10           stars
11           review
12         }
13       }
14     }
15   `;
16
17   interface Variables {
18     itemId: string;
19   }
20
21   interface Data {
```

```
22      item: {
23        id: string;
24        ratings: Rating[];
25      };
26    }
27
28    export const useItemReviews = (itemId: string) => {
29      return useQuery<Data, Variables>(GET_ITEM_REVIEWS, {
30        variables: {
31          itemId,
32        },
33      });
34    };
```

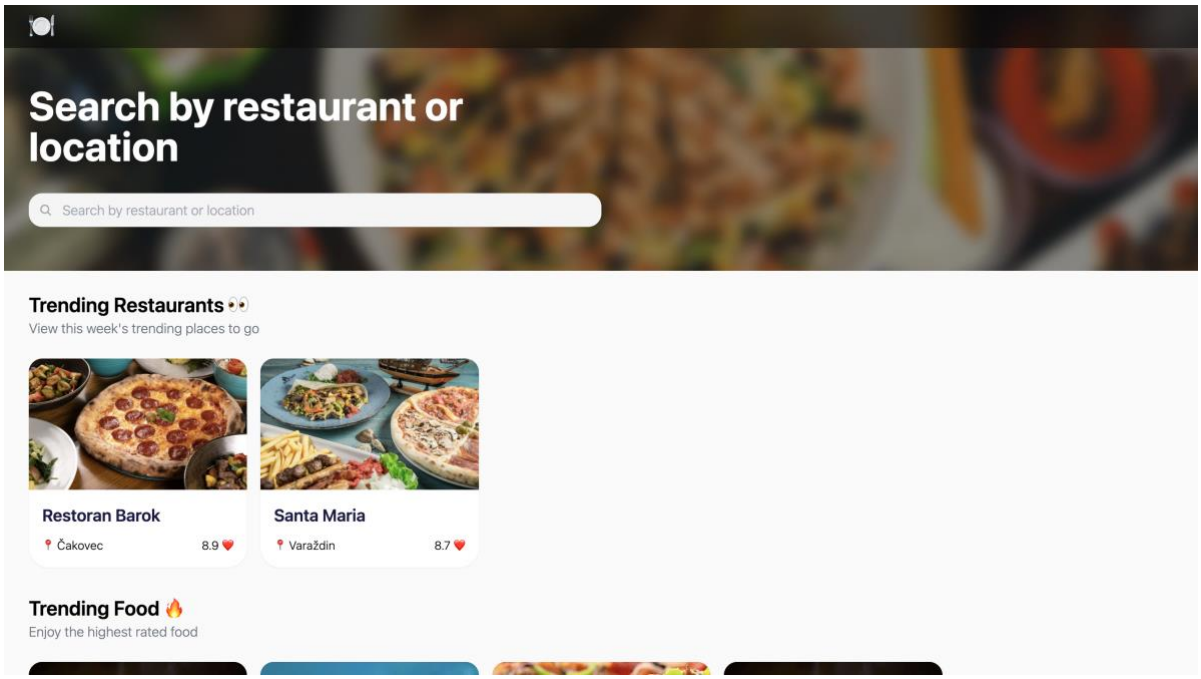
Korisnik također može dodati recenziju, odnosno ocjenu za neko jelo što je prikazano na slici 59. Pri slanju recenzije koristi se mutacija `rateItem`.



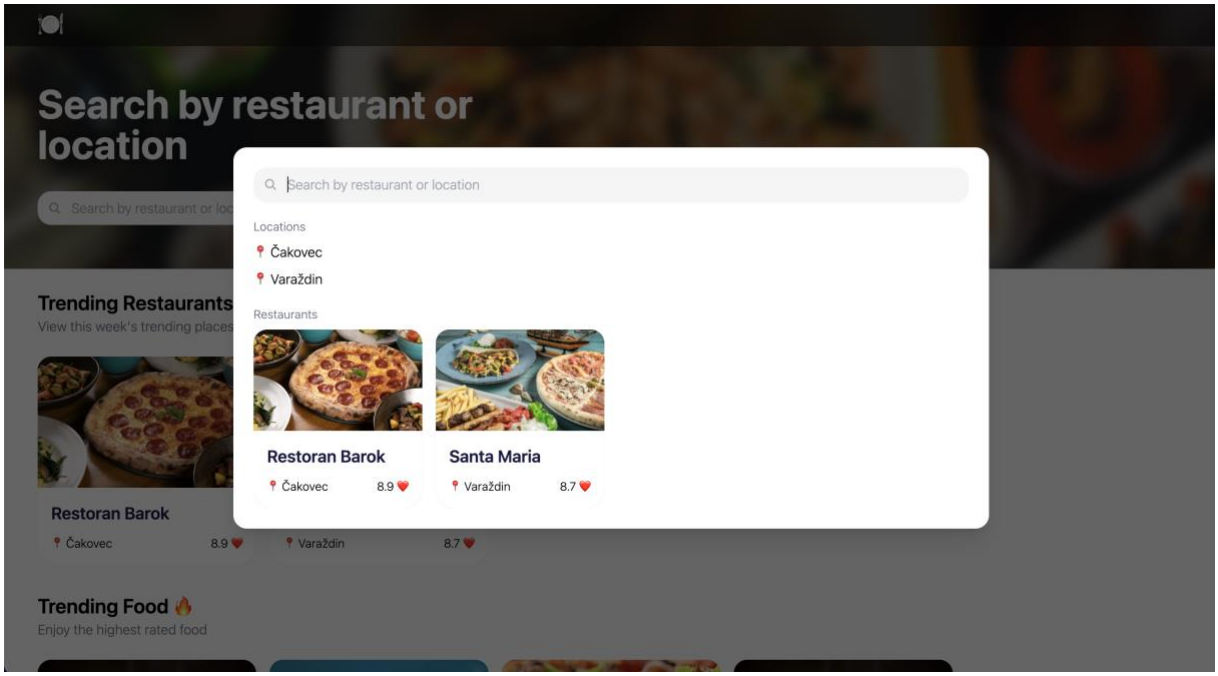
Slika 59: Dodavanje ocjene i recenzije jelu

6.4.8. Početna stranica

Sa strane korisnika, odnosno gosta, osim što je moguće vidjeti digitalni jelovnik nekog restorana također je moguće na početnoj stranici vidjeti trenutno najbolje ocijenjene restorane i najbolje ocijenjena jela (slika 60). Osim toga, restorane je moguće pretraživati prema nazivu ili prema lokaciji (slika 61).



Slika 60: Prikaz početne stranice aplikacije



Slika 61: Prikaz pretraživanja restorana na početnoj stranici prema nazivu ili lokaciji

7. Zaključak

Cilj ovog rada bio je pojasniti koncept implementacije i primjene GraphQL upitnog jezika pri izradi web servisa. Na početku rada su pojašnjeni neki osnovni koncepti modernog weba kao što su HTTP protokol, REST servisi koji se temelje na HTTP protokolu te potom sam GraphQL upitni jezik. Isto tako, spomenute su neke razlike između REST servisa i GraphQL servisa te navedene njihove prednosti i nedostaci. Potom su dotaknuti pojmovi računarstva u oblaku i arhitekture bez poslužitelja koji danas s razlogom privlače mnogo pažnje.

GraphQL, za razliku od drugih upitnih jezika kao što je SQL, nije jezik za izravnu komunikaciju s bazom podataka, već jezik koji definira pravila i oblik komunikacije klijenta s API poslužiteljem. GraphQL specifikacija je otvoreni standard koji opisuje pravila i karakteristike jezika te pruža upute za izvršavanje GraphQL upita. U usporedbi s REST servisima gdje se zahtjevi šalju na više krajnjih točaka, GraphQL upiti šalju se samo na jednu krajnju točku. Iako GraphQL rješava neka ograničenja REST servisa, i dalje postoje neki problemi na koje se često nailazi pri implementaciji - overfetching problem i N+1 problem. Ono što je najveća prednost GraphQL-a jest njegova fleksibilnost što timovima omogućuje brže iteracije pri razvoju proizvoda.

Upravljanje infrastrukturom često predstavlja velik izazov za organizacije pa se one osvrću na primjenu arhitekture bez poslužitelja što predstavlja model računarstva kojim brigu o održavanju infrastrukture propuštamo pružatelju usluga (npr. AWS). Na taj se način organizacije mogu više fokusirati na sam razvoj aplikacije te uz to iskoristiti mnoge druge prednosti ovakve arhitekture kao što su smanjeni troškovi, automatsko skaliranje itd.

Iako sam već bio upoznat s GraphQL upitnim jezikom, izradom ovog rada produbio sam znanje o samoj implementaciji GraphQL servisa te o načinima kako riješiti neke česte probleme koji se javljaju kod implementacije. Smatram da je GraphQL tehnologija koja nudi velike prednosti, no isto tako ona nije zamjena za REST arhitekturu već njoj alternativa. Obradom ovog aspekta weba naučio sam mnogo o GraphQL servisima i primjeni arhitekture bez poslužitelja što vjerujem da će mi uvelike doprinijeti u budućem razvoju web aplikacija.

Popis literature

[1] T. Rascia, "An Introduction to GraphQL", 2021. [Na internetu]. Dostupno na: https://www.taniarascia.com/introduction-to-graphql/ [Pristupljeno: 23-tra-2022]

[2] M. Landeiro, "Analysis of GraphQL performance: a case study" [Diplomski rad]. Instituto Superior de Engenharia de Porto, Portugal, 2019, Dostupno na: https://www.proquest.com/openview/6d7cffabf92955f2b102c11ff9ddfe4a [Pristupljeno: 25-tra-2022]

[3] "TypeScript" (bez dat.) [Na internetu]. Dostupno na: https://www.typescriptlang.org/ [Pristupljeno: 29-tra-2022]

[4] "Nest.js Introduction", 2021, [Na internetu]. Dostupno na: https://docs.nestjs.com/ [Pristupljeno: 29-tra-2022]

[5] "React" (bez dat.) [Na internetu]. Dostupno na: https://reactjs.org [Pristupljeno: 29-tra-2022]

[6] "Next.js Pages" (bez dat.) [Na internetu]. Dostupno na: https://nextjs.org/docs/basic-features/pages [Pristupljeno: 2-svi-2022]

[7] "AWS Cloud Products" (bez dat.) [Na internetu]. Dostupno na: https://aws.amazon.com/products [Pristupljeno: 2-svi-2022]

[8] "Serverless Framework Concepts", 2022. [Na internetu]. Dostupno na: https://www.serverless.com/framework/docs/providers/aws/guide/intro [Pristupljeno: 3-svi-2022]

[9] "What is HTTP? How it Works and Related Safety Concerns", 2021 [Na internetu]. Dostupno na: https://neeva.com/learn/what-is-http [Pristupljeno: 4-svi-2022]

[10] "What is HTTP?" (bez dat.) [Na internetu]. Dostupno na: https://www.cloudflare.com/en-gb/learning/ddos/glossary/hypertext-transfer-protocol-http/ [Pristupljeno: 5-svi-2022]

[11] "HTTP" (bez dat.) [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/HTTP [Pristupljeno: 5-svi-2022]

[12] "What is an API?" (bez dat.) [Na internetu]. Dostupno na: https://www.cloudflare.com/en-gb/learning/security/api/what-is-an-api/ [Pristupljeno: 6-svi-2022]

[13] L. Gupta, "REST Architectural Constraints", 2022. [Na internetu]. Dostupno na: https://restfulapi.net/rest-architectural-constraints/ [Pristupljeno: 8-svi-2022]

[14] "What is a REST API?", 2020 [Na internetu]. Dostupno na: https://www.redhat.com/en/topics/api/what-is-a-rest-api [Pristupljeno: 8-svi-2022]

[15] "What Is OpenAPI?" (bez dat.) [Na internetu]. Dostupno na: https://swagger.io/docs/specification/about/ [Pristupljeno: 10-svi-2022]

[16] B. Clark, "What is GraphQL: History, Components, and Ecosystem" [Na internetu]. Dostupno na: https://levelup.gitconnected.com/what-is-graphql-87fc7687b042 [Pristupljeno: 11-svi-2022]

[17] K. Stemmier, "What is GraphQL? GraphQL introduction", 2021. [Na internetu]. Dostupno na: https://levelup.gitconnected.com/what-is-graphql-87fc7687b042 [Pristupljeno: 12-svi-2022]

[18] "GraphQL schema basics" (bez dat.) [Na internetu]. Dostupno na: https://www.apollographql.com/docs/apollo-server/schema/schema [Pristupljeno: 13-svi-2022]

[19] S. Stubailo, "The Anatomy of a GraphQL Query", 2021. [Na internetu]. Dostupno na: https://www.apollographql.com/blog/graphql/basics/the-anatomy-of-a-graphql-query/ [Pristupljeno: 13-svi-2022]

[20] B. Nedelcu, "The Graph in GraphQL", 2020. [Na internetu]. Dostupno na: https://dev.to/bogdanned/the-graph-in-graphql-1l99 [Pristupljeno: 14-svi-2022]

[21] "Schemas and Types" (bez dat.) [Na internetu]. Dostupno na: https://graphql.org/learn/schema/ [Pristupljeno: 14-svi-2022]

[22] "Introspection" (bez dat.) [Na internetu]. Dostupno na: https://graphql.org/learn/introspection/ [Pristupljeno: 15-svi-2022]

[23] R. Derks, "Documenting GraphQL APIs", 2021. [Na internetu]. Dostupno na: https://hackernoon.com/documenting-graphql-apis [Pristupljeno: 16-svi-2022]

[24] L. Losoviz, "Code-first vs. schema-first development in GraphQL", 2020. [Na internetu]. Dostupno na: https://blog.logrocket.com/code-first-vs-schema-first-development-graphql/ [Pristupljeno: 16-svi-2022]

[25] N. Burk, "The problems of 'Schema-First' GraphQL Server Development", 2019. [Na internetu]. Dostupno na: https://www.prisma.io/blog/the-problems-of-schema-first-graphql-development-x1mn4cb0tyl3 [Pristupljeno: 17-svi-2022]

[26] "GraphQL is the better REST" (bez dat.) [Na internetu]. Dostupno na: https://www.howtographql.com/basics/1-graphql-is-the-better-rest/ [Pristupljeno: 17-svi-2022]

[27] “Introduction to cloud computing” (bez dat.) [Na internetu]. Dostupno na: https:// www.accenture.com/us-en/insights/cloud-computing-index [Pristupljeno: 18-svi-2022]

[28] “What is AWS?”, 2017. [Na internetu]. Dostupno na: https://intellipaat.com/blog/what-is-amazon-web-services-aws/ [Pristupljeno: 18-svi-2022]

[29] “Cloud Computing Models” (bez dat.) [Na internetu]. Dostupno na: https://aws.amazon.c om/types-of-cloud-computing/ [Pristupljeno: 18-svi-2022]

[30] N. Fee, "What Is Serverless Architecture? Key Benefits and Limitations", 2020. [Na internetu]. Dostupno na: https://newrelic.com/blog/best-practices/what-is-serverless-architecture [Pristupljeno: 18-svi-2022]

[31] “Amazon S3” (bez dat.) [Na internetu]. Dostupno na: https://aws.amazon.com/s3/ [Pristupljeno: 19-svi-2022]

[32] "AWS Lambda Concurrency: The Complete Guide" (bez dat.) [Na internetu]. Dostupno na: https://lumigo.io/aws-lambda-performance-optimization/aws-lambda-concurrency/ [Pristupljeno: 20-svi-2022]

[33] J. Beswick, "Operating Lambda: Performance optimitzation - Part 1", 2021. [Na internetu]. Dostupno na: https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-1/ [Pristupljeno: 20-svi-2022]

[34] "Nest.js Modules", 2022. [Na internetu]. Dostupno na: https://docs.nestjs.com/modules [Pristupljeno: 21-svi-2022]

[35] H. Adel, 2021. [Na internetu]. Dostupno na: https://unchained.shop/blog/n-plus-one-problem-in-graphql [Pristupljeno: 21-svi-2022]

[36] D. Agren, "How we came to create a new image placeholder algorithm, BlurHash", 2019. [Na internetu]. Dostupno na: https://blog.wolt.com/hq/2019/07/01/how-we-came-to-create-a-new-image-placeholder-algorithm-blurhash/ [Pristupljeno: 26-svi-2022]

[37] "BlurHash", 2022. [Na internetu]. Dostupno na: https://github.com/woltapp/blurhash/ [Pristupljeno: 26-svi-2022]

[38] "Caching in Apollo Client", 2022. [Na internetu]. Dostupno na: https://www.apollographql.com/docs/react/caching/overview/ [Pristupljeno: 21-svi-2022]

Popis slika