Izrada web aplikacije korištenjem programskom jezika Go, okvira gRPC i biblioteke React

Bogdan, Kevin

Master's thesis / Diplomski rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:076326

Rights / Prava: Attribution 3.0 Unported/Imenovanje 3.0

Download date / Datum preuzimanja: **2024-03-29**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Kevin Bogdan

Izrada web aplikacije korištenjem programskog jezika Go, okvira gRPC i biblioteke React

DIPLOMSKI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Kevin Bogdan

Matični broj: 46215/17–R

Studij: Informacijsko i programsko inženjerstvo

Izrada web aplikacije korištenjem programskog jezika Go, okvira gRPC i biblioteke React

DIPLOMSKI RAD

Mentor/Mentorica:

Prof. dr. sc. Danijel Radošević

Varaždin, kolovoz 2022.

Kevin Bogdan

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 aplikacija pomoću programskog jezika Go sa backend strane, baze podataka PostgreSQL, te biblioteka React.js i Netxt.js za frontend stranu. Kroz uvod objašnjene su općenite stvari vezane uz web aplikacije te razvoj istih. Nakon toga objašnjene su najvažnije stvari koje razlikuju programski jezik Go od ostalih programskih jezika, napravljene su usporedbe različitih protokola za komunikaciju: gRPC, REST i usporedba HTTP protokola (HTTP1 i HTTP2), te prednosti i mane istih. Slijedi uvod frontend dio gdje su objašnjene biblioteke React i Next.js i usporedba CSS-a sa SCSS-om. U drugom dijelu nalaze se opisi i primjeri implementacije izrađene web aplikacije oglasnika. Aplikacija ima veći fokus na samu implementaciju, odnosno sadrži manji broj različitih značajka, ali implementacija istih je odrađena na generički način sa fokusom na mogućnost ponovne upotrebe. Primjeri implementacije biti će usko povezani sa konceptima opisanima u teorijskom dijelu rada. Na samom kraju slijedi zaključak i osobno mišljenje na odrađenu temu.

Ključne riječi: web aplikacija, Go, React, Next.js, gRPC, HTTP, PostgreSQ

Sadržaj

Sadržaj iii
1. Uvod 1
2. Tehnologije 2
2.1. Go (Golang) 2
2.1.1. Goroutines 3
2.1.2. Channels i Mutex 4
2.2. JavaScript i TypeScript 5
2.3. React.js i Next.js 7
2.3.1. React.js 7
2.3.1.1. Redux 8
2.3.2. Next.js 9
2.3.3. Sass (SCSS) 11
2.4. RPC i gRPC 11
2.4.1. Protocol buffers 12
2.4.2. HTTP/2 14
3. Implementacija aplikacije 16
3.1. Klijent i poslužitelja 17
3.1.1. ERA 18
3.1.2. Forme 19
3.1.3. Autentifikacija 22
3.1.4. Middleware 23
3.1.5. Upravljačka ploča 25
3.1.6. Cache 30
3.1.7. Pred renderiranje 34
3.1.8. Internacionalizacija 36
4. Implementacija i analiza performansi 40

4.1. Node.js – REST 40
4.2. Go – REST 42
4.3. Go – gRPC 44
4.4. Analiza 45
5. Zaključak 47
6. Literatura 48
Popis slika 50
Popis tablica 51

1. Uvod

Kako danas sve više alata i programa prelazi na web, potreba za razvojem web aplikacija visokih performansi raste iz dana u dan. Jedan od pokazatelja je veliki razvoj novih biblioteka i alata za olakšani razvoj istih. Gotovo svi danas imaju pristupa internetu, velika većina ljudi se koristi internetom na dnevnoj bazi. Web aplikaciji može pristupati svako tko ima pristup internetskoj vezi, stoga smatram da je razvoj alata i programa kroz web aplikaciju puno učinkovitiji nego razvoj desktop aplikacija. Istovremeno razvoj desktop aplikacija zahtjeva razvoj aplikacija za više platforma: MacOS, Windows, Linux što povećava trošak razvoja, trajanje razvoja i otežava održavanje istog. Slična situacija je i kod mobilnih aplikacija, potrebno je podržati u većini slučajeva dvije platforme: iOS i Android.

Upravo ovi nabrojani nedostatci razvoja aplikacija za desktop ili mobilne platforme potaknule su me za razvojem web aplikacije. Kako su performanse bitan faktor za korisnika i korisničko iskustvo odlučio sam koristiti programski jezik Go koji je po performansama gotovo jednak programsku jeziku C ili C++ i okvir gRPC koji je nekoliko puta brži od njegovih alternativa REST i GraphQL.

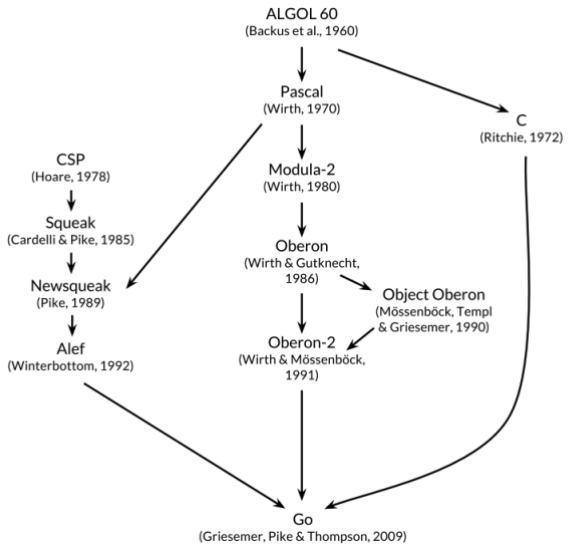
Kroz rad istražit će se teorijska osnova korištenih tehnologija, te će se potkrijepiti teorijska osnova kroz praktični dio, odnosno izradu same web aplikacije.

2. Tehnologije

Web aplikacija se može podijeliti na dva dijela, klijentski i poslužiteljski dio. Za izradu web sučelja (klijent) koristile su se tehnologije: TypeScript, React.js, Next.js i SCSS, a za izradu poslužitelja programski jezik Go. Klijent i poslužitelj komuniciraju preko gRPC okvira. Potrebno je istaknuti okvir Next.js koji je zapravo zaslužan za same funkcionalnosti na klijentskoj strani, te je važno napomenuti kako se on temelji na biblioteci React.js. Klijentski dio pisan je pomoću programskog jezika TypeScript. TypeScript je nad skup programskoj jezika JavaScript. Poslužitelj je zaslužan za komunikaciju sa bazom podataka, u ovom slučaju radi se o PostgreSQL bazi podataka.

2.1. Go (Golang)

Go je programski jezik koji su razvili Robert Griesemer, Rob Pike i Ken Thompson u Google-u u rujnu 2007., a najavljen je u studenom 2009. Go je brz, statički tipiziran, kompilirani jezik. Mehanizmi konkurentnosti olakšavaju pisanje programskog koda koji izvlači maksimalne performanse iz više jezgrinih sustava. Kompiliranje Go programskog koda je vrlo brzo što je još jedna od bitnih značajki programskog jezika. [1] Go je jezik opće namjene pa se samim time doista može koristiti u različitim domenama poput izgradnje poslužitelja, CLI aplikacija, strojnog učenja, DevOps-a. [2]



Slika 1. Graf izumitelja programskog jezika Go (izvor: https://books.google.hr/books?hl=en&lr=&id=SJHvCgAAQBAJ&oi=fnd&pg=PT8&dq=go+programming+language&ots=qAjGE_j1w7&sig=r46oN668IdUmNa5gv9dd5-8c8EU&redir_esc=y#v=onepage&q=go programming language&f=falsewri)

2.1.1. Goroutines

Konkurentnost u programiranju je sposobnost različitih dijelova programa da se izvršavaju izvan redoslijeda ili djelomičnim redoslijedom bez utjecaja na konačno rezultat, Go u sebi sadrži implementaciju modela paralelnosti. Svaka aktivnost koja se trenutno izvršava naziva se *goroutine*. Goroutina je lagana nit kojom upravlja Go runtime. U sekvencijalnom programu funkcije se izvršavaju jedna po jedna, odnosno kad jedna završi druga kreće sa izvršavanjem, ali u paralelnom programu se mogu izvršavati dvije ili više istovremeno, neovisno jedna o drugoj. [2]

Primjer dolje prikazuje dvije petlje od kojih jedna ispisuje brojeve bez goroutine, odnosno sinkrono, a druga koristi ključnu riječ 'go' prije poziva metode za ispis. Prvi ispis prikazuje brojeve od 0-25 i to pravilnim poretkom, iz razloga što se *ispisiBrojSinkrono* funkcija zove tek kada je prethodna izvršena. Kod drugog ispisa vidimo brojeve od 0-25 no bez pravilnog poretka, odnosno poredak se može promijeniti kod svakog izvršavanja programa, a razlog je upravo konkurentnost gdje se svih 25 poziva metode *ispisiBroj* poziva i izvršava gotovo istovremeno.

```
var wg sync.WaitGroup

func ispisiBrojSinkrono(broj int) {
  fmt.Printf("%v ,", broj)
}

func ispisiBroj(broj int) {
  fmt.Printf("%v, ", broj)
  wg.Done()
}

func main() {
  for i := 0; i < 25; i++ {
    ispisiBrojSinkrono(i)
  }

  for i := 0; i < 25; i++ {
    wg.Add(1)
    go ispisiBroj(i)
  }

  wg.Wait()
}
```

Ispis:

0 ,1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10 ,11 ,12 ,13 ,14 ,15 ,16 ,17 ,18 ,19 ,20 ,21 ,22 ,23 ,24

4, 0, 1, 2, 3, 20, 18, 19, 14, 13, 22, 21, 8, 5, 6, 7, 23, 10, 9, 11, 15, 16, 24, 12, 17

2.1.2. Channels i Mutex

Kanali služe kao kanali između Go rutina. Kanal je komunikacijski mehanizam koji služi za komunikaciju između goroutina. [2]

Prema zadanim postavkama slanje i primanje je blokirano sve dok druga srana nije spremna što dopušta goroutinama da se sinkroniziraju bez eksplicitnih zaključavanja varijabli [1].

- definiranje kanala: *kanal := make(chan string)*
- slanje vrijednosti na kanal: *kanal <- „Pozdrav“*
- prihvaćanje vrijednosti iz kanala: *poruka := <- kanal*

```
func sum(s []int, c chan int) {
  sum := 0
  for _, v := range s {
    sum += v
  }
  c <- sum
}
func main() {
  s := []int{1, 2, 3, -1, -2, -3}

  c := make(chan int)
  go sum(s[:len(s)/2], c)
  go sum(s[len(s)/2:], c)
  x, y := <-c, <-c

  fmt.Println(x, y, x+y)
}
```

Ispis: 6, -6, 0

Ukoliko nam komunikacija nije potrebno, već nam je jedino bitno da nam samo jedna goroutina pristupa varijabli kako bi se izbjegli konflikti koristimo koncept uzajamnog isključivanja (*mutual exclusion - mutex).*

```
type SafeCounter struct {
  mu sync.Mutex
```

```
  v  map[string]int
}

func (c *SafeCounter) Inc(key string) {
  c.mu.Lock()
  c.v[key]++
  c.mu.Unlock()
}

func (c *SafeCounter) Value(key string) int {
  c.mu.Lock()
  defer c.mu.Unlock()
  return c.v[key]
}

func main() {
  c := SafeCounter{v: make(map[string]int)}
  for i := 0; i < 1000; i++ {
    go c.Inc("somekey")
  }

  time.Sleep(time.Second)
  fmt.Println(c.Value("somekey"))
}
```

Ispis: 1000

Ukoliko ne bi koristili mutex u goroutinama, pomoću zastavice *-race* prilikom pokretanja programa dobili bi rezultat koji nije jednak očekivanom i sličnu poruku:

WARNING: DATA RACE

Read at 0x00c0000a6180 by goroutine 8

2.2. JavaScript i TypeScript

JavaScript je lagani, interpretirani ili pravodobno kompilirani programski jezik. JavaScript je najpoznatiji kao skriptni jezik za izradu web stranica, no danas mu to nije jedina namjena. Zahvaljujući Node.js-u, React.js-u (React Native) i ostalima, JavaScript se koristi i za izradu poslužitelja, mobilnih aplikacija (iOS i Android), pa čak i za strojno učenje. [7]

TypeScript je nastao zbog nedostatka strogih tipova u JavaScriptu. TypeScript je nad skup JavaScript-u pa mu tako dodaje dodatnu sintaksu koja omogućava lakše uočavanje pogrešaka

kroz editor, kod se pretvara u JavaScript kod pa samim time se može koristiti bilo gdje umjesto JavaScript-a.[8]

Kroz primjer možemo vidjet njihove razlike i objasniti zašto je TypeScript bolji prilikom rada u većem timu i pri smanjenju pogrešaka. Prvi primjer prikazuje kod u programskom jeziku JavaScript. Važno je uočiti primjer gdje kreiramo objekt *user2* koji sadrži neke podatke koji opće nisu bitno za metodu *printUserData* koja zapravo samo ispisuje ime i prezime korisnika. Prilikom izvršavanje ovog koda dobivamo *undefined* vrijednost koja može često uzrokovati pad aplikacije. Također nemamo nikakvih upozorenja prilikom pozivanja metode sa krivim parametrima.

```
const printUserData = (user) => {
  console.log("User's first and last name: ", user.firstName, " ", user.lastName);
};

const user = {
  firstName: "Foo",
  lastName: "Bar",
};
const user2 = {
  random: "data",
};

printUserData(user);
printUserData(user2);
```

U primjeru ispod koristimo TypeScript. Razlika u kodu nije prevelika, no ipak je ima, dodano je sučelje *User* koji sadrži dva polja *firstName* i *lastName* koji su tipa *string* i to sučelje pridodajemo parametru *user* u metodu *printUserData*, te varijablama *user* i *user2*. Nakon toga greška se već pojavljuje u samom IDE-u koja nam govori kako *user2* nema polja koja su mu propisana preko sučelja. Ukoliko bi maknuli sučelje od varijable *user2* i pokušali tako pozvati metodu, IDE bi javio grešku da varijabla nema potpis koji je potreban metodi.

```
interface User {
  firstName: string;
  lastName: string;
}

const printUserData = (user: User) => {
  console.log(
    "User's first and last name: ",
```

```
    user.firstName,
    " ",
    user.lastName
  );
};

const user: User = {
  firstName: "Foo",
  lastName: "Bar",
};

const user2: User = {
  random: "data",
};

printUserData(user);
printUserData(user2);
```

2.3. React.js i Next.js

Zajednička stvar React.js-u i Next.js-u je njihova namjena, odnosno koriste se u istu svrhu, za izgradnju interaktivnih i dinamičkih web aplikacija. U pozadini koriste programski jezik JavaScript ili njegov nad skup TypeScript.

2.3.1. React.js

React.js je JavaScript biblioteka za izradu korisničkih sučelja, nastala 2011. od strane Facebook-a. [3] Neke specifičnosti React.js-a su virtualni DOM i JSX ekstenzija. Kako je DOM manipulacija skupa, cilj biblioteke je bio smanjiti broj manipulacija na minimum, to su postigli virtualnim DOM-om i samim time obrađuje minimalni broj operacija na stvarnom DOM-u. JSX (Javascript Syntax Extension) je XML-proširenje koji omogućuje pisanje JavaScript-a i HTML-a u istoj datoteci, a preprocesori kao što su Babel, Webpack zatim prevode u JavaScript. [5]

Jedan od ciljeva u programskim rješenjima je ne ponavljanje koda, što je React.js uspio sa uvođenjem komponenata. Komponente su manji dijelovi web aplikacije (npr. gumb, forma,

navigacijska traka) koji jednom kada su kreirani mogu biti ponovo iskorišteni u bilo kojem dijelu aplikacije. Kako bi aplikacije bile interaktivne, odnosno kako bi promjene podataka uzrokovane od strane korisnika bile vidljive, React.js je uveo *State*. *State* je zapravo objekt koji sadrži neke podatke, *State* je definiran unutar same komponente. Objekti koji se dijele između komponenata nazivamo *Props*. Takvi objekti se mogu samo dijeliti od vrha prema dnu, odnosno roditeljska komponenta može podijeliti objekt samo sa svojim podređenim komponentama. Na prvu se možda ova rečenica čini uredu, no upravo zbog ovakvog načina dijeljenja podataka između komponenata nastaje problem kod većih aplikacija, gdje jedan objekt treba koristiti velik broj komponenata, koje ni ne moraju biti podređene komponenti koja sadrži taj objekt, dolazimo do problema upravljanja stanja (state management). Kako riješiti taj problem? Redux. Još jedan problem React.js biblioteke je što se u potpunosti renderira na strani klijenta u web pregledniku. Iz tog razloga SEO i Google rangiranje su prilično loši zbog nepostojećih HTML elemenata (npr. h1, h2, meta) koji zapravo služe Google-u za prepoznavanje namjene web aplikacije. Taj problem rješava Next.js.

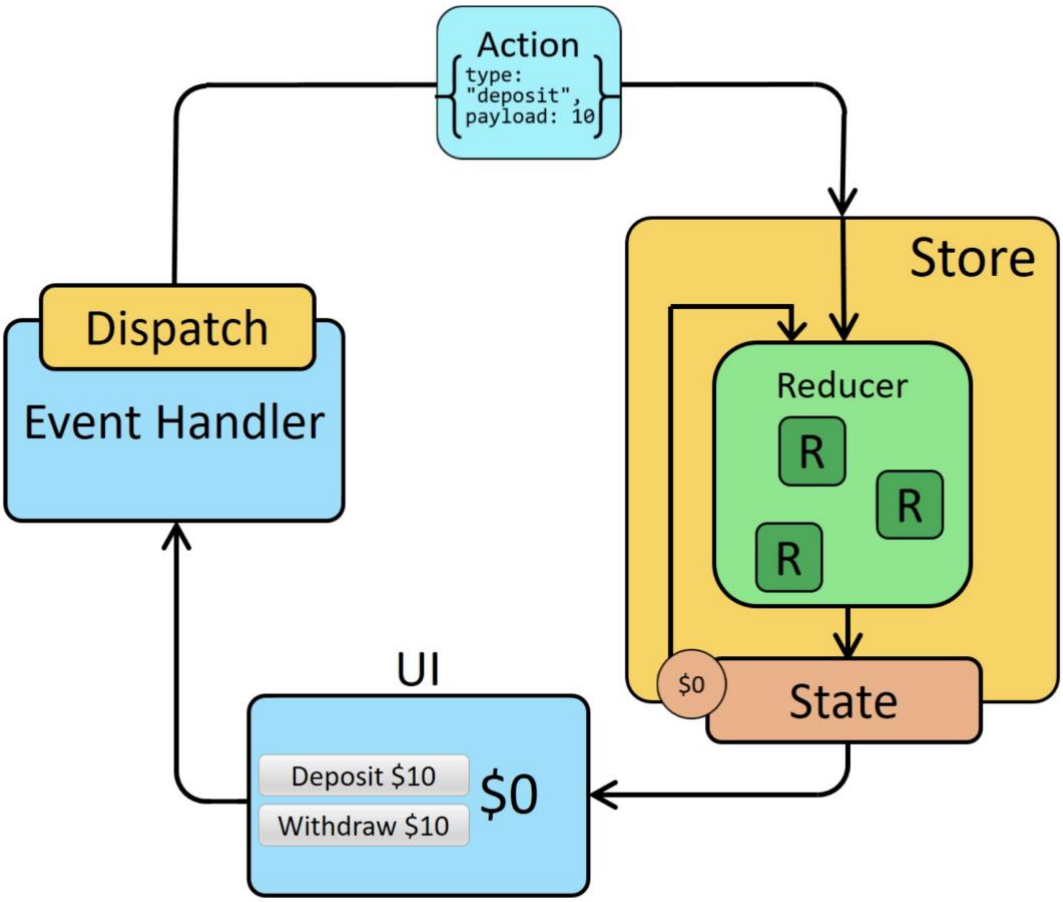
2.3.1.1. Redux

Redux je biblioteka za upravljanje i centralizaciju stanja. [6] U izradi web aplikacije koristio sam Redux, točnije Redux Toolkit koji je nastao nakon Redux-a. Za samo lakše otklanjanje pogrešaka koristio sam Redux DevTools, ekstenziju za pretraživač Google Chrome. Ekstenzija služi za pregled stanja, izvršavanje akcija, testiranja.

Redux Toolkit je nastao kako bi riješio probleme Redux-a:

- olakšano postavljanje trgovine (store)
- smanjen broj paketa
- smanjenje ponavljajućeg koda

Redux se sastoji od akcija, reduktora i trgovine (*actions, reducers, store)*. Svaka komponenta pristupa pojedinom stanju unutar trgovine pomoću selektora – rješavamo problem potrebe dijeljenja objekata direktno iz roditeljske komponente u djecu. Svaka komponenta može promijeniti stanje slanjem akcije na temelju koje reduktor napravi promjenu stanja u trgovini.



Slika 2. Životni ciklus Redux-a

2.3.2.Next.js

Next.js je razvojni okvir i temelji se na biblioteci React.js. Prema službenoj stranici, Next.js pruža najbolje iskustvo razvoja web aplikacija uz sve značajke koje su potrebne za razvoj istih: hibridno statičko i poslužiteljsko renderiranje, podrška za TypeScript, pametno grupiranje, prethodno dohvaćanje rute. Osim navedenih značajaka, Next.js sadrži još nekolicinu alata: automatska optimizacija slika, internacionalizacija (domain ili subdomain),

analitici, optimiziranost produkcijskog koda bez dodatnih konfiguracija, SSG i SSR, usmjeravanje pomoću datotečnog sustava. [5]

Kako je prije opisano, React.js ima SEO problem, Next.js iako je baziran na React.js-u sadrži mogućnost renderiranja stranica u vrijeme izrade (SSG) ili u vrijeme zahtjeva (SSR), što bi značilo da se umjesto praznog HTML-a koja sadrži samo *body* oznaku, web preglednik dobiva cijelu HTML strukturu sa svim pripadnim HTML elementima.

Usmjeravanje (routing) pomoću datotečnog sustava zapravo znači da za razliku od pisanja ruta pomoću koda kao što je to u React.js-u, Next.js automatizirano prepoznaju rute na temelju datotečnog sustava. Kada je datoteka dodana unutar 'pages' direktorija, ona automatski postaje dostupna kao nova ruta. Za navigaciju koristi se komponenta *Link* koja je dostupna unutar paketa *next/link.* Kroz primjere možemo prikazati nekoliko slučajeva.

U primjerima će se koristiti imaginarna domena *oneplace.hr*.

- za rutu *oneplace.hr/oglasi/auto-moto* potrebno bi bilo dodati direktorij *oglasi* unutar direktorija *pages*, a zatim dodati datoteku *auto-moto.tsx* unutar direktorija *oglasi*
    - *pages/oglasi/auto-moto.tsx -> oneplace.hr/oglasi/auto-moto*
- za rute s dinamički parametrima, npr. *oneplace.hr/oglas/auto-moto/10,* gdje bi broj 10 predstavljao ID automobila i koji je zapravo dinamički podatak, potrebno bi bilo kreirati direktorij *oglas*, unutar tog direktorija kreiramo direktorij *auto-moto* te na samom kraju datoteku *[id].tsx*. Vrijednost unutar zagrada označava dinamički segment rute.
    - pages/oglas/auto-moto/[id].tsx -> oneplace.hr/oglas/auto-moto/:id (oneplace.hr/oglas/auto-moto/10)

Next.js *Image* komponenta dostupna u paketu *next/image* je proširenje HTML elementa <*img*>. Uključuje razne ugrađene optimizacije koje pomažu pri postizanju dobrih rezultata kod alata poput Lighthouse. Lighthouse je alat za mjerenje kvalitete, brzine i točnosti web aplikacija. Neke od ugrađenih optimizacija [5]:

- poboljšanje performansa posluživanjem slike prilagođene veličine na temelju uređaja
- automatsko sprječavanje kumulativnog pomicanja izgleda
- učitavanje slika samo kada su u okviru za prikaz
- promjena veličine slika na zahtjev, čak i za slike pohranjene na udaljenim poslužiteljima

2.3.3. Sass (SCSS)

Sass (Syntactically Awesome Style Sheet) je preprocesorski skriptni jezik koji se kompilira u CSS (Cascading Style Sheets), dok je CSS stilski jezik koji se koristi za opisivanje vizualnog dijela dokumenta napisanog u HTML-u. [10] Kako se navodi u dokumentaciji, Sass omogućuje korištenje varijabli, ugniježdenih pravila, mixina, funkcija i još mnogo toga što nije dostupno u CSS-u. [11]

Slika ispod prikazuje primjer koda na temelju kojeg možemo vidjeti jednu od prednosti SCSS-a, a to je da dinamički možemo definirati različita pravila što smanjuje ponavljanje istog koda i pojednostavljuje čitanje koda.

SCSS

```scss
@mixin define-emoji($name, $glyph) {
  span.emoji-#{$name} {
    font-family: IconFont;
    font-variant: normal;
    font-weight: normal;
    content: $glyph;
  }
}

@include define-emoji("women-holding-hands", "👭");
```

CSS

```css
@charset "UTF-8";
span.emoji-women-holding-hands {
  font-family: IconFont;
  font-variant: normal;
  font-weight: normal;
  content: "👭";
}
```

Slika 3. Primjer koda u SCSS i CSS (izvor: https://sass-lang.com/)

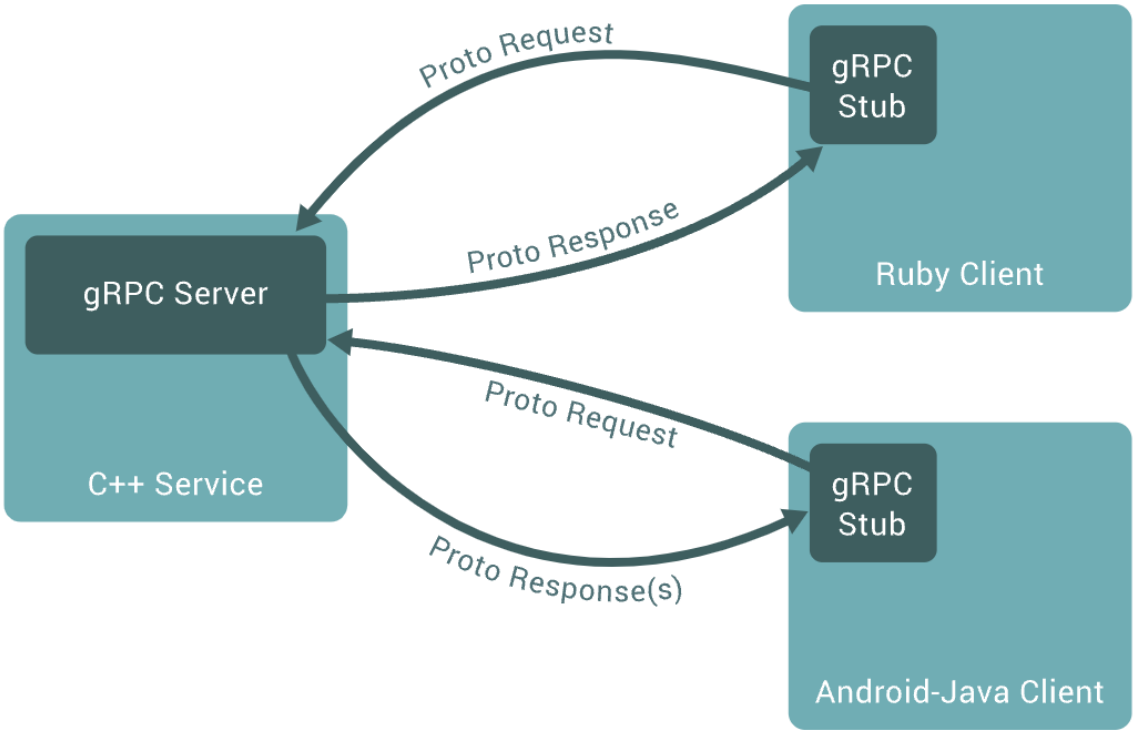
2.4. RPC i gRPC

RPC je komunikacijski protokol koji pozivanjem procedure za izvršenjem u drugom adresnom prostoru radi na način kao da je lokalni poziv procedure, bez da se eksplicitno kodiraju detaljni za udaljenu interakciju, a razvijen je od strane Bruce Jay Nelsona 1981. godine. Koristi model klijent-poslužitelj, gdje klijent šalje zahtjev, a poslužitelj vraća odgovor. [12]

GRPC je moderan RPC okvir koji može raditi u bilo kojem okruženju. Učinkovito povezuje usluge unutar i između podatkovnih centara s podrškom za balansiranje opterećenja, praćenje, provjeru ispravnosti i autentifikaciju, također primjenjiv je i u distribuiranim sustavima. [13] Što je zapravo pogodno za današnje potrebe iz razloga što se danas softverski sustavi temelje na distribuiranim sustavima koji međusobno komuniciraju.

Pri razvijanju gRPC aplikacije potrebno je definirati servisno sučelje. Definicija sučelja sadrži informacije o parametrima metoda, formate poruka koje se koriste pri pozivanju i pri vraćanju odgovora metoda. Jezik za definiranje poznat je kao IDL. Iz definiranih usluga može se generirati kod na strani poslužitelja, no isto tako i na strani klijenta neovisno o programskom jeziku koji se nalazi na jednog ili drugoj strani. [13]

Slika ispod prikazuje primjer komunikacije gRPC poslužitelja i gRPC klijenta (Stub), također vidljivo je da su korišteni različiti programski jezici za pojedini servis, C++ kao poslužitelj, a Ruby i Android-java kao klijenti.



Slika 4. Tok komunikacije gRPC (izvor: https://grpc.io/)

Osim standardnog toka komunikacije gdje klijent šalje zahtjev, a poslužitelj vraća odgovor, gRPC nudi 4 vrste metoda [13]:

- unary – klijent šalje zahtjev poslužitelju i dobiva natrag odgovor, kao uobičajeni poziv funkcije
- server streaming – klijent šalje zahtjev poslužitelju i dobiva stream za čitanje niza poruka natrag, istovremeno se zadržava redoslijed poruka unutar poziva
- client streaming – klijent kreira niz poruka i šalje ih poslužitelju, nakon što klijent završi, poslužitelj vraća svoj odgovor, isto tako redoslijed poruka je zadržan
- bidirectional streaming – obje strane šalju slijed poruka pomoću toka čitanja i pisanja neovisno jedno o drugom, poruke se mogu slati naizmjenično bez nekog strogog pravila, no i dalje je redoslijed poruka zadržan

Prema zadanim postavkama gRPC koristi među spremnike protokola (protocol buffers), Google-ov mehanizam za serijalizaciju strukturiranih podataka, iako se može koristiti s drugim formatima podataka kao što je JSON. [13]

2.4.1. Protocol buffers

Kao što je već navedeno, među spremnici su Google-ov mehanizam neutralan u odnosu na jezik ili platformu, ujedno je i manji, brži i jednostavniji od ostalih formata. Kroz nekoliko koraka objašnjeni su najvažniji pojmovi koji služe za definiranje servisa, a definiranje istih se obavlja u datotekama sa ekstenzijom *.proto*. Definicija poruke započinje ključnom riječi *message*, nakon koje slijedi ime, a polja u poruci započinju tipom polja (npr. string, int32) te imenom polja nakon kojeg slijedi znak '=' i indeks polja, odnosno kroz primjer: *tip ime = indeks*. Definicija servisa započinje ključnom riječi *service*, pa zatim slijedi ime servisa. Servis sadrži u sebi RPC definicije koje izgledaju slično standardnim metodama u programskim jezicima, a format je sljedeći: *rpc ime (Zahtjev) returns (Odgovor){}.*

U ovom primjeru definiran je servis za dohvaćanje svih učenika za određeni razred. Na samom početku definiramo format poruke, odnosno objekt koji će predstavljati učenika, poruka *Ucenik*. Ova poruka će se kasnije koristiti u odgovoru od strane poslužitelja.

```
message Ucenik {
  string ime = 1;
  string prezime = 2;
  ...
  int32 razred_id = 3;
}
```

Isto tako definiramo poruku *Razred*, sa poljem *razred_id* kako bi poslužitelj znao za koji želimo dohvatiti učenike.

```
message Razred {
  int32 razred_id = 1;
}
```

Kako želimo da nam metoda vrati listu učenika, potrebno je definirati još jednu poruku *Ucenici*. Osim formata koji je naveden iznad, polje može sadržavati i opcionalni dio koji se nalazi na samom početku definicije polja. Opcionalni dio može biti:

- singular – polje se može ponoviti nula ili jedan puta, istovremeno to je i zadano pravilo ukoliko se ne specificira drugačije
- repeated – polje se može ponoviti nula ili više puta, gdje se poredak zadržava

```
message Ucenici {
  repeated Ucenik ucenici = 1;
}
```

Nakon definiranje poruka, slijedi definicija servisa, u ovom primjeru kreiran je servis *Skola* sa metodom *DohvatiUcenikeZaRazred*. Metoda *DohvatiUcenikeZaRazred* prima kao parametar poruku *Razred* putem koje saznaje *ID* razreda za koji želimo dohvatiti učenike, a kao odgovor vraća poruku *Ucenici*.

```
service Skola {
  rpc DohvatiUcenikeZaRazred(Razred) returns (Ucenici) {}
}
```

Za generiranje koda koristi se alat Protoc. Primjer komadne za generiranje Go koda.

```
protoc --go_out=$mydir --go-grpc_out=$mydir $name.proto
```

2.4.2. HTTP/2

Povijest HTTP protokola započinje 1991. gdje se prvi put upoznajemo sa verzijom HTTP/0.9, nedugo nakon toga dolazi HTTPS koja se koristi sa SSL certifikatima, te nakon toga HTTP/1.0 i HTTP/1.1 koji sadrži značajke koje koristimo danas, poput trajnih veza, pred memoriranje, objekt zaglavlja, slanje u komadima. Zatim 2015. dolazi HTTP/2 sa svojim značajkama i ciljem za smanjenjem trajanja učitavanja web aplikacija bez ljudskih napora u smislu razvoja programa. Neke od važnijih značajki HTTP/2 [15]:

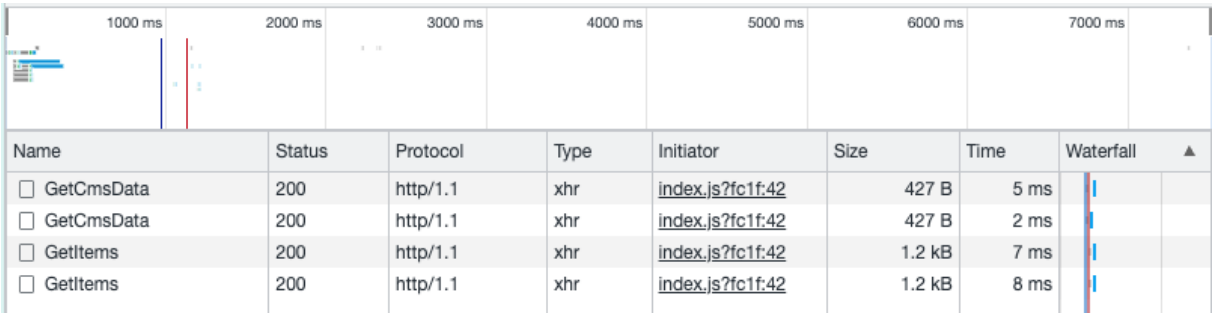
- uvodi koncept gdje poslužitelj predviđa resurse koji će biti potrebni klijentu i gura ih prije nego što klijent podnese zahtjev uz mogućnost da klijent može odbiti resurse
- uvodi koncept multipleksiranja koji naizmjenično izmjenjuje podatke bez blokiranja i preko jedne TCP veze
- HTTP/2 je binarni protokol što uvelike povećava učinkovitost u smislu sigurnosti, kompresije i multipleksiranja
- Koristi HPACK algoritam za kompresiju zaglavlja koji je otporan na napade i koristi statično Huffmanovo kodiranje

Tablica 1. HTTP verzije i usporedba

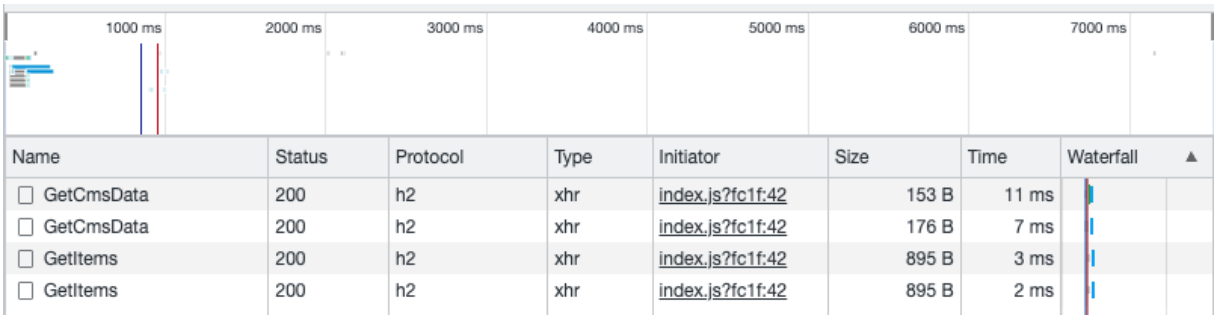
	HTTP/1.0	HTTP/1.1	HTTP/2
Glavne značajke	Svaka TCP veza služi za jedan zahtjev i odgovor	Omogućuje ponovno korištenje veze, no slobodan tok nije bio izvedivo rješenje	Koristi multipleksiranje gdje se preko jedne TCP veze mogu razmjenjivati poruke bez blokiranja
Status kodovi	16 status kodova; loše specificiranje grešaka	24 status kodova; prijava pogrešaka je brža i učinkovitija	Ostaje isto kao i kod HTTP/1.1

Mehanizam autentifikacije	Korisni osnovnu shemu provjere koja nije sigurna jer se podatci šalju u čistom tekstu	Relativno siguran zbog NTLM autentifikacije	Ostaje isto kao i kod HTTP/1.1 ali napredak postoji u novim značajkama TLS-a kao što je pogreška veze tipa Inadequate_Security
Pred memoriranje	Podrška pred memoriranja putem zaglavlja If-Modified-Since	Proširuje podršku uz dodatna polja u zaglavlju kao cache-control	Ostaje isto kao i kod HTTP/1.1 sa značajkom pred memoriranja od strane poslužitelja
Web promet	HTTP/1.1 omogućuje brže učitavanje web stranica i smanjuje promet u odnosu na HTTP/1.0, no TCP se sporo pokreće što može dovesti do zagušenja mreže		Korištenjem multipleksiranja i „server push“ značajke smanjeno je vrijeme učitavanja stranica

Na temelju teorijskog dijela odlučio sam testirati isto u praksi. Slike ispod prikazuju zahtjeve putem HTTP/1.1 i HTTP/2 protokola. Slika 5 prikazuje zahtjeve putem HTTP/1.1 protokola, a to je vidljivo u stupcu 'Protocol', a slika 6 prikazuje zahtjeve putem HTTP/2 protokola, također vidljivo u stupcu 'Protocol'. U ovom testiranju fokus je na veličini zahtjeva i odgovora, a vrijednost je vidljiva u stupcu 'Size'. Možemo vidjet kako je veličina smanjena između 25% i 50% što zaista potvrđuje teoriju iznad. Brzina zahtjeva ovdje nije ključna iz razloga što su ovi zahtjevi kreirani na lokalnoj mašini te su varijacije uvijek pristupne.



Slika 5. HTTP/1.1 zahtjevi



Name	Status	Protocol	Type	Initiator	Size	Time	Waterfall
GetCmsData	200	h2	xhr	index.js?fc1f:42	153 B	11 ms	
GetCmsData	200	h2	xhr	index.js?fc1f:42	176 B	7 ms	
GetItems	200	h2	xhr	index.js?fc1f:42	895 B	3 ms	
GetItems	200	h2	xhr	index.js?fc1f:42	895 B	2 ms	

Slika 6. HTTP/2 zahtjevi

3. Implementacija aplikacije

Praktični dio rada se sastoji od opisa i implementacije aplikacije, a podijeljen je na dva dijela, prvi dio opis i implementacija poslužitelja, a drugi dio sadrži opis i implementaciju klijenta. Aplikacija je zapravo oglasnik sa neodređenom, odnosno, aplikacija je implementirana na generički način gdje se zapravo sve može dinamički kreirati. To bi značilo da ukoliko vlasnik aplikacije odluči dodati novu kategoriju ili potkategoriju, to može napraviti bez potrebe prisustva programera. Isto vrijedi i za polja koja pojedina kategorija sadrži, uzmimo za primjer auto-moto

industriju. Automobili imaju značajke kao što su snaga, pogon, boja, karoserija i slično, dok s druge strane možemo usporediti informatičku industriju, uzmimo za primjer laptopa, laptop opet imaju neke svoje značajke koje se razlikuju od automobila, npr. broj jezgri procesora, veličina ekrana. Cilj je bio implementirati sve značajke bez pretjeranih korištenja biblioteka ili paketa trećih strana kako bi se dobio stvaran dojam i osjećaj razvoja programskih rješenja.

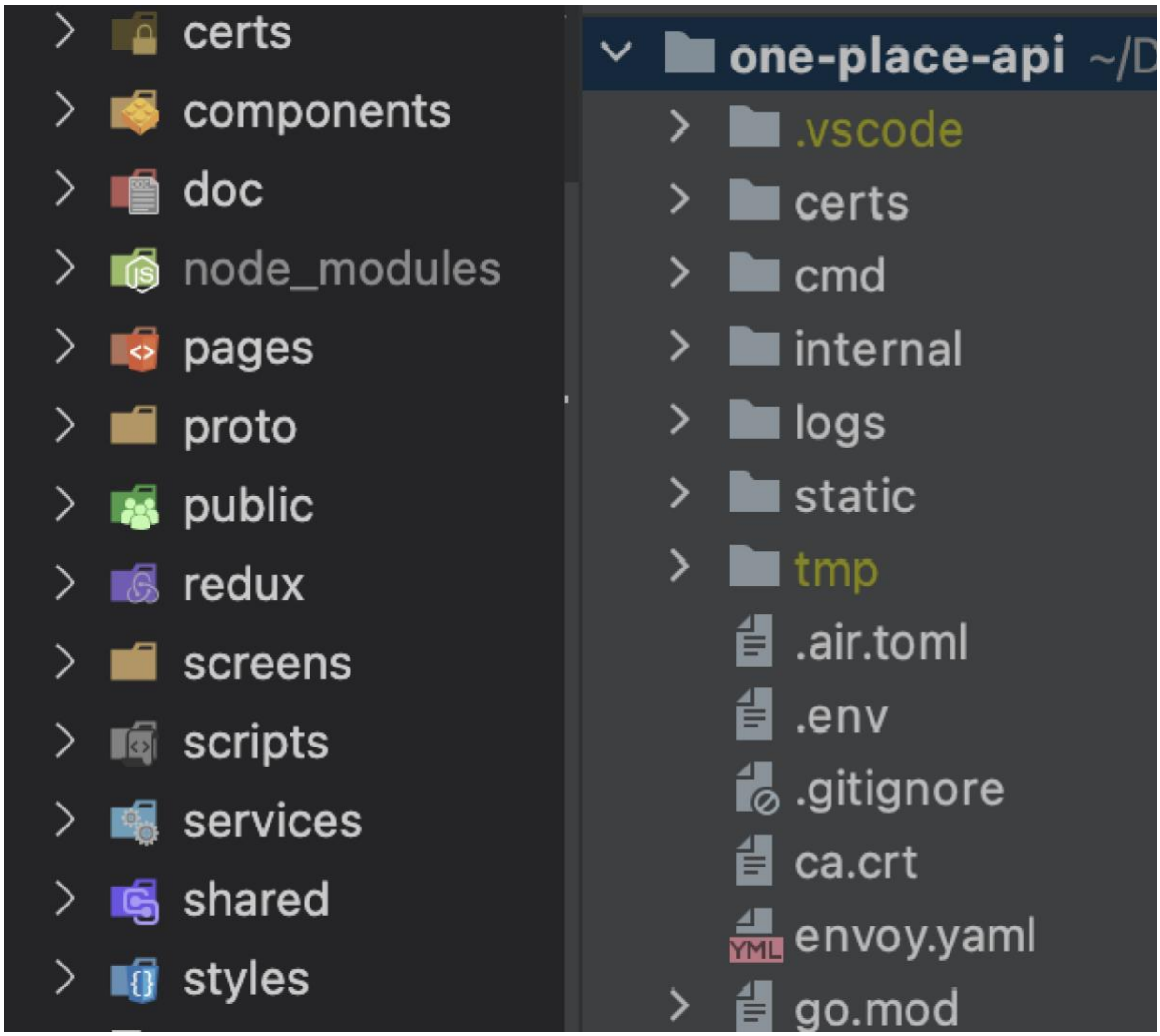
3.1. Klijent i poslužitelja

Za implementaciju klijenta koristi se programski jezik TypeScript sa bibliotekama React.js i Next.js uz biblioteku Redux Toolkit za upravljanjem stanja aplikacije, osim Redux Toolkit-a važan paket koji se još koristio je *grpc-web* koji se zapravo zaslužan za mogućnost korištenja gRPC protokola u web pregledniku. Struktura direktorija podijeljena je prema namjeni. Direktorij *certs* isto kao i kod poslužitelja sadrži SSL certifikate za kreiranje sigurnosne veze. Slijedi jedan od važnijih direktorija a to je direktorij *components* koji sadrži komponente od kojih je izgrađena web aplikacija. Zatim već spomenuta značajka Next.js-a, usmjeravanje pomoću datotečnog sustava koji se nalazi u direktoriju *pages*. *Proto* direktorij sadrži generiran kod na temelju *.proto* datoteka. Cijela logika Redux-a se nalazi u direktoriju *redux*. Direktorij *services* sadrži logiku za komunikaciju sa poslužiteljem. Na samom kraju nalazi se direktorij *shared* koji zapravo sadrži sve stvari koje nisu direktno namijenjene za pojedinu stvar već postoji mogućnost da se koriste u bilo kojem dijelu aplikacije ili za bilo koju funkcionalnost, neki primjeri toga su HOC (Higher-Order Components), konstante, funkcije koje se često koriste u svim dijelovima aplikacije.

Za implementaciju poslužitelja koristi se programski jezik Go i gRPC protokol, a razvojno okruženje Goland koji je proizveden primarno za Go od strane tvrtke JetBrains koja već dugi niz godina proizvodi jedne od najboljih razvojnih okruženja. Na stranu poslužitelja uvrštava još i PostgreSQL bazu podataka kojom Go komunicira pomoću upravljačkog programa *pgx*. Kod kreiranja strukture i rasporeda datoteka po direktorijima koristio sam se pravilima dostupna na *poveznici*. U direktoriju *certs* nalaze se SSL certifikati za kreiranje sigurnosne veze (HTTPS), unutar *cmd* direktorija nalazi se *main.go* datoteka koja je zapravo početna točka pri pokretanju cijelog sustava. Direktorij *static* služi nam za posluživanje statičkih datoteka vanjskom svijetu, a *logs* za spremanje logova aplikacije koji se kasnije mogu pregledavat po potrebi. Najvažniji direktorij je zapravo *internal* u kojem se nalazi gotova cijela funkcionalnost same aplikacije. U direktoriju *grpc* imamo nekoliko servisa

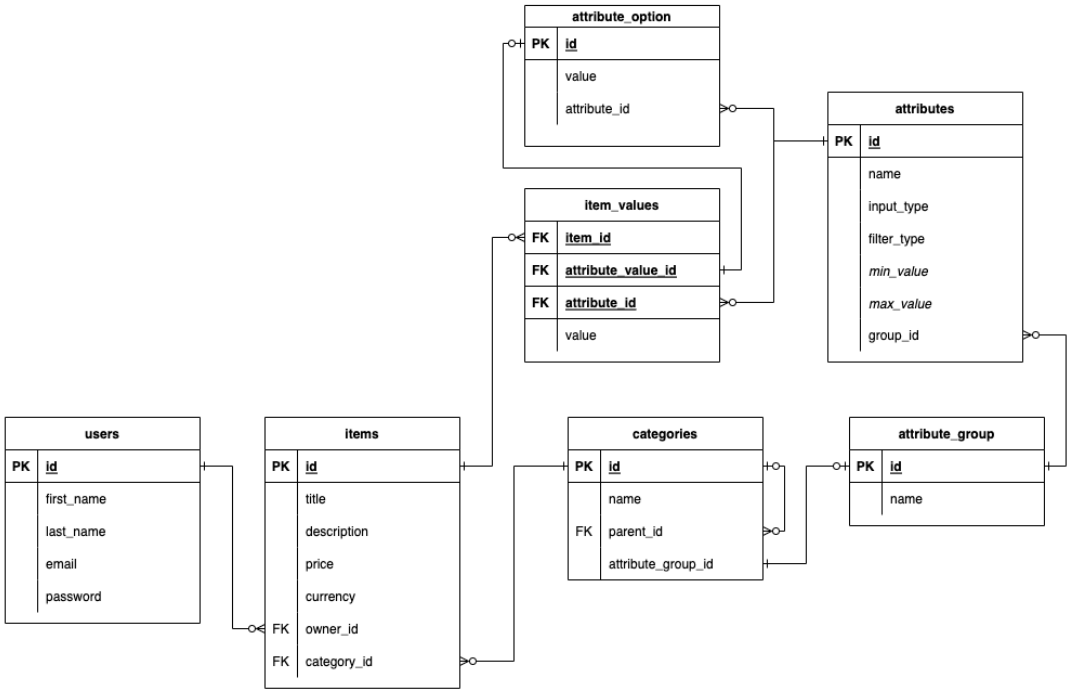
- auth – servis zaslužan za autentifikaciju
    - prijava, registracija, provjera JWT tokena, middleware
- cms – servis za upravljanje sadržajem
    - akcije upravljačke ploče

- items – servis za upravljanje artikala



Slika 7. Prikaz strukture poslužitelja i klijenta

3.1.1.ERA



Slika 8. ERA model baze podataka

Na slici 8 prikazan je ERA model baze podataka koja služi za spremanje podataka vezanih uz aplikaciju. Naša aplikacija pruža mogućnost registracije i prijave korisnika, stoga imamo tablicu *users*. Svaki korisnik može kreirati oglase, odnosno kreirati novi proizvod, stoga je korisnik povezan relacijom jedan naprema nula ili više sa tablicom *items* koja sadrži proizvode. Proizvodi imaju svoju pripadajuću kategoriju, npr. auto-moto, laptopi, knjige, kategorije su spremljene u tablicu *categories.* Jedna kategorija može sadržavati nula ili više proizvoda pa je iz toga razloga relacija jedan naprema nula ili više. Kako bi omogućili dinamičnost kroz aplikaciju, svaka kategorija ima svoju grupu atribute (*attribute_group*), a svaka grupa atributa svoje atribute (*attributes)*. Atribute dijelimo na one sa padajućim izbornikom (select) i one sa slobodnim unosom od strane korisnika (input). Na samom kraju relacija između proizvoda i pripadnih vrijednosti može se vidjeti relacijom *items, item_values, attribute_option* i *attributes*.

3.1.2. Forme

Kako se forme često koriste na klijentskoj strani, kreirao sam generičku komponentu *GenericForm.tsx* koja kao parametre (*props*) prima objekte na temelju kojih se generira forma i pripadna funkcionalnost forme. Kroz nekoliko isječka koda možemo vidjeti na koji način se zapravo komponenta može koristiti.

Na samom početku definiramo dva *interface-a* koji nam služe kako bi osigurali tipiziranost tijekom kreiranje forme i smanjili moguće greške. *RegistrationUserData* sadrži polja potrebna za formu za registraciju, a *RegistrationFormFields* osigurava da kao *key* u polje možemo unijeti samo vrijednost iz prijašnje kreiranog sučelja, a kao *value* može poprimit tip *FieldProps* koji nam zapravo govori kakvog je tipa polje za unos u formi (npr. input, select, checkbox).

```
export interface RegistrationUserData {
  firstName: string;
  lastName: string;
  email: string;
  password: string;
  repeatPassword: string;
}

export type RegistrationFormFields = {
  [key in keyof RegistrationUserData]: FieldProps;
};
```

Sada kada smo osigurali tipiziranost možemo krenuti sa kreiranjem objekata za inicijalne vrijednosti forme (*initialFormValues*) i objektom za polja u formi (*formFields*). Uočimo kako smo ovdje zapravo iskoristili gore kreirana sučelja i time dobili tipiziranost i automatsko dovršavanje od IDE-a.

```
const initialFormValues: RegistrationUserData = {
  firstName: "",
  lastName: "",
  email: "",
  password: "",
  repeatPassword: "",
};

const formFields: RegistrationFormFields = {
  firstName: {
    label: "Ime",
    type: "text",
  },
```

```
  lastName: {
    label: "Prezime",
    type: "text",
  },
  email: {
    label: "Email",
    type: "email",
  },
  password: {
    label: "Lozinka",
    type: "password",
  },
  repeatPassword: {
    label: "Ponovljena lozinka",
    type: "password",
  },
};
```

Komponenta osim objekata za kreiranje forme, prihvaća i validacijsku shemu koja služi za validaciju unosa od strane korisnika, za validaciju sam koristio paket *yup*. Kreirano sučelje *RegistrationUserData* također možemo ponovno iskoristiti i ovdje *yup.SchemaOf<RegistrationUserData>*. Pomoću metode *object()* kreiramo shemu sa pripadajućim poljima i njihovim pravilima za validaciju. Paket *yup* već sadrži neke predefinirane validacije kao što su *string, email, min, max,* ali omogućuje i dodavanje prilagođenih validacija.

```
const registrationUserDataSchema: yup.SchemaOf<RegistrationUserData> =
  yup.object({
    firstName: yup.string().required("Polje ime je obavezno"),
    lastName: yup.string().required("Polje prezime je obavezno"),
    email: yup.string().email().required("Polje email je obavezno"),
    password: yup
      .string()
      .min(5)
      .max(50)
      .required("Polje lozinka je obavezno"),
    repeatPassword: yup
      .string()
      .required()
      .oneOf([yup.ref("password"), null], "Lozinke se ne poklapaju"),
  });
```

Nakon što su objekti kreirani, možemo zapravo iskoristiti komponentu *GenericForm* na način ispod. Osim objekata koje smo kreirali iznad, prosljeđujemo još metodu (*handleFormSubmit*) koja se poziva kada se forma podnese. Kako React.js sam po sebi nema dobru podržanost za forme, odlučio sam koristiti paket *Formik*.

```
<GenericForm
  formFields={formFields}
  initialValues={initialFormValues}
  submitHandler={handleFormSubmit}
  validationSchema={registrationUserDataSchema}
  buttonText="Registracija"
/>
```

Slika ispod prikazuje zapravo formu koju smo dobili koracima iznad, te primjer validacijske poruke definirane u *yup* schemi. Istim koracima možemo kreirati bilo koju formu.

Slika 9. Primjer forme za registraciju sa porukom greške

3.1.3. Autentifikacija

Kako bi registracija iz prijašnje sekcije bila pravilno implementirana moramo se pobrinuti o sigurnosti i autentifikaciji korisnika. Sigurnost je jednim dijelom odrađena pomoću SSL certifikata, odnosno korištenja HTTPS veze koja kriptira sav promet između klijenta i poslužitelja. Osim toga potrebno je također i kreirati hash vrijednost zaporke i kao takvu spremiti u bazu podataka, kako se prilikom napada ne bih mogla pročitat lozinka u čitljivom obliku. Osim registracije korisnika, aplikacija ima mogućnost prijave korisnika. Kod prijave

korisnika ponavlja se ista radnja gdje se kreira hash vrijednost zaporke na poslužitelj i zatim uspoređujemo sa hash vrijednosti iz baze podataka. Kada govorimo o autentifikaciji moramo i spomenuti sesiju korisnika. Sesija sadrži podatke o korisniku koji su potrebni za rad sa aplikacijom. U mojem slučaju za enkripciju podataka o korisniku korišten je JWT token (JSON Web Token). Isječak koda ispod prikazuje kako se uz pomoć *bcrypt* paketa dobiva hash vrijednost zaporke. Metoda *GenerateFromPassword* osim zaporke prima i parametar *cost*. Cost je mjera koliko puta treba pokrenuti *hash*, veći broj ponavljanja poboljšava sigurnost ali i smanjuje samo brzinu kreiranje hash-a.

```
func HashPassword(password string) (string, error) {
  bytes, err := bcrypt.GenerateFromPassword([]byte(password), 8)
  return string(bytes), err
}

hashedPassword, err := HashPassword(userData.Password)
// e.g. $2a$14$/CvSTzC5xr3JtFFaw/kmneeEddFzcr/VGnLm7zlhPD21WHnOXPzTG
```

3.1.4. Middleware

Prilikom prijave korisnika generira se već spomenuti JWT token, u token spremamo ID korisnika i email. Zatim pomoću paketa *jwt-go* i HMAC-SHA metoda potpisivanja kreiramo token i potpisujemo ga dodatno pomoću tajne koja je spremljena kao varijabla okoline.

```
claims := &Claims{
  Id:    user.Id,
  Email: user.Email,
  StandardClaims: jwt.StandardClaims{
    ExpiresAt: time.Now().Add(60 * time.Minute).Unix(),
  },
}

token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
tokenString, err := token.SignedString([]byte(os.Getenv("JWT_SECRET")))
```

Neke od ruta na aplikaciji su namijenjene samo prijavljenim korisnicima, kao što je npr. stranica korisničkog profila. Na toj stranici dohvaćamo proizvode koje je korisnik kreirao. Kako bi osigurali da korisnik smije dobit odgovor na taj zahtjev, potrebno je proslijedit JWT token kroz *metada* zahtjeva. Nakon što zahtjev stigne do poslužitelja, poslužitelj provjerava da li je ruta zaštićena, ukoliko je, poziva middleware koji provjerava valjanost JWT tokena i zatim kroz

metadata prosljeđuje ID korisnika i email. Na samom kraju zahtjev dolazi do rute za koju je i prvobitno namijenjen i u tom trenutku poslužitelj je siguran da je korisnik prijavljen.

Tijekom implementacije zaštićene rute označavamo sa dodatno kreiranim polje u već spomenutim *proto* datotekama. Dodavanjem polja *option (common.is_private) = boolean;* označavamo zaštićenost metode u određenom servisu:

```
rpc GetUserItems(Empty) returns (GetItemsResponse) {
  option (common.is_private) = true;
}
```

U implementaciji trenutno koriste se samo unarni pozivi te je stoga potrebno kreirati *middleware* koji se koristi za provjeru autentifikacije i potrebno ga je proslijediti kao argument tijekom kreiranja poslužitelja. Metoda *withServerUnaryInterceptor* u sebi sadrži metodu *authorize* koja zapravo poziva metodu *ValidateAuth* iz drugog servisa.

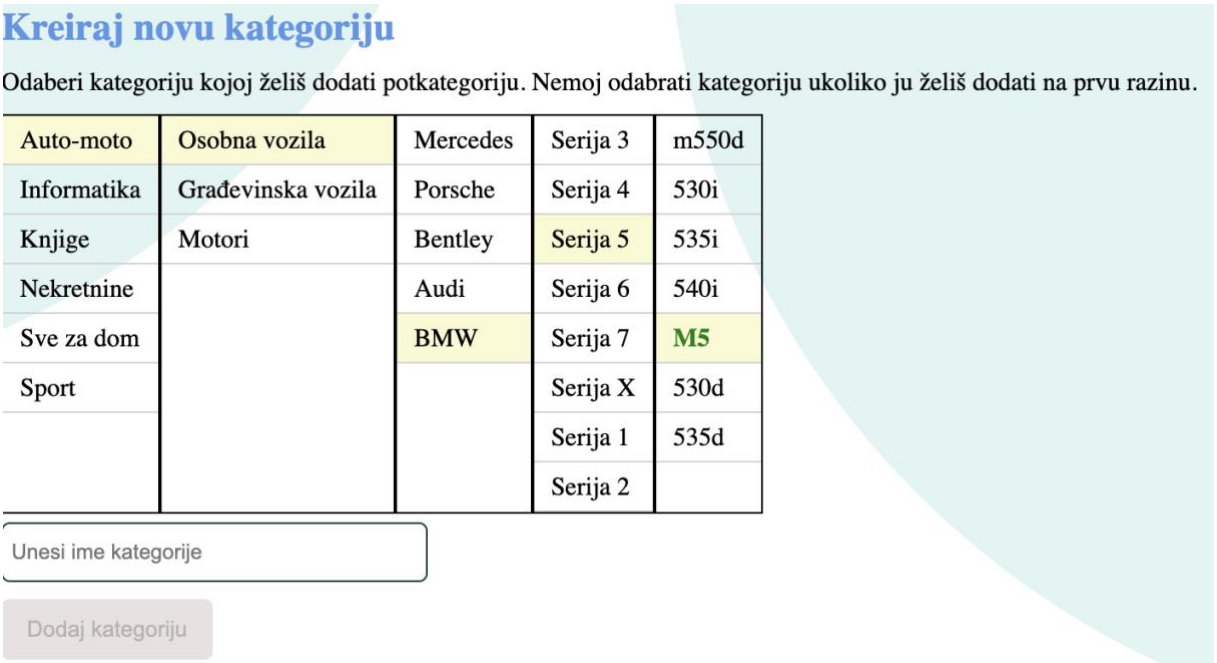
```
s := grpc.NewServer(
  withServerUnaryInterceptor(),
)
…
func authorize(ctx *context.Context) error {
  md, _ := metadata.FromIncomingContext(*ctx)

  *ctx = metadata.NewOutgoingContext(*ctx, md)
  res, err := AuthService.ValidateAuth(*ctx, &auth.ValidateUserRequest{})

  if err != nil {
    return status.Errorf(codes.Unauthenticated, err.Error())
  }

  md = metadata.Pairs(
    "userId", strconv.Itoa(int(res.UserData.Id)),
  )
  *ctx = metadata.NewIncomingContext(*ctx, md)
  return nil
}
```

Kod koji se nalazi ispod zapravo prikazuje na koji način se provjerava valjanost JWT tokena, vrlo slična procedura kao i kod kreiranja, a ovaj isječak koda se zapravo nalazi u metodi koja je korištena iznad za provjeru autentifikacije korisnika *ValidateAuth*.

```
token, err := jwt.ParseWithClaims(jwtToken, &Claims{}, parseKey(mySigningKey))

if err != nil {
  return nil, status.Error(codes.Unauthenticated, "invalid JWT token")
}

if claims, ok := token.Claims.(*Claims); ok && token.Valid {
  return &authPb.ValidateUserResponse{
    IsAuthenticated: true,
    UserData: &authPb.UserData{
      Id:    claims.Id,
      Email: claims.Email,
    },
  }, nil
} else {
  return nil, err
}
```

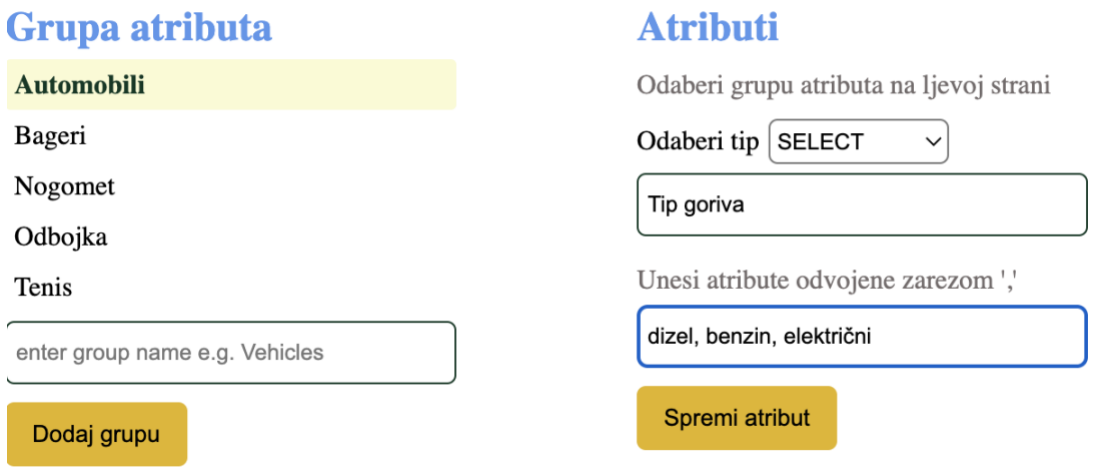
3.1.5. Upravljačka ploča

Prvi dio upravljačke ploče koristi se za kreiranje kategorija i potkategorija, broj kategorija i potkategorija nije ograničen te je pri samom dodavanju odmah vidljiv u drugim dijelovima aplikacije gdje bi se lista kategorija mogla koristiti, npr. kod pretrage proizvoda neke kategorije. Unosom imena kategorije i pritiskom



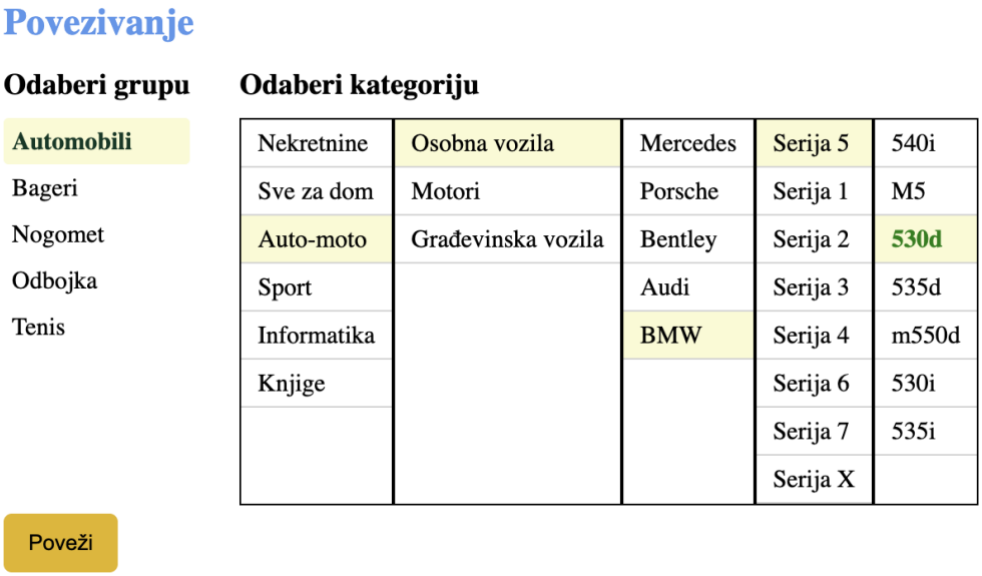
Slika 10. Kreiranje kategorija i potkategorija

Druga sekcija upravljačke ploče koristi se za kreiranje grupe atributa, svaka grupa atributa ima svoj naziv kojim raspoznajemo različite grupe, te za kreiranje atributa za određenu grupu. Atribut može biti tipa padajućeg izbornika (SELECT) ili slobodno unosa (INPUT). Kreirani atributi kasnije se koriste u formi za kreiranje proizvoda i za filtriranje proizvoda u kategoriji kojoj je grupa atributa pridodana. Slika ispod prikazuje listu grupa, te trenutno odabranu *automobili*. Forma na desnoj strani koristi se za kreiranje atributa, a atributi se automatski dodjeljuju odabranoj grupi atributa, u ovom slučaju kreirali bi atribut 'tip goriva' sa unaprijed definiranim vrijednostima 'dizel', 'benzin' i 'električni'. Kreirat ću još dodano atribut *Oblik karoserije* sa vrijednostima: limuzina, coupe, kabrio, karavan i atribut *Snaga* tipa slobodnog unosa kako bi kasnije mogli vidjeti to u akciji.



Slika 11. Kreiranje grupe atributa i atributa

Posljednja sekcija upravljačke ploče služi za povezivanje gore kreiranih dijelova, odnosno povezujem grupe atributa sa pojedinom kategorijom kako bi ona poprimila te vrijednosti. Slika ispod prikazuje povezivanje grupe atributa *Automobili* sa kategorijom *BMW Serija 5 530d*.

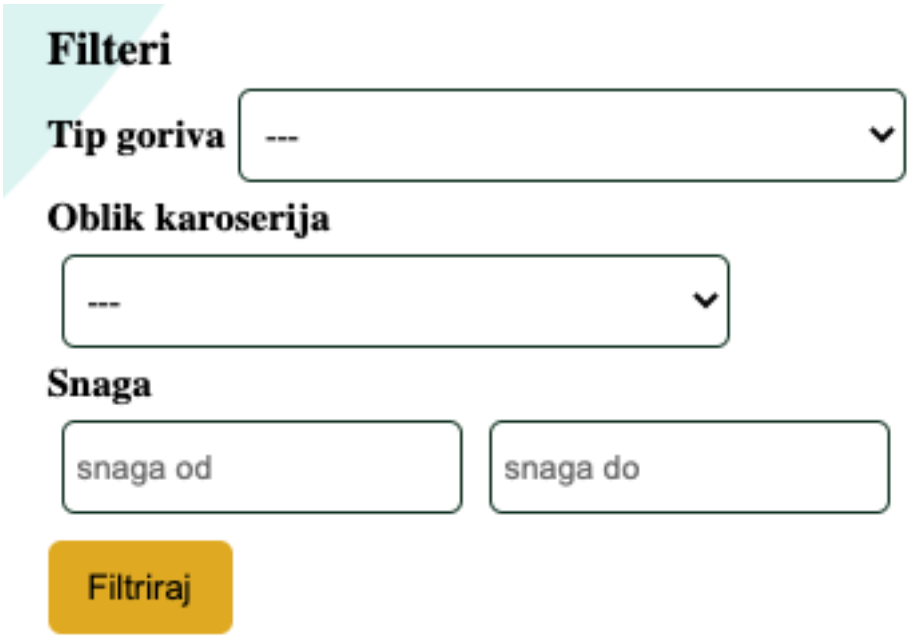


Slika 12. Povezivanje grupe atributa i kategorije

Nakon što smo dodali atribute, dodijelili ih grupi i povezali grupu sa kategorijom, pri kreiranju proizvoda za tu grupu (*BMW Serija 5 530d)* forma izgleda kao na slici ispod. Vidimo generičke atribute *naziv, opis, cijena* i *valuta*, te naknadno dodane u prijašnjim koracima: *Tip goriva, Oblik karoserija* i *snaga*.

Slika 13. Forma za kreiranje proizvoda

Osim mogućnosti pretraživanja proizvoda po kategoriji, proizvodi se mogu dodatno filtrirat na temelju atributa koji su dodijeljeni kategoriji. Sad već možemo vidjet svrhu atributa i koliko zapravo sama aplikacija postaje fleksibilnija. Atributi koji imaju tip 'SELECT' dobivaju pripadajući padajući izbornik (tip goriva, oblik karoserija), a slobodni unos 'INPUT' dobiva dva polja za unos vrijednosti *od* i *do*.



Slika 14. Prikaz filtera

Slijedi dio koda koji se koristi za slanje zahtjeva za dohvaćanje proizvoda, na početku dohvaćamo servis koji želimo koristit, kako se ovdje radi o dohvaćanju proizvoda, potreban nam je *itemsService*. Zatim inicijaliziramo zahtjev (GetItemsRequest) i pomoću setter-a postavljamo odgovarajuće vrijednosti. Nakon toga na dohvaćenom servisu pozivamo metodu *getItems* i prosljeđujemo kreirani zahtjev.

```
/**
 * Returns items from that category, returns all items if category ID is equal '0'
 *
 * @param categoryID - category ID from DB
 * @returns Promise with list of items
 */
export const getItems = async (
```

```
  categoryID?: string,
  title?: string
): Promise<GetItemsResponse.AsObject> => {
  return new Promise((resolve, reject) => {
    const itemsService = getServerService("itemsService");
    const getItemsRequest = new GetItemsRequest();

    categoryID && getItemsRequest.setCategoryId(categoryID);
    title && getItemsRequest.setTitle(title);
    itemsService?.getItems(getItemsRequest, (err, res) => {
      if (err) {
        reject();
      }
      resolve(res.toObject());
    });
  });
};
```

Možemo uočiti način kreiranja i poziva zahtjeva, svaka stvar je strogo tipizirana i poziv na poslužitelj zapravo izgleda kao običan poziv metode. Slična stvar je i na poslužiteljskoj strani, ispod možemo vidjeti metodu *GetItems*. Metode primaju dva parametra, od kojih je prvi *Context* i drugi *request* koji je pripadnog tipa za tu metodu, a vraća *response* pripadnog tipa za tu metodu i *error* ukoliko on postoji. Kako ne koristim nikakav ORM paket koji bi olakšao kreiranje upita, ali istovremeno i usporio same upite, kreiranje upita se mora obaviti ručno, odnosno upiti su onakvi kakvi bi se inače pisali direktno za bazu podataka.

Metoda na samom početku kreira upit za dohvaćanje proizvoda i provjerava postoji li upit za pripadnu kategoriju već u memoriji, ukoliko postoji vraća klijentu proizvode iz memorije, time smanjujemo opterećenost baze i povećavamo brzinu odgovora. Ukoliko odgovor nije spremljen u memoriju, metoda dohvaća konekciju na bazu podataka pomoću koje kasnije šaljemo upite. Prvi upit na bazu podataka služi za dohvaćanje svih potkategorija određene kategorije koja je proslijeđena kao polje u *request* objektu. Zatim nakon toga pomoću metode *getCategories* dobivamo listu potkategorija prigodnu za obradu na klijentskoj strani. Isto to ponavljamo za dohvaćanje proizvoda sa pripadnim atributima iz te kategorije. Također dohvaćamo i potkategorije ali samo jednu razinu niže u odnosu na prijašnji upit gdje dohvaćamo rekurzivno do zadnjeg djeteta.

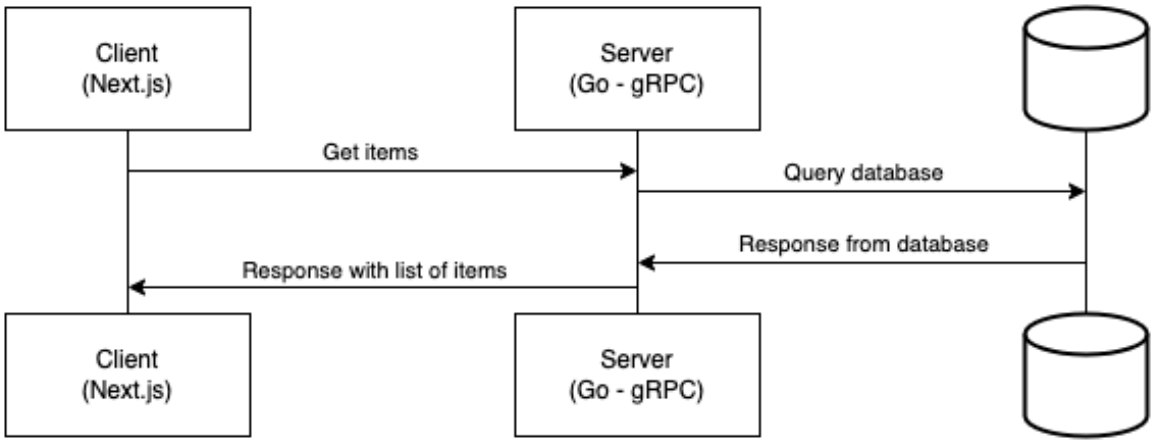
```
func (s server) GetItems(ctx context.Context, request *itemsPb.GetItemsRequest)
(*itemsPb.GetItemsResponse, error) {
```

```
  itemsQuery := qItems.GetItemsAttributeByCategory()

  cachedResult, cacheQuery := checkCache(itemsQuery, request.GetCategoryId())
  if cachedResult != nil {
    return cachedResult, nil
  }

  dbConn := database.GetDBConnection()

  rows, err := dbConn.Query(ctx, cms.GetAllChildCategories(), request.GetCategoryId())
  if err != nil {
    return nil, errors.New("failed to query subcategories")
  }
  defer rows.Close()

  allSubcategories, err := getCategories(rows, err)

  rows, err = dbConn.Query(ctx, itemsQuery, pq.Array(allSubcategories))
  if err != nil {
    return nil, errors.New("failed to query items")
  }
  defer rows.Close()

  resItems := getItems(rows)

  subCategories := getChildCategories(ctx, request, rows, dbConn)

  res := itemsPb.GetItemsResponse{Items: resItems, Categories: subCategories}
  cache.CacheService.AddNewEntry("getItems", cacheQuery, res)
  return &res, nil
}
```

3.1.6. Cache

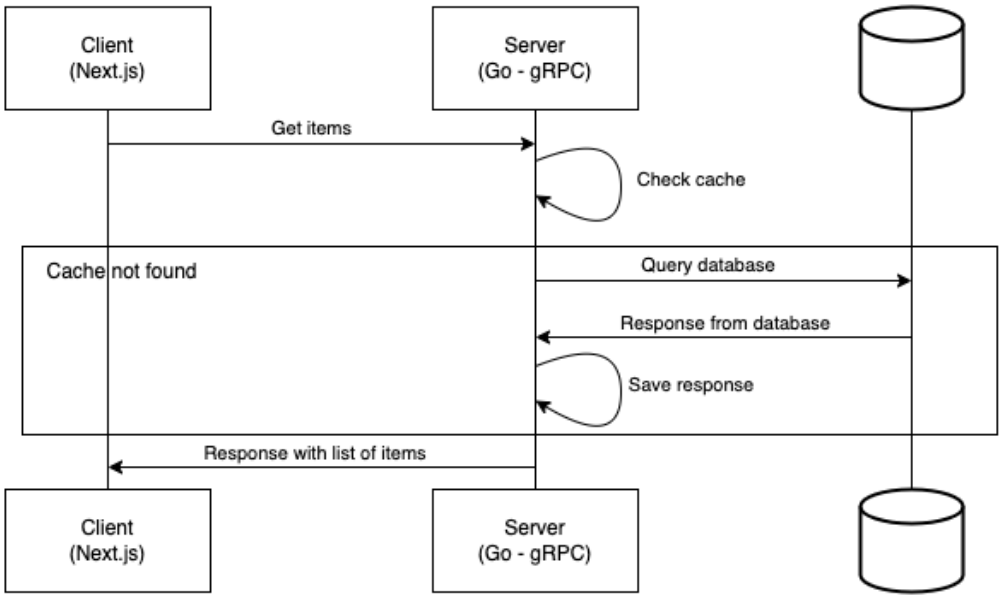
Cache implementacija je implementirana generički i fokusirana je na spremanje odgovora od strane baze podataka u memoriju kako bi se smanjio broj zahtjeva na samu bazu podataka. Time smo skratili vrijeme trajanja odgovora od strane poslužitelja, smanjili opterećenost baze podataka i povećali korisničko iskustvo kod korištenja web aplikacije.

Slika 7 prikazuje način obrade zahtjeva (u našem slučaju GetItems) bez implementiranog sustava za pred memoriranje. Klijent šalje zahtjev za listom proizvoda prema poslužitelju, poslužitelj šalje upit na bazu podataka, te nakon odgovora, prosljeđuje isti klijentu, odnosno korisniku s druge strane.



Slika 15. Prikaz toka bez cache-a

Slika 8 prikazuje način obrade zahtjev sa implementiranim sustavom pred memoriranja. Klijent također šalje zahtjev za listom proizvoda prema poslužitelju, no poslužitelj ovaj puta prvo provjerava ukoliko je odgovor na taj upit već u memoriji, te ukoliko je, šalje ga kao odgovor klijentu. Ako odgovor na taj upit ne postoji u memoriji, šalje se zahtjev na bazu podataka, odgovor se sprema u memoriju i šalje se odgovor klijentu.



Slika 16. Prikaz toka sa cache-om

Slijedi cache implementacija na poslužitelju. Na samom početku definirana je metoda *NewCache* koja se koristi pri samom pokretanju poslužitelja te kao parametar prima *config Config* objekt koji sadrži neke od konfiguracijski parametara. Jedan od bitnijih je *InvalidationInterval* koji zapravo služi za invalidaciju cache-a i brisanje podataka iz memorije ukoliko nisu ponovno dohvaćeni u tom periodu.

```
func NewCache(config Config) {
  CacheService = Cache{database: make(map[string]map[string]cacheEntry)}
  ticker := time.NewTicker(config.InvalidationInterval)
  quit := make(chan struct{})
  go func() {
    for {
      select {
      case <-ticker.C:
        invalidateCache()
      case <-quit:
        ticker.Stop()
        return
      }
    }
  }()
}
```

Nakon što smo implementirali metodu za kreiranje cache-a, možemo krenut na metodu za spremanje u cache. Implementirana je metoda *AddNewEntry* koja služi za spremanje odgovora baze podataka u memoriju. Kao parametre prima *name* koji predstavlja apstraktnu grupu svih upita za jedan dio aplikacije, npr. *items, cmsData* kako bi kasnije mogli izvršiti invalidaciju nad cijelom grupom. Ukoliko grupa ne postoji, kreiramo grupu i zatim dodajemo zahtjev i odgovor u grupu.

```
func (c *Cache) AddNewEntry(group string, query string, response interface{}) {
  if _, ok := CacheService.database[group]; !ok {
    CacheService.database[group] = make(map[string]cacheEntry)
  }
  CacheService.database[group][query] = cacheEntry{
    query:    query,
    valid:    time.Now().Add(time.Hour),
    response: response,
  }
}
```

Sljedeća važna metoda je *GetFromCache* koja zapravo dohvaća odgovor baze podataka koji je spremljen u memoriji pod imenom koje je zapravo upit korišten za dohvaćanje iz baze podataka.

```
func (c *Cache) GetFromCache(query string) interface{} {
  for _, cacheEntry := range CacheService.database {
    for _, cacheEntryInner := range cacheEntry {
      if cacheEntryInner.query == query {
        return cacheEntryInner.response
      }
    }
  }
  return nil
}
```

Nakon implementacija cache-a, pogledajmo rezultate na realnom primjeru. Pri prvom pokretanju aplikacije, cache memorija je prazna, te trajanje zahtjeva iznosi 76.794ms:

```
Request - Method:/items.ItemsService/GetItems  Duration:76.793891ms
```

Nakon prvog zahtjeva odgovor spremio u cache memoriju, te sada trajanje zahtjeva iznosi samo 9.576ms:

Request - Method:/items.ItemsService/GetItems Duration:9.576294ms

3.1.7. Pred renderiranje

Prema zadanim postavkama Next.js unaprijed renderira svaku stranicu, odnosno generira HTML za svaku stranicu umjesto da se to obavlja na klijentskoj strani pomoću JavaScript-a. Svaki generirani HTML povezan je s JavaScript kodom koji je minimalan, odnosno sadrži samo kod koji je potreban za tu stranicu, tako se smanjuje vrijeme učitavanja stranice, tek kada preglednik učita stranicu, pokreće se njen JavaScript kod i tako stranica postaje interaktivna. Ovaj proces se naziva *hydration*. [4]

U Next.js-u postoje dva oblika pred renderiranja, a razlika je u tome kada se generira HTML stranica [4]:

- statičko generiranje – preporučeno – HTML je generiran tijekom izgradnje i ponovno se koristi na svakom zahtjevu
- renderiranje na strani poslužitelja – HTML se generira na svaki zahtjev

Neke stranice zahtijevaju dohvaćanje vanjskih podataka za pred renderiranje, u tom slučaju postoje dva načina:

- sadržaj ovisi o vanjskim podacima – koristi se *getStaticProps*
- ruta stranice ovisi o vanjskim podacima – koriste se *getStaticPaths* (obično uz *getStaticProps*)

SSR (Server-Side Rendering) metoda za prikaz web stranica koje se pripremaju na serveru i šalju klijentu već unaprijed kreirane, kao što je bilo spomenuto, koristi se često za SEO i Google rangiranje. Ukoliko stranica zahtijeva dohvaćanje vanjskih podataka koji se često ažuriraju, potrebno je izvesti funkciju *getServerSideProps*. [4]

Kroz primjere u aplikaciji prikazat će se razlika u njihovoj HTML strukturi. U prvom primjeru koristit će se SSR sa metodom *getServerSideProps.* Implementacija se nalazi u datoteci *pages/search/[category].tsx*. U metodi *getServerSideProps* dohvaćamo sve proizvode

i atribute pripadne kategorije, zatim u objektu *props* (koji mora biti prisutan prilikom bilo koje od metoda pred renderiranje) vraćamo odgovore koje smo dobili od poslužitelja.

```
export const getServerSideProps: GetServerSideProps<
  Category_SSRProps
> = async ({ query, locale }) => {
  const data = await serverServices.itemsService.getItems(
    query?.category?.toString()
  );
  const fields = await serverServices.itemsService.getCategoryFields(
    query?.category?.toString() || ""
  );

  return {
    props: {
      data,
      fields,
      ...(await serverSideTranslations(locale || "en", ["common"])),
    },
  };
};
```

Nakon gore naveden implementacije, Next.js može na poslužiteljskoj stranici generirati HTML stranicu pomoću podataka dobivenih od strane poslužitelja. Na samom kraju možemo provjeriti izgled HTML koda, ovdje je prikazan samo mali isječak no možemo vidjeti kako se u jednom dijelu nalaze naši proizvodi.

```
<div class="ItemsScreen_subcategories__wBHp5">
  <h3>Subcategories</h3>
  <div class="ItemsScreen_subcategories__list__uG5BO">
    <div class="ItemsScreen_subcategories__list__item__T9ER_">Osobna vozila</div>
    <div class="ItemsScreen_subcategories__list__item__T9ER_">Građevinska vozila</div>
    <div class="ItemsScreen_subcategories__list__item__T9ER_">Motori</div>
  </div>
</div>
....
<div class="ItemsScreen_items__DIHCJ">
  <h3>Items</h3>
  <div class="ItemList_wrapper__0Pfhy">
    <div class="ListItem_wrapper__JZLDV">
      <h3>BMW 530d</h3>
```

```
      <p>Prodajem BMW 530d odlično stanje</p>
      <p>30000
        <!-- -->
        <!-- -->€
        <!-- -->
      </p>
    </div>
  </div>
</div>
```

Ukoliko ne bi koristili jednu od metoda za pred renderiranje već bi iste podatke dohvaćali na klijentskoj stranici, dobili bi HTML kod bez prikazanih proizvoda i ostalih stvari. Isječak HTML koda bez korištenja pred renderiranja prikazan je ispod. Uočimo kako ovdje unutar *ItemsScreen_items* div-a više nemamo nikakve elemente koji bi nam ukazivali na postojanje proizvoda na toj stranici, barem bi drugi alati kao Google crawler-i tako to interpretirali.

```
<div class="ItemsScreen_items__DlHCJ">
  <h3>Items</h3>
  <div class="ItemList_wrapper__0Pfhy"></div>
</div>
```

3.1.8. Internacionalizacija

Internacionalizacija, poznatija kao i18n (i – osamnaest slova – n) je proces planiranja i implementacije proizvoda tako da se lako mogu prilagoditi određenim jezicima, taj proces se naziva lokalizacija. [16]

Next.js ima ugrađenu podršku za internacionalizirano usmjeravanje. Na temelju danog popisa lokaliteta, zadanog lokaliteta i lokaliteta specifičnih za domenu Next.js će automatski upravljati usmjeravanjem. Značajka Next.js-a je također automatsko otkrivanje lokaliteta, kada korisnik posjeti aplikaciju Next.js će pokušati automatski otkrit koju lokalizaciju korisnik preferira na temelju zaglavlja *Accept-Language* i trenutne domene. Isto tako automatski će dodati atribut *lang* oznaci <*html*>. [9]

Dvije vrste upravljanja lokalizacijom putem usmjeravanja:

- *sub-path routing* – lokalizacija se nalazi u URL-u
  - */en/blog*
  - */hr/blog*
- *domain routing* – lokalizacija na različitim domenama
  - *oneplace.*com

- *oneplace.hr*

Praktični dio aplikacije sadrži lokalizaciju na dva jezika: hrvatski i engleski. Paket koji sam koristio na strani klijenta je *next-i18next*. Razlog korištenja paketa ja zato što paket podržava značajke Next.js-a kao što su SSG i SSR. Sami prijevod sprema se u *json* datotekama koje su spremljene na poslužitelju, razlog toga je lakša mogućnost proširenja funkcionalnosti upravljačke ploče sa mogućnosti dodavanja, mijenjanja i brisanja određenih zapisa u datoteci lokalizacije. Isječak koda iz *json* datoteke za lokalizaciju prikazan je ispod.

```
"navbar": {
  "home": "Početna",
  "registration": "Registracija",
  "login": "Prijava",
  "profile": "Profil",
  "items": "Proizvodi",
  "dashboard": "Upravljačka ploča"
},
```

Kako bi mogli koristiti lokalizaciju na strani klijenta potrebno je dohvatiti prijevode sa poslužitelja i prema dokumentaciji paketa omotati komponentu HOC komponentom *appWithTranslation*. Metoda osim komponente također i prihvaća konfiguraciju koja je također vidljiva ispod.

```
export default appWithTranslation(wrapper.withRedux(MyApp), nextI18NextConfig);
```

Kroz konfiguraciju definirali smo putanju do poslužitelja gdje se nalaze *json* datoteke, te *cache* objekt koji nam zapravo govori koliko dugo se prijevod zadržava u memoriji na strani klijenta kako ne bih prilikom svakog pregleda stranice nepotrebno dohvaćali podatke koji se ionako jako rijetko mijenjaju.

```
module.exports = {
 i18n: {
   defaultLocale: "en",
   locales: ["en", "hr"],
   backend: {
     loadPath: `${basePath}/static/locales/{{lng}}/{{ns}}.json`,
   },
 },
 cache: {
   enabled: true,
```

```
    expirationTime: 5 * 60 * 1000,
  },
  ns: ["common"],
  use: [I18NextHttpBackend],
};
```

U primjer možemo vidjeti način korištenja lokalizacije za navigacijsku traku, odnosno za primjer iznad. Definiramo rute koje nam se nalaze u navigacijskoj traci i u polje *name* upisujemo putanju, a u *href* rutu na koju vodi poveznica i polje *requireAuth* pomoću kojeg zabranjuje pristup određenoj ruti ukoliko korisnik nije autentificiran. Nakon definicije ruta pomoću *useTranslation* hook-a koji dolazi iz istoga paketa koristimo metodu *t* kojoj prosljeđujemo putanju do pripadnog prijevoda.

```
const routes: Route[] = [
  {
    name: "navbar.home",
    href: "/",
    requireAuth: false,
  },
  {
    name: "navbar.login",
    href: "/login",
    requireAuth: false,
  },
  {
    name: "navbar.registration",
    href: "/registration",
    requireAuth: false,
  },
  {
    name: "navbar.profile",
    href: "/profile",
    requireAuth: true,
  },
  {
    name: "navbar.dashboard",
    href: "/dashboard",
    requireAuth: true,
  },
];
```

```
const { t } = useTranslation();

const renderNavbarLinks = () => {
   return routes.map((route) => {
     if (!route.requireAuth) {
       return (
         <Link key={route.name} href={route.href}>
           <a>{t(route.name)}</a>
         </Link>
       );
     }
   });
 };
```

4. Implementacija i analiza performansi

Kako bi dodatno usporedili performanse pojedinih programskih jezika i pripadnih arhitektura, implementirao sam tri servisa. Jedan u programskom jeziku Node.js i arhitekturom REST, a druga dva u programskom jeziku Go i arhitekturama REST i gRPC.

Za testiranje REST arhitekture koristimo *ab – Apache HTTP server benchmarking tool*, a za testiranje gRPC arhitekture koristimo *ghz*.

Značajke alata ghz [17]:

- može koristiti proto datoteku, paket ili *server reflection*
- pregled rezultata u različitim formatima: CLI, CSV, JSON, HTML i InfluxData
- prilagođavanje podataka zahtjeva
- testiranje *unary*, *streaming* i *duplex* pozive

Alat *ab* služi za analizu HTTP poslužitelja (mjerenje performansi). Izvorno se koristio samo za testiranje Apache HTTP poslužitelja, ali njegov generička narav dopušta testiranje bilo kojeg web poslužitelja koji podržava HTTP/1.0 ili HTTP/1.1 [18].

4.1. Node.js – REST

Implementacija REST servera koji vraća JSON objekt na ruti „/“. Objekt sadrži *key message* i *value* koji je zapravo tekst veličine 5kb. Server se pokreće na portu *3030*.

```
const hostname = "127.0.0.1";
const port = 3030;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader("Content-Type", "application/json");
  res.end(
    JSON.stringify({
      message: lorem,
    })
  );
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Pokretanjem naredbe „*ab -k -c 150 -n 5000 http://127.0.0.1:3030*“ pokrećemo izvršavanje testiranja.

Naredba iznad ima definirani broj paralelnih virtualnih korisnika koji šalju zahtjeve, zastavica 'k', u našem slučaju je to 150. Također, definirali smo broj zahtjeva pomoću zastavice 'n', u našem slučaju broj zahtjeva iznosi 5000.

Rezultati testiranja:

```
Server Hostname:        127.0.0.1
Server Port:            3030

Document Path:          /
Document Length:        5046 bytes

Concurrency Level:      150
Time taken for tests:   0.723 seconds
Complete requests:      5000
Failed requests:        0
Keep-Alive requests:    0
Total transferred:      25765000 bytes
HTML transferred:       25230000 bytes
Requests per second:    6916.15 [#/sec] (mean)
Time per request:       21.688 [ms] (mean)
Time per request:       0.145 [ms] (mean, across all concurrent requests)
Transfer rate:          34803.61 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   0.5      1       5
Processing:    10   21   7.8     19      57
Waiting:        5   12   5.4     11      49
Total:         11   21   8.1     19      58

Percentage of the requests served within a certain time (ms)
  50%     19
  66%     22
  75%     24
  80%     24
  90%     28
  95%     37
```

```
 98%     58
 99%     58
100%     58 (longest request)
```

Na temelju rezultata možemo vidjeti da je prosječno vrijeme trajanja zahtjeva 21.68ms i brzina prijenosa 34803.61 Kbytes/sec. Većina zahtjeva obradila se u brzinom prosječnog trajanja zahtjeva, no uočavamo da je za zadnjih 5% zahtjeva potrebno bilo gotovo dvostruko više od prosječnog vremena trajanja zahtjeva što bi moglo značiti da povećanjem broja zahtjeva, trajanje zahtjeva bi mogao drastično rasti.

4.2. Go – REST

Implementacija REST servera koji vraća JSON objekt na ruti „/“. Objekt sadrži *key message* i *value* koji je zapravo tekst veličine 5kb. Server se pokreće na portu *3020*.

```
func sayHello(w http.ResponseWriter, r *http.Request) {
  msg := map[string]string{
    "message": lorem,
  }
  w.Header().Set("Content-Type", "application/json")
  json.NewEncoder(w).Encode(msg)
}

func main() {
  assignToLorem()
  http.HandleFunc("/", sayHello)
  log.Fatal(http.ListenAndServe(":3020", nil))
}
```

Pokretanjem naredbe „*ab -k -c 150 -n 5000 http://127.0.0.1:3020*“ pokrećemo izvršavanje testiranja. Naredba iznad ima definirani broj paralelnih virtualnih korisnika koji šalju zahtjeve, zastavica 'k', u našem slučaju je to 150. Također, definirali smo broj zahtjeva pomoću zastavice 'n', u našem slučaju broj zahtjeva iznosi 5000.
Rezultati testiranja:

```
Server Hostname:        127.0.0.1
```

```
Server Port:            3020

Document Path:          /
Document Length:        5077 bytes

Concurrency Level:      150
Time taken for tests:   0.389 seconds
Complete requests:      5000
Failed requests:        0
Keep-Alive requests:    0
Total transferred:      25825000 bytes
HTML transferred:       25385000 bytes
Requests per second:    12860.55 [#/sec] (mean)
Time per request:       11.664 [ms] (mean)
Time per request:       0.078 [ms] (mean, across all concurrent requests)
Transfer rate:          64867.89 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    6   1.0      6       8
Processing:     1    6   0.9      6       8
Waiting:        0    6   1.0      6       8
Total:          4   11   1.6     11      16

Percentage of the requests served within a certain time (ms)
  50%     11
  66%     12
  75%     12
  80%     13
  90%     13
  95%     14
  98%     14
  99%     15
 100%     16 (longest request)
```

Na temelju rezultata možemo vidjeti da je prosječno vrijeme trajanja zahtjeva iznosilo 11.664ms i brzina prijenosa je iznosila 64867.89 Kbytes/sec što bi značilo da bi odabirom ovih tehnologija u ovakvim okolnostima ubrzali zahtjeve za gotovo 50%. Također, odstupanja su vrlo mala što bi moglo značiti da povećanjem broja zahtjeva ili korisnika ne bi trebalo doći do drastičnih promjena u trajanju zahtjeva.

4.3. Go – gRPC

Implementacija gRPC servera koji vraća objekt veličine 5kb. Server se pokreće na portu *50000*.

```
func (s server) SayHello(_ context.Context, _ *hello.Empty) (*hello.Response, error) {
  return &hello.Response{Message: lorem}, nil
}

func main() {
  assignToLorem()
  lis, err := net.Listen("tcp", ":50000")

  if err != nil {
    log.Fatalf("Failed to listen #{err}")
  }

  s := grpc.NewServer()

  hello.RegisterHelloServiceServer(s, &server{})

  if err := s.Serve(lis); err != nil {
    log.Fatalf("Failed to start server #{err}")
  }
}
```

Pokretanjem naredbe `ghz -c 150 -n 5000 --proto=./go-grpc/hello.proto localhost:50000 --call=hello.HelloService/SayHello –insecure` pokrećemo testiranje sa istim parametrima, 150 istovremenih virtualnih korisnika i 5000 zahtjeva, dobiveni rezultati su sljedeći:

```
Summary:
  Count:  5000
  Total:  203.72 ms
  Slowest:  17.63 ms
  Fastest:  0.14 ms
  Average:  3.52 ms
  Requests/sec: 24543.38

Response time histogram:
```

```
0.144  [1]    |
1.893  [1147] |∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎
3.641  [1621] |∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎
5.390  [1537] |∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎
7.138  [420]  |∎∎∎∎∎∎∎∎∎∎
8.887  [172]  |∎∎∎∎
10.635 [55]   |∎
12.384 [31]   |∎
14.133 [6]    |
15.881 [6]    |
17.630 [4]    |

Latency distribution:
  10 % in 0.90 ms
  25 % in 2.06 ms
  50 % in 3.36 ms
  75 % in 4.58 ms
  90 % in 5.89 ms
  95 % in 7.36 ms
  99 % in 10.38 ms

Status code distribution:
  [OK]   5000 responses
```

Na temelju rezultata uočavamo kako ovdje imamo najbrže prosječno vrijeme koje iznosi 3.52ms, isto tako broj zahtjeva po sekundi iznosi 24543.38. Ovdje je distribucija solidna, također se velik broj zahtjeva zapravo kreće okvirno oko prosječnog vremena trajanja zahtjeva no vidimo i zahtjeve koji su potrajali duže. No kako je gRPC temeljen na HTTP/2 postoji mogućnost da ovo nije baš pravedna usporedba, iako je postigao najbolje rezultate vjerujem da bi postigli još bolje da se ponovo iskorištava kreirana veza između klijenta i poslužitelja.

4.4. Analiza

Nakon testiranja slijedi analiza rezultata, najvažniji podatci prikazani su u tablici dolje.

- servis 1: Node.js – REST
- servis 2: Go – REST
- servis 3: Go - gRPC

Tablica 2. Analiza rezultata

Parametar	Servis 1	Servis 2	Servis 3
Total time	0.723 s	0.389 s	0.203 s
Requests/sec	6916,15	12 860,55	24 543,38
Time per request (avg)	26.69 ms	11.66 ms	3.52 ms

Na temelju tablice možemo vrlo lako zaključit da je servis 3 uvjerljivo brži od ostalih. Usporedimo li rezultate *Requests/sec* i *Total time*, možemo uočiti da je gotovo brži za 50% od servisa 2, te gotovo 70% brži od servisa 1, dok je istovremeno *Time per request (avg)* manji za 70% u usporedbi za servisom 2, te 85% u odnosu na servis 1.

5. Zaključak

Cilj ovog rada bio je upoznati se sa konceptima implementacije i primjene novijih programskih jezika i ostalih tehnologija, te potkrijepiti teoriju kroz praktični dio, odnosno izradu web aplikacije. Na samom početku rada objašnjene su tehnologije koje su se koristile, te su istaknute njihove najvažnije značajke. Objašnjeni su programski jezici Go, JavaScript i TypeScript, te biblioteke React.js, Redux i Next.js. Nakon toga objašnjen je servis gRPC, HTTP/2 protokol i njegova usporedba sa prijašnjim verzijama, te na samom kraju objašnjena je i prikazana implementacija teorijskih dijelova kroz samu web aplikaciju.

Za vrijeme izrade web aplikacija pojavio se velik broj problema sa kojima se nisam prije susretao baš iz razloga što su gotovo sve navedene tehnologije nove za mene, a neke od njih se čak i tek sada počinju sve više i više koristiti u stvarnom svijetu. Iz tog razloga nema ni toliko resursa na internetu gdje se može naći rješenje za određeni problem kao što bi to bilo kod tehnologija koje su već duži niz godina u upotrebi.

Na samom kraju mogu reći kako je razvoj same aplikacije bio vrlo zanimljiv, pokušao sam se što manje ponavljati kroz slične ili gotovo iste funkcionalnosti, već radije isprobati implementirati što više različitih značajki same aplikacije: lokalizacija, pred renderiranje, caching, middleware, autentifikacija. Uočio sam također da svaka tehnologija ima svoju svrhu, pa tako ne bih mogao reći da bi gRPC mogao nadomjestiti REST ili obrnuto bez da se zna u koju će se svrhu koristiti. Kroz rad možemo vidjeti kako su Go i gRPC uvelike brži od REST-a, no brzina razvoja aplikacije je također znatno sporija i male promjene mogu zahtijevati puno više vremena za njihovu prilagodbu. Stoga bi moje neko mišljenje bilo da se koristi gRPC samo na poslužitelju i koristi se mikro servisna arhitektura i kreira se REST sučelje za vanjske upite, odnosno upite sa klijentske strane. Obradom ovog rada sakupio sam dosta novog znanja, iskustva i smatram kako će mi uvelike doprinijeti u budućem razvoju rješenja i van ovih tehnologija.

6. Literatura

[1] Go (bez dat.) [Na internetu]. Dostupno na: https://go.dev/

[2] Alan A.A. Donovan, Biran W. Kernighan „The Go Programming Language“, 2016. [Na internetu]. Dostupno na: https://books.google.hr/books?hl=en&lr=&id=SJHvCgAAQBAJ&oi=fnd&pg=PT8&dq=go+programming+language&ots=qAjGE_kVte&sig=861B_HFLdqp8YkW_uNISse3C5mg&redir_esc=y#v=onepage&q=go%20programming%20language&f=false

[3] „React“ (bez dat.) [Na internetu]. Dostupno na: https://reactjs.org/

[4] „Next.js“(bez dat.) [Na internetu]. Dostupno na: https://nextjs.org/

[5] Vipul A. M., Prathamesh Sonpatki „ReactJS by Example - Building Modern Web Applications with React“. 2016. [Na internetu] Dostupno na: https://github.com/srjainapur/Spring-Spring-Boot-Microservices/blob/master/ReactJS%20by%20Example%20-%20Building%20Modern%20Web%20Applications%20with%20React.pdf

[6] „Redux“ (bez dat.) [Na internetu] Dostupno na: https://redux.js.org/

[7] „JavaScript“ MDN (bez dat.) [Na internetu] Dostupno na: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[8] „TypeScript“ (bez dat.) [Na internetu] Dostupno na: https://www.typescriptlang.org/

[9] „Next.js, Vercel“ (bez dat.) [Na internetu] Dostupno na: https://nextjs.org/

[10] „CSS“ MDN (bez dat.) [Na internetu] Dostupno na: https://developer.mozilla.org/en-US/docs/Web/CSS

[11] „SASS“ (bez dat.) [Na internetu] Dostupno na: https://sass-lang.com/

[12] Linda Rosencrance, Brein Matturro „Remote Procedure Call (RPC)“, 2021 [Na internetu]. Dostupno na: https://www.techtarget.com/searchapparchitecture/definition/Remote-Procedure-Call-RPC

[13] „gRPC“ (bez dat.) [Na internetu] Dostupno na: https://grpc.io/

[14] Kasun Indrasiri, Danesh Kuruppu, „gRPC Up & Running“, 2020 [Na internetu]. Dostupno na: https://books.google.hr/books?hl=en&lr=&id=883LDwAAQBAJ&oi=fnd&pg=PR2&dq=grpc&ots=juuTgS5DAD&sig=ogcj5qlJtHh1No31wlKojnb0s0M&redir_esc=y#v=onepage&q=grpc&f=false

[15] „HTTP/1.x vs HTTP/2 – The Difference Between the Two Procols Explained“ (bez dat.) [Na internetu] Dostupno na: https://cheapsslsecurity.com/p/http2-vs-http1/#:~:text=HTTP2%20is%20much%20faster%20and,then%20the%20page%20loads%20faster

[16] „internationalization (i18n)“, 2011 [Na internetu]. Dostupno na: https://www.techtarget.com/whatis/definition/internationalization-I18N

[17] „ghz“ (bez dat.) [Na internetu] Dostupno na: https://ghz.sh/

[18] „ab – Apache HTTP server benchmarking tool“ Apache (bez dat.) [Na internetu] Dostupno na: https://httpd.apache.org/

Popis slika

Slika 1. Graf izumitelja programskog jezika Go (izvor: https://books.google.hr/books?hl=en&lr=&id=SJHvCgAAQBAJ&oi=fnd&pg=PT8&dq=go+programming+language&ots=qAjGE_j1w7&sig=r46oN668IdUmNa5gv9dd5-8c8EU&redir_esc=y#v=onepage&q=go programming language&f=falsewri) 2

Slika 2. Životni ciklus Redux-a 9

Slika 3. Primjer koda u SCSS i CSS (izvor: https://sass-lang.com/) 11

Slika 4. Tok komunikacije gRPC (izvor: https://grpc.io/) 12

Slika 5. HTTP/1.1 zahtjevi 15

Slika 6. HTTP/2 zahtjevi 16

Slika 7. Prikaz strukture poslužitelja i klijenta 18

Slika 8. ERA model baze podataka 19

Slika 9. Primjer forme za registraciju sa porukom greške 22

Slika 10. Kreiranje kategorija i potkategorija 26

Slika 11. Kreiranje grupe atributa i atributa 26

Slika 12. Povezivanje grupe atributa i kategorije 27

Slika 13. Forma za kreiranje proizvoda 28

Slika 14. Prikaz filtera 28

Slika 15. Prikaz toka bez cache-a 31

Slika 16. Prikaz toka sa cache-om 32

Popis tablica

Tablica 1. HTTP verzije i usporedba 14

Tablica 2. Analiza rezultata 46